

APPLICATIONS OF POINTER ALIAS ANALYSES

BY

TASNEEM KAOCHAR

A Thesis Submitted to The Honors College

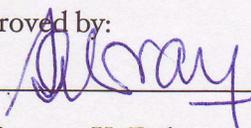
In Partial Fulfillment of the Bachelor's Degree with Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

DECEMBER 2008

Approved by:



Dr. Saumya K. Debray

Department of Computer Science

STATEMENT BY AUTHOR

I hereby grant to the University of Arizona Library the nonexclusive worldwide right to reproduce and distribute my thesis and abstract (herein, the "licensed materials"), in whole or in part, in any and all media of distribution and in any format in existence now or developed in the future. I represent and warrant to the University of Arizona that the licensed materials are my original work, that I am the sole owner of all rights in and to the licensed materials, and that none of the licensed materials infringe or violate the rights of others. I further represent that I have obtained all necessary rights to permit the University of Arizona Library to reproduce and distribute any nonpublic third party software necessary to access, display, run, or print my thesis. I acknowledge that University of Arizona Library may elect not to distribute my thesis in digital format if, in its reasonable judgment, it believes all such rights have not been secured.

Signed: Fayneem Koochar

CONTENTS

1	Introduction	4
2	Background	5
2.1	Pointers and Their Usage in Programming Languages	
2.2	High Level Alias Analysis	
2.3	Low Level Alias Analysis	
3	Overview of Case Studies	9
4	FA Analysis	10
4.1	The Problem	
4.2	Analysis Implementation	
4.3	Analysis Application	
4.4	Experimental Results and Evaluation	
5	Dynamic Assembly Level Analysis	16
5.1	The Problem	
5.2	Analysis Implementation	
5.3	Analysis Application	
5.4	Experimental Results and Evaluation	
6	Value Set Analysis	19
6.1	Analysis Implementation	
6.2	Analysis Application	
6.3	Experimental Results and Evaluation	
7	Conclusion	24
8	Future Work	26
9	Acknowledgments	27
10	References	28

I. INTRODUCTION

Consider for a moment the tremendous growth of electronic devices such as cellphones, MP3 players, digital cameras, DVD players and GPS systems within the last decade. With each passing year, these devices - known as embedded devices because of special-purpose computer system that reside within them - are shrinking in size and weight while performing increasingly advanced functionalities. How are engineers achieving these seemingly contradictory goals?

At the heart of this problem is the need to reduce the size of the special-purpose computer system that reside within each embedded device. One compelling choice is to identify and eliminate any unnecessary code present in the computer system and thereby reduce the memory footprint of the operating system [5, 8]. In order to carry out this process - known as dead code elimination - the use of a pointer analysis is an absolute must.

Pointer alias analyses play a critical role in various areas of computer science research. The objective of an alias analysis is to answer a central question: can a given memory location be accessed in multiple ways? This question arises due to the existence of a *pointer* data type in many programming languages that make it possible for two expressions to refer to the same mutable location in memory. To address this question, an analysis is necessary in order to identify all the potential aliases - multiple references to the same storage location in memory - that may occur when a program is executed [10]. In the example of the compaction of operating systems (OS) within embedded devices described above, the fact that most OS code make use of pointer data types necessitates the use of a pointer alias analysis to analyze the OS and identify any repetitious and/or unnecessary code.

There are numerous classifications of pointer alias analyses in the research field. In this thesis, an alias analysis is broadly defined under two categories: high (source) level alias analysis and low (assembly) level alias analysis. A high level alias analysis attempts to identify potential aliases by recovering information from a high level representation of the source code. Contrastingly, a low level alias analysis seeks to disambiguate memory relationships by analyzing the low level representation of the source code. Most tools that exist today analyze programs written in high level languages rather than deal with low-level assembly because the latter presents many new challenges [11]. Unlike assembly level alias analyses, high level alias analyses that work on intermediate representation built from the source code can take advantage of source level semantic information and thus yield more precise results [20]. In many cases, however, researchers have access solely to the program assembly code and therefore any useful program analysis must be able to handle low-level assembly. The decision to use a high level analysis versus a low level analysis is thus often based on the needs of the client application.

In this thesis the terms *pointer alias analysis*, *pointer analysis*, and *alias analysis* are used interchangeably to refer to a static code analysis that seeks to establish the possible runtime values of each pointer present in the code. The remainder of the thesis is organized as follows. Chapter 2 provides a thorough discussion of pointers and pointer alias analyses, including their necessity for analyzing programs that employ pointers or pointer-like structures. Chapter 3 provides an overview of the three different types of pointer algorithms that I have implemented. Chapters 4 to 6 then go into an in-depth discussion of each of the implementations, providing a detailed look at the challenges that emerged during the process and the specific application of each implemented analysis. Chapter 7 summarizes the conclusions and Chapter 8 describes possible future extensions to each implementation. The thesis concludes with a series of acknowledgments in Chapter 9.

2. BACKGROUND

Although research on pointer alias analyses dates back to the late 1970s, the increasing demand for more accurate and/or cost efficient analysis algorithms makes this topic a focal point in computer science research today. A pointer alias analysis is essential for analyzing programs written in languages that employ pointers and pointer-based data structures, such as C, C++, Java and Objective C. The purpose of a pointer analysis is to determine the storage or memory locations a pointer may point to when a program is executed. All pointer analyses employ a static code analysis technique in which the program being analyzed is never actually executed but rather the analysis gathers information by exploring *all* the possible execution paths that could occur if the program were executed.

2.1 Pointers and Their Usage in Programming Languages

A pointer is a data type that refers to another value stored in memory using its address (see figure 1). One can obtain the value to which a pointer refers by *dereferencing* the pointer. In its most fundamental form, a pointer is a memory address and thus can be treated as such. Pointers are directly supported in languages such as C, C++, Pascal and most assembly languages. Other popular languages, such as Java, make heavy use of pointers in behind-the-scenes implementation of the language and thus shield its users from the challenges that arise through pointer usage. Pointers are primarily used for creating references and are essential for constructing many data structures, including lists, queues and trees, as well as for passing data between different parts of a program.

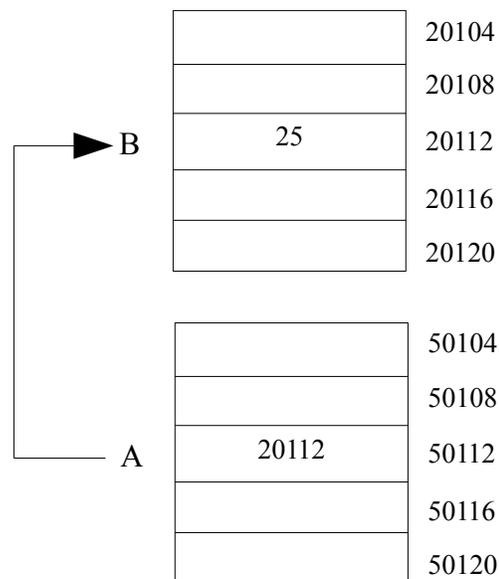


Figure 1: Pointers
Pointer *A* points to memory location *B* which stores integer values.

The diagrams on the following page offer a brief look at some common pointer usages using C code. Figure 2 illustrates simple pointer manipulation in C using a integer pointer. Figure 3 shows an example specific to C and C-like languages in which a function pointer may be used to indirectly call a function.

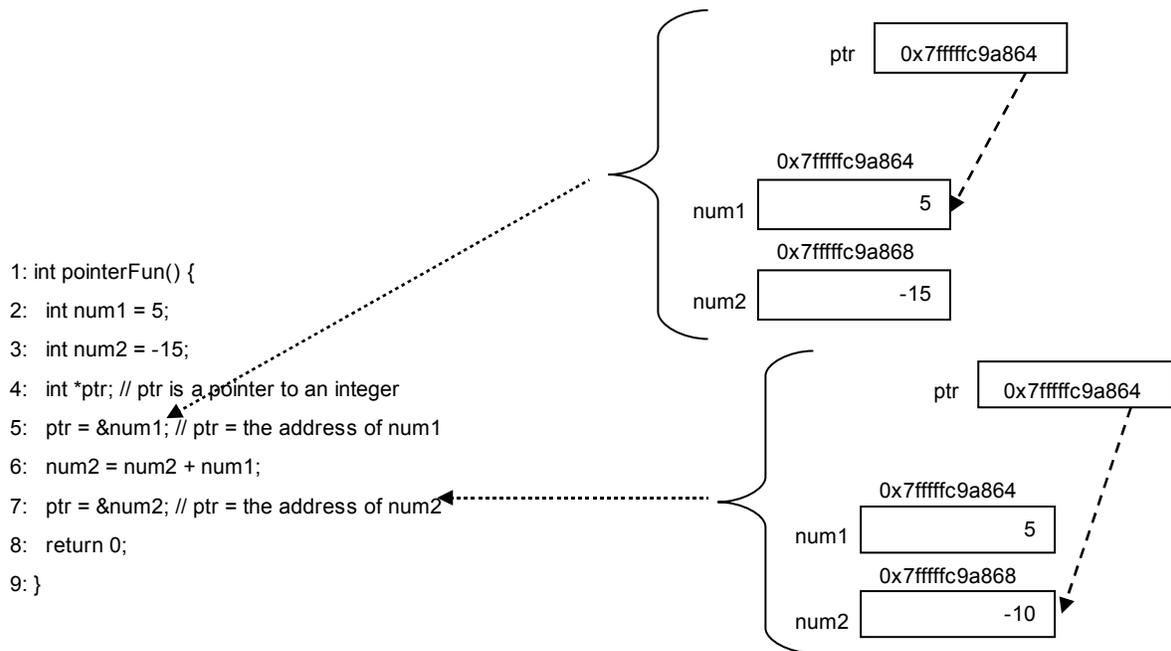


Figure 2: Simple pointer usage (in C)

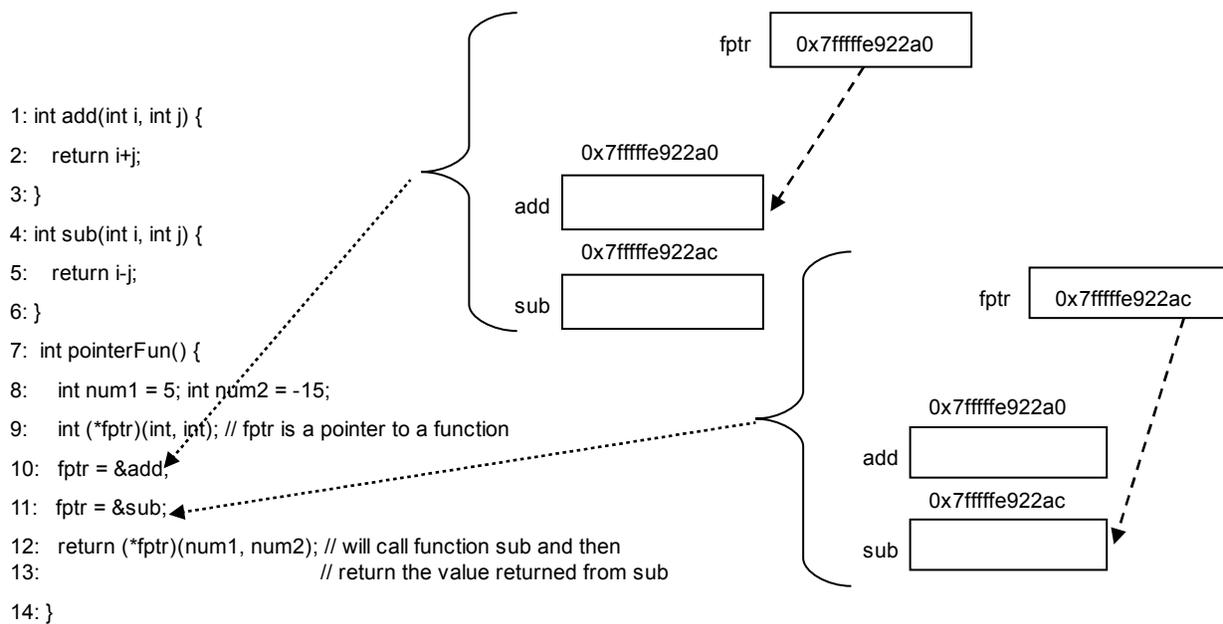


Figure 3: Indirect call made through a function pointer (in C)

While both of the examples demonstrated in figures 2-3 are trivial, they highlight the essential functionality of pointers. In figure 3, line 12 contains an indirect function call via the use of a function pointer, *fptr*. In this particular example it is not difficult to determine that the function call on line 12 will call function *sub* because we can see from the sequence of statements executed that the most recent assignment of *fptr* was to *sub* (line 11). However, imagine a scenario where the right-hand side of the assignment on line 11 was replaced by a call to a function *foo* defined in a distant part of the code which returns a function pointer. Imagine that *foo* itself consisted of multiple calls to functions defined elsewhere. When the call to function *foo* returns and the assignment on line 11 is carried out, it is impossible to determine what function *fptr* points to at this point of execution without a thorough analysis of the code. This scenario highlights the ambiguity that can arise when one is trying to determine the potential values of a pointer at a given point in a program's execution.

In order to disambiguate the relationship between any two given pointer expressions, an analysis must make either one of two assertions. The first and weaker assertion is that for some execution path *P*, two memory references *a* and *b* refer to the same memory location; this is known as a may-alias relationship. The second and stronger conclusion, known as a must-alias relationship, asserts that for *all* execution paths, two memory references *a* and *b* refer to the same memory location. Due to the nature of this problem, all pointer alias analyses are undecidable and thus, the implementation of any alias analysis is based on an approximation algorithm [14]. Existing algorithms for pointer alias analysis differ considerably in their precision and cost. However, all analyses are conservative and therefore guaranteed to report all possible pointer relationships that could actually occur at runtime. An analysis *A* is said to be more precise than analysis *B* if solution set computed by *A* is a subset of the solution set of *B*. It is generally agreed that more precise algorithms are usually much more costlier to compute but whether the additional cost is worth the degree of accuracy that such algorithms yield is debatable [16]. The worst-case time complexities of existing analyses range from almost linear to doubly exponential [11], although most often the worst-case behavior is not indicative of typical performance.

There are numerous classifications of pointer alias analyses in the research arena. Although some of these classifications will be discussed in various parts of this thesis, the broadest classification referenced in this thesis divides a pointer alias analysis into two categories: high level (source level) alias analysis and low level alias analysis. A source level alias analysis can be further classified based on the type of information it collects as it analyzes program code. The following two subsections provide a thorough discussion of each type of analysis and the benefits and drawbacks that each carries.

2.2 High Level Alias Analysis

A pointer alias analysis performed at the program source code level (or an intermediate high level code representation) is known as a high level alias analysis. By nature, high level alias analyses can take advantage of source level semantic information and therefore provide more accurate disambiguations of pointer references. A source level analysis can be described in terms of several properties: flow-sensitivity, context-sensitivity and type-sensitivity [10].

Flow-sensitive alias analyses take into account control flow within the program code and thus produces separate aliasing information for different points in the control flow of the program. Contrastingly, flow-insensitive analyses produce aliasing information for the entire program, disregarding both flow of control and statement execution order. Therefore, flow-insensitive analyses, while less expensive than their

counterparts, are more conservative in their approximations of pointer relationships. Context-sensitive alias analyses distinguish between different invocations of a procedure within the program code. For instance, if a procedure P is called from procedure \mathcal{Y} and procedure \mathcal{Z} , a context-sensitive analysis will produce distinct alias information for both $P\mathcal{Y}$ and $P\mathcal{Z}$. A context-insensitive analysis ignores such distinctions and therefore produces a solution set that is more conservative and hence less precise. (As discussed earlier, the smaller the solution set of an analysis, the more precise the analysis is considered to be in relation to other analyses. On the same note, the more conservative an analysis is in its approximations of pointer relationships, the larger the size of the solution set it will produce.)

The final classification of source level alias analyses pertains to the use of type information. Type-sensitive alias analyses make use of type information when deriving aliasing relationships [20]. For instance, a basic type-sensitive alias analysis would recognize that two references a and b that are of incompatible types cannot alias each other. Although the incorporation of type information in creating aliasing relationships can significantly improve the results of an alias analysis, such information is only available at the source code level and therefore in scenarios where users do not have ready access to the program source code, an alternative approach must be undertaken for code analysis.

2.3 Low Level Alias Analysis

An alias analysis that analyzes program assembly code (available via the program binary) is known as a low level alias analysis. Because of the nature of assembly code, the techniques used for source code analysis are insufficient for analyzing executable code. Below is a list of some noteworthy challenges that arises when dealing with assembly code (compiled from a publication by Brumley and Newsome [3]):

- Assembly code lacks the presence of any expressive types and therefore any form of heuristic based on type information is obsolete.
- Assembly code lacks any notion of function abstractions and control flow is exclusively defined by either unconditional or conditional jumps to locations.
- Memory in assembly is laid out as one, big contiguous chunk of storage, making it difficult to determine where the allocation of one object ends and the next one begins.
- Heavy use of address arithmetic in assembly makes such arithmetic difficult to ignore. Many source level alias analyses do not take into account address arithmetic but all low level analyses must do so in order to successfully handle memory dereferences at the assembly level, which almost always involve address arithmetic.
- Widespread use of indirect jumps in assembly makes it difficult to predict flow of control.

The use of a low level alias analysis becomes necessary when source code is not available, such as in the case of malware analysis where the malware is only available in its executable form. In addition to situations where its use is a must, low level alias analyses can provide some significant benefits over source level analyses. Whereas source level analyses typically support a specific high level language, low level analyses make no assumptions pertaining to the syntactic constructs defined in certain high level languages and can therefore handle code that may originally have been written in multiple languages. Some source code, while being primarily written in one high level language, can also contain inlined assembly code; only a low level analysis would be able to analyze such sites of inlined assembly code.

While the implementation of an alias analysis that processes assembly code can be significantly more challenging than the implementation of a source level alias analysis, its usage is sometimes necessary and in many cases, such an analysis can provide information that otherwise would be lost.

3. OVERVIEW OF CASE STUDIES

The following three chapters outline the three different program analyses that I have implemented in the course of my undergraduate research career. Each analysis was implemented because it was deemed to be the most appropriate – in terms of cost, efficiency and degree of accuracy - for the problem being addressed. Chapter 4 discusses my implementation of a source level flow insensitive alias analysis with the goal of identifying and eliminating unreachable code in the Linux kernel and thereby compacting its size to better satisfy the needs of an embedded device . Chapter 4 and 5 discuss two different approaches to addressing the same problem: determining the transition from seemingly benign to actual malicious code in a malware binary. Chapter 4 describes a standalone, rudimentary dynamic assembly level analysis that adds instructions to the existing assembly code in order to exert some control over the execution of the binary. Chapter 5 discusses the implementation of a more sophisticated and detailed analysis that can process assembly code at a finer granularity.

4. FA ANALYSIS

The ability to identify and eliminate unused code in an application is becoming increasingly significant as the amount of memory space available in many popular special-purpose computer systems becomes limited [6]. A perfect example of this is embedded devices. Over the past decade, the use of embedded devices such as cell phones, MP3 players, digital cameras, microwave ovens and DVD players, has evolved rapidly with the continuous desire of consumers for newer, smaller and slicker-looking gadgets. The urge to decrease size and weight, reduce power consumption, and lower production cost has limited the amount of memory available to the computer systems embedded within these devices. In order to satisfy the ever increasing desire to execute more and more sophisticated applications – such as encryption and speech recognition – these systems must now be able to run larger programs on limited space. The accumulation of these desires leads to increasingly large programs running on limited memory space, and as a result, the need for an operating system that can be tailored to run sophisticated applications while using a small amount of memory is invaluable.

4.1 The Problem

One of the most popular operating systems (OS) that vendors use for embedded devices is Linux, which has been freely available to the public since its creation. Built as a general-purpose OS, however, Linux is less sensitive to the limited resources of an embedded device, in particular, the reduced memory availability. Thus, code compaction or reducing the code size of the Linux kernel - the central component of the OS responsible for managing system resources such as memory - is critical in tailoring the Linux OS to better suit the environment of an embedded device. The applications in the Linux kernel contain much more data than codes for greater functionality than what is needed or desired for an embedded device. Thus, identification and elimination of unused code in the kernel (as per the requirements of the device) - referred to as dead code elimination - is integral in the compaction of the overall operating system [9].

In order to identify unused or dead code in a given program, it is necessary to construct a call graph for the program that provides information about all the calling relationships between different functions in the program. Such a program call graph can be used to determine, given a set of input functionalities desired of the program, what part of the program code will never be called in order to carry out those functionalities. With the construction of an accurate call graph, one can identify code that is never executed and therefore can be eliminated. Hence, it is necessary to carefully choose and implement an alias analysis that can construct the most precise program call graph while maintaining a manageable cost.

4.2 FA Analysis Implementation

The flow-insensitive alias (FA) analysis, a type of pointer alias analysis developed by Zhang et al. [22, 23], has been previously shown to produce the most accurate call graphs in a comparison test with other analysis algorithms [16]. In addition to ignoring the order in which statements are executed in the code (hence, flow-insensitive), the FA analysis is also context-insensitive, field sensitive and type sensitive. Being both flow- and context-insensitive means that the analysis algorithm has a low cost – running in almost linear time in terms of the size of the program and the size of the produced call graph – and low precision. For the purposes of creating a call graph for the Linux kernel, however, the low precision is tolerable because obtaining even some information about calling relationships in a relatively short

execution time is invaluable.

The FA analysis algorithm begins by constructing an object name for each memory location found in the code. As the kernel code is processed, the analysis merges object names together into sets called equivalence classes. Each equivalence class represents a set of aliases. At the end of the analysis, a graph that contains the alias sets for all function pointers in the code is created. Once this graph is constructed, determining the potential targets of an indirect call (made through a function pointer) is simple. If an indirect call uses a function pointer p , then the possible targets of that call are all the function object names in the same equivalence class as p . Figure 4 shows a small step-by-step example of how – given a series of source code statements (taken from figure 3 in Chapter 2) – the analysis identifies the set of object names and groups them into equivalence classes.

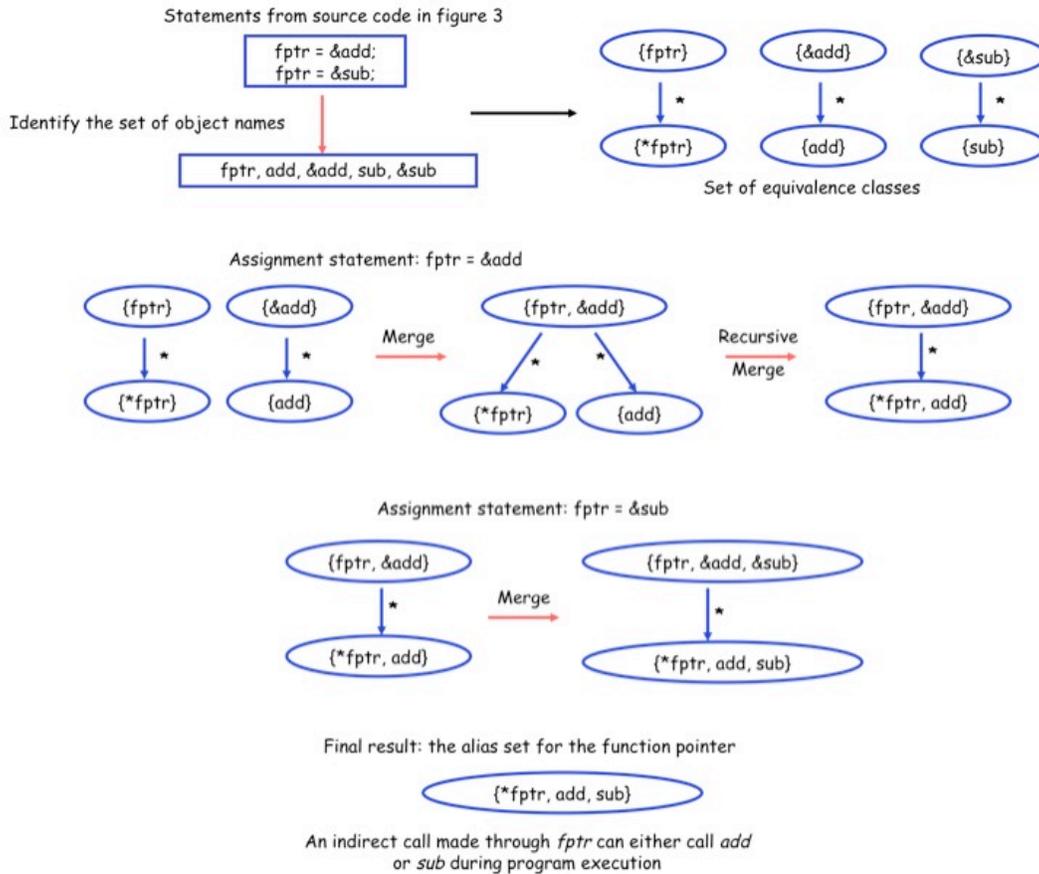


Figure 4: Set of equivalence classes produced from the source code in figure 3

Note: * = *pointer dereference* symbol = the value at the location to which a pointer points to; & = *address of symbol* = the address in memory where variable is located

4.3 Application of the FA Analysis

The construction of a program call graph for the Linux kernel consists of the following two steps:

1. Converting the C source code of the Linux kernel into compilation unit summaries using an XML markup that preserves semantic information pertaining to the high level language, such as the types of variables. Since flow of control statements are not taken into consideration by the FA analysis, this information is not present in the final compilation unit summaries. This transformation of C source code into compilation unit summaries (that would then serve as input to the FA analysis) was performed by John Trimble, a former undergraduate student in our research group who has worked extensively with the FA analysis algorithm, including writing his own implementation in Python [20].
2. My implementation of the FA analysis, written in the C programming language, loops through each compilation unit summary produced in step 1 until all summaries have been processed. In order to process the XML summaries, my implementation uses Libxml2, a XML C parser and toolkit developed for the gnome project [15]. Libxml2 parses a XML file and produces a tree data structure which can then be traversed by the client - in this case, my program - in order to process each node information. Throughout this processing phase, the equivalence classes defined in the FA analysis algorithm are generated. At the end, the analysis dumps the call graph for the kernel, which consists of the following:
 1. for each function with indirect calls – print the list of potential targets of that function (as a result of the indirect calls)

Figure 5 presents the code for a sample Hello program that consists of an indirect function call. Figures 6 and 7 contain the resulting compilation unit summary and call graph produced by the FA analysis for the Hello program, respectively. These figures originally appeared in Trimble's thesis [20] and are reproduced here as reference.

```
void hello(){ printf("hello world\n"); }

void main(void){
    (void *f)();
    f = &hello;
    f();
}
```

Figure 5: Hello program code

```

<compunit>
  <function id="hello">
    <decl id="hello">
      <namespace/>
      <type-ref key="1"/>
    </decl>
    <functioncall>
      <decl id="printf">
        <namespace/>
      </decl>
      <param num="1">
        <decl id="@cststring1">
          <namespace/>
          <type-ref key="2"/>
        </decl>
      </param>
    </functioncall>
  </function>
  <function id="main">
    <decl id="main">
      <namespace/>
      <type-ref key="1"/>
    </decl>
    <assignment>
      <decl id="f">
        <namespace>
          <decl id="main">
            <namespace/>
          </decl>
        </namespace>
      </decl>
      <type-table>
        <type-ref key="1"/>
      </type-table>
    </assignment>
  </function>
  <type-table>
    <type-ref key="1"/>
  </type-table>
</compunit>

```

Figure 6: Hello program compilation unit summary

```
main:hello,
```

Figure 7: Hello program call graph produced by FA analysis

Once the program call graph of the Linux kernel code has been constructed, it is then used by the code compaction software to compact the kernel. The flowchart in figure 8 summarizes this sequence of events.

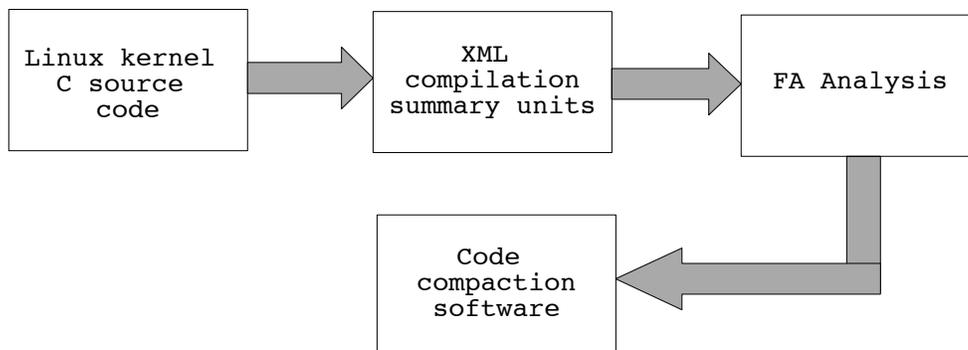


Figure 8: Overview of Linux kernel compaction apparatus

4.4 Experimental Results and Evaluation

A pointer alias analysis can be evaluated based on several metrics, the most common of which are accuracy of results and speed (low execution time). The desire for accuracy in an analysis is obvious but efficiency is an equally important characteristic because often times alias analyses are used by client programs as a small part in an apparatus that is already complicated and expensive. Thus the motivation to keep the alias analysis itself simple and inexpensive is high. In the case of the FA analysis and its role in the kernel code compaction apparatus (see figure 8), it was crucial to have an analysis that would process the entire kernel in a relatively short time. Additionally, while the FA analysis was specifically applied to the kernel compaction software in our research work, we wanted to implement an inexpensive alias analysis that would be an attractive choice for other applications as well.

Since the source level code of the Linux kernel was converted into a XML representation, the implementation of the FA analysis was not required to be written in any specific language. One of the primary motivations of writing the implementation in C was to take advantage of the flexibility that C data structures allow in order create an efficient analysis. My initial implementation of the FA analysis took upwards of 8 hours to process the entire Linux kernel, the approximate size of which is 781,826 bytes. Using program profiling tools such as `gprof`, the GNU profiler [8], I was able to identify several execution bottlenecks - areas of the code where most of the execution time was being spent - in my implementation code. Mostly all of the identified bottlenecks pointed to sections of the code that used expensive string comparison operations to process tree nodes. After modifying my implementation to make use of a more efficient string table lookup to process the tree nodes, I was able to reduce the execution time of the analysis to 2.5 hours. This speedup, while significant, was still far from an ideal execution time for a pointer alias analysis. Further use of `gprof` gave mixed results as to any remaining bottlenecks in the code, making it difficult to determine the underlying source of the execution time overhead.

In regards to the accuracy of the program call graph produced by the FA analysis, results varied depending on the input size. When analyzing individual modules of the Linux kernel, the analysis proved to be fairly accurate, producing a call graph that had at most twenty potential targets for any given function with indirect calls. When analyzing the entire kernel code, however, imprecision crept into the analysis results, generating a call graph that for some functions with indirect calls, produced hundreds of potential targets.

It is difficult to pinpoint exactly the source of the disappointing results of the FA analysis implementation. The following are some hypotheses regarding the outcome:

1. The Libxml2 parser may have played a role in the poor performance of the FA analysis. The XML parser was repeatedly invoked by the analysis in order to process each compilation unit summary. However, due to the existence of the parser's source code in a separate library (not available to us), the GNU profiler was unable to perform any profiling analysis on the parser. This may explain why the profiler produced such mixed results at times, unable to pinpoint the execution time bottlenecks in the code.
2. Information lost in the process of converting the Linux kernel source code files into XML compilation unit summaries may have led to the imprecision observed in the FA analysis results.

5. DYNAMIC ASSEMBLY ANALYSIS

Security of computer systems has been a key component of computer science research for many decades but in recent years, with an explosive surge in malware attacks, computer security has come to the forefront of research investigations. Today, there is a growing market for security products that can detect and eliminate viruses and worms from a computer system before any harm is done. Thus, the need for tools to better understand a binary executable and determine whether it exhibits any suspicious characteristics commonly observed in malicious code is critical for ensuring computer security.

5.1 The Problem

Recently, one of the most common virus detection methods employed by the software security industry involves a brute-force test on the suspect file by trying to match byte patterns from the file to a database of known malware byte patterns. Many malware writers today fight against such detection methods by transmitting malware in a packed or encrypted form and thereby scrambling the bytes in the executable [21]. In a packed malware, the malicious code is originally encrypted to evade detection by antivirus scanners and is only unpacked (decrypted) and executed when the malware is run. When the malware is executed, control jumps to the unpacker routine, descrambling the bytes until the malicious code is visible, and then jumps to the recently unpacked malware code. Because of the nature of packed malware, any binary exhibiting the characteristics of a self-modifying piece of code is flagged as suspicious.

This packed format presents an obstacle for researchers wishing to analyze malicious code, many of whom turn to dynamic code analysis techniques for the task [12]. Most of the existing dynamic analyses used in malware detection function by executing each instruction in the binary and scanning memory in search for matches with the known malware database. While such an analysis can successfully detect a packed malware, any information distinguishing the part of the malware code that is responsible for carrying out the unpacking (the unpacker routine) versus the part that actually executes the unpacked and malicious code is unknown. The lack of such information prevents researchers from forming any concrete ideas about the control flow graph of the malicious code and therefore, fully understanding the behavior of the malware.

5.2 Dynamic Assembly Analysis Implementation

Our first, simple approach in determining whether a binary executable is a packed malware and then identifying the point of transfer from the unpacker routine to the actual malicious code in the binary involves a brute-force dynamic assembly analysis. The objective of the analysis is to process and execute the binary instruction-by-instruction until a point deemed to be the transition point is reached. The transition point is defined to be the moment right before the analysis executes an instruction that has been modified by a previously executed instruction.

Before the analysis phase begins, an initial disassembly of the binary (using an existing disassembler tool) breaks down the executable into basic blocks and within each basic block, into a list of assembly instructions. The dynamic assembly analysis processes the global list of instructions generated from the disassembly and maintains a list of all the memory locations that have been written to by the instructions, referred to as the write set.

For each assembly instruction, the analysis performs the following:

1. Prior to executing the instruction, a check is performed to see if the address of the instruction to be executed exists in the `write set`
2. If the instruction address exists in the `write set`, the analysis stops because the transition point has been found.
3. If the instruction address does not exist in the `write set`, the analysis executes the instruction. If the instruction is an instruction that writes to memory, the destination address of the instruction (the location where it writes to memory) is added to the `write set`.
4. Repeat the process for next instruction.

My implementation of the dynamic assembly analysis involved adding the above described functionalities in the form of additional assembly instructions inserted into the global instruction list. The implementation also included the addition of assembly instructions that ensured the safe manipulation of the stack and all the registers prior to the execution of each original instruction from the binary. While I implemented the described instrumentations of the global instruction list, a key part of the instrumentation required the usage of address translation routines that was implemented earlier by a graduate student in our research group.

5.3 Application of Dynamic Assembly Analysis

The dynamic assembly analysis that we implemented was applied to a known packed malware: the Hybris C email worm. (figure 9 displays the unpacker code for this malware.) The malware initialization begins by loading registers with the address (`0x401000`) and size (5418 words) of the region to be unpacked, and the decryption key. The code then iterates through a loop, decrypting the specified region until the counter goes to zero. At this point the execution drops out of the loop and jumps to the unpacked region, which now contains the descrambled, malicious code. While unpackers employed by other malware may vary in certain aspects, all unpackers share the common self-modifying characteristic evident in Hybris C; that is, the property of modifying memory to create new code that was not present in the original binary.

<i>Instr</i>	<i>Address</i>	<i>x86 assembly code</i>	<i>Explanation</i>
...	0x401000:	...	<i>encrypted malware body</i>
I_0	0x4064a8:	<code>movl %edx ← \$0x152a</code>	<i>register %edx ← size of region to be unpacked</i>
I_1	0x4064ad:	<code>movl %eax ← \$0x401000</code>	<i>register %eax ← start address for unpacking</i>
I_2	0x4064b2:	<code>movl %esi ← \$0x44b3080</code>	<i>register %esi ← initial decryption key</i>
I_3	0x4064b7:	<code>subl (%eax) ← %esi</code>	<i>decrypt the word pointed at by %eax</i>
I_4	0x4064b9:	<code>addl %esi ← \$0x2431400</code>	<i>update decryption key</i>
I_5	0x4064bf:	<code>addl %eax ← \$4</code>	<i>update code pointer</i>
I_6	0x4064c2:	<code>decl %edx</code>	<i>decrement counter</i>
I_7	0x4064c3:	<code>jne .-6</code>	<i>loop back to I_3 if the counter is nonzero</i>
I_8	0x4064c5:	<code>jmp 0x401000</code>	<i>jump to unpacked code</i>

Figure 9: Hybris C email worm

5.4 Experimental Results and Evaluation

The dynamic assembly analysis was tested on the Hybris-C and Hybris-D email worms. Hybris-D is an extension of the Hybris-C worm and the code for both malware is almost identical except for the technique each employs in directing

control to the unpacked code once the unpacker routine has completed. In the case of Hybris-D, instead of employing the more obvious single jump statement for this purpose (as is evident in Hybris C, see figure 9, I_8), it uses two instructions – a push and a return – to push the address of the unpacked malware body onto the stack and then jump to that address. The dynamic assembly analysis was successfully able to recognize the suspicious behavior of these two malware and identify the transition from unpacker code to unpacked malware body in each. In the case of the Hybris C example shown in figure 9, the correctly identified transition point is $I_8 : jmp\ 0x401000$.

While the dynamic assembly analysis successfully recognized the Hybris-C and Hybris-D worms as malicious binaries and identified the transition point in each code, both of these examples are relatively small and simple. When dealing with more complex and larger malware files, the approach taken by the dynamic assembly analysis will be significantly less efficient because it requires the instrumentation of each individual instruction that appears in the original binary prior to the execution of the unpacked malware body. Most packed malware today employs complex unpacker routines that consist of at least hundreds of instructions. Individual instrumentation of each of these instructions would be costly, resulting in a high execution time for the analysis. Furthermore, dynamic analysis of malware binaries is also subject to certain vulnerabilities that are discussed in detail in the next chapter.

6. VALUE SET ANALYSIS

The drawbacks of the dynamic assembly analysis described in the previous chapter has necessitated the need for a more sophisticated approach to addressing the problem of malware detection and behavior analysis. One of the major drawbacks with analyzing malware binaries dynamically is that it allows the malware to employ dynamic defenses such as anti-debugging defense, time bombs and logic bombs [5, 12]. A malware binary with an anti-debugging defense attempts to detect whether the binary's execution is being monitored by the host system and if so, does not carry out the execution of the malicious code. Other examples of dynamic defenses include time bombs, which cause the malware to be activated only on certain times or dates, and logic bombs, which cause the malware to be activated based on the detection of some environmental trigger. A dynamic analysis of a malware binary only explores one execution path of the malware and therefore in the case where dynamic defenses are employed, cannot always recognize the suspicious nature of the code. Hence, in order to analyze packed malware binaries that may also make use of dynamic defenses, it is necessary to conduct a static analysis that explores all possible execution paths of the binary.

Our method for analyzing packed malware statically can be summarized into two phases: (1) given an initial disassembly of a binary executable, identify code unpacking and find the associated unpacker routine and (2) transform the unpacker code into a customized unpacker that can then be emulated to unpack the malicious binary. A fundamental component of this method is a pointer alias analysis that is necessary for both (a) the identification of transition points that signal where control may be transferred to unpacked, malicious code and (b) the identification of the actual static unpacker routine. Alias information obtained from an analysis is essential in determining the possible targets of indirect memory operations and indirect control transfers in the malware binary.

The nature of this work requires a low level pointer analysis that can obtain alias information from an executable in the absence of source code, as is necessary when analyzing viruses and worms. Developing a pointer alias analysis algorithm that can be used to analyze potential malware executables with a satisfactory degree of precision is nontrivial. Thus, we chose to resort to existing literature to identify a suitable algorithm rather than formulating our own. The extensive work of Gogul Balakrishnan as published in his dissertation in December 2007 [2] provided us with a fairly recent, well-written and thoroughly detailed alias analysis algorithm, the performance and precision of which can be further improved, if so desired, by additional optimizations outlined by the author.

6.1 Value Set Analysis Implementation

Our alias analysis implementation based on Balakrishnan's dissertation work involved two distinct tasks. The first task was to implement all the necessary internal data structures such as representations of individual memory regions, abstract locations corresponding to variable-like entities in an executable, value sets denoting the set of addresses in a memory region and finally, mappings from abstract locations to value sets and from memory regions to value sets. Once all the data structures were in place, the next step was to implement the flow-sensitive, context-sensitive, intraprocedural analysis algorithm that aims to explore a program's behavior for all possible inputs and all possible states that the program can reach.

At the topmost level, the value set analysis deals with a data structure known as the `AbsEnv`, which represents the abstract environment of the program at any given point during its execution. The `AbsEnv` maintains a list of all the memory regions defined in the program. The analysis classifies all data objects into three distinct categories of memory regions: (1) *global* region - for memory locations that hold global data, (2) *activation record* region - for memory locations pertaining to a particular function or procedure (local data), and (3) *malloc* or heap region - for memory locations allocated at a particular heap site. In order to represent the potential value of any given memory location, Balakrishnan's algorithm employs several different levels of data structures, the most basic of which represents the value as a set, defined by a strided interval. Figure 10 provides a detailed look at the data structures involved in the value set analysis algorithm. While the implementation of the value set analysis was a group effort, I was personally responsible for implementing most of the underlying functionalities of the data structures shown in figure 10.

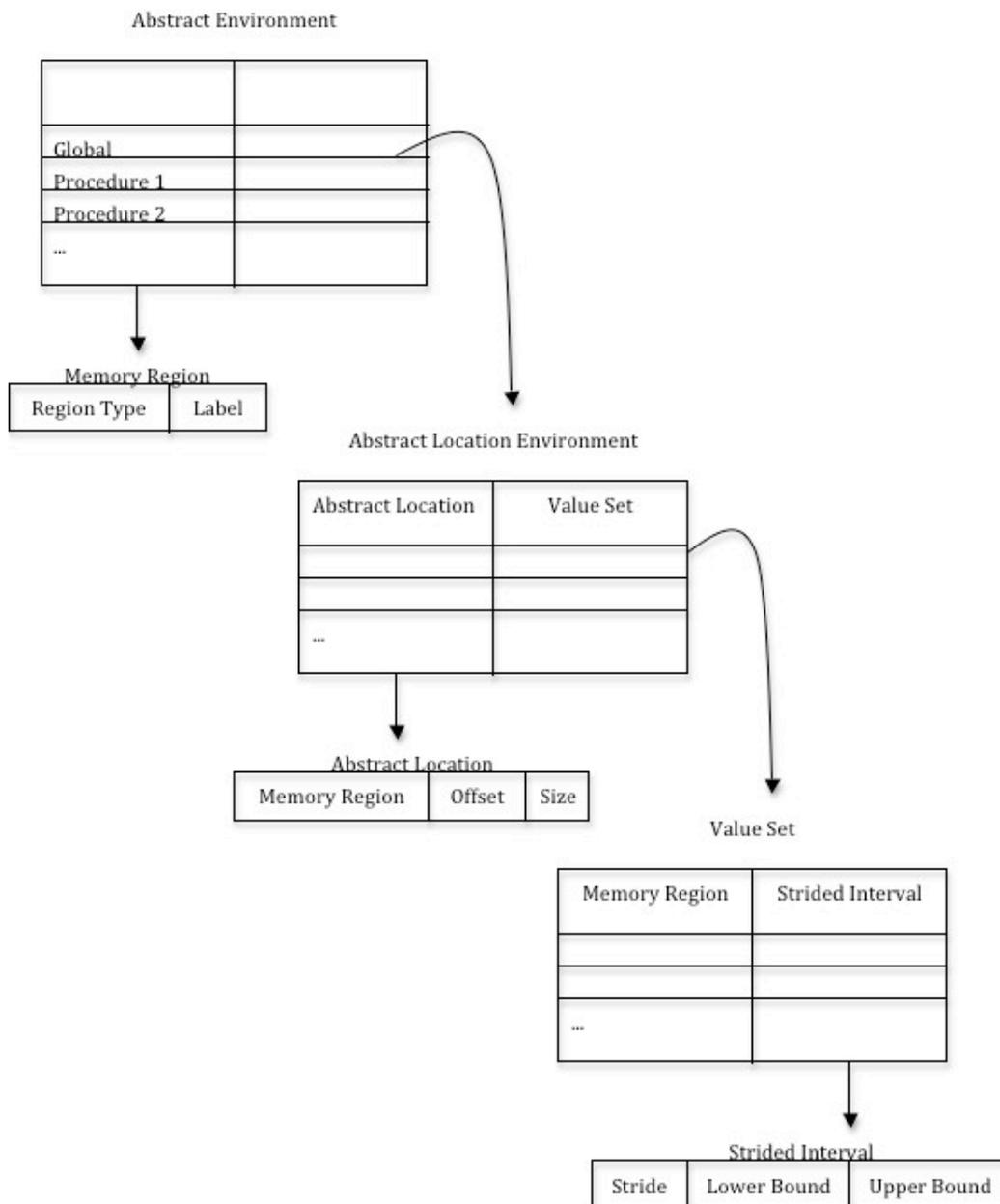


Figure 10: Value set analysis data structures

The value set analysis processes the malware binary instruction-by-instruction, updating all components of the current abstract environment (AbsEnv) for the program as necessary. In addition to keeping track of all data objects, the AbsEnv also maintains information pertaining to the potential value(s) of each of the registers at any given time.

6.2 Value Set Analysis Application

The value set analysis plays a critical role in our desire to determine whether a given binary executable exhibits the characteristics of a packed malware and if so, identify the transition point from the unpacker routine to malicious code. Before the analysis phase begins, the binary is disassembled using PLTO, a disassembler tool developed by our research group [19]. The disassembly partitions the original malware code into basic blocks and within each basic block, into a list of assembly instructions. The value set analysis then processes this global list of instructions, determining the possible target addresses of indirect memory operations in the disassembled code. Using the information gathered by the value set analysis, a list of potential transition points (or transfer of control to the unpacked code) is generated. The list generated is referred to as the set of “potential” transition points in order to account for imprecision in the binary level alias analysis. For each potential transition point t , the results of the value set analysis is used to determine the set of memory locations that can be modified along the execution paths to t to identify the static unpacker. Once the static unpacker has been extracted, it is transformed in order to eliminate any dynamic defense code and add instrumentation to identify the true transition point in the malware. Finally, the static unpacker is emulated to observe its effects on memory and hence, the remainder of the malware code.

Figure 11 provides an overview of the sequence of events described above. Step 1, parts of step 2, step 3 and most of step 4 was implemented by other members of my research group. In addition to my work on the value set analysis, I also implemented the instrumentation of the static unpacker (in its transformation phase) and the final emulation. Many parts of the code used for the instrumentation and emulation of the static unpacker was recycled from the dynamic assembly level analysis implementation described in Chapter 5.

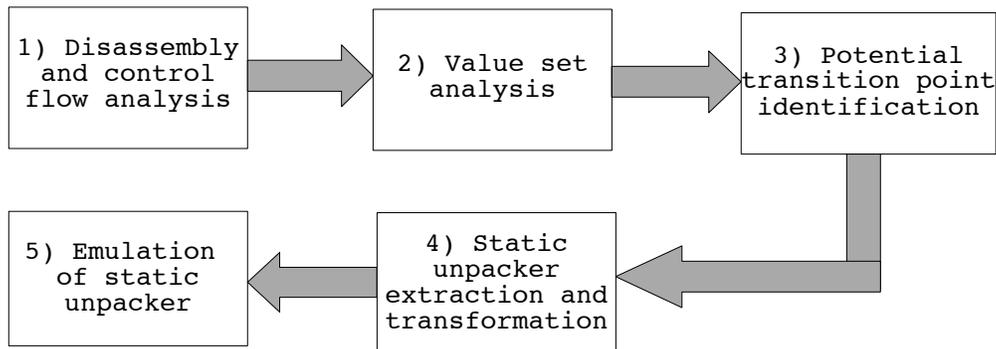


Figure 11: Overview of malware detection apparatus

6.3 Experimental Results and Evaluation

The static unpacker was tested on the original Hybris-C email worm described in Chapter 6 as well as several variants of the Hybris code. The variants, constructed by a doctoral student in our group, was specifically handcrafted to incorporate different types of dynamic defenses. Figure 12(a) shows the control flow graph of the original Hybris code. In this example, while the packed code begins with B₀ at address 0x401000, the malware begins execution with the unpacker routine at B₁ (address 0x4064a8). As the loop in the unpacker routine executes (block B₂), it overwrites the garbage instructions that are visible in B₀ with actual malicious code. Figure 12(b) shows a variant of the Hybris code that begins execution at B_{dd} by loading a value into register %eax and then conditionally branching to the unpacker routine if the value is nonzero.

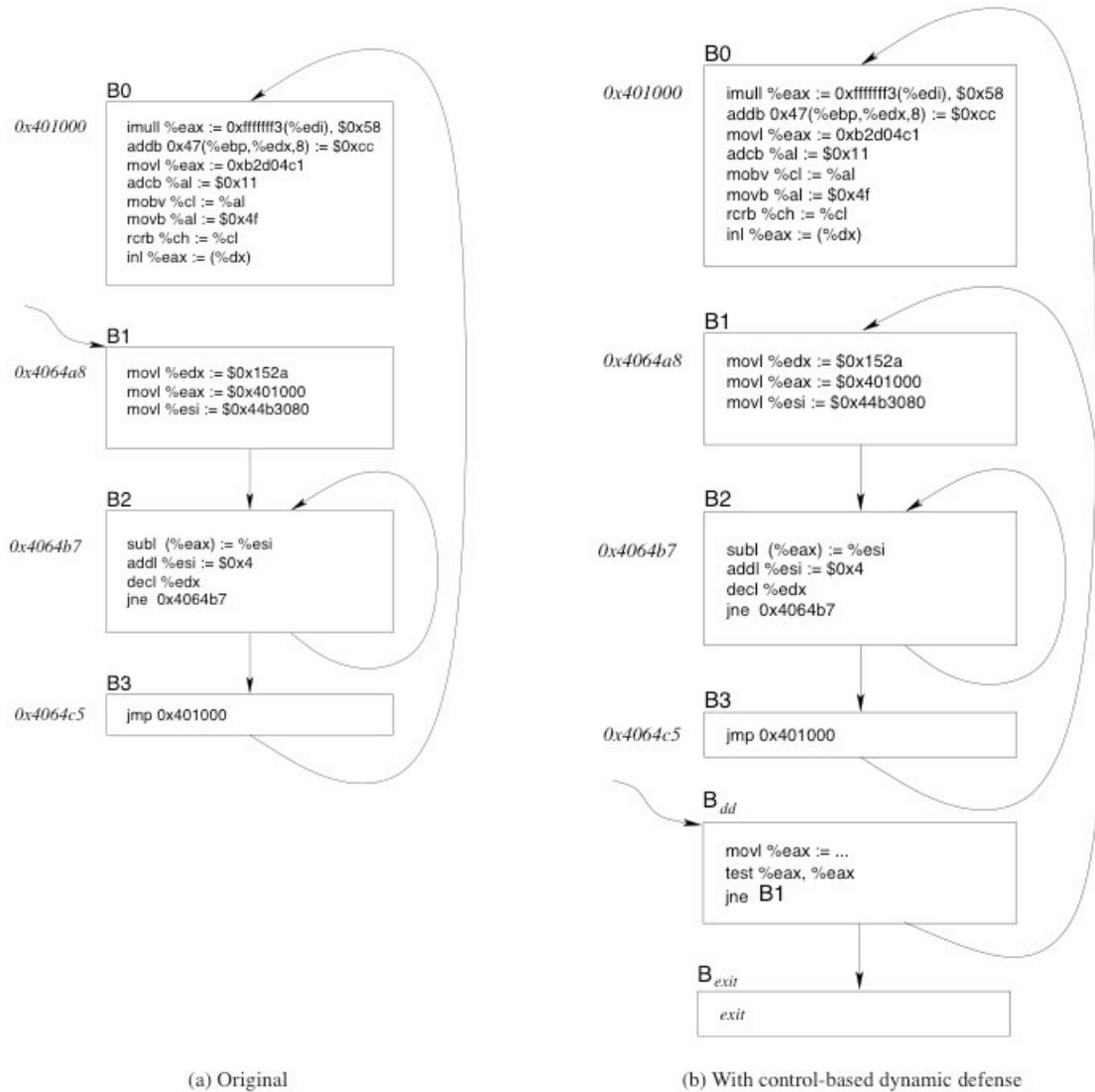


Figure 12: Two versions of the Hybris-C worm

Our prototype static unpacker was able to successfully identify the transition point, extract, modify and emulate the unpacker routine in each malware. In the case with the dynamic defense, the static unpacker correctly identified and eliminated the dynamic defense code.

7. CONCLUSION

After nearly two years of research work relating to pointer alias analyses, I have learned that implementations of most pointer analyses – even ones based on relatively simple algorithms such as the FA analysis – are nontrivial. While implementations of binary level alias analyses are much more complex and difficult than source level analyses, the latter is by no means a walk in the park. Below are several lessons I have learned over the course of my research experience:

1. Familiarity with the code base that one will be interacting with during the course of writing any implementation is crucial in successfully completing a good implementation. In the ideal case, we would all be working with our own code all the time and hence there would be no need to understand code written by another programmer. In reality, however, we are often given the task of continuing to build or extend a project that has been previously implemented by one or more programmers. In these situations, I have learned that it is essential to understand what others have written thus far before beginning one's own implementation. Sometimes we are lucky and the previous authors are available and willing to help in the understanding of their code in person or via email. Other times we are not so lucky and the task becomes more difficult.

While working on the FA analysis implementation, I decided to not worry myself with the GCC frontend, implemented by John Trimble, because it was independent of the analysis implementation. In retrospect, I wish I had spent more time understanding the frontend that produced the XML compilation unit summaries that my implementation had to process. I believe that had I spent some time in understanding Trimble's transformation of Linux kernel source code into XML summaries, I might have had a better understanding of the imprecision that was observed in my analysis results and its potential causes.

Additionally, my work on the FA analysis has taught me the uncertainties that accompany any usage of libraries created by other research groups. Like in the case of most libraries available for free on the web, the writer(s) of the Libxml2 parser (that I used in my FA analysis implementation) provided an API (application programming interface) outlining all the functionalities that the parser supported. The source code for the parser was not readily available to all its users and hence, I was never able to confirm our hypothesis that the inefficiency evident in my analysis implementation was somehow related to the parser implementation.

Learning from the FA analysis work, when it came to implementing the value set analysis outlined by Gogul Balakrishnan, instead of using the analysis API that Balakrishnan and his research group made available to other academic research groups, we decided to build our own implementation from scratch. This gave us the advantage of being able to modify Balakrishnan's algorithm for our own purposes and made us well aware and knowledgeable about the code base we were working with.

As I have learned from experience, familiarity with one's code base is critical in hunting down program bugs and understanding program behavior.

2. Accuracy always trumps efficiency and any optimizations should be implemented at the end only if there is good indication that the benefits it provides will outweigh the cost of the additional work. In each of the analyses I have helped to implement thus far, our first and foremost goal has always been accuracy. Often times, this meant that our initial implementation was always the most simple and straight-forward method of addressing the task, even if the methodology was inefficient.

In the case of the FA analysis, my first implementation involved a simple linked-list data structure to process all the tree node information received from the parser. In my later, more efficient implementation, I made several optimizations to the code, including the addition of other data structures, such as a hash table for string lookup and comparisons.

Likewise, in the course of our value set analysis implementation, we decided to keep things simple, implementing only the first part of Balakrishnan's algorithm: an intraprocedural analysis. While the addition of an interprocedural phase to the analysis will most likely be necessary in the future, we wanted to have a simple, running analysis that we could test using our basic test cases.

During the implementation of both the dynamic assembly analysis and the value set analysis, we also considered several optimization algorithms. When dealing with assembly level code, the optimal functionality for any program analysis is to be able to recognize when the code enters a loop and instead of emulating all the iterations of that loop (a very expensive and time consuming task), make some smart conclusions about the data objects whose values the loop modifies after emulating the effects of one or at most two iterations of the loop. There are some existing complex optimization algorithms that deal particularly with these types of situations. For the dynamic assembly analysis, no optimizations were implemented. For the value set analysis, we initially considered implementing one such optimization known as widening and its inverse, narrowing. Simply put, in the case when the value of a data object is being modified inside a loop, a complex widening and narrowing optimization [4] would recognize the repeated modification and make an intelligent conclusion about the final value of the data object at the end of the loop.

Because of the complexity involved in implementing such an optimization, at this time we have decided to resort to a very simple version of widening in our analysis. In our current implementation of the value set analysis, when the value of a data object is recognized as being continually modified inside a loop (usually after two iterations), the analysis immediately sets the value set corresponding to that object as unknown. In this manner, we are not being precise in our implementation but at the same time, we are ensuring that incorrectness does not creep into our analysis results. For the Hybris-C virus variants on which we have tested our analysis thus far, our implementation has proved to be sufficient. It is likely, however, that down the road we will be implementing these optimizations when dealing with more complex malware that necessitate the use of a smarter analysis.

It is difficult to quantify the amount of work, time and effort that has gone into the implementations discussed in this thesis. As is the case with most big programming projects, each implementation phase not only involved writing many lines of code – at least several thousand – but also countless hours of debugging, the joys of which only a programmer can appreciate.

8. FUTURE WORK

For the Linux kernel code compaction project, the source level FA analysis provided aliasing information only at the function level. This, unfortunately, limits the amount of unreachable code in the Linux kernel that can be identified and therefore eliminated. If we were able to collect more information from the Linux kernel source code, an extension to the FA analysis may be made to provide further finer grained aliasing information, specifically relating to memory references [20]. This information would, in turn, be beneficial to the code compaction software.

In regards to the binary level value set analysis, several extensions of the current implementation is planned for the future in order to successfully handle a larger collection of malware. Due to time constraints, our initial implementation of Balakrishnan's value set analysis consisted only of an intraprocedural phase. In order to handle procedure calls as well as indirect jumps and calls, we would like to extend our current implementation to include an interprocedural analysis phase. Additionally, we would like to explore possible implementations of several optimizations briefly discussed in Balakrishnan's thesis, specifically a sophisticated widening algorithm [4] and an affine-relation analysis [17].

9. ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Saumya K. Debray for his continuous support, encouragement and guidance throughout my undergraduate research career. Special thanks to John Trimble for making my transition into the FA analysis implementation project smooth and pain-free. I would like to thank Kevin Coogan – a mentor and colleague who stars as the anonymous graduate student several times in this thesis – for his patience and wisdom during our work together on the assembly level alias analyses. Special thanks to Gregg Townsend for his clear articulation of Balakrishnan's value set analysis. Finally, I would like to thank all the current and former members of the SOLAR research group – especially Dr. Greg Andrews, Somu Periyayagam, Drew Davidson, Haifeng He, Joe Roback and Keith Fligg – who have helped me grow academically and individually during my undergraduate years at the University of Arizona.

All my love and thanks to my dear family – Abbu, Ammu, Apu and Safat – for the never-ending love, support and faith.

10. REFERENCES

- [1] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Conf. on Compiler Construction (CC)*, pages 5–23, April 2004.
- [2] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*, Ph.D. dissertation and Tech. Rep. TR-1603, Computer Sciences Department, University of Wisconsin, Madison, WI, August 2007.
- [3] D. Brumley and J. Newsome. Alias analysis for assembly. Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.
- [4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. Principles of Programming Languages (POPL)*, 1977. [wdiening]
- [5] A. Danielescu. Anti-debugging and anti-emulation techniques. *CodeBreakers Journal*, 5(1), 2008. <http://www.codebreakers-journal.com/>. [dynamic defenses]
- [6] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.
- [7] Saumya K. Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Symposium on Principles of Programming Languages*, pages 12–24, 1998.
- [8] GNU gprof. <http://gnu.huihoo.org/gprof-2.9.1/gprof.html>.
- [9] Haifeng He, John Trimble, Somu Perianayagam, Saumya Debray, and Gregory Andrews. Code compaction of an operating system kernel. In *Symposium on Code Generation and Optimization*, 2007.
- [10] Michael Hind. Pointer analysis: Haven’t we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE’01)*, Snowbird, UT, 2001.
- [11] M. Hind and A. Pioli. Which pointer analysis should I use? In *ISSTA ’00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.
- [12] Lord Julus. Anti-debugging in win32, 1999. VX Heavens. <http://vx.netlux.org/lib/vlj05.html>.
- [13] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proc. Fifth ACM Workshop on Recurring Malcode (WORM 2007)*, November 2007. [dynamic analysis]
- [14] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.
- [15] Libxml2 XML C parser. <http://xmlsoft.org/>.
- [16] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise and efficient call graph construction for c programs with function pointers. *Journal of Automated*

Software Engineering. 2004.

- [17] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. Principles of Programming Languages (POPL)*, 2004. [affine]
- [18] P. Ször. *The Art of Computer Virus Research and Defense*. Symantek Press, February 2005. [dynamic analysis]
- [19] Benjamin Schwarz, Saumya Debray, Gregory Andrews, Matthew Legendre. PLTO: A link-time optimizer for the intel ia-32 architecture. In *Proc. 2001 Workshop on Binary Translation*.
- [20] John Trimble. *Combining High Level Alias Analysis with Low Level Code Compaction of the Linux Kernel*. University of Arizona. Honors thesis, 2006.
- [21] Andrew Walenstein, Arun Lakhotia. The Software Similarity Problem in Malware Analysis. In *Dagstuhl Seminar Proceedings: Duplication, Redundancy, and Similarity in Software*. 2007.
- [22] Sean Zhang. *Practical Pointer Aliasing Analyses for C*. PhD thesis, 1998.
- [23] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Foundations of Software Engineering*, pages 81–92, 1996.