

# AWE: Improving Software Analysis through Modular Integration of Static and Dynamic Analyses\*

Ruben E. Brown

Roger Khazan

Michael Zhivich

MIT Lincoln Laboratory  
244 Wood Street, Lexington, MA USA  
{rebrown,rkh,mzhivich}@ll.mit.edu

## ABSTRACT

AWE is a prototype system for performing analysis of x86 executables in the absence of source code or debugging information. It provides a modular infrastructure for integrating static and dynamic analyses into a single workflow. One of the major challenges with performing analysis of modern software is the amount of data that must be analyzed by a human to determine software behavior. This challenge is further compounded by the number of different tools and extensive expertise required to perform such analyses. The AWE system addresses this challenge in two ways: first by focusing analyst's attention on a prioritized subset of software features of importance, and second by simplifying analysis through an integrated static and dynamic analysis workflow.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification – Formal methods, Validation.

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering

## General Terms

Algorithms, Security, Verification

## Keywords

Malicious software analysis, static analysis, dynamic analysis

## 1. Introduction

The AWE system is an assistive tool that facilitates both initial understanding and complete understanding of malicious software when used by a human analyst. AWE improves accuracy and speed of malicious sample analysis by automating repetitive and error prone tasks through integrated static and dynamic analyses. In

addition, AWE directs the analyst to points and regions of interest within the software sample. This directed analysis provides a substantial reduction in the amount of data that must be examined to determine the behavior of the software sample.

## 1.1 State of Software Analysis

Currently, software analysis is a largely manual process requiring extensive expertise and use of a large number of specialized tools. The general process followed is detailed in Figure 1. This process requires the analyst to approach the analysis with a number of non-integrated tools, each requiring a separate set of skills. The final result of each piecemeal analysis is then hand-gathered into a single report. The lack of tool integration results in an analysis following a single pass of dynamic and static analyses, often limited to an examination of the strings produced by the disassembler due to time constraints.

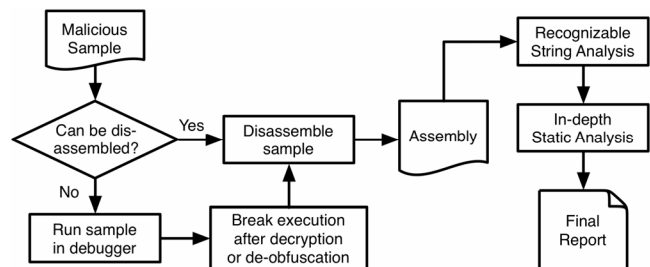


Figure 1. User-driven manual analysis of malicious code.

A more sophisticated piece of malicious software can drive the analyst to cycle back and forth between static analysis and dynamic decryption. However, the more the analyst cycles, the more onerous the lack of integration between the static and dynamic analysis tools becomes, as the analyst spends an ever increasing amount of time performing manual data transport from tool to tool rather than examining disassembly of interest.

## 2. Goal

The AWE project seeks to reduce the amount of time necessary to understand malicious software by reducing the amount of time that must be spent on “low-yield” effort that does not directly result in a better understanding of the software behavior. Eliminating “low-yield” tasks such as separating compiler generated pre-ambles from authored code and manually transporting data between analysis tools would enable the analyst to focus on “high-yield” efforts such as deciphering worm targeting loops or extracting encryption methods.

\* This work was sponsored by the Department of Defense under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

### 3. AWE Analysis Workflow

AWE is an integrated static and dynamic analysis system for the Analysis of Windows Executables. It combines several of the most important tools into a cohesive semi-automated workflow, resulting in a single Integrated Analysis Environment, analogous to modern Integrated Development Environments.

A Windows executable for the Intel x86 processor is provided to both static and dynamic analyzers. The user drives interactions between static and dynamic analyses through a graphical user interface provided by a popular disassembler IDA Pro. Analysis data is stored in an intermediate program representation that describes control flow and data flow graphs, and contains information about known call sites, unresolved indirections, variable values captured at runtime and dynamically detected control flow traces. This process is detailed in Figure 2.

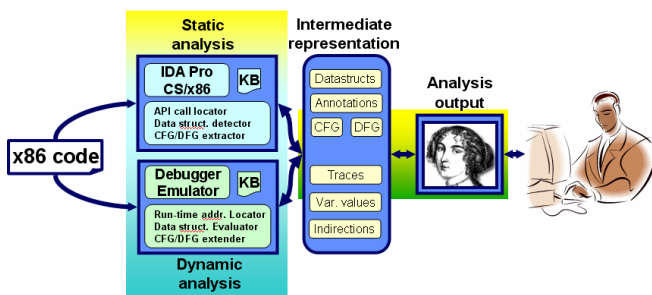


Figure 2. AWE cyclic processing architecture.

Multiple tools are involved in each part of this combined analysis. IDA Pro and CodeSurfer for x86 executables (CSx86) provide the foundation for the AWE static analysis, while OllyDbg acts as the debugger of choice for capturing runtime information [2,3,4]. AWE provides additional analysis on top of the basic functionality of these subcomponents. An example of how AWE fits into the analyst's workflow using CSx86 and OllyDbg is sketched out in Figure 3.

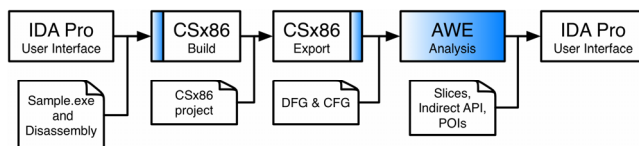


Figure 3(a). AWE static analysis workflow.

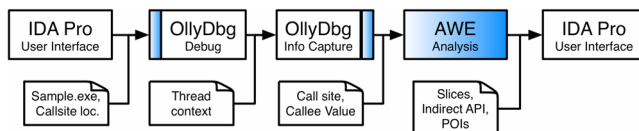


Figure 3(b). AWE dynamic analysis workflow.

AWE provides a subcomponent interface for both CSx86 and OllyDbg (indicated by small colored blocks in the diagram). This interface allows the analyst working in IDA Pro to benefit from other program analysis tools in a semi- to fully-automated fashion. Furthermore, AWE provides additional analysis based on integrating the results of the subcomponent analyses and presents the analyst with additional information in the form of data flow and control flow slices, resolved indirect API calls and points of interest.

### 4. Modular Architecture

As an integrated system AWE provides several layers of communication and data sharing between different analysis tools. In particular, AWE consists of intercommunicating modules inserted into the IDA Pro disassembler, the CodeSurfer static analysis tool, and the OllyDbg debugger. Each of these modules has a different role in the AWE system, and requires different implementation to cope with the constraints imposed by the program in which it is operating. Figure 4 presents the logical organization of these tools in terms of user and program interaction and information flows.

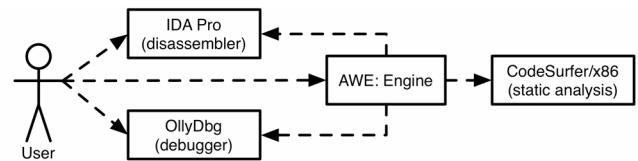


Figure 4(a). AWE subcomponent control interaction.

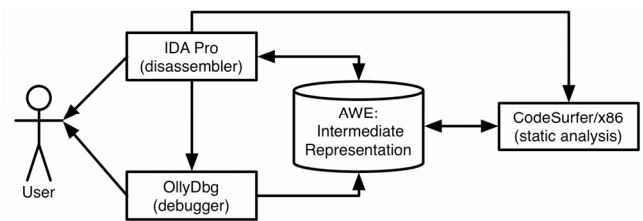


Figure 4(b). Information flow within AWE.

The AWE analysis system operates on information gathered from each module and centralized in the intermediate representation. This intermediate program representation is similar to the internal representation used by a compiler during program compilation. The core of the intermediate representation is a system dependence graph that consists of a conjoined data and control flow graph augmented with additional program information, such as indirect call site targets and variable values.

### 5. The Analyses

This section discusses the details of the AWE's more important methods and procedures. Except where noted, these analysis algorithms operate on the intermediate program representation described in Section 4. These analyses progressively reveal additional information and iteratively augment the control flow, data flow, and other summary data in the intermediate representation.

#### 5.1 Indirect Call Resolution

The AWE system provides two methods for automating indirect call resolution. One approach is a fully automated static analysis pass for resolving indirect call targets that are calculated using *LoadLibrary* and *GetProcAddress* API call pairs. The other is a semi-automated dynamic analysis mechanism for resolving any callee address at run time.

##### 5.1.1 Callee Resolution via Static Analysis

The AWE system can perform indirect call resolution using static analysis based on the system dependence graph. This static analysis performs several operations. First, all call sites whose callees have not been resolved by IDA Pro are collected into a list of unresolved indirect calls. Each unresolved call site is examined by performing a backward data flow search of the system dependence graph to find a call to *GetProcAddress*. The first argument to each *GetProcAddress*

call is a handle to the library module containing the callee function, while the second argument contains a pointer to a string naming the callee function. The library name is then recovered by a second backward data flow search from the push of the first argument to each *GetProcAddress* call to find the associated *LoadLibrary* call or calls. *LoadLibrary*'s only argument is a string representation of the DLL name. This results in a set of possible library name and function name pairs for each unresolved call.

If the set of possible library and function names for an unresolved call contains only a single pair, then the callee is considered resolved, and the system dependence graph is updated with control flow edges to connect caller and callee and data flow edges to connect arguments (inputs) and return value (output). The control flow update creates an edge from the call site to the callee if the callee is already present in the disassembly. The data flow update connects the call site to the argument pushes and return value determined by looking up the callee by library and function name in the AWE library models.

### 5.1.2 Callee Resolution via Dynamic Analysis

The AWE system can perform indirect call resolution using dynamic analysis based on observation of call site targets at breakpoints during execution of the software in the OllyDbg debugger. When a captured callee address corresponds to a known system API call, the library and function name are reported to the AWE IDA plug-in, and system dependence graph is updated as in the case of static call resolution. Callee addresses are translated into library and function name pairs using a method extracted from a pre-existing OllyDbg plug-in named OllyDump. If the captured callee address does not correspond to a known system API call, it is still reported to the AWE IDA plug-in; however, only a control flow update is performed.

## 5.2 Name and Type Propagation

The AWE system performs several types of name propagation to help analysts navigate the disassembly. AWE detects functions which only marshal arguments and call another function, and propagates the wrapped function name up to the call context of the wrapper by renaming the wrapper function. AWE can also summarize the set of call sites contained in a function by pushing documentation for these call sites to the head of the calling function, effectively enabling direct call graph navigation in the IDA Pro disassembly. Finally, AWE applies and propagates the names of arguments to push instructions preceding newly resolved indirect API calls.

Nevertheless, without propagation of associated type information for the variables in a function, propagation of argument names is of limited use. In order to assist the analyst with determining the types of registers and addresses, AWE provides an automatic and a semi-automatic method of type propagation. Both methods start with known type information for the arguments to resolved callees. The automatic method for type propagation is triggered when a system API or other modeled library call is resolved. The semi-automatic method allows analysts to experiment with the results of a type widening algorithm which propagates both named and size-based type throughout all the operands of the current function.

## 5.3 Code Block Detection and Extraction

Software is composed of many basic blocks of code, each providing some small part of the program behavior. Logical groups of these

code blocks collectively provide higher-level software functionality. However, detecting and extracting the logical groupings of interest from a piece of software can be quite difficult in practice, especially when the software under analysis repeatedly uses a previously unknown obfuscation or encryption scheme. In this case, it is necessary to extract and analyze code generated at runtime in order to reveal the software's function. In a combined static and dynamic analysis package such as AWE we can go one step further and detect the logic which generates executable code so that any code decrypted by that logic can be analyzed even if it is never executed.

### 5.3.1 Loop Detection

The most easily extracted code group is the function or sub-routine abstraction, and because that abstraction is systematically enforced by most high-level programming languages it is detected by most disassemblers. However, in order to detect logical programming units that perform repeated processing, such runtime code generation logic, we need focus on detecting the programming mechanism for performing repeated actions – the loop. This capability improves the ability of the system to detect higher-level behaviors and assist analysts in understanding and defeating software that employs sophisticated obfuscation techniques, such as self-decrypting code.

The AWE system can detect loops in control flow contained within a single function context. This detection is provided by a method that blends a simple depth-first search (DFS) of the statically detected control flow with a more complex dominator analysis that expands the DFS detected path through the loop into a more complete collection of the nodes contained in the loop. This blended algorithm detects nested loops and properly identifies loops with multiple entry and exit points. The algorithm consists of two stages: first, a single path through a loop is obtained using a depth-first search; then, the set of entry and exit points is determined via dominator and post-dominator analysis.

Loops can be detected fairly simply using a depth-first search on the control flow graph: revisiting a graph node along a path which already contains that node indicates presence of a loop. The AWE loop detector uses this method to extract a single path through each loop. Note that this search must start at function (or program) root and continue until all nodes in the control flow graph have been visited in order to detect and generate paths for loops that are enclosed by or follow the first detected loop. Each of these loop paths can now be considered independently and represents a minimum set of nodes for each detected loop.

Each node in the loop path is now examined to find entry and exit points to the loop. A node is considered a potential entry point if it contains an inbound control flow edge from a node which is not in the set of loop path nodes. Similarly, a node is considered a potential exit point if it contains an outbound control flow edge to a node which is not in the set of nodes along the loop path. Once the potential entry points have been detected, it is necessary to determine, for each potential entry point, whether the non-loop node from which the incoming edge originates is contained by a different valid path through the loop, or if it is actually part of completely separate control flow. In the first case, we must add the newly detected valid loop path to the set of loop nodes and remove the potential entry point. In the second case, the potential entry point is an actual entry point to the loop. A similar examination is required for each of the potential exit points, in order to develop the final set of loop exit points.

Detection of these additional valid loop paths and invalidation of potential entry points can be performed using dominator analysis. Node **A** is said to dominate a node **B** if and only if all paths from the program root reaching node **B** in the control flow pass through node **A**. For any pair of potential entry points **{A, B}** it follows that if **A** dominates **B** then **B** cannot be an entry point to the loop containing **A** and **B**, and that there exists a set of paths from **A** to **B** which is completely contained in the loop. The domination relationship can be validated with a depth-first search backwards along the control flow from **B** to **A**: if the search manages to reach the function root (or more generally the program root) without passing through **A**, then **B** is not dominated by **A**. However, if every path explored reaches **A**, then **A** dominates **B**, and all of those paths are valid loop paths which can be added to the set of loop nodes.

Similarly, if we consider a post-dominator property, where one node **A** is said to post-dominate a node **B** if and only if all paths from node **B** to the program exit point pass through node **A**, we can perform an analogous analysis to validate potential exit points and add additional valid loop paths arising from this validation.

Following reduction of potential entry and exit node sets and the expansion of the loop node set to include all the dominated paths, one further consideration is necessary to produce a complete description for the loop. We need to extract the set of terminal nodes which if reached indicate that the loop has been exited. These nodes are simply the set of non-loop nodes targeted by each validated exit point. To track loop exit AWE allows the analyst to set breakpoints automatically at the addresses corresponding to these control flow graph nodes.

### 5.3.2 Extracting, Typing, and Analyzing Code Blocks

To extract code blocks the analyst selects a single suspect loop or requests that AWE monitor all intra-function loops using the loop detection capability described above. Once a loop has been identified, the program is executed in the debugger, and an AWE plug-in to OllyDbg captures all the data produced by the loop. As each piece of data is produced, it is added to a data group using one of the following heuristics:

- Data is produced by a single instruction
- Data is produced by a single loop
- Data is produced by many instructions in adjacent memory locations
- Data is produced by many instructions and placed in a location targeted by an existing control flow edge

Each group is associated with a meta-type, such as “data” or “executable code”. This meta-type starts as “data” and can be changed either by the analyst or by an automatic trigger, such as assigning the meta-type “executable code” to a data group when execution under the debugger reaches an address in that group.

The AWE system takes additional post-processing actions when the meta-type change occurs. For example, designating the group of all data produced by a specific write to memory in a specific loop as “executable code” prompts disassembly and static analysis of all data in the group. The data group is transferred from the debugger to the disassembler, new data segments are created, and if the type change was prompted by the debugger entering the code block, a link is added between the *call/jmp* site and the generated code at the targeted address causing IDA Pro to disassemble and analyze the newly detected code block. AWE then applies a basic static analysis

to the code block and passes it to CSx86 for additional static analysis, the result of which is imported back into the AWE intermediate representation and is immediately available to the analyst.

### 5.3.3 Validating Code Block Extraction

We have validated this method of performing automated extraction and analysis of executable code on a simple encrypted program. This program consisted of several XOR “encrypted” functions that were “decrypted” by a single piece of logic, applied to each function immediately prior to execution. AWE detects functions as they are decoded, and they immediately appear in the disassembler complete with static analysis of paths through the function that were not executed during the run that decrypted the function.

The fundamental strengths of this approach are that it is independent of the encryption method and not only analyzes code rooted at an entry point to the decrypted code block, but also recognizes that other data produced by the same logic is likely executable code and analyzes that as well. Furthermore, because this approach only relies on finding the entry point of the currently executing decrypted block rather than finding a global original entry point, it can defeat even decryption which repeatedly uses the same decryption loop to unpack code in a piecemeal fashion rather than in a single pass.

## 6. Case Study with NIMDA

In order to validate the benefits of the AWE system we conducted an in-depth study of the Nimda worm. Nimda is a complex multi-vector worm which provides a number of challenges to the analyst. Its sheer volume makes finding important information about its behavior a difficult task: the IDA Pro disassembly of the worm contains 15,373 instructions, which constitute 350 pages if printed out single-spaced at 44 lines per page. Thus, in order to exhaustively analyze the worm, the analyst must in theory undertake reading a full novel’s worth of disassembled binary code! To give a real world example – it took Ryan Russell at Security Focus forty hours to complete his initial analysis of Nimda.

```

361731C3 loc_361731C3:
361731C3 lea    eax, [ebp+var_4]
361731C6 mov    [ebp+var_4], ebx
361731C9 push  eax
361731CA push  8004667Eh
361731CF push  edi
361731D0 call  dword_3617AC18
361731D6 test  eax, eax
361731D8 jnz   loc_361732E7
361731DE lea  eax, [ebp+var_1C]
361731E1 push  10h
361731E3 push  eax
361731E4 push  edi
361731E5 call  dword_3617AC58

```

Figure 5. Excerpt from Nimda before AWE analysis.

### 6.1 Analyzing Nimda without AWE

Nimda analysis is a daunting task that is further complicated by a simple obfuscation that the worm practices in the form of using indirect calls to system libraries. These indirect calls are not disassembled into meaningful disassembly with callees, thereby completely obscuring Nimda’s use of network APIs to scan and attack hosts. An excerpt of Nimda’s attack on IIS over port 80 is shown in Figure 5. While this section of the disassembly is of

distinct interest to an analyst, its importance is not at all clear from screenshot of the IDA Pro generated disassembly displayed here.

In this example the API calls that could hint to an analyst that this region is of interest are obscured by indirect calls appearing only as `call dword fixed_address`. The values at these fixed addresses are generated in a non-local and oblique fashion, as a series of optimized interleaved calls through a register to `GetProcAddress` and `LoadLibrary`. The interleaving makes it easy for a human analyst to mistake the result of a previous call to `GetProcAddress` for the value being stored at the fixed address, and thus misinterpret the call routines and behavior of this section of code.

## 6.2 Analyzing Nimda with AWE

Figure 6 shows a screenshot of the IDA Pro disassembly after AWE has augmented it with the results of static analysis. This screenshot includes three windows: IDA Pro disassembly window, a window containing a list of points of interest within the sample, and a third window containing a backward data flow slice from the newly revealed `connect` call to its arguments.

This section of code is now of obvious interest, and has been pointed out to the analyst as such by the “Analysis Points of Interest” window. Furthermore, the backward data slice from the `connect` call displayed in the lower right makes it clear that this is an attack on port 80, by connecting the `50h` argument passed to `hostshort` to the `port` field of the `name` argument passed to the `connect` call.

With AWE we see certain definite advantages. The existing disassembly is augmented with function and variable names resulting from resolving the indirect calls automatically, systematically, and correctly. Points of interest are presented to the analyst in a separate window, which the analyst can use to navigate

to the interesting sections of the disassembled executable. AWE also provides properly terminated backward data flow slices from each of the points of interest illustrating how the API call arguments are generated. It is worth noting that all of these windows are tightly integrated into IDA Pro, a tool the majority of analysts are already using. Because of this tight integration these windows work in the same way as built-in IDA Pro windows and provide an intuitive method for navigating the disassembly.

## 6.3 Summary of Nimda Results

AWE resolves the obscured indirect calls and picks out those call sites that are most likely to be of interest, based on an extensible list of library functions designated as points of interest. The default list includes functions for network communication such as `connect`, `send` and `recv`, functions that perform process creation and control, and the set of file and registry editing functions. Furthermore, AWE allows the analyst to rapidly examine the argument creation for each of these points of interest in the form of a backward data flow slice. Even when these slices are capped at a maximum path depth of ten instructions, each slice contains the necessary information to understand the behavior of the worm. Considering just these slices, rather than the complete disassembly, provides an order of magnitude reduction in the amount of data that must be examined by a human.

In the case of Nimda, AWE reduces the original 15,373 instructions in IDA Pro to just 967 instructions of interest that are organized in the form of data slices. AWE also focuses the analyst’s attention from the original 1,146 call instructions to 178 calls of interest. Using its static analysis capabilities, AWE resolves 153 of the 161 indirect calls enabling a much better understanding of Nimda’s functionality.

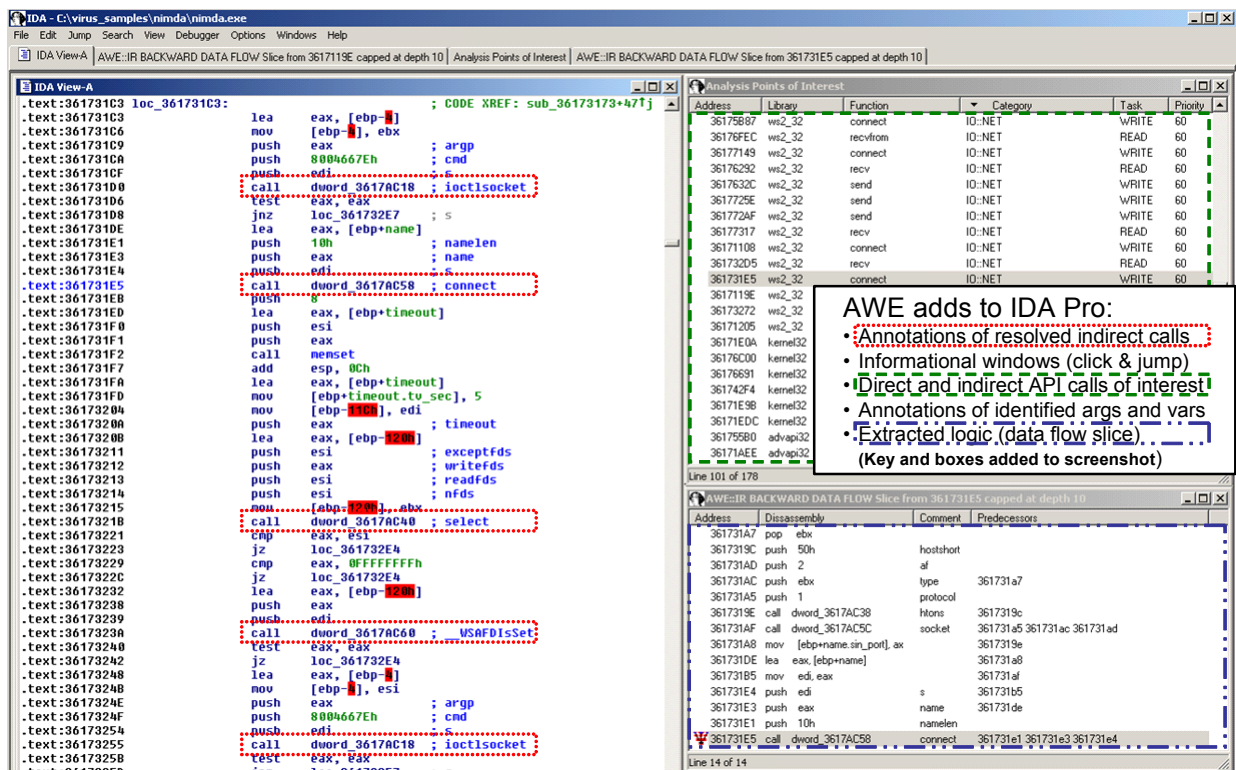


Figure 6. Excerpt from Nimda after AWE analysis.

AWE resolved the obscured indirect calls, picked out and displayed call sites of interest, and provides an order of magnitude reduction in the number of instructions the analyst must examine in only four minutes when run in a VMWare virtual machine on a 3.6 GHz workstation. The speed of this analysis can drop when using more complex sets of build options available in later versions of CodeSurfer.

## 7. Related Work

The development of an integrated static and dynamic analysis tool such as AWE requires a considerable maturity in pre-existing static and dynamic analysis theory and commercial tools.

Prior works, such the development of value set analysis for binary executables by Balakrishnan and Reps [1] at the University of Wisconsin and the extension of this technology into the investigation of obfuscated code by Lakhotia and Kumar at the University of Louisiana [6,7], lay the groundwork for the AWE static analysis.

In addition to static analysis research, research into dynamic monitoring of statically analyzed malicious executables such as that preformed by Rebek and Khazan [8] forms a basis for our dynamic analysis practice. Additional methodologies for the integration of static and dynamic analysis are discussed in position statements such as that of Ernst at MIT [10].

Commercial work in the development of sophisticated reverse engineering tools, such as DataRescue's IDA Pro disassembler and OllyDbg debugger, make the jump to sophisticated integrated static and dynamic analysis possible. The development of the AWE tool was facilitated by the rapid commercialization of Balakrishnan's VSA system by Grammatech. The integration of VSA and IDA Pro based disassembly forms the basis of the CodeSurfer for x86 tool [3] that AWE uses as part of its static analysis process.

## 8. Conclusion

The AWE system demonstrates the potential power of an integrated analysis environment. By blending static and dynamic analyses into a single cohesive workflow AWE produces a more efficient analysis process. Human analysis effort is focused on tasks that more rapidly yield understanding of the interesting software behavior, while lower yield tasks are avoided or automated. This process, when combined with the data reduction and logic extraction capabilities of the system, shows great promise for improving speed and accuracy of binary analysis.

### 8.1 Lessons Learned

The integration of static and dynamic analysis is particularly effective in decreasing human effort when we were able to chain together multiple steps of dynamic and static analysis into a single automated process. For example, de-obfuscation and capturing encrypted code was greatly simplified by using dynamic events,

such as entry into a generated code block, to kick off an analysis chain that included automated code capture, disassembly, and static analysis of the decrypted code. This automation saves an analyst from having to execute each necessary step manually. Existing and future tools could benefit from expanding their autonomous understanding of such triggering events and their association with higher-level analysis behaviors to continue on the path towards automated integrated analysis.

## 9. Acknowledgements

We would like to thank Tom Reps and Gogul Balakrishnan at the University of Wisconsin and the entire staff of GrammaTech Inc, particularly Tim Teitelbaum and Radu Gruian, both for granting us access to the pre-release versions of the CodeSurfer for x86 and for supporting our work. We would also like to thank Tim Leek for his input and insight throughout this project and Rob Cunningham for his oversight and guidance.

## 10. References

- [1] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 -- A platform for analyzing x86 executables. In *Proc. Int. Conf. on Compiler Construction*, April 2005.
- [2] DataRescue. The IDA Pro disassembler and debugger. <http://www.datarescue.com>
- [3] GrammaTech. CodeSurfer. <http://www.grammatech.com>
- [4] O. Yuschuk. OllyDbg. <http://www.ollydbg.de>
- [5] E. Skoudis and L. Zeltser. *Malware fighting malicious code*. Pearson Education, Inc, Upper Saddle River NJ, 2004.
- [6] A. Lakhotia and E. U. Kumar. Abstract stack graph to detect obfuscated calls in binaries. In *Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'04)*, Chicago, IL, 2004.
- [7] M. Venable, M. R. Chouchane, M. E. Karim and A. Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In *Lecture Notes in Computer Science*, volume 3548, pp 1—18, 2005.
- [8] J. Rabek, R. Khazan, S. Lewandowski and R. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, Washington, D.C., Oct. 2003.
- [9] J. Bergeron, M. Debbabi, M. M. Erhioui and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *8th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1999.
- [10] M. Ernst. Static and dynamic analysis: synergy and duality. In *Workshop on Dynamic Analysis (WODA)*, 2003.