Noname manuscript No.
(will be inserted by the editor)

Concordia: Google© for Malware

Timothy Daly · Rick Linger · Stacy Prowell · Luanne Burns

Received: March 26, 2010 / Accepted: February 23, 2011

Abstract This paper introduces Concordia [4], a new architecture for automating generalization of program structures and recognition of common patterns for malware analysis. By using massively parallel processing on large malware program sets Concordia can recognize common code sequences, such as loop constructs, if-then-else structures, and subroutine calls, as well as common subroutine sequences. The Concordia architecture generalizes the recognized elements for collection into invariant forms. The invariant forms can be used by an analyst to understand a particular program. These invariant forms can be used to classify large numbers of programs automatically.

Concordia gathers and organizes expertise from malware specialists in a way that can be used by organizations to recognize and respond to malware threats. Recent trends have shown that threats are becoming more focused and specific, which means that standard, signature-based approaches to malware will no longer be effective. Research and development in new malware analysis technology is required to respond to this trend.

The end result could be characterized as a "Google© for Malware", for rapidly recognizing malicious software.

Keywords Concordia, malware, parallel, generalization

Perception means making things discrete – Leslie Lamport

1 Learning to Swim during the Flood

Malware is flooding the Internet. CERT currently has a catalog of many millions of malware programs, with new ones arriving at a rate above 50,000 per week. Not all of these programs are unique and not all of them are malicious.

Unfortunately there is no generic classification scheme. The current systems use "signatures", which are small fragments of the program, to recognize threats that have been seen in many places. This technique has several shortcomings. It is effective only

Timothy Daly E-mail: daly@cert.org · Rick Linger E-mail: rick.linger@gmail.com · Stacy Prowell E-mail: prowellsj@ornl.gov · Luanne Burns E-mail: luanne.burns@jhuapl.edu

after the fact. It depends on malware that does not change its signature. It depends on the fact that the malware is widespread.

Signature verification will not be effective against the new wave of organization- and system-specific targeted attacks. We need to leverage our expertise, capture it into an automated system, and design the system so it can rapidly adapt to incoming threats.

In some ways this is the same kind of problem that credit card companies have. In response, they created automated methods to detect patterns of card usage which might be fraudulent. The potentially bad card usage can be dealt with by agents trained to handle specific problems.

Concordia aims to automate detection of patterns found in malware so that potentially bad programs can be dealt with by analysts trained to handle specific problems. In addition, as the agents learn new information there are mechanisms to add their expertise back into the system.

2 Sorting the Pretty-good from the Maybe-bad

Much malware analysis today requires resource-intensive use of highly skilled analysts. Unfortunately this does not scale. We cannot hire enough analysts to deal with the volume of malware. We need to do the analysis at machine speeds. But we also need to be able to capture the expertise of an analyst so it can be used, shared, and automated.

Concordia is designed to gather and organize expertise from malware specialists in a way that can be used organizations to recognize and respond to malware threats. Recent trends have shown that threats are becoming more focused and specific, which means that standard, signature-based approaches to malware will no longer be effective.

2.1 Whitelisting and Blacklisting

Concordia uses classification and generalization to categorize incoming programs. This categorization can be most useful along the “trust” axis between fully “whitelisted” (trusted) programs and fully “blacklisted” (malicious) programs.

Concordia can be independently trained on known-good programs, such as normal Windows© and Linux© programs which ship with the operating system. These can be verified, checksummed, and tagged as “trusted”. Concordia can also be independently trained on the known malicious malware programs. These can be verified, checksummed, and tagged as “malicious”.

Independent, multi-layer perceptrons can be trained to offer “opinions” along this trust axis or other axes based on attributes of interest. These “opinion generators” can be added or ignored to match the task.

2.2 Embracing Uncertainty

Christopher Bishop, in his recent Turing award lecture[1], uses a combination of bayesian reasoning, graph theory, and message passing to create an efficient way to search very large, uncertain, problem spaces. In particular, he stresses the development of models of the domain in order to structure the search space. His models are mathematically well-founded. His code for rapidly constructing models is available for download.

In the malware domain, the idea of a “family” relationship is a likely basis for a model. Malware authors tend to use automated tools to obfuscate their code and there are a limited number of these tools. They also tend to use categories of techniques, such as buffer overflows, against common targets, such as the browser. A model of these “families” of attack vectors (essentially a “threat matrix”) gives a firm foundation for exploiting Bishop’s bayesian graphs. Bishop is careful to point out that his techniques scale to handle large-scale datasets.

2.3 Cord Enzymes

Concordia grinds programs into chunks (a process called “chipping”), breaking programs into sequences of length 2, of length 3, of length 4, and so on to form chunks of fixed size. These chunks are stored in a repository that contains the sum total of all of the information related to the chunks and the original program. These chunks form the raw material that is used for analysis. They are categorized, generalized, associated with invariant forms, associated with pattern matching routines to form “enzymes” which recognize the chunk patterns, and used as source patterns for SQL-style queries by an analyst.

By creating pattern matching routines with each chunk, the “enzyme”, Concordia can recognize the chunk in a larger program. Consider the repository of Concordia as a large pool of “enzymes” that can match up with portions of a new program and attach themselves. Programs can be “dipped” into this pool. The result is a copy of the program with information attached.

This information is generalized with the goal of forming human-readable results that analysts can use to quickly identify program structure and function.

3 Tagging Dead Fish is not Learning to Swim

3.1 Slow manual analysis

Currently we analyze programs by hand. This approach is extremely costly. It involves human labor and it uses our most precious resource, the malware analyst, to all parts of code, even sections that may be intentionally inserted no-op code, that is, code with no functional effect and thus irrelevant to the problem.

Current processes for malware analysis are overwhelmed by the scale of the problem. Due to the volume of programs, almost all of them are put in a catalog for later analysis. Unfortunately, due to the volume and the lack of good automation, almost none of the information contained in these programs is useful. Since the programs are in binary form we do not have the source code. This means we must have tools which work bottom-up to understand the actual program behavior from instruction sequences. Malware by its nature is structured and obfuscated to make this hard for humans and machines.

We must find a way to leverage the material we have in hand to help us recognize incoming malware that has similar function. Only by using a deep analysis of the malware catalog can we hope to extract this information.

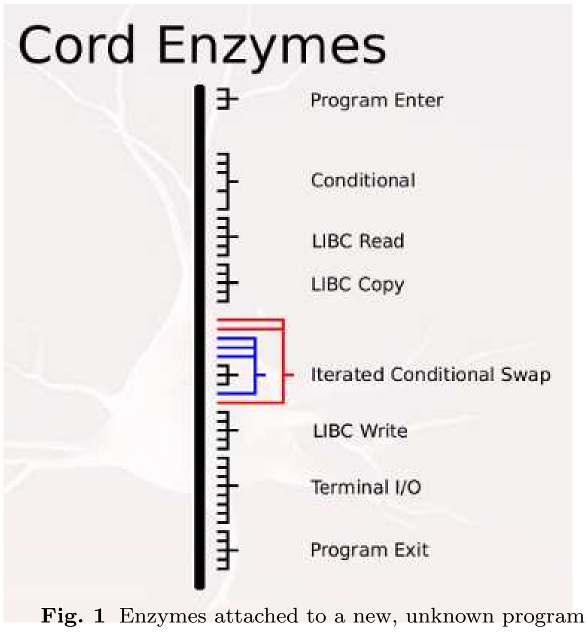


Fig. 1 Enzymes attached to a new, unknown program

3.2 Working at machine speeds

Automating the analysis task is the obvious solution. But we need to base the automation on a firm mathematical foundation. Function Extraction (FX) technology uses the ground truth semantics of the machine instructions to deliver mathematically correct results. Recent in-progress research has shown that FX analysis can work across wildly varying machine architectures and instruction sets. The original FX work was based on the Intel© instruction set but the new research uses the PIC18 microprocessor, a harvard architecture with a split program and data memory.

This shows that we can leverage FX technology to extract function semantics, not only from main line processors, but also from popular embedded processors used in supervisory control and data acquisition (SCADA) systems. Since these are deployed to control critical infrastructure such as the electric power grid it is vital that we develop the technology to recognize and react to attacks.

The mathematical foundation gives us confidence that the behavior we extract is the exact behavior of the program. All malware programs have the weakness that no matter how they obfuscate the code, they cannot obfuscate the ultimate behavior.

3.3 Moving to the leading edge

The ultimate goal is to be able to process, analyze, and classfiy new programs in real time with high confidence. This will allow isolation of potential malware to be stopped at the border.

The work necessary to build the classification system involves a large up-front effort in machine learning, model building, training and expertise capture. Concordia can then be fed a continuous stream of questionable programs. With sufficient hardware bandwidth it should be possible to process new programs in real time.

Using delta-sigma modulation techniques we can provide an incremental feedback scheme to track changes in the incoming stream. These feedback loops can be used to recognize “drift” from the “family” categories. This “drift” can be used to propose new “family” nodes in the Bishop graphs and sub-block statistical techniques can be used to validate discovery of a potential new node in the family graph. This would correspond to recognizing changes such as a new packer program or a new browser attack vector.

4 Learn, Generalize, and Predict

Concordia is loosely based on the neocortex model of the brain[8,9] using the idea of layers of structure to organize the system internals. Machine Learning [15] techniques are used within the layers to take advantage of the best available techniques for learning and adapting to the task.

4.1 Brain architecture

In order to organize Concordia we break the system into several layers so that we can separate concerns and focus our attention. Jeff Hawkins proposes a mechanism whereby the brain can learn from low-level patterns.

Hawkins perceives six physical layers of neurons in the human cortex. He speculates that these six layers have a series of functions based on levels of abstraction, classification, and generalization of input stimuli. Whether or not the brain actually functions as proposed, the architecture and separation of concerns that Hawkins defines is a good starting point for generalization of incoming information.

4.2 Machine Learning

Concordia is a mixture of unsupervised and supervised learning [7,14,15].

Unsupervised learning involves clustering data [10] without knowing the correct answers. Metrics for clusters are provided by functions on code chunks, such as behavior computed by Function Extraction[13]. The system is looking to provide structure to the data. This is used in the first three layers of Concordia to form groups of program fragments.

The key problem of the unsupervised learning is to develop metrics which can be used to distinguish between similar and dissimilar program fragments.

We have been developing Function Extraction technology which allows us to compute the “as built” behavior of a code chunk. This is ideal in a Concordia environment because we choose to work with small, usually linear code fragments. We have also proposed a process for uniform state and time space analysis, called Lambda abstraction. This allows us to recognize patterns in both space and time.

Supervised learning provides a data set with the correct answers. The machine learns the association between the data and the answers. Training values are provided by the metric functions applied to code chunks. The hypothesis function takes a new program and predicts whether it is malware.

Supervised learning can be applied by the analyst to associate tags with prior unsupervised clusters. It is also used to create new clusters based on user queries from the prediction layer. This is used mainly in the upper three layers.

Because Concordia is a continous processing system new decisions and new programs are fed back to the system as a form of reinforcement learning. The reward function is provided by the analyst feedback.

4.3 MapReduce

It is important to understand the MapReduce idea [5] since it is used heavily throughout Concordia. MapReduce is needed to handle the large volume. Google© processes many datasets larger than one Petabyte using MapReduce. Hadoop [22] is a free, open source implementation of MapReduce. We quote from Dean and Ghemawat:

> MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model...
>
> The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs....
> *Map*, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the *Reduce* function.

To process the code fragments from a single program;

```
map(String program, List chipsOfSize2)
  for chip in chipsOfSize2:
    output('twochip',chip)
```

This might generate millions of key-values pairs of ('twochip',code-sequence)

> The *Reduce* function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges together these values to form a possibly smaller set of values.

The corresponding *Reduce* function receives the tag ('twochip' in this case) and a list of all of the length-2 code sequences.

```
reduce(String tag, Iterator listOfCodeSequences)
  bucket := hashtable()
  for code in listOfCodeSequences:
    bucket(metric(code)) := code
  for key in hashtable:
    output(key,hash-value)
```

This would group the list of code fragments of length 2 into hash buckets based on one of the metrics, forming sets of equivalent code.

4.4 Neocortex architecture

Following Hawkins [9], Concordia is organized into six layers, similar in spirit to the neocortex of the brain. Given the brain's obvious strength at pattern recognition we use it as an organizational guide.

Concordia's six layers are Recognition, Equivalence, Classification, Generalization, Invariance, and Prediction. The flow of information through the system transforms a program into useful information.

Recognition breaks programs into many-sized chunks of code in a process we call "chipping". Equivalence examines each chip and adds measurements. Classification tries to create sets by examining the measures. These first three layers are (mostly) classical unsupervised learning techniques.

Once we have the sets of code chips, Generalization tries to name them. The Invariance layer chooses a standard representation for the pile. Finally, the Prediction layer interacts with the end user to capture the human-level expertise and deliver results. Human input to the system drives these upper three layers as a form of supervised learning.

The system can be fed code sequences with associated tags. This "known good" information, such as a tagged library, allows Concordia to know the names of various routines, such as "sort", "strlen", etc. Any program code that has the same behavior and ends up in the pile with these tagged fragments is now known.

The system can answer queries about programs which provides different ways to cluster the input. This gives a Google-like ability to search for interesting properties.

Given a new program, the main question to be answered is, "Is this program malicious?".

4.5 Recognition

In this layer, the MapReduce process is applied to chip and tag the program. There are multiple, parallel MapReducers each of which is chipping the program to their particular size. There are also multiple, parallel MapReducers working on many programs at the same time.

The basic steps of the MapReducer chipper program (the lowest level) follow the MapReduce plan.

1. Concordia forks a master node and worker nodes
2. The master node assigns map, combine and reduce functions to workers
3. The map worker connects to its assigned input stream for chipping
4. Input data chunk is replicated locally
5. Chips are written to local mapper storage
6. The combine worker locally reduces the data to vectors
7. Reduce workers connect to map storage nodes and output storage

4.6 Equivalence

This layer is concerned with feature extraction of the code chunks. We want to define equivalence functions so that multiple expressions of the same operation, for example, swapping two elements, end up in the same class. The end result of the Equivalence process is a set of structures called a cord, similar to Google's protobuf structures but with a richer format. It holds metrics associated with each code fragment.

In this layer, MapReduce is applied multiple times, each time moving the data into a more abstract state. We are essentially classifying each code chunk by each feature vector. Each pass refines the results of the prior pass, splitting the sets according to each feature.

Feature computation mechanisms try to abstract away from the actual code sequence in order to enlarge the set of programs that fall into a given pile. There have been several proposed feature refinements, three of which are discussed here.

4.6.1 Lambda Reduction

Lambda reductionn is accomplished by multiple passes of MapReduce, the first pass abstracts away the exact locations, the second abstracts away the exact machine specific instructions, and additional passes use normal compiler techniques like lifetime analysis and code lifting to reduce the result to a time-and-space neutral form.

In this example we show how a swap sequence which involves both registers and memory is first transformed by choosing a machine-neutral numeric naming convention. Next we change to machine-neutral move instructions, making memory-to-memory, memory-to-register, and register-to-register operations uniform. Finally, we apply a lifetime analysis to extract a machine-neutral swap sequence.

4.6.2 Function Extraction [13]

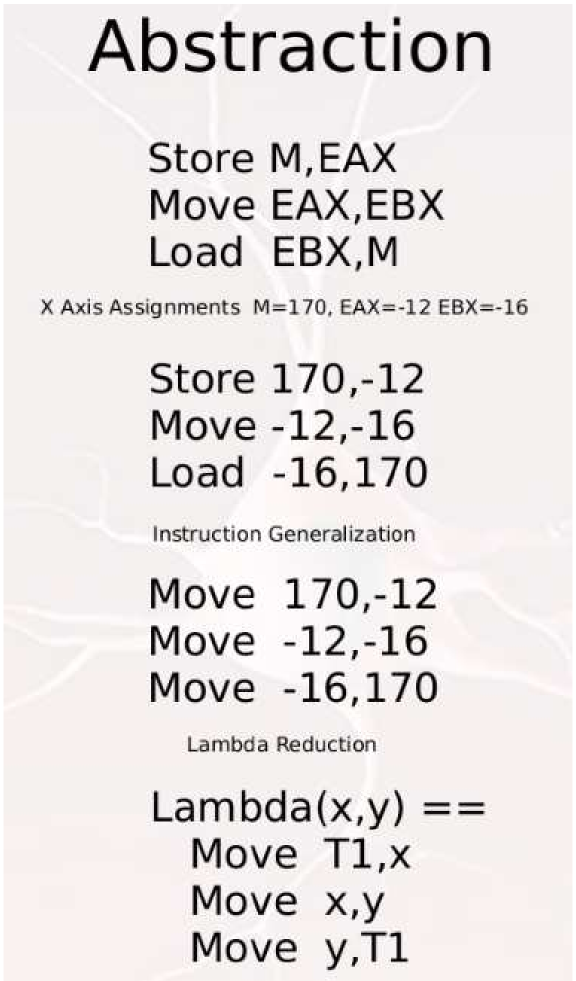
Function extraction (FX) technology under development by CERT is directed to automated computation of the full functional behavior of programs. The FX process begins by expressing the semantics of each machine instruction in an input program as a conditional concurrent assignment (CCA). The CCA captures all of the functional behavior of each instruction including side-effects such as 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 through the mathematical process of function composition to create new combined CCAs that capture the net behavior of the program.

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, including removing long sequences of instructions that do nothing but confuse the analyst. Behavior computation reveals when 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 functionals across different implementations.

Because of the "many implementation, one function" property the clusters of program fragments have stong behavior bonds. Unlike other measures of similarity, FX can recognize equivalent behavior despite large variations in code sequences.



Abstraction

```
Store M,EAX
Move EAX,EBX
Load  EBX,M
```

X Axis Assignments M=170, EAX=-12 EBX=-16

```
Store 170,-12
Move -12,-16
Load  -16,170
```

Instruction Generalization

```
Move  170,-12
Move  -12,-16
Move  -16,170
```

Lambda Reduction

```
Lambda(x,y) ==
  Move  T1,x
  Move  x,y
  Move  y,T1
```

Fig. 2 Lambda reduction of a swap sequence

4.6.3 Perceptron Recognizers [2]

We use a form of perceptron recognizers without reference to the geometric language. Instead we have a cord picture language that recognizes certain source features, applies a set of filters, and then joins the results. This is a disjunctive normal form of pattern. These patterns can be cascaded to form higher-level perceptron recognizers.

We start with a "seed" set of recognizers for common patterns like conditionals, loops, call-stack processing, subroutine entry and exit. At a higher level we have recognizers for initialization which is a loop of assignments, search which is a loop of compares, maps which is a loop of function calls, and other such common loop patterns.

We also have structural recognizers to handle memory shape, finding common patterns such as arrays, linked lists, and structs. Building upon these are specific recognizers for things like structs used as objects which combine storage pointers with function pointers and other such common patterns.

The perceptron recognizers can be improved by user or analyst input using the picture cord language which will be explained in the section on Prediction.

4.7 Classification

MapReduce [23] is used in machine learning at Google© for their AdSense advertising machinery. Hadoop [22] is Yahoo's open source version of the same technology. MapReduce is a several step process:

1. Map(k,v) $->$ ($k^{'}$,$v^{'}$) – Associate a key and a value for every unit of input
2. Combine($k^{'}$,$v^{'}$) by $k^{'}$ $->$ ($k^{'}$,$v^{'}$[]) – Locally sort by the keys and collect the results into arrays of values by key (at the mapper nodes)
3. Group($k^{'}$,$v^{'}$) by $k^{'}$ $->$ ($k^{'}$,$v^{'}$[]) – Globally sort by the keys and collect the results into arrays of values by key (at the reducer nodes)
4. Reduce($k^{'}$,$v^{'}$[]) $->$ $v^{''}$ – Reduce the array values into a final result.

The k are keys and the v are values. Concordia uses this key-value pair in many ways. Code sequences are tagged where the key is the code sequence and the value is the tag. This allows code sequences with known meanings, such as a sort from a library routine, which are tagged on input and carried through the process. Using this technique we are able to extract meaningful names for code sequences.

Because our data is more complex we adopt a structured model for data whereas MapReduce normally uses strings. Recently Google© has support for *protomsg* structured data [17] but we use a data structure, called a cord, with a richer format.

Choosing training examples with the smallest function margins for a support vector machine (this is, an optimal margin classifier) allows us to reduce the noise of choosing separation hyperplanes. These hand chosen examples can be fed into the MapReduce machinery and get associated with the code.

In general, the support vector machine model needs a kernel that will strongly distinguish between inputs. Because FX is invariant in the face of many code sequences it serves as a useful kernel for the support vector machine.

Once we have reduced the input to sets of equivalence classes we want to classify the code fragments along various axes. There are two kinds of classification techniques applied. One kind is from unsupervised learning where we try to extract the natural clustering using k-means techniques. Large numbers of clusters will emerge due to the variation in code fragments.

A second kind of clustering is from supervised learning where we have known-good data points and we cluster around those points. Given the large number of code fragments the supervised clusters will only classify a small subset reliably. This subset will be the matches for the tagged input sequences.

The multiple clustering metrics, similar to feature functions in [21], serve as a basis for classification. The elements of the feature vector are derived from the training data using functional expansion. The features of each class are derived from the functions $\phi_{ij}(x)$ (e.g. $\phi_{i1}(x)$ might be the value assigned by the lambda reduction function for the code sequence x). These values are combined in a polynomial with weights c_{ij}.

Maximizing the polynomial

$$f(x) = \sum_{1}^{n} c_{ij}\phi_{ij}(x)$$

provides the centers for clustering. This is similar to k-means clustering but provides a way to reduce the dimensions of the space while still allowing complex functions of the arguments for metrics.

4.8 Generalization

Now we move to supervised learning. In the first three layers, the unsupervised learning algorithms were general purpose since they had no external criteria to measure their progress. Now we have task-specific information, provided by the user, to judge and direct our attention.

One of the primary tasks is to name pieces of code with meaningful labels. This reification step has two components, a bottom-up approach and a top-down approach.

In the bottom-up approach there are many different sets of code. We would like to meaningfully label each pile. One technique is "name injection". We tag a library of code and feed it into the system. Each of these tagged code fragments will end up classified into a pile. For instance, we can feed in the string length (STRLEN) library routine. When the Function Extraction classifier is run this code will fall into a pile of other semantically equivalent code fragments. We can safely conclude that all of the code in this pile represents STRLEN.

By judicious choice of labeled inputs we can find whole classes of routines for sorting, searching, argument parsing, network access, etc.

In the top-down approach the user applies labels to code sequences of programs being examined. These labels are then applied to the code sets representing the program chunk. Since this information is remembered by the system it becomes a method to capture human expertise. One user can benefit from the insights of others.

4.9 Invariance

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

Internally we use a language derived from Chandy and Misra's Unity [3] language. This language gives us an algebra we can use to compute invariant behavior of program segments.

An interesting topic to explore is the meta-characterization of malware into classes such as denial-of-service, rootkit, worms, keyloggers, and other human-chosen ideas. These clearly need to be imposed in a top-down fashion.

We know from the analyst that certain library routines were tagged and added to the repository. Code sequences are very sensitive to compiler switch settings. We know the compiler switches that were associated with this particular code sequence so when we find this code sequence in the program being analyzed we can infer that the whole program was likely compiled with similar switches.

4.10 Prediction

The repository represents the accumulated experience of analyzing many malware programs 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 can do a SQL-style SELECT from the repository. It would be possible to find all programs that were compiled with GCC 4.2, using the virtual instruction set, and accessing the network. At minimum, it might be possible to conclude that this program is already in the repository but with some different kind of obsfuscation applied.

In addition, the Unity language [3] give us a notation to accurately describe the type of behavior we are trying to isolate. The end user has a "cord picture language" that allows expressions to be filtered and joined in a disjunctive normal form. These expressions can be applied in the classification layer to select a subset of the program chunks for further study.

The Unity language derives from Hoare's work [11] on Communicating Sequential Processes (CSP) and there is an isomorphism between CSP and Petri Nets [20], a visual notation. Presenting program results using this Petri Net notation allows us to graphically represent and manipulate key concepts, such as "dead nodes". These dead nodes are currently found by using Function Extraction technology where it is known as no-op elimination.

In addition to Unity, Christopher Bishop [1] stresses the importance of expressing expertise in graphical format. Bishop's Infer.net technology offers the ability to efficiently walk multi-million node networks using only partial information. Using his graphical technique we can develop "family" relationships between malware. This relationship tree gives us the likelyhood of identifying a piece of code as malware based on its association with other malware containing similar code sequences. Given the size of our data and the partial amount of information this technique is ideally suited for making statistically valid inferences efficiently.

These user interfaces to Concordia gives the analyst a powerful set of tools to visualize the behavior of programs. The tools are mathematically sound so the results can be manipulated with confidence.

5 Conclusion

Concordia has the ability to cope with existing threats. We can leverage the large catalog of malware at CERT as primary material for learning categories. In particular, we can use previously identified malware in a supervised learning role to establish the identity of some of the categories.

One way to leverage Concordia is as a real-time classifier for incoming programs. In fact, given the machine learning technology it might be possible to derive a set of features which are relatively easy to compute but which form a strong indication of the type of program. If these classifiers and boundaries exist it would be possible to derive standalone tools for first-guess classification.

One key advantage of Concordia is the ability to adapt to dynamically changing threats. New threats will form their own clusters. These new clusters provide an early warning system for emerging threats.

A recent change in the malware landscape is the emerging threat of company-specific malware. That is, malware which is not spread as spam but is specific to one target. Anti-virus companies would never see these attacks since they are not generally available. And even if they did see the attack it would be too late. Concordia can see these attacks as they happen which allows the programs to be quarantined at the door.

Concordia has the ability to capture and share analyst expertise so any annotations provided by the analyst can be pushed into the repository, tagged, and made available to other analysts. This allows the ongoing accumulation of expertise, making the system ever more effective.

6 Draw on all we know to make sound decisions

Concordia draws on several disparate streams of work to synthesize a tool for handling the volume and complexity of malware.

We draw on parallel processing (MapReduce) because the malware programs are independent and can be processed in parallel. This gives us the ability to handle the volume.

We draw on the machine learning area to provide an architecture to structure both the unsupervised and supervised learning. This gives us the ability to adapt to changing threats in the incoming stream. It also provides a basis for generalization.

We draw on formal methods in program understanding, such as Function Extraction, to provide a mathematically sound basis for computing program behavior. This gives us strong guarantees that the classification algorithms in machine learning can use.

We draw on formal methods of program description, such as Unity, to provide a mathematically sound method of describing and manipulating program behavior. Simplification techniques applied to Unity programs provide another means of generalization.

We draw on mathematically sound visual descriptions, such as Petri Nets, to provide a means of presenting program descriptions to the analyst in intuitive ways. The same mechanism allow construction of queries.

We draw on Bishop's bayesian graph networks to build a "family" structure which captures human expertise about malware. This provides a way to understand connections between apparently disparate attacks. Recognition of "family" techniques in programs implies that the programs have a common attacker source.

Ultimately, Concordia's synthesis of these streams into a cohesive tool will allow rapid and flexible response to malware. In the hands of an analyst it gives deep insight into program behavior. In the hands of the end user it gives a trip-wire warning similar to credit card fraud mechanism.

References

1. Bishop, Christopher, "Embracing Uncertainty: The new machine intelligence" IET Prestige Lecture Series Turing Lecture 2010 `http://scpro.streamuk.com/uk/player/Default.aspx?wid=7739`
2. Bongard, M. "Pattern Recognition", 1970 Spartan Books ISBN 0-87671-118-2
3. Chandy, K. Mani and Misra, Jayadev "Parallel Program Design" Addison-Wesley 1988 ISBN 0-201-05866-9

4. Daly, Timothy and Burns, Luanne "Concurrent Architecture for Automated Malware Classification" Hawaii International Conference on System Sciences 43 Jan. 5-8, 2010
5. 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`
6. 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
7. Duda, Richard and Hart, Peter "Pattern Classification and Scene Analysis" pp10-256 Wiley Interscience 1972 ISBN 0-471-22361-1
8. Eggermont, Jos J. "Learning - The Neocortex" pp246-279 from "The Correlative Brain" Springer-Verlag 1990 ISBN 0-387-52326-X
9. Hawkins, Jeff "How the Cortex Works" pp106-176 from "On Intelligence" 2004 Henry Holt and Company ISBN 0-8050-7853-3
10. "Hierarchical Clustering SAS/STAT (R) 9.2 Users Guide" `http://www.sas.com`
11. Hoare, C.A.R, Brookes, S.D., Roscoe, A.W. "A Theory of Communicating Sequential Processes" 1984 JACM pp560-599 ISSN:0004-5411
12. 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
13. 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
14. Marsland, Stephen "Machine Learning. An Algorithmic Persepctive" Chapman and Hall/CRC 2009 ISBN 978-1-4200-6718-7
15. Ng, Andrew. "Machine Learning (CS229)" `http://www.youtube.com`
16. Perrott, R.H. "Parallel Programming" Addison-Wesley Publishing, 1987 ISBN 0-201-14231-7
17. `http://code.google.com/p/protobuf`
18. 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
19. 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
20. Sheldon, Frederick T., Kavi, Krishna M., Kamangar, Farhad A. "Reliability Analysis of CSP Specifications: A New Method Using Petri Nets" Proceedings of the AIAA Computing in Aerospace 10 Conference, pp317-326, March 28-30, 1995
21. Tou, Julius and Gonzalez, Rafael "Pattern Recognition Principles" Addison-Wesley Publishing 1974 ISBN 0-201-07586-5
22. White, Tom "Hadoop: The Definitive Guide" O'Reilly 2009 ISBN 978-0-596-52197-4
23. Zhao, Jerry and Pjesivac-Grbovic, Jelena "MapReduce - The Programming Model and Practice" `http://research.google.com/archive/papers/mapreduce-sigmetrics09-tutorial.pdf`