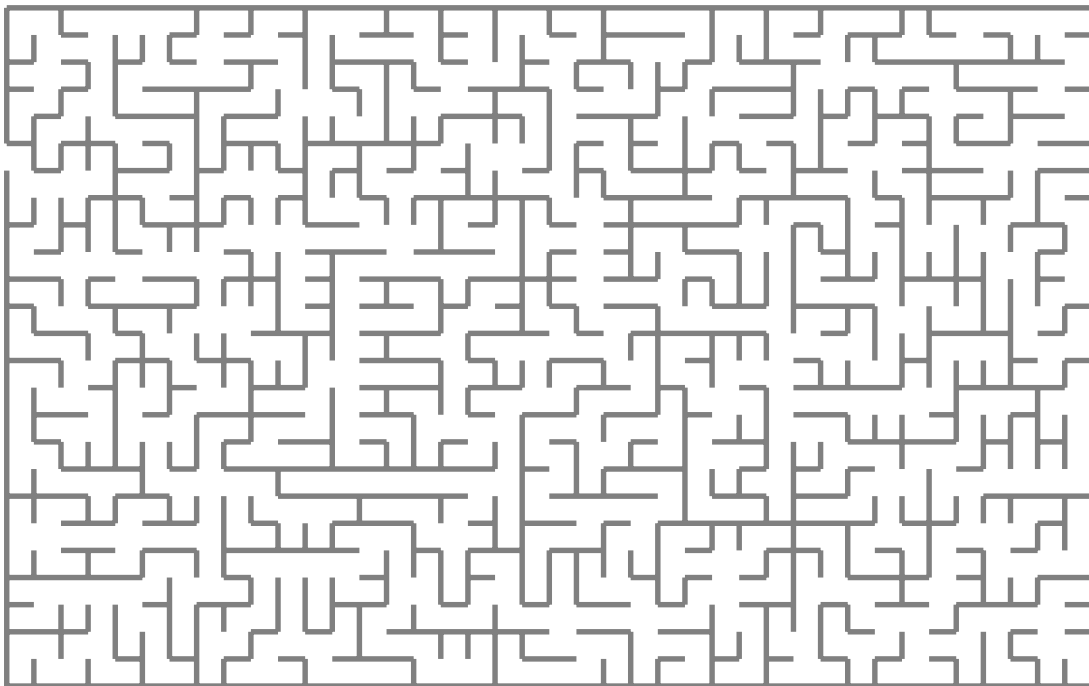


Proceedings of the First International Workshop on
Code Based Software Security Assessments
— CoBaSSA 2005 —

November 7th 2005, Pittsburgh, PA



Organized by Leon Moonen and Spiros Mancoridis

Co-located with the 12th IEEE Working Conference on Reverse Engineering (WCRE)

Contents

Code Based Software Security Assessments	1
Schedule	3
Pattern Matching Security Properties of Code using Dependence Graphs <i>by John Wilander & Pia Fåk</i>	5
Hardware-based Control Flow Monitoring to Prevent Malicious Control Flow Redirection <i>by Nidhi Shah & Linda M. Wills</i>	9
Towards Disk-Level Malware Detection <i>by Nathanael Paul, Sudhanva Gurumurthi & David Evans</i>	13
Adversarial Software Analysis: Challenges and Research <i>by Andrew Walenstein & Arun Lakhotia</i>	17
CoBaSSA Working Session	21

Workshop on Code Based Software Security Assessments

Background

Our technological society has become more and more dependent on software that is used to automate everyday processes. This dependence increasingly exposes us to the security threats that originate from malicious software (malware) such as computer viruses and worms and software vulnerability exploits such as remote execution of code or denial of service attacks. Moreover, this exposure is not limited to computer systems but is spreading to common appliances such as mobile phones, PDAs and consumer electronics such as media centers, personal video recorders, etc. since a growing number of these products are made extensible and adaptable by means of embedded software.

The proliferation of malware and exploits requires that action is taken to tackle these issues and evaluate software security to prevent the damage and costs (e.g., data loss, productivity loss, recovery time) that result from security incidents. This calls for measures to assure that a software system has the desired security properties, i.e. that it is free of malware and vulnerabilities.

Objectives

The purpose of this workshop is to bring together practitioners, researchers, academics, and students to discuss the state-of-the-art of software security assessments based on reverse engineering of source or binary code (as opposed to software security assessments that look at the software process that was applied). This includes research on topics like source & binary code analysis techniques for the detection of software vulnerabilities (e.g. detect if code has potential buffer overflow problems) or analysis for the detection of malicious behaviour (e.g. detect if code contains exploit/virus/worm).

Topics of interest

CoBaSSA topics of interest include, but are not limited to:

- Mitigating stack- or heap-based buffer overflow attacks
- Re-modularizing legacy code for privilege separation
- Code tamper-proofing and obfuscation
- Detecting vulnerabilities in trust management and authentication
- Best practices practices for secure coding
- Analysis of computer virus and worm code
- Case studies analyzing system software vulnerability (e.g., CORBA, EJB, DCOM, Windows, Linux)
- Binary code disassembly and anti-debugging
- Race condition detection
- Copy protection schemes
- Analysis of the implementation of cryptographic algorithms

Dissemination of results

All accepted position papers are available in digital proceedings from the workshop's web-page:

<http://swer1.tudelft.nl/leon/cobassa2005/>

The workshop's results will be summarized in a workshop report that will also be available from this website.

Workshop Organizers

CoBaSSA 2005 is organized by:

- Leon Moonen (Delft University of Technology & CWI, The Netherlands)
(Leon.Moonen@computer.org)
- Spiros Mancoridis (Drexel University, USA)
(spiros@drexel.edu)

Schedule

During the workshop, we will have five 30 min presentations, each followed by 10 minutes for questions and discussion. In the afternoon, we will have a longer working/discussion session to establish a list of *open issues in software security assesment research and practice*. Participants are asked to prepare for the latter by thinking up their own top 3 of open issues before the workshop.

8:30	Opening & participants introduce themselves
9:00	Presentation: <i>Pattern Matching Security Properties of Code using Dependence Graphs</i> by John Wilander & Pia Fåk.
9:30	Questions & Discussion
9:40	Presentation: <i>Hardware-based Control Flow Monitoring to Prevent Malicious Control Flow Redirection</i> by Nidhi Shah & Linda M. Wills
10:10	Questions & Discussion
10:20	Coffee break
10:40	Presentation: <i>Towards Disk-Level Malware Detection</i> by Nathanael Paul, Sudhanva Gurumurthi & David Evans
11:10	Questions & Discussion
11:20	Presentation: <i>Adversarial Software Analysis: Challenges and Research</i> by Andrew Walenstein & Arun Lakhotia
11:50	Questions & Discussion
12:00	Lunch
13:00	Invited Presentation: <i>Best practices for secure coding</i> by Robert C. Seacord.
13:30	Questions & Discussion
13:40	Working session "Open issues in software security assesments"
15:10	Coffee break
15:30	Wrap up & conclusions: lessons learned, next steps.
16:00	(workshop ends)

Pattern Matching Security Properties of Code using Dependence Graphs

John Wilander and Pia Fåk, {johwi, x05piafa}@ida.liu.se
Dept. of Computer and Information Science, Linköpings universitet

Abstract

In recent years researchers have presented several tools for statically checking security properties of C code. But they all (currently) focus on one or two categories of security properties each. We have proposed dependence graphs decorated with type-cast and range information as a more generic formalism allowing both for visual communication with the programmer and static analysis checking several security properties at once. Our prototype tool GraphMatch currently checks code for input validation flaws. But several research questions are still open. Most importantly we need to address the complexity of our algorithm for pattern matching graphs, the accuracy of our security models, and the generality of our formalism. Other questions regard the impact of security property visualization and heuristics for ranking of potential flaws found.

Keywords: security properties; dependence graphs; static analysis

1 Introduction

In November 2002 we published a comparative study of five static analysis tools checking C code for buffer overflows and format string vulnerabilities [19]. We used micro benchmarks and our study showed that tools performing lexical analysis produced a lot of false positives (52% to 71%), while syntactical and semantical analysis had problems with too many false negatives (70% to 73%). The latter mainly due to poor vulnerability databases, not the underlying techniques.

Since then many more tools have been developed [1, 3, 5, 12, 6, 15]. The research behind these tools and prototypes is excellent and the empirical results are promising, but it is not evident if and how the techniques can be combined to solve several security problems at once. They all (currently) focus on one or two categories of security properties each and make use of quite different system models, methods of analysis, and also require different amounts of user involvement. In our studies of the modeling formalisms used in the tools we identified a specific problem in modeling security

properties of code—the *dual modeling problem*.

Some security problems are typically described as “If you do A you must do B” (e.g. *input validation*). Such properties are best modeled as good programming practice—“do like this”. Other security problems are described as “If you do A then you must not do B” (e.g. *double free*). Such properties are best modeled as bad programming practice—“do not do like this”. For a formalism to be able to cover the great variety of security properties it needs to be able to model both good and bad programming practice. The dual modeling problem is closely related to *safety* and *liveness* properties of code [11].

A drawback of static analysis tools in general is that they only *detect* vulnerabilities and therefore the user has to know how to patch the code. The aforementioned tools only offer textual information about analysis results. We believe visual information can be helpful for programmers.

Engler and Musuvathi have pointed out the problem of reporting huge amounts of potential bugs as the result of static analysis and model checking—“It’s not enough to find a lot of bugs. (...) What users really want is to find the 5-10 bugs that really matter ...” [13]. Therefore we believe it is necessary to automatically *rank* the bugs reported from a security analysis tool.

Our research goal is to implement a tool that can:

- check several types of security properties;
- visually communicate with programmers; and
- rank the severity of potential flaws.

1.1 Paper Overview

In Section 2 we present decorated *dependence graphs* as a generic modeling formalism for code security properties covering control-flow, data-flow, type and range information. Models of two security vulnerability types—integer flaws and double `free()` are shown in Section 3 and 4. Section 5 and 6 briefly explain the implementation of our prototype tool and present our initial results. Finally, Section 7 covers future work and open research questions.

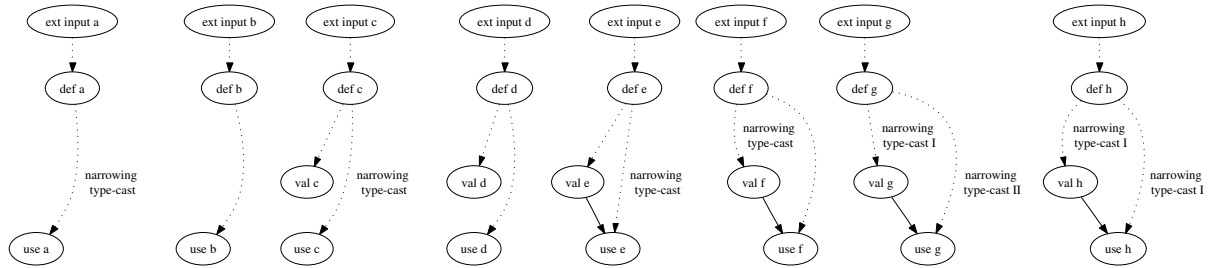


Figure 1. Eight incorrect graph patterns for integer validation. Solid edges represent control dependence and dotted edges represent data dependence. The vertices are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and sensitive use of the variable (use). Roman figures tell if type-casts are different. Severity ranking from left to right stretching from validation point absent to validation with implicit narrowing type-casts.

2 Dependence Graphs

We have proposed decorated dependence graphs as a more generic formalism for visualizing security properties, and performing static analysis of C code [18].

Dependence graphs were first presented by Ottenstein and Ottenstein as an intraprocedural intermediate form—the *program dependence graph*, or *PDG* [14]. Vertices represent statements and predicates (program points), and edges represent control- and data-flow *dependence*. A program point *B* is *control dependent* on another program point *A*, if *A* controls whether *B* is executed or not. A program point *B* is *data dependent* on a program point *A* if some variable *x* is defined in *A* and later used in *B* without any new defines in-between. This means that only necessary temporal constraints are encoded in the graph—it does not include a complete control-flow graph. The interprocedural version, called *system dependence graph*, or *SDG*, was presented by Horwitz *et al* [9]. Dependence graphs were designed to allow for deep analysis of code. They are the underlying structure for *program slicing* [16].

Several so called *narrowing integral type-casts* have constituted security vulnerabilities. Chen *et al* have studied this category of security bugs and summarized the insecure conversions [4]. To detect such flaws we decorate the original SDGs be with type information, specifically implicit type conversions. Type conversion information belongs to edges in the SDG since it is the data-flow between two program points that can include such a conversion, and a program point can be data-flow dependent on several others.

Weber *et al* have used SDGs decorated with range constraint information for string buffers to statically detect buffer overflow vulnerabilities [15]. We will use this technique to check both buffer and integer ranges.

3 Integer Flaws

Several security vulnerabilities prove that handling integers is difficult. The problems mostly arise when integers are used as memory offsets, in pointer arithmetic, and/or when the integer representation changes from signed to unsigned or vice versa [1, 2, 10]. For proper input validation in such sensitive cases, two crucial steps need to be taken; (1) validate integral variables so that narrowing type-casts do not lead to unintended behavior, and (2) validate upper and lower bounds of user affected integral variables before they are used in memory references and calculations.

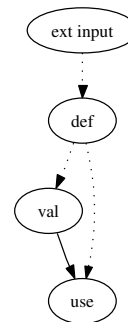


Figure 3. Correct code pattern for integer input validation.

The graph to the left is an example of a model of good programming practice—correct integer input validation. The integer has to be validated before it is used which means that the use point (use) has to be control dependent on the validation point (val), and both use point and validation point have to be data dependent on the input without narrowing type-casts. Modeling of validation points is abstracted away from these models. Using range constraints is a feasible way of doing this [1].

To allow for severity ranking of reported flaws we can encode the dual to the correct code pattern, ending up with a collection of incorrect code patterns, i.e. models of bad programming practice (see Fig. 1).

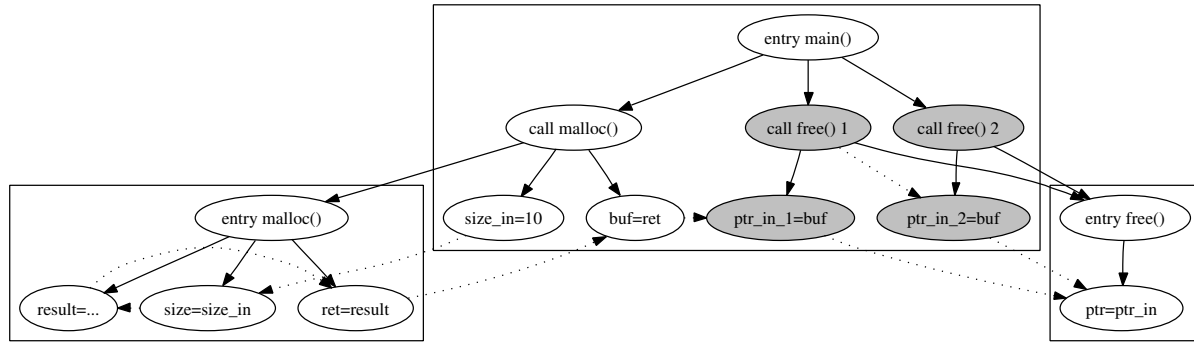


Figure 2. Incorrect graph pattern for malloc and free, where free is called twice. The grey nodes are the bad programming model where two free use the same pointer (shown by a data dependence).

How to model external input is not obvious. Still, many bugs become security vulnerabilities because the user can affect data input. The solution is system and API specific. Apart from file accesses and command line arguments we have followed the pointers by Wheeler who mentions *Environment variables* as untrustworthy sources [17], and Ashcraft and Engler who add another three categories—System calls, routines that copy data from user space, and network data [1].

4 The Double free () Flaw

To allocate heap memory, a C program calls `malloc()` and gets a pointer to the allocated memory as return value. When the program is done using the memory it has to be released, which is done with a call to `free()`. To keep track of which parts of heap memory are allocated and which are free, the operating system has to store information. For scalability reasons this information is stored together with each allocated chunk of memory; it is stored “in-band”. When memory is freed the in-band information is used to relink the memory chunk with the list of free memory.

Normally, attempting to free the same memory twice or more will lead to undefined behavior, often a *segmentation fault*. But if an attacker can change the memory in between two calls to `free()` he or she can inject false in-band information and potentially compromise the process.

This is an example of bad programming practice. We show in Fig. 2 why the double free has to be modeled as a bad security property. The bad model *contains* the good one. If we ignore one of the calls to `free()` we have a match for correct usage of `free()`. Thus we cannot say a piece of code is secure simply because we have pattern matched a good use of `free()`, we also have to look for bad use of `free()`.

5 Tool Implementation

We have implemented a prototype tool called *GraphMatch* that performs pattern matching using dependence graphs. We build graph models of the programs with Gram-matech’s tool *CodeSurfer* [8]. Currently GraphMatch can detect integer input validation flaws by following a straight forward algorithm (compare with vertex labels in Fig. 3):

1. Begin at some external input vertex (ext input)
2. Follow transitive data-flow to match definitions (def)
 - (a) Follow data-flow to all sensitive uses (use)
 - (b) Follow data-flow to all validations (val)
3. Check that all the sensitive uses from 2(a) are control-dependent on some validation in 2(b)

If some part of the program model deviates from the model of correct integer input validation it is reported as a potential flaw. This algorithm has a complexity of $O(E * V^h)$, where E and V are the number of edges and vertices in the program model, and h is the depth of the security property model.

6 Initial Results

The GraphMatch prototype performs well on our synthesized micro benchmarks whereas real-life applications pose a harder problem. We checked `wu-ftpd 2.6-4` which consists of approx. 20.000 lines of code and produces a dependency graph with approx. 130.000 vertices. An analysis for integer input validation flaws took 15h on a 2.66 GHz Pentium 4. GraphMatch produced three warnings, two false positives and one true positive. The false positives were due to inaccuracy of our “sensitive use” model. The true positive was clearly a missing input validation but didn’t seem exploitable.

7 Future Work

Defining the modeling formalism was the first step toward a single tool able to check for several security properties. Apart from modeling other security properties and checking them with real-life code, we have several open research questions to address:

Complexity. Not too surprisingly, our initial results show that our graph matching has high complexity. It might be that dependence graph matching can be reduced to the *subgraph isomorphism problem* which is shown to be NP-complete [7]. Even so, we will investigate how heuristic trade-offs leading to unsoundness and/or incompleteness can affect practical performance.

Accuracy. How much does the inevitable inaccuracy of the underlying program analysis affect the accuracy of our pattern matching?

Generality. Are dependency graphs suitable for modeling a great variety of security properties of code? Are they suitable for analysis of other languages than procedural ones such as C?

Usability. Can visualization of code properties with dependence graphs help the programmers fix vulnerable code? Can it help in secure programming education?

Heuristic Ranking. Can we find effective heuristics for ranking of potential security bugs found through analysis?

Model Updates. Will our security property database be fairly static or will it need continuous updates with new flavors of the security properties?

References

- [1] K. Ashcraft and D. Engler. Using programmer written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.
- [2] Blexim. Basic integer overflows. Phrack Magazine 60 <http://www.phrack.org/phrack/60/p60-0x0a>, 2002.
- [3] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington DC, USA, 2002.
- [4] W. Chen, B. Rudiak-Gould, and B. Schwartz. Automatic detection of implicit type cast errors in C. Paper in graduate course, <http://www.cs.berkeley.edu/~wychen/papers/261.ps>, 2002.
- [5] B. V. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.
- [6] J. S. Foster, R. Johnson, J. Kodumal, T. Terauchi, U. Shankar, K. Talwar, D. Wagner, A. Aiken, M. Elsmann, and C. Harrelson. Cqual: A tool for adding type qualifiers to C. <http://www.cs.umd.edu/~jfoster/cqual/>, 2003.
- [7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
- [8] Grammatech Inc. Codesurfer. <http://www.grammatech.com/products/codesurfer/>.
- [9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.
- [10] M. Howard. Reviewing code for integer manipulation vulnerabilities. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp>, April 2003.
- [11] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
- [12] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Helsinki, Finland, 2003.
- [13] M. Musuvathi and D. Engler. Some lessons from using static analysis and software model checking for bug finding. In *Proceedings of the Second Workshop on Software Model Checking*, Boulder, Colorado, USA, 2003.
- [14] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, Pittsburgh, Pennsylvania, 1984.
- [15] M. Weber, V. Shah, and C. Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001.
- [16] M. Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, San Diego, California, USA, 1981.
- [17] D. A. Wheeler. Secure programming for Linux and Unix HOWTO v3.010. <http://www.dwheeler.com/secure-programs/>, March 2003.
- [18] J. Wilander. Modeling and visualizing security properties of code using dependence graphs. In *Proceedings of the Fifth Conference on Software Engineering Research and Practice in Sweden (to appear)*, Vasteras, Sweden, <http://www.idt.mdh.se/serps-05/>, October 2005.
- [19] J. Wilander and M. Kamkar. A comparative study of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002.

Hardware-based Control Flow Monitoring to Prevent Malicious Control Flow Redirection

Nidhi Shah & Linda Mary Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology
{nidhi,linda.wills@ece.gatech.edu}

Abstract

Control flow redirection attacks have now advanced from traditional stack smashing to other types such as heap exploits or format string exploits. Many approaches have been proposed to solve the problem, the majority of which require modifying, recompiling, relinking or reloading the source code and are mainly focused on stack smashing. In this paper we propose a hardware-based scheme, which automatically extracts a model of the nominal branching behavior of an application as it runs and detects when an anomaly in this behavior occurs. The scheme is not only able to catch traditional stack smashing but also newly emerging heap and format string exploits. The key advantage of this approach is that it provides protection to existing legacy applications with vulnerabilities that allow malicious control flow redirection without requiring access to the original source code, which may not always be readily available. Also this technique does not require software support (e.g., relinking or operating system plugins) and is transparent to the end user.

1. Introduction

Control flow attacks have become more pervasive in recent years. The goal of these exploits is to maliciously modify control flow to execute injected code or malicious parameters [4]. The common problem of buffer overflows has been addressed in limited ways, but current solutions require the source code, extensive debugging information, or the location of the buffer overflow vulnerability in the code. Most techniques also address only stack smashing forms of control flow redirection. Attacks, such as format string and heap exploits allow an attacker to overwrite precise memory locations with malicious data and thus can easily defeat the common protection mechanism that makes use of a canary value, as done in Stackshield [3] and Stackguard [1]. Current approaches also do not provide protection from pointer subterfuge or arc injection, which attacks any pointer variable that is used for control flow instead of just the return address stored on the stack [4].

In this paper, we propose a dynamic, hardware-based approach that does not require changing the application nor the instruction set architecture (ISA) interface to the hardware. Our approach is

independent of the language in which the application is written and the operating system. This approach can support legacy applications for which only the binary is available and requires no extra support from the application development process.

Other hardware-based protection schemes include information flow tracking proposed by Suh et al. in [6] and Crandall in [5]. The problem with this approach is it requires modification throughout the microarchitecture, including changing the reorder buffer, reservation stations, data and instruction paths, main memory, and cache, to include tags on all operands that specify integrity bits. This approach also requires changes to the operating system to detect all attacks. Extra cycles to propagate integrity bits are required with every instruction, which incurs substantial performance overhead. Careful implementation of tagging of the incoming data and its flow is very important to avoid the self-revocation problem pointed out by J. Crandall in [5], in which all data becomes tagged as untrusted.

We propose a runtime system that monitors a subset of instructions, which implement *indirect branches* to recognize opportunities for control flow redirection. Indirect branches are jumps or conditional tests that compute their target addresses at runtime. As discussed in following sections, this scheme can be implemented with slight modification to the existing hardware and incurs only negligible overhead which is absorbed into the already incurred branch penalty. Control flow attacks do not happen on every cycle and so not every computation cycle should pay a penalty for detection. Redirection can be detected as a specialized case of control flow misprediction which is handled as part of the speculative execution that is supported by most modern microprocessors.

In this paper, we describe the approach in detail after a brief background on the problem. We then present a proof of concept implemented on the Intel IA-32 architecture and discuss its advantages compared with previous approaches. Initial experiments with this approach successfully detected the infamous exploit for the *wu-ftp* format string vulnerability, as well as a custom function pointer attack that tests the ability to detect pointer subterfuge [4]. The custom application implemented an *echoserver* having a buffer overflow that exploited a function pointer in the application. This shows that scheme can protect against new attacks.

2. Background

A control flow attack takes place when an attacker overwrites the target of a branch by exploiting some vulnerability that allows overwriting a memory location anywhere in the system [4]. The attacker can only affect branches that compute their target address at runtime (known as indirect branches) as compared to branches whose target address is computed at compile time and encoded in the binary instruction.

Dynamic linking and dynamic binding are frequently used software techniques to facilitate code reuse and code sharing. It provides the programmer the flexibility to provide multiple options for the execution stream, only one of which is dynamically chosen depending on runtime parameters. Various implementations of these can be seen as:

- Global Offset Table (GOT) entries for dynamic library linking
- DTORS section in C/C++ for storing constructor/destructor address
- Function pointers such as c++ vtables, callbacks for dynamic binding.

Each of these has a specific purpose and a different implementation. For example, dynamic linking may be coded as jump tables, function pointers as program variables and virtual functions using a VPTR function pointer table. However, at the ISA level, all are implemented as *indirect branches*. Subcategories can be *call* indirect, *jmp* indirect, memory indirect or register indirect etc.

For all indirect branches, the target address is computed and modified at runtime. This is in contrast to direct branches, such as if-then-else, which have a definite target, computed statically by the compiler and encoded in the instruction. Direct branches cannot be easily attacked since the instructions are in the code section and are not modified at runtime. Indirect branches, on the other hand, make frequent use of target tables, stored at memory location that can be read and written to allow modification of the target addresses dynamically. This makes indirect branches a prime target for attacks.

3. Hardware-based Control Flow Modeling

We propose a hardware-based control flow monitoring scheme that dynamically keeps track of indirect branch target ranges and detects when control flow is redirected. . Figure 1 shows the conceptual view of our modifications to the microarchitecture. It depicts the instruction execution sequence in typical four stage pipeline architecture

with indirect branch monitoring inserted after execution stage. When an indirect branch is encountered, the processor keeps track of the target of the branch as part of normal speculative execution using dynamic branch prediction hardware. The next time the same branch is encountered, it is expected to be in the predetermined range of the first target . The actual implementation logic will differ to optimize and extend the monitoring logic.)

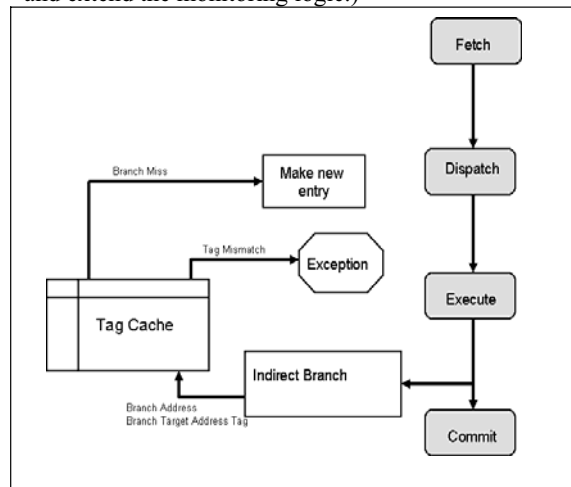


Figure 1 Conceptual Architecture View

The key issues that we are exploring include: 1) the feasibility of characterizing the target range of indirect branches 2) hardware requirements 3) performance optimization.

3.1. Target Range

Results of empirical studies performed on Spec2000 benchmarks with a widely used architectural simulator, simplescalar, show that targets of a given indirect branch stay within the range of a few megabytes of its target when first encountered. Figure 2 (a) shows that the 99th percentile of branch targets fall within a 16MB code range, which can be represented by an 8 bit common tag. In the majority of the cases, an attacker has to inject the code somewhere and modify an indirect branch target address location to store a pointer to the injected malicious code. That injection would be either in a different file or in a section other than the code section, which allows both read and write of data. Normally, the code and data sections are far enough apart to have different virtual address ranges and thus can be identified if the top address bits are compared. To implement this, we introduce a *tag cache*, which will store the tag to record the branch target range history. Most indirect branch targets are predictable. A notable exception is the return address used in procedure calling. Return addresses can have a wide range of branch address as shown in Figure

2(b) since a function can be called from anywhere. Especially in the case of library functions that can be called from a completely new binary. Fortunately for return addresses, hardware-based protection mechanisms already exist [1] that are complementary to our technique and which can be easily combined with it.

3.2. Tag Cache

The tag cache stores the tag representing the target range for the indirect branch. It is indexed by branch address with an index function, which is selected depending on size and configuration of the

branch address should be mapped precisely on the tag cache, which can be done by storing either the tag of branch address, if the index is a function of branch address, or the full branch address if the index function is independent of the branch address.

The tag cache hardware requirement can be significantly reduced if it is implemented as an extension to existing special microarchitectural support for indirect branch prediction, as shown in Figure 3(b). Section 3.2.1 describes this in detail.

3.2.1. Indirect Branch Prediction

Considering the dynamic nature of indirect

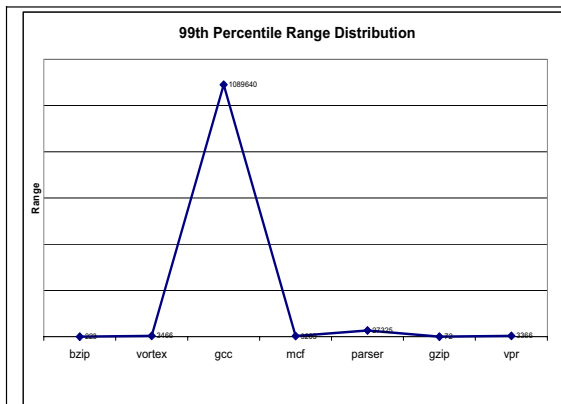


Figure 2(a) 99th Percentile of indirect branch target range is below 16MB

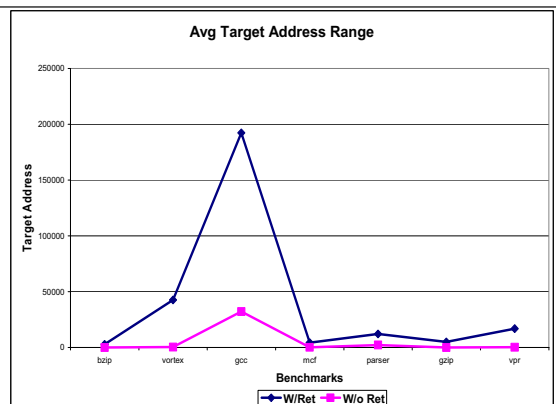


Figure 2(b) Return has wide address range as compared to Other indirect branches

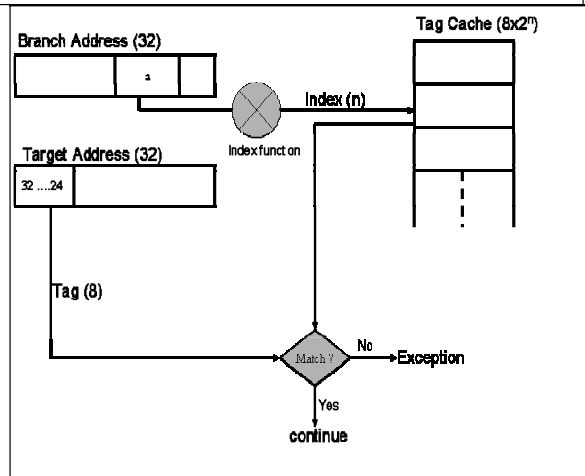


Figure 3 (a) Tag Comparator to Detect Malicious Control Flow ($n = \#$ of indexing bits, 2^n entries)

tag cache. The tag represents the higher bits of target address as shown in Figure 3(a). The empirical studies from Spec benchmarks reveal that the lower 24 bits of the target address vary. The tag cache stores the 8 most significant bits of the target address. Depending on the number of entries, the size of the tag cache can be variable. To accurately predict, the

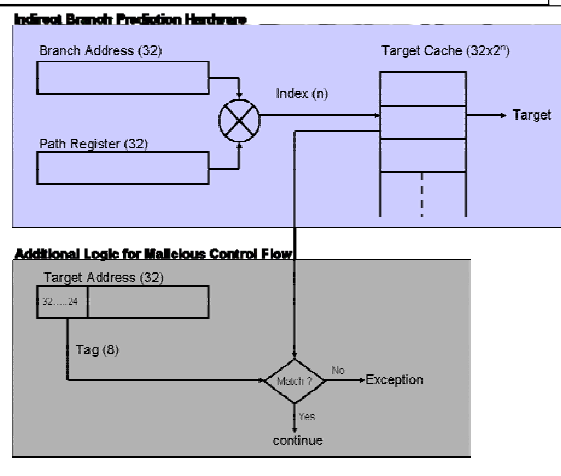


Figure 3 (b) Extension to Indirect Branch Prediction for Malicious Control Flow Detection

branches, Patt et al. proposed path based indirect branch prediction that stores the multiple target-path combination of a given branch in a hardware repository named the *target cache* [7]. Their proposed approach requires expensive hardware resources but in turn substantially increases the accuracy of indirect branch prediction (e.g., by 93.4%

for Spec benchmark *gcc*) and thus received a wide acceptance in processor designs.

We propose extending the target cache which is implemented as part of the indirect branch prediction instead of implementing a separate tag cache. The target cache stores indirect branch target addresses. The range can be determined by retrieving the 8 bit tag from the target address. The cache is updated when the indirect branch is executed and the target address is verified. The same cycle can be used to verify the tag as well as saving processor resources.

4. Experimentation and Results

We are focusing our initial feasibility experiments on Intel's IA-32 architecture due to its pervasiveness. We are using the academic version of Simics simulator from Virtutech to test the proposed approach with real world intricacies. We tested the proposed scheme against various attack categories described by Pincus et.al [13], using vulnerable versions of *WU-FTP* and a test version of *echoserver* with bufferoverflow.

In the case of *wu-ftp*, a dynamically link library call was exploited by overwriting a GOT entry for `printf`. At first, the proposed hardware generated a false positive for this case but further investigation revealed the way dynamic library linking is handled. The dll address is unknown at compile time and is loaded dynamically on the client machine. The compiler creates a static table which is known as the Program Linkage Table (PLT) that stores the routine to populate the GOT at runtime. The first call to the dynamically linked function jumps to the PLT entry which then populates the GOT and subsequent calls are directed to the GOT entry directly, bypassing the PLT. To correctly detect the indirect branch range, hardware has to detect such a pattern. We implemented pattern recognition logic that removed all the false positives generated by this particular construct. We are investigating more such patterns to avoid other types of false positives.

The implementation and performance cost is negligible as this scheme can be implemented with simple extensions to existing indirect branch prediction hardware. The only execution overhead is incurred when there is indirect branch misprediction.

5. Ongoing and Future Work

Currently we are also exploring ways to extend hardware-based recognition over additional architectures. The runtime recognition of various types of indirect branches (*call* or *jmp*, *memory indirect* or *register indirect* etc) is currently relying on properties of CISC (Complex Instruction Set Computer) instructions. The complex, orthogonal addressing modes, strict conventions on the roles of

registers, and multiple micro-operations bundled into a single complex instruction provide a rich context in which to recognize the role of a single instruction. To target RISC (Reduced Instruction Set Computer) instructions, we are exploring recognition of multi-instruction fragments, maintaining contextual information along the way.

Another area of ongoing research is dealing with attacks that are passive in nature. Our approach requires a short learning period to model the nominal control flow behavior. The first time the processor encounters a branch, it assumes the target is a legitimate address and the actual detection logic begins with the next encounter. On systems that are running a server, an attacker may first try to scan the ports to decide whether system is running a vulnerable server. The application gets a few cycles of correct execution before the attack may happen, allowing the learning period to occur. However, when the attacker is posing as a server and looking for vulnerable applications to connect to it, the victim machine may not get that learning period and may run the application the first time itself just to connect to that server. Avoiding this type of attack requires range detection logic to be incorporated into the binary which may be possible as more sophisticated binary reverse engineering tools emerge [8].

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCR-0092552 and CCR-0209179.

References

- [1] C. Cowan, et. al, "FormatGuard: Automatic Protection From printf Format String Vulnerabilities," 10th USENIX Security Symposium, Aug. 2001
- [2] C. Cowan, et al., "Stackguard: Automatic Detection and Prevention of Buffer-Overflow Attacks," Proc. 7th USENIX Security Symp., Jan.1998.
- [3] StackShield. Available online: <http://www.angelfire.com/sk/stackshield>, Aug. 2004.
- [4] J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," *IEEE Security and Privacy*, July 2004.
- [5] J. Crandall and F. Chong, "A Security Assessment of the Minos Architecture," Workshop on Architectural Support for Security and Anti-Virus, Oct. 2004.
- [6] E.Suh, J.W. Lee, D.Zhang, S.Devadas, "Secure program execution via dynamic information flow tracking," ASPLOS, pp. 85 – 96, October 13-16,2004
- [7] P. Chang, E. Hao, Y. N. Patt, "Target Prediction for Indirect Jumps," ISCA, Denver, 1997.
- [8] L. Vinciguerra, L Wills, N.Kejriwal,P. Martino, and R. Vinciguerra, "An Experimentation Framework forEvaluating Disassembly and Decompilation Tools for C++ and Java," WCRE, November 2003.
- [9] D. Ye and D. Kaeli, "A Reliable Return Address Stack: A Microarchitectural Feature to Defeat Stack Smashing," Workshop on Architectural Support for Security and Anti-Virus, Boston, MA, October 2004.

Towards Disk-Level Malware Detection

Nathanael Paul Sudhanva Gurumurthi David Evans
University of Virginia, Department of Computer Science
Charlottesville, VA
{nate, gurumurthi, evans}@cs.virginia.edu

Abstract

Disk drive capabilities and processing power are steadily increasing, and this power gives us the possibility of using disks as data processing devices rather than merely for data transfers. In the area of malicious code (malware) detection, anti-virus (AV) engines are slow and have trouble correctly identifying many types of malware. Our goal is to help make malware detection more reliable and more efficient by using the disk drive's processor. Using the extra processing power available on modern disk drives can provide significant advantages in detecting malware including reducing the traditional AV engine's workload on the host CPU by partitioning the workload between the host AV engine and the disk drive, improving the detection of stealth malware by providing a low-level view of the system, and recognizing virus behavior by observing disk I/O traffic directly. Several research questions must be addressed before these benefits can be realized: how to correctly partition work between the AV engine and the disk drive processor, how to design interfaces between the operating system (OS) or host AV engine and the disk drive that provide satisfactory performance without compromising security, and how to recognize malicious behavior based on the dynamic analysis of low-level data accesses.

Keywords: dynamic analysis, malware detection, virus detection, disk drive processor.

1. Introduction

Malware detectors face three major challenges: they must have false positive rates very close to zero, they must have minimal performance overhead, and they must be able to detect a large number of known viruses and an unlimited number of possible variants. Moving malware detection and response to the disk drive processor offers promising opportunities on all three of these problems since the disk can analyze all I/O traffic with little overhead. Current methods of virus detection can have an overhead potentially as high as 129% [21]. In fact, one company is marketing a hardware coprocessor explicitly for virus detection to lessen AV overhead by using regular expressions along with other techniques [20], and Symantec has a patent on a device (or software) that can be queried to match a regular expression on data and block that data from storage if there is a match [7]. These hardware solutions present a bottleneck, since higher disk activity will require higher loads on a centralized hardware coprocessor. Our goals are to speedup and improve the accuracy of virus detection without requiring specialized hardware.

Recent trends are making storage devices more aware of application needs (application-aware) and making devices actively perform application-specific code on the storage system itself (active storage) [4, 18]. For instance, rather than doing encryption in software, Seagate recently announced a drive that can perform hardware-based

encryption [16]. A drive's processor is about three generations behind a high-end modern CPU [1], and it is currently underutilized. Sivathanu et al. model a semantically-smart disk using a Pentium III 550 Mhz processor, and this trend of increasing processing power should continue as Application-Specific Integrated Circuit (ASIC) technology continues to provide higher speeds within disk power constraints [18]. We have the opportunity to have work done in the disk processor nearly for free. A moderate amount of extra computation on the hard drive CPU scales better than a larger I/O load, since we can hide the processing with the physical data transfers that will incur longer mechanical delays. In using the disk processor to aid malware detection, the I/O is already transferred by the disk, so the only cost is what we incur by also examining the request and data.

The next three sections explore areas in malware detection that significantly benefit from additional processing power. Section 2 deals with the issues in partitioning the AV engine workload between the primary CPU and the disk drive. If the AV engine can let an active disk perform work on its behalf while freeing up the host CPU to service application tasks, then this will increase overall system throughput. Section 3 discusses augmenting low-level protection of machines from rootkits. Many rootkit detection tools depend on a true low-level view of the system, and the closest component of the non-volatile state of the system is the disk drive. Our last example discusses the dynamic analysis of disk I/O.

2. Partitioning Workload

Some of the main actions that an AV engine performs are heuristic scanning, signature matching, and emulation [19]. Heuristic scanning is used to quickly identify traits of a program to act as a filter to more expensive scanning techniques (like emulation), but the most dominant action is simple string matching. Emulation is also an expensive action, and the AV software must be selective on the programs it chooses to emulate and how long it performs emulation. On a single processor system, the AV engine overhead can cost up to 129% [21]. Although the disk drive processor is a suitable place to offload some of the burden of an AV engine, we must first address several issues including: partitioning the workload between the AV engine and the disk processor, minimizing I/O delay, and the interaction between the AV and the disk.

Partitioning malware scanning is especially important if we can exploit parallelism for the scanning process. For this design, we envision an active and application-aware disk where the disk acts independently of the host AV engine.

Since we are using a disk processor, we are limited to a small cache, typically 8–16MB. This limits the disk scanning to use a smaller signature database. The size of the Symantec AV update executable for October 8, 2005 is 8.86 MB [12], but our signatures will be at the lower level of disk blocks. We face the challenge of creating small signatures with low false positive rates. While using the disk cache for signatures, we will also need to study the trade-off of the OS using the disk block cache.

Some polymorphic malware may be difficult to identify, since encryption can be used before writing to the disk. Although knowing the content of I/O traffic can be more useful, the disk may still identify malware according to the location of a written block even if the block is encrypted. Some malware can be recognized through simple string matching, enabling the disk to recognize the malware whenever a known string is written to the disk. Of course this would not work for encrypted I/O, so a signature for encrypted I/O could capture the ways in which the PE file was being manipulated. For instance, one questionable PE file modification is overwriting the entry point of execution. We must be careful to keep the false positive rate low if we use these types of heuristic defenses.

These designs warrant further study into how to minimize I/O delay from regular disk requests and how to set up the communication channel between the disk and the OS or AV engine. We cannot overload the disk processor to the point where regular I/O suffers, but we will harness the idle processing power currently available. The second problem of establishing a secure line of communication is not possible with current PC design, but there are some things we can do to raise the bar for a successful system compromise discussed in the following section.

3. Dynamic Analysis of I/O Requests

In this application area, we use the drive in a more active capacity to scan the disk traffic for malicious activity in

conjunction with the host AV engine. Higher-level detection code may not easily detect the malicious behavior, but the disk may see malicious I/O and stop it before anything bad happens.

For example, the W32/Funlove virus infects local and networked drives by adding a small amount of data (“Fun Loving Criminal”) to the end of each PE file [6]. As the virus enumerates the network shares, it writes to remote PE files through memory mapped file I/O. On-access virus scanners work by catching certain file events like open, close, and create, and they could not detect the virus without more information. Funlove used memory-mapped I/O to compromise remote machines, so traditional scanners could not recognize the virus until it had already infected the disk [19]. Szor goes on to recommend other defense mechanisms including behavior blocking and a network IDS, but an active disk scanning for anomalous traffic (e.g., malicious traffic that installs itself as a Windows service) can prevent intrusions like Funlove with much less cost than an IDS.

One consequence of scanning I/O patterns at the disk-level is that we lose semantic information that we would otherwise have in a typical AV engine. The disk will only receive requests that read or write blocks at a given location, but the disk needs to know what these blocks map to in order to do something useful. One way in which we could bootstrap the disk is by providing a mapping of disk blocks to applications at OS installation. Without talking to the OS, we will need to provide all necessary semantic information about the file system at the OS installation. This comes at a cost – not being able to securely update the disk software after installation. If we need a more extensible design, then we can allow interaction between the AV and disk drive, but we run the risk of having these interfaces compromised.

Some current, and most future PCs, are likely to support a Trusted Platform Module (TPM) that enables a secure bootstrapping process. This has benefits in preventing virus propagation, but as long as users are able to install additional software, TPMs will not stop all viruses. However, we can use a TPM like Intel’s LaGrande for secure communication between the AV and the disk [10]. If the AV engine has a trusted part of code protected by hardware, then the AV engine can attest the calls the disk makes and notify the user to take additional actions if necessary.

4. Detecting Rootkits

Rootkits are a form of malware that are installed by an attacker to keep stealth or secretive access to a machine. This is often accomplished by altering some part of the OS [9]. Rootkit detection tools like Strider Ghostbuster [22], RootkitRevealer [3], and Blacklight [5] perform a high-level scan and a low-level scan of a machine looking for a discrepancy between the scan reports. The high-level scan will use the Windows API or a command like “dir /s /b”, and the low-level scan will read the Master File Table (MFT), raw hive files (Windows registry), and the kernel process list.

The security of these mechanisms relies on the difficulty of implementing code that could intercept these low-level reads and construct false MFT, hive, or process data. While this

may give detection tools the ability to detect most current rootkits, it is only a matter of time until rootkits are developed that can fool the low-level scans. The rootkit detector will call some API to read the low-level data, and this can always be hooked. The detector could implement the functionality of a disk device driver itself to communicate directly to the hardware, but some rootkits are now un hiding files when they detect a rootkit detector performing a file system scan, so the rootkits remain undetected while the unhidden files do not get reported as a difference between the two scans [15]. This battle between rootkits and detection tools will inevitably continue.

To setup the disk to perform a low-level scan, the disk must have more information about the disk blocks. We bootstrap this information at the OS installation, and the disk then has the associations for registry data and the MFT. The kernel process information is kept in memory at run-time, so this information is not accessible from the disk. Since the disk will be performing the file system scan, the scan has the advantage of remaining undetected from any rootkit.

To recover from a low-level malware infection, we can set up the disk to protect certain disk blocks associated with core OS files again specified at OS installation time. Although the OS may have system restore data (e.g., MS Windows), these blocks would not be writable to any code outside of the disk drive. The main problem is performing an update to a protected OS file. Assuming no rootkit has compromised the machine, the user could download an update and override the protection mechanism to apply the update. Of course, this does force the user to trust the OS to not have been compromised. Any time one of the protected blocks is overwritten, the disk can make a backup to some portion of the disk that is not accessible to the OS, in case restoration is needed later. If the disk does indeed find a discrepancy between the high and low level scans, then the latest known clean block can be restored, and we can perform another system scan to make sure all traces of the malware are removed.

The advantages of this approach far outweigh the overhead of storing a few KB/MB on the disk. Capacity is not at a premium, since we currently have consumer disks reaching as high as one-half TB now [8]. The main weaknesses in this approach are updates of registry data and communication between the AV engine and the disk. When the user installs an application it can potentially destroy the integrity of the registry, but the disk may be able to detect a malicious update to the registry in some cases. For the updates that are not deemed malicious, we must either depend on the recovery mechanism in case it is later identified as malicious or allow the disk to receive updates about the protected disk blocks from the OS.

To compare the high-level and low-level scan, the OS will need the low-level scan information, or the disk will need high-level scan information. Because the design requires comparing the results from the OS and disk, the communication link is again a vulnerable target. As suggested in the previous section, we can make use of a TPM. If a TPM is not available, then we have at least raised

the bar of system compromise.

Another example of low-level AV software being circumvented by rootkits is within a filesystem filter driver [9]. To process an I/O request packet (IRP) in Windows, the IRP is passed through a chain of filter drivers before reaching the lowest-level device driver [11]. Many AV engines install their own filter driver in this chain to process incoming IRPs, but even these filters could be circumvented. Using the disk for scanning ensures that malicious traffic can be scanned while the scan itself cannot be circumvented.

5. DADDIO

We are building a new tool to *Dynamically Analyze Disk Drive I/O* (DADDIO) while offloading the CPU workload and aiding in low-level malware detection. DADDIO can provide interfaces to the AV engine to perform string matching and for viewing the low-level filesystem details, and it will analyze disk I/O for malicious activity. If the AV uses software interfaces to DADDIO, then we must use a TPM to use DADDIO securely.

We may be able to leverage DADDIO without a TPM, but we will lose the capability to communicate securely with the host OS. To perform services on behalf of the host AV engine, DADDIO will throttle its own execution workload if the I/O performance suffers. DADDIO's other main action of scanning for malicious disk I/O will be performed during each write to the disk. Reads do not matter if we assume no malicious blocks exist on the disk before DADDIO is activated and DADDIO can prevent malicious writes to the disk.

Recovery from detected malicious I/O traffic can be done without interaction from the AV engine. Without communication with the host OS, we do not risk compromise of the communication channel, but DADDIO has no way of indicating a problem to the user. At the worst, DADDIO can simply suspend all disk I/O. Once users observe the system has frozen from the suspended disk, they will most likely perform a reboot, erasing the malware from the system. Note that this eradicates the malware, since DADDIO prevented it from ever writing to the disk. If the virus activity can be isolated, DADDIO can continue to service regular disk I/O while denying disk access to the malicious process performing I/O. One possibility is to adopt the failure-oblivious computing approach introduced by Rinard et al. [14], and simply write different data on the drive than the malicious code. This approach has proven effective in masking memory errors to keep a faulty server running [14]. Our aim is to allow the OS to safely continue execution without malicious corruption.

6. Related Work

Others have also studied the idea of providing AV services outside of the host machine AV engine. Work by Pennington et al. studied an AV implementation on an NFS server [13]. In hardware, Silberstein [17], Tarari [20], Symantec [7] have proposed ideas to offload AV computation from the main host.

Silberstein observes the signature matching overhead for the open-source AV tool, ClamAV [2], can be as high as 40%. He suggests using a hardware coprocessor to assist in string matching.

Pennington's implementation of an IDS uses an NFS server to support a rule-based IDS system [13]. For their IDS, they require a filesystem separate from the host, so they can guarantee protection for the IDS administration system even if the host has been compromised. However, this protection does not extend to the client, so a compromised client (e.g., trojan) could allow unauthorized access to data on the NFS server [Gobioff99]. Our design protects the local machine by placing an AV engine directly on the disk drive that can partition the workload between the main AV engine and the disk drive AV engine.

The Tarari [20] and Symantec [7] implementations describe designs that can also be used as a hardware or networked offloading engine, respectively. Both designs are for offloading the workload of AV engines, but neither one provides for offloading on the local machine.

One critical difference is that our design can capitalize on the disk already being on the critical path of the data, so we are now analyzing the data rather than just performing data transfers. We avoid the potential bottlenecks of having to pass data through dedicated (and more expensive) hardware solutions.

7. Conclusion

Active and application-aware disks can be useful in malware detection by offloading part of the AV's offload, providing a closer view of the low-level filesystem, and scanning for malicious disk access patterns using low-level disk I/O. We advocate using the disk drive processor for malware detection, and we have presented three motivating examples showing the advantages of using the disk processor for detecting viruses and rootkits. Current virus scanning can be expensive (e.g., emulation and string scanning), and the virus scanner must minimize its own overhead. Lowering the host AV scan-time frees up more processing time for more analysis or for performing other application tasks. If we can capitalize on detailed dynamic analysis of disk I/O, we may be able to avoid more viruses like W32/Funlove. We are currently investigating ways to partition the load between the host CPU and the disk drive CPU and are identifying the best techniques that can benefit from low-level disk block information.

Acknowledgements

The authors thank Peter Szor for helpful discussions regarding this work. This work was supported in part by grants from the National Science Foundation (NSF CAREER CCR-0092945 and NSF ITR EIA-0205327).

References

- [1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks. *Eighth Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.
- [2] Clam AntiVirus. <http://www.clamav.net/>.
- [3] Bryce Cogswell and Mark Russinovich. *SysInternals RootkitRevealer*. <http://www.sysinternals.com/utilities/rootkitrevealer.html>.
- [4] Information Storage Industry Consortium. *Data Storage and Systems (DS2) Roadmap*. January 2005.
- [5] F-Secure. Blacklight. <http://www.f-secure.com/blacklight/>.
- [6] F-Secure Virus Descriptions: FunLove. <http://www.f-secure.com/v-descs/funlove.shtml>.
- [7] John K. Hile, Matthew H. Gray, and Donald L. Wakelin. *In Transit Detection of Computer Virus with Safeguard*. US Patent 5319776.
- [8] Hitachi Deskstar 7K500. <http://www.hitachigst.com/portal/site/en/menutitem.8f07a3c3d3a7a12d92b86b31bac4f0a0/>.
- [9] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.
- [10] Intel Corporation. *Lagrande Technology Architectural Overview*. September 2003. http://www.intel.com/technology/security/downloads/LT_Arch_Overview.pdf.
- [11] Rajeev Nagar. *Windows NT File System Internals*. O'Reilly, September 1997.
- [12] Norton AntiVirus Virus Definitions. October 8, 2005. <http://definitions.symantec.com/defs/20051008-002-i32.exe>.
- [13] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A.N. Soules, Garth R. Goodson, and Gregory R. Granger. Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior. *12th USENIX Security Symposium*. Washington D.C. August 2003.
- [14] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebe, Jr. Enhancing Server Availability and Security Through Failure-Oblivious Computing. *Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.
- [15] Joanna Rutkowska. *Thoughts about Cross-view based Rootkit Detection*. June 2005. http://invisiblethings.org/papers/crossview_detection_thoughts.pdf.
- [16] Seagate Press Release. *Hardware-based Full Disc Encryption Security*. June 8, 2005. <http://www.seagate.com/cda/newsinfo/newsroom/releases/article/0,,2732,00.html>.
- [17] Mark Silberstein. *Designing a CAM-based Coprocessor for Boosting Performance of Antivirus Software*. March 2004. http://www.technion.ac.il/~marks/docs/AntivirusReport_revised_version.pdf.
- [18] Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Second USENIX Conference on File and Storage Technologies (FAST)*, March 2003.
- [19] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.
- [20] Tarari. *Anti-virus Content Processor*. <http://www.tarari.com/antivirus/index.html>.
- [21] Derek Uluski, Micha Moffie, and David Kaeli. Characterizing Antivirus Workload Execution. *Workshop on Architectural Support for Security and Anti-Virus (WASSA)*. October 2004.
- [22] Yi-Min Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski. *Detecting Stealth Software with Strider Ghostbuster*. MSR-TR-2005-25.

Adversarial Software Analysis: Challenges and Research

Andrew Walenstein and Arun Lakhotia

Center for Advanced Computer Studies

University of Louisiana at Lafayette

walenste@ieee.org, arun@louisiana.edu

Abstract—Most work on reverse engineering and program analysis has been performed in the realm of friendly code—code produced without the intent or concerted effort to hinder its analysis. We outline what we see as the main qualitative and technical differences between research on analyzing friendly and adversarial code, provide a survey of research issues for the field, and highlight areas of long term research interest.

I. INTRODUCTION

Reverse engineering is reverse engineering is reverse engineering. Program analysis is program analysis is program analysis. Right?

Perhaps. The argument presented below is that, when dealing with software written by an *adversary*, one enters into a distinctive arena of reverse engineering and program analysis. For our purposes here we say an adversarial relationships between software producers and analysts occur whenever a program producer does not wish the software holder to do something by analyzing the program. Classic adversaries include software demo writers versus crackers who try to circumvent software limitations, and video game vendors versus game modders. And, of course, in the area of trust and security assessment it includes malware authors (“malware” being the now-common generic term for malicious programs such as worms and viruses) versus the various security tools such as anti-virus (AV) scanners.

The task of analyzing adversarial software presents a collection of challenges that appears to set it apart from the problem of reverse engineering and analysis of software written by friendly parties. Solutions that are often satisfactory for analyzing friendly software no longer work well, or even fail utterly. The specter of failure can shift emphasis away from traditional goals such as correctness and efficiency. The focus of research turns to countering the difficulties generated by adversaries. A game is started in which the analyst seeks to *harden* her code analysis infrastructure against *attacks*. We have begun to call this area “ASA”: Adversarial Software Analysis. The goal of ASA is to remedy to headaches created by adversaries.

We would wish to distinguish ASA from “malware analysis” because it makes sense to also include the analysis of benign code—such as commercial video games—which happen not to be friendly toward analysis, and yet are not considered malicious programs *per se*. And ASA is intended to be more encompassing than the term “de-obfuscation”, even if Christodorescu and Jha [1] present a fine argument for viewing

the malware/anti-malware arms race as an obfuscation/de-obfuscation game. Our reasoning is that, although obfuscation is absolutely the key weapon used by adversarial code producers, de-obfuscation is usually only a sub-goal of ASA, which is to defeat what the adversary is guarding against. For instance, we would want to include machine-learning based malware classification techniques (e.g., see Schultz *et. al* [2]) as research explored within the ASA framework even if it is not directly attacking an obfuscation problem.

While perhaps no technical advantage is conferred by introducing a new name to an already confusing vocabulary space, we hope that it can help focus attention on a core set of problems and research associated with adversarial code. The focus on hardening the analysis appears to us to be the defining characteristic of ASA. Research problems in the area would thus be couched in terms such as “vulnerabilities”, “failures” and “reliability”. The search for theoretical apparatus would systematize knowledge along such dimensions. Empirical evaluation would, naturally, focus attention on how well the goal of hardening is being met.

In the remainder we overview major research challenges within ASA and broadly outline possible research avenues for the future. Our experiences in analyzing malware will be used as a prototypical problem to drive the exposition.

II. ASA CHALLENGES

The adversarial relationship that exists between malware authors and defenders—such as AV scanners—is perhaps paradigmatic. AV scanners seek to inspect programs so as to decide if they are trustworthy. Malware authors actively seek to undermine this determination. Often the AV test involves matching program features to various patterns on blacklists or whitelists, such as lists of suspicious behavior patterns. Abstractly speaking, this is a classification exercise, so the game is one of classification/defeating classification.

It is clearly possible to enlist the arsenal of techniques from reverse engineering and program analysis in making such classifications. For instance, to find suspicious program behavior patterns of suspect system calls, a model checker might be used. The model checker would be used to check for control paths that could execute the malicious behavior. The hope is that it could find the intended malice even for executables that have never been seen, or have been obfuscated in various ways, including via junk code insertion.

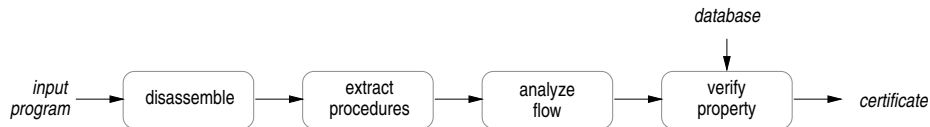


Fig. 1. Classic reverse engineering / program analysis pipeline.

This basic approach has been attempted in our laboratory [3]. An industrial-strength disassembler (IDA Pro¹) was employed to disassemble the machine code and identify procedures. Standard control graph extraction and flow analyses were used. A popular model checker was employed. The overall architecture of the approach matches Figure 1. This minimal pipeline architecture is also commonly found in reverse engineering and program analysis. It is typically staged, requiring complete and accurate output to be passed from one phase to another. In our case we required call and flow information. Several other related approaches to finding behavior patterns via static analysis include the works by Bergeron *et. al* [4], Christodorescu *et. al* [1], and Kruegel *et. al* [5]. They employ similar architectures and maintain similar sets of exacting processing requirements, such as the need for complete or accurate flow graphs.

A wrench is thrown into the works by adversaries attacking the analysis. Our detector was thwarted by one of the first viruses we tried it on. It obfuscated its system calls, making our flow graphs erroneous and rendering the pattern detecting component impotent despite its model checking sophistication. Attacks like these are the nature of an adversarial relationship; responses to such adversarial actions circumscribe the bounds of ASA. The venerable difficulties in reverse engineering and program analysis still plague ASA, of course: the problems of correct decompilation and abstraction; accurate points-to analysis; tool interchange languages, and so on. However it is the wrenches being thrown that become a key concern.

Three concerns appear to move to the fore: (1) recognizing failures and their impacts, (2) understanding vulnerabilities to attack, and (3) hardening against attacks to improve reliability and graceful degradation of functionality. Some initial thoughts on these points are outlined below.

A. Failures

Failures in ASA might be understood by application established failure analysis techniques. Common steps are to inventory failures by mode, effect, and cause. This might be done at component (e.g., disassembler) or system (e.g., whole pipeline) granularities.

1) *Modes*: Apart from program faults (i.e., bugs, incorrect design), the two main failure modes of interest to ASA are: (i) *hard failure*—failing to produce (complete) output, and (ii) *soft failure*—producing erroneous or inaccurate output, particularly if it is without appropriate notification. In our case the control flow extraction failed softly and gave no indication of that.

¹Trademark of Datarescue, see datarescue.com.

2) *Effects*: Tracing effects of failures typically involves tracing causal chains and examining the impacts of each failure mode. Figure 1, might be treated as a “reliability block diagram” [6], and be used to trace effects and estimate reliability. A simple processing chain leads to a brittle system because each stage is a potential single point of failure. In our particular example, all the virus writer needed to do was obfuscate calls to cause the whole chain to fail.

3) *Causes*: The causes we are interested in for ASA are attacks by adversaries. Obfuscating the system calls was the cause in our example.

B. Vulnerabilities

A buffer overflow is said to be a system vulnerability that can be used as a point of entry for a worm. These vulnerabilities are not the sort being presently considered. Rather, we are concerned with vulnerabilities in the analysis software or environment. These afford adversaries ways of defeating the analyses.

We cannot propose a full taxonomy of such vulnerabilities, but we can suggest at least three different categories:

1) *Static analysis vulnerabilities*: These are vulnerabilities present because of limitations in the processing infrastructure or in the basic capabilities of implemented or known algorithms. These include what Linn and Debray [7] call “computationally difficult static analysis problems”, such as constructing accurate points-to information. An adversary could craft her code in such a way that the points-to information is too imprecise. We can say these vulnerabilities cannot in general be avoided: they may be NP hard, map to the halting problem, or simply require more resources than can be made available.

2) *Assumption vulnerabilities*: A typical analysis system makes myriad assumptions about the format of the input. Often these are based on a formal or informal convention. Each such assumption can potentially be a target for attack. For example, *typically* procedure calling conventions for Intel x86 architectures are coded using `call` and `ret` instructions. However this is merely a convention and it is naive to assume that all possible programs will conform to it. Likewise, the default way of padding bytes for non-executing bytes in a code segment is to use instructions that will not foul traditional linear-sweep disassembly. Violating this assumption fouls a linear-sweep disassembly [7].

3) *Infrastructure/environment vulnerabilities*: These are exploitable weaknesses the processing infrastructure. Examples such vulnerabilities include: anti-debugging techniques [8] which make debugging techniques fail; and executable loading

flaws that expose exploits even without intentionally being technically executed [9].

Note that these vulnerabilities are partly dependent on one another. Many of the conventions underlying the assumptions exist because they simplify the problem class, making it easier (or even possible) to perform the analyses correctly. For instance, without conventions for laying out executable code it might well be impossible to guarantee correct disassembly. This point will be returned to later.

Viewing failures as being caused by vulnerabilities provides a different viewpoint as compared to the one offered by the notion of a obfuscation/de-obfuscation game. Christodorescu *et. al* [1] and Collberg *et. al* [10] describe multiple classes of obfuscation attacks. While obfuscations certainly are a type of cause for failure, focusing on the obfuscation may deflect scrutiny about the assumptions being made. For example, from the vulnerabilities point of view, linear-sweep disassembly makes an assumption about the input that allows non-code bytes to force it to erroneously disassemble the code bytes. From this viewpoint the code itself is not being obfuscated at all if only the (let us assume) irrelevant padding bytes are being doctored. Similarly, if an AV scanner is being evaded by systematically renaming register, it may make sense to wonder about the strictness of the matching methods rather than blaming obfuscation (e.g., why should swapping two general-purpose registers make a program less scrutable?).

C. Hardening and reliability concerns

In ASA it can be assumed that adversaries force the analyst into ensuring their analysis infrastructure is hardened, i.e., that the vulnerabilities are minimal in number and the remaining ones are made difficult to attack. This can be expected to, in turn, shift the design and evaluation goals.

1) *Design goals*: Problems that might otherwise have been swept under the rug (e.g., “assume we have an accurate and complete flow graph”) may become a make-or-break concern. Speculative, probabilistic, and imprecise methods may become acceptable. In a compilation environment a program analysis technique is unlikely to be adopted if it occasionally produces incorrect results. An adversary may force the analysis to at least make a best guess.

2) *Evaluation goals*: Since hardening is the goal in ASA, measuring the amount achieved may be expected to be the main evaluation criterion. This may take several forms. One possibility is to measure the increase in accuracy of results in the presence of various adversarial attacks. The experiment by Kruegel *et. al* [11] provides a good example. They measured the difference in the accuracy of their disassembler in the presence of specific obfuscation attacks. In the context of malware detection, a common method for assessing accuracy is the use of classification precision measures such as the ROC measure [2]. A second possible evaluation method could be to measure an increase in the number of conditions that can be handled effectively by some technique. Perhaps an example of this type is the evaluation by Lakhotia *et. al* [12] who measured how well a transformation filter normalized

variations in malware to simplify the matching problem for AV scanners.

III. RESEARCH DIRECTIONS: HARDENING

We have suggested that ASA be considered an area defined characteristically by a concern for hardening analysis against adversarial attack; that the theoretical apparatus of the research area would systematize knowledge about failures, vulnerabilities, and robustness; and that evaluation would evaluate progress towards the goal of hardening the analysis. Here we speculate on future directions that research in ASA may take. To ease exposition we use the differences between Figure 1 and Figure 2 as a visual index to the areas in which future focus may be turned. Differences between the two figures are highlighted by bolding the changes.

A. Feedback and opportunistic processing

In our model checking-based malware detector we had a strictly phased pipeline. Disassembly had to be completed before control flow could be extracted. This is a strictly “bottom up” notion of processing, and is reflected by the unidirectional pipeline architecture of Figure 1. This makes sense if one views the problem of generating a control flow graph as one that first requires determining the control instructions.

Yet as the paper by Kruegel *et. al* [11] implies, one may reasonably view the problem as being circularly defined: one first needs to understand the control flow to know which bytes could or could not be included in a possible control sequence. This is an instance where the removal of an assumption makes a problem harder to solve. The chicken-and-egg circularity is tackled by performing speculative flow extraction followed by a process of re-investigating the disassembly portion and a probabilistic decision procedure to classify a byte as code or data. Instead of a purely bottom-up approach it is a multi-directional approach. This is reflected in the bidirectional arrows in Figure 2.

Generalizing this observation the processing begins to resemble the shift to an opportunistic model of interpretation, such as occurred early in speech understanding [13]. A key ideas in such approaches includes the use of a blackboard or agent architecture which allows processing to be made in the *ad hoc* order in which progress can be made by any component—allowing fluid mixing of top-down and bottom-up processing.

B. History and cohort information management

Our model-checking malware detector employed a blacklist at the final stage; patterns of suspected malicious functionality were encoded as predicates to check. The blacklist was generated by examining other malicious programs to discover common behaviors. As such, the blacklist can be viewed as a knowledge base regarding a class of malicious programs. It was used to aid in the classification process, making it a type of knowledge-based classification. This is shown as a database input in Figure 1.

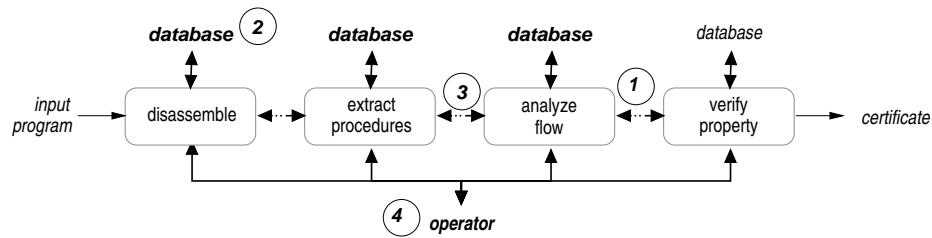


Fig. 2. Visual index for possible research directions.

Knowledge could be helpful in other processing steps also. The disassembler of Kruegel *et. al* [11] can be viewed as an example. In their disassembler, a decision needs to be made at the byte interpretation stage as to which bytes to consider as code or not. They used a probabilistic decision procedure based on what could be called (in Bayesian probability terms) the *prior probabilities* of single and paired bytes. These prior probabilities were measured by counting occurrences in what was presumed to be a representative samples of real programs. Thus a knowledge base or historical information might usefully be added to any of the phases. This is reflected in the bidirectional arrows in Figure 2. In the diagram the links to the database are also bidirectional, to indicate the possibility that the analyzers may learn over time based on the stimuli they receive. In the general case some types of machine learning or case-based reasoning might be employed.

C. Inexact methods and confidence tracking

Our model-checking model detector needed to perform flow analysis so that potential executions of malicious behavior patterns could be found. We used conservative flow analysis, which might entirely miss possible flow edges in the presence of computed branches.

One potential way forward is to add speculative flow edges to the flow graph. This would remove a vulnerability that allows the hiding of malicious code by ensuring all flow edges to it are impossible to determine statically. However since there is more uncertainty as to the true possibility of the speculated flow occurring, it should perhaps be accorded less weight in the later pattern matching phases. Generalizing this suggestion, one future research direction is in exploring ways of representing inexact or fuzzy information, and of tracking confidence to various working representations. This possibility is reflected in the dashed flow lines between the processing components of Figure 2.

D. Human-computer cooperation

A clear avenue for future research involves smoothly integrating human cooperation so that automated analysis limitations are overcome. This is indicated in Figure 2 by the lines indicating interaction with an operator.

IV. CONCLUSION

Current research in analyzing malware appears to fall into an area of reverse engineering and program analysis in which the

hardening of the analysis techniques is the primary concern. If true, then the defining issues, theories, and evaluation criteria will concern vulnerabilities, failures, reliability, and hardening techniques. Our brief review of emerging research appears to match such a viewpoint; hopefully the ASA view can help crystallize a common understanding, and to direct future attention.

REFERENCES

- [1] M. Christodorescu and S. Jha, ‘Static analysis of executables to detect malicious patterns,’ in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 169–186.
- [2] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, ‘Data mining methods for detection of new malicious executables,’ in *IEEE Symposium on Security and Privacy*, 2001, pp. 38–49.
- [3] P. K. Singh and A. Lakhota, ‘Static verification of worm and virus behavior in binary executables using model checking,’ in *Proc. of the 2003 IEEE Workshop on Information Assurance*, 2003, pp. 298–300.
- [4] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, ‘Static detection of malicious code in executable programs,’ in *Proceedings of the International Symposium on Requirements Engineering for Information Security SREIS’01*, 2001, pp. 1–8.
- [5] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, ‘Polymorphic worm detection using structural information of executables,’ in *Proceedings of the 8th Symposium on Recent Advances in Intrusion Detection (RAID’2005)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2005, (to appear).
- [6] A. Abd-Allah, ‘Extending reliability block diagrams to software architectures,’ Department of Computer Science, University of Southern California, Tech. Rep. USC-CSE-97-501, 1997.
- [7] C. Linn and S. Debray, ‘Obfuscation of executable code to improve resistance to static disassembly,’ in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, 2003, pp. 290–299.
- [8] P. Ször, ‘Attacks on Win32,’ in *Proceedings of Virus Bulletin Conference 1998*. Virus Bulletin Ltd., 1998, pp. 57–84.
- [9] B. Moore, ‘Bugger the debugger: Pre interaction debugger code execution,’ Security-Assessment.com, Ltd., Apr. 2005, last accessed June 2005. [Online]. Available: <http://www.security-assessment.com/Whitepapers/PreDebug.pdf>
- [10] C. Collberg, C. Thomborson, and D. Low, ‘A taxonomy of obfuscating transformations,’ Department of Computer Science, University of Auckland, Tech. Rep. TR 148, July 1997.
- [11] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, ‘Static disassembly of obfuscated binaries,’ in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 255–270.
- [12] A. Lakhota and M. Mohammed, ‘Imposing order on program statements to assist anti-virus scanners,’ in *Proceedings of the 11th Working Conference on Reverse Engineering*, 2004, pp. 161–170.
- [13] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy, ‘The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty,’ *ACM Computer Surveys*, vol. 12, no. 2, pp. 213–253, 1980.

CoBaSSA Working Session

The goal of this working session is to come up with some sort of top 10 (or research agenda if you like) of issues that we consider interesting and/or important to look at (both from research and from practice perspective).

To allow for more structured collection and discussion of such a list of issues, we propose to use the following approach which is basically implementing a “distributed ranking & merging process”:

Distributed ranking & merging process

Time needed: approximately 90 minutes in 3 phases of resp 30, 30–40 and 20-30 minutes.

First phase: “distributed ranking”

1. Every participant gets a “ranking sheet” and writes down a personal top 3 of important issues (in the area of code based software security assessments) that should be worked on.
2. Everyone chooses a neighbor.
3. Together they look at both sheets and rank all items in the combined list between 1 and 6, with 6 being most important.
4. if (number_of_iterations < 5)
 - they swap sheets and choose a new neighbor with which they haven't 'interacted' yet and continue with step (3)else
 - everyone has a sheet with 3 issues having 5 numbers behind it, they sum these into one total for each issue.fi

Second phase: "harvesting & merging"

With 3 subjects per sheet and 5 iterations the highest possible (summed) ranking that could be achieved is $(3*2*5=)$ 30. In pseudo code the harvesting & merging goes as follows:

```
rank_of_interest = 30
while ( rank_of_interest > 0 & issues_collected < 10) do
  if ( someone has an issue ranked with rank_of_interest on their sheet)
    write down issue on flipsheet or powerpoint
    if ( anyone has a related/overlapping issue on their sheet)
      discuss with group if indeed related
      if ( so)
        add/merge in description on flipsheet or powerpoint
        and strike through on participant sheets
      fi
    fi
  fi
  rank_of_interest--
od
```

Final phase: "discussion & refinement"

In the final phase, we look at the collected issues and discuss if their descriptions need refinements. In addition, we check if we all agree on this list and investigate if there are still important issues which were somehow missed or underrated, etc.

Ranking Sheet

The issue collection and ranking sheet is included on the next page.

Issue	Rank a	Rank b	Rank c	Rank d	Rank e	Total Rank
A:						
B:						
C:						

Author (for clarification questions):