An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java

Lori Vinciguerra
ALPHATECH, Inc.
loriv@alphatech.com

Linda Wills, Nidhi Kejriwal
Georgia Institute of Technology
{linda, nidhi}@ece.gatech.edu

Paul Martino
Alphah Software

Ralph Vinciguerra
Trinity Research and Development, Corp.

Abstract

The inherent differences between C++ and Java programs dictate that the methods used for reverse engineering their compiled executables will be language-specific. This paper looks at the history of decompilers, disassemblers, and obfuscators in C++ and Java and presents the current state of the art for binary reverse engineering. An experimentation framework for evaluating tools is described, including methodology, benchmark programs, metrics, and reverse engineering tasks. Preliminary results of experiments conducted so far to assess the capability of a small select set of chosen popular tools are given. These results reveal language-specific differences in the feasibility of the binary reverse engineering tasks on input programs with varying degrees of obfuscation (e.g., stripped vs. unstripped binaries). In addition, the results reveal the relative effort required to complete a task and an assessment of the value of the tools and techniques.

Keywords: disassembly, decompilation, obfuscation, binary reverse engineering, binary translation.

1. Introduction

Computer software is at the heart of most of the technological and scientific assets of our country: from communications and commerce infrastructure to critical national security applications. These applications often contain legacy code that needs to communicate with newer software components. Often, the original source code is not available, requiring integrators of the newer components to reverse engineer the legacy interface from binary executables, sometimes adding new code to complete the interface. Sometimes, the legacy software must be ported to a new hardware architecture (for example, allowing the software to run on a faster, parallel computer). Some reverse engineering is initiated from a need to duplicate a legacy system.

Software reverse engineering techniques recover the inherent structure of programs, including data structures and algorithms used, components and their interrelationships, and the overall architecture and design. Software reverse engineering enables several forms of transformation, including:

- Binary to binary (binary translation and optimization, platform retargeting)
- Binary to source (decompilation, disassembly)
- Source to source (software re-engineering, language translation).

Several motivations drove the work performed and presented in this paper, including understanding the state of the art for binary-to-binary applications such as translation and understanding what disassembly and decompilation tools can reveal from executables. The latter is motivated by a need to understand how well software protection mechanisms work from a developer’s point of view (e.g., licensing software), and understanding the usefulness of reverse engineering tools for tracking down security flaws and vulnerabilities in binary code.

Our goal is to understand what tools are available and how to systematically assess their effectiveness. We have developed an experimentation framework for evaluating binary reverse engineering tools, which we call BinREEF (Binary Reverse Engineering Experiment Framework). This includes a set of C++ and Java benchmark binary programs representing varying degrees of obfuscation (e.g., stripped vs. not, optimized vs. unoptimized) and a set of reverse engineering tasks ranging from common, general questions (e.g., recovering the calling relationships) to application-specific tasks (e.g., recovering key parameters from a particular benchmark program).

Experimentation frameworks for studying reverse engineering tools and techniques are scarce. Notable isolated examples are the WELTAB [29] legacy election software that has been used as a common data set for evaluating source-level reengineering tools for large systems, a structured tool demonstration [23] designed to evaluate visualization and exploration tools for program comprehension tasks, and the C++ Extractor Test Suite (CppETS) [21, 22], a C++ benchmark which has been used to study fact extraction by parser-analyzer tools. Like CppETS, BinREEF includes a focused set of reverse
Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)
1095-1350/03 $ 17.00 © 2003 IEEE

data, such as jump tables for implementing indexed or instructions are encoded, may contain non-executable segment of C/C++ binaries, where the assembly

This paper is organized as follows: The next section summarizes the key issues in binary reverse engineering and the state of the art in addressing them, focusing primarily on language differences in C/C++ and Java. Section 3 describes BinREEF for empirically assessing these tools. Preliminary results are given in section 4. The results of experiments conducted so far on a select set of popular tools reveal language-specific differences in the feasibility of the binary reverse engineering tasks.

2. Key Issues in Disassembly and Decomposition: Language Differences between C/C++ and Java

Decomposition techniques utilize code binaries and recover high level language information, including expressions, procedure calls and parameter lists, while removing low-level machine-dependent details such as register usage and condition codes. Disassembly is a subset of this process and sometimes an early phase of decompilation that recovers assembly language from an executable.

Large differences in the state of the art binary reverse engineering tools exist between those developed for C/C++ versus those developed for Java. These differences arise due to language differences between C/C++ and Java as well as differences in the needs of applications that have traditionally driven the development of these tools. The Java language produces its executable format in a manner that is very different from C++. The Java class file contains Java bytecode according to the Java Virtual Machine (JVM) Specification. Some critical differences between C++ and Java executables are in the following areas.

Recovering data type information. Binaries for C/C++ and most other traditional programming languages are completely devoid of type information. Type recovery and propagation techniques are needed [10, 16]. Java, on the other hand, is a type safe language. This type safety is ensured at execution time by the Bytecode Verifier that examines all uses of data on all branches to locate the presence of any type collision. For verification to be decidable, complete type information must be present within the Java binary.

Distinguishing instructions from data. The “text” segment of C/C++ binaries, where the assembly instructions are encoded, may contain non-executable data, such as jump tables for implementing indexed or indirect jumps [3, 6], alignment bytes, or virtual tables for virtual method dispatching in object-oriented code [25]. This can cause the disassembler to create incorrect assembly code when data is misinterpreted as instructions [20]. It also complicates procedure and virtual method recovery [5, 25]. This problem has dominated much of the work on C/C++ disassembly tools (the main approaches to attacking this problem are summarized below). This problem does not occur in Java bytecode since instructions and data are explicitly separated as a fundamental design goal of the language. As a result all entry points to an executable are known and recovery of all instructions from entry points is a trivial task.

Platform-independence or retargetability. One of the defining characteristics of Java is its platform independence, which has benefits for portability, code reusability, enabling code mobility, and simplifying disassembly. This is in contrast to C/C++ binaries, which are highly machine-dependent. This especially complicates the recovery of high-level procedure calls, including which parameters are passed, which values are returned, and which local variables are used. Procedure calling mechanisms vary with the operating system and hardware architecture, so machine-independent procedural analysis is nontrivial and typically involves making use of specifications of procedure calling conventions for various machine and OS-specific binary interface descriptions. In addition, special-purpose idioms are often used in assembly programs that are specific to a particular machine (e.g., assuming two’s-complement number representations). Recognizing these idioms requires libraries of machine-specific patterns.

Although JVM implementations vary from platform to platform, all resulting class files from a Java compile contain more information than the resulting object file from a C++ compilation. Java bytecode contains all of the necessary information to reconstruct a Java source file: method names, member names, and a set of instructions. The set of instructions that are generated are constrained and easily anticipated when analyzing the class file. For example, the instruction iload will load an integer value onto the stack and the instruction fload will load a float value onto the stack. Bytecode instructions are openly available via the JVM Specification.

Standard library function identification. Disassemblers are difficult to write due to the variety of output of each vendor’s compiler. It is necessary to identify calls to standard library functions within the code. The development of compiler and library signatures [4] attacks the problem of library function identification. Library signatures can be used to reverse the task of a static linker by encoding for a piece of code, the particular library subroutine and compiler used for a given library file. Compile signatures help determine which compiler generated the binary, while library prototypes help
2.1. C/C++ Disassembly and Decompilation

Survey

The existing C/C++ disassemblers can be categorized by their approach to separating code from data, which is a key challenge for C/C++ binaries. The main static approaches are summarized here.

Linear sweep (LS) – this technique disassembles the segment of the binary reserved for instructions (e.g., the “text” segment), one instruction at a time in a linear fashion. It breaks down when there is data embedded in the text segment (e.g., a jump table or alignment bytes) [20].

Extended linear sweep (ELS) – Schwarz, et al [20] describe an extension of linear sweep in which jump tables are identified based on contiguous chunks of relocatable addresses. This technique is based upon the unreliable assumption that relocation information is available in the binary.

Recursive traversal (RT) – this technique disassembles instructions by following all control flow paths through the code [24]. It can miss a path if indexed or indirect jumps are used, since all targets of the control transfer cannot be precisely determined statically.

Data-flow guided RT (DRT) – this technique [6] augments RT with data flow analysis, using slicing and forward substitution to more accurately determine the extent of jump tables and the target addresses for indexed jumps. The technique relies on determining jump table sizes by analyzing bounds checks on indexed jumps. This will fail if bounds checking is not performed. (Interestingly, an obfuscation technique to protect binaries from this disassembly technique would be to remove bounds checking, but that would trade-off security by increasing the vulnerability to buffer overflow-based attacks.)

Hybrid ELS/RT – Schwarz [20] described an approach in which their extended linear sweep algorithm is run redundantly with a recursive traversal technique. This allows disassembly errors to be detected when the results do not concur.

Speculative disassembly – this technique is orthogonal to LS and RT. It keeps track of which portions of the binary have been disassembled and attempts to fill in gaps in disassembly coverage by speculatively disassembling the code. It marks the disassembled instructions as speculative and is able to back out by discarding them if an invalid disassembled instruction is detected. This depends on being able to detect when a disassemble instruction is invalid, which might not be possible if there is ambiguity between data and instruction encodings. This technique is helpful for indirect jumps and virtual tables in object-oriented languages [25]. However, it also generates a lot of noise in the disassembly in the form of unnecessary decodings [8].

Interactive disassembly – some disassemblers [1, 13] rely on having a human in the loop to control which portions of the binary should be disassembled. An important empirical question is whether people are any better at disambiguating instructions vs. data than automated tools.

Research into decompilation began over 30 years ago [11], but most techniques have been developed in the past decade [12]. The majority of these techniques rely on static analysis; however it is becoming clear that complementary dynamic analysis and run-time support for augmenting the static analyses is needed [9, 26, 27].

Most of the early disassemblers and decompilers for C/C++ were restricted to specific hardware platforms (e.g., dcc [2, 4] and XDASM [30]) or were compiler-specific (e.g., DisC [14] and decomp [18]). Retargetability of disassembly and decompilation techniques became an important design goal as they were applied to binary translation and optimization tasks. The New Jersey Machine Code (NJMC) Toolkit [17] takes a machine description specification as input and generates a disassembler for that machine. The NJMC Toolkit has been built upon by the University of Queensland Binary Translator (UQBT [7, 8]) which, although developed for binary translation, performs decompilation from source binary to a low-level C form as an intermediate step in
binary translation. The tool then uses a C compiler as a macro assembler back end to generate executables on a particular target machine. This open source tool has been applied to a broad range of different source/target platform pairs. UQBT is able to recover data type information, local variable scope and temporary variable stack usage, procedure abstractions (even recognizing user-defined function calls when the binary is stripped of symbol table and relocation information), structured control flow and data flow information, call graphs, and standard library calls when dynamically linked. UQBT requires knowing which hardware architecture the source binary was compiled to and it must be one of a variety that UQBT supports. It does not require information about the compiler used to create the binary. UQBT is the most advanced of existing decompilation tools for C. There is ongoing work to extend it to handle object-oriented code, primarily focusing on the difficult issue of recovering virtual method calls [25], using a hybrid static and dynamic analysis approach.

The most sophisticated disassembly tool available commercially is DataRescue’s IDA Pro [13]. It is an interactive disassembler, capable of recovering and propagating data type information and object references, determining function boundaries and calling relationships, recognizing stack variables and standard library functions, and deriving control flow graphs. It has limited cross-referencing abilities (e.g., identifying which function or methods use a local variable), but no general data flow analysis capabilities. It does not recover class hierarchy information. IDA Pro has a plug-in architecture and is programmable for advanced users. It supports virtually all processors on the market. However, it works best if processor type and compiler information is provided; otherwise the user may guide the disassembly interactively and define processor specifications using the IDA Pro SDK, provided that the user has a good understanding of the assembly language and addressing modes of the processor.

2.2. Java Disassembly and Decompilation Survey

The compilation of Java source code into bytecode allows for the creation of highly sophisticated reverse engineering tools. The first reverse engineering tools were disassemblers, the predecessors to decompilers. Early disassemblers included javap and dis. The javap command can produce the bytecode listing for the Java class file. Dis produces a listing of the machine code for C++ object files. Versions of dis are also available for Java class files.

Mocha was the first Java decompiler, written by Hanpeter van Vliet and released shortly before his death in 1996. Mocha is mostly a pattern matching decompiler that works when class files are compiled specifically with the Sun compiler. Due to its extensive use of pattern matching, Mocha is not useful with class files produced on other machines. Other pattern-matching, or first-generation, decompilers include SourceTec, WingDis, DeJaVu, and Krakatoa.

Mocha generated significant interest in the area of Java reverse engineering. The author’s untimely death led several companies to fill the void in the marketplace. The first commercial Java decompilers, WingDis and SourceAgain [15], were released in 1997. SourceAgain was the first of the second-generation Java decompilers. The second-generation products, SourceAgain, JAD, and JODE, produce source code via more advanced techniques that do not make extensive use of pattern matching. As a result, these decompilers operate on binaries from any compiler with any optimization setting. In the specific case of SourceAgain, source code can be produced even from hand-assembled class files (in which no source existed in the first place).

The advent of both freeware and commercial class file obfuscation tools led to the creation of third-generation decompilers, better known as de-obfuscators. These de-obfuscators not only decompile source code, but also recognize known obfuscations and invert them in the best way possible. SourceAgain and JAD are examples of products that implement de-obfuscation.

Java decompilers are best described in terms of basic functionality. There are three classes of Java decompilers

- **Disassemblers** – produce an human readable bytecode assembly of a classfile
- **Decompilers** – produce Java source code from a classfile
- **De-obfuscators** – advanced decompilers that undo intentional code obfuscation

The early success of the first Java decompilers caused serious concern among Java developers that source code was being revealed whenever class files were distributed. This led to the creation of several commercial and research Java obfuscation tools. The first obfuscator, Crema, was in fact, written by the author of the first decompiler.

The first-generation obfuscators simply renamed variable and type symbols to less revealing names. Therefore, instead of having rich class hierarchies and variable names such as “window” and “slider” a class file contains the symbols “a” and “b” or other non-informative names. While information is lost in the process of renaming, it does not prevent decompilers from producing reusable source code from these “name-mangled” class files.

In the early days of Java obfuscation it was believed that small ad-hoc modifications to the bytecode would fundamentally break a decompiler. Early obfuscators such as HashJava added a spurious bytecode sequence to every method and this caused Mocha to code dump. Since these ad-hoc modifications are easy to detect and undo, little
Reverse engineering is not limited to decompilers and obfuscators. Many binary re-engineering tools (or instrumentation libraries) have been created to facilitate the implementation of advanced code coverage and cross-reference tools. Many of these tools are research and non-commercial in nature.

3. Experimentation Framework

We developed BinREEF as a way to systematically and empirically assess the capabilities of available binary reverse engineering tools. It focuses on providing a test suite of programs with multiple versions of each program generated by applying varying degrees of software protection (i.e., combinations of stripped, license protected, optimized, or no obfuscation).

The primary objective of the experimentation framework is to determine whether a set of increasingly difficult reverse engineering tasks is possible to accomplish on programs of varying degrees of protection, using a given tool under evaluation. An interesting and important experiment result would be if a task cannot be feasibly accomplished due to some underlying intractability in the task. If a task can be accomplished, some characterization of the level of effort required will be useful in determining the strength and effectiveness of different techniques for preparing and guarding the executable.

BinREEF contains these elements [19]:

1. Experiment objectives - the problem addressed by the experiment
2. Response variable(s) - outcomes determined during experiments
3. Control factors - parameters identified for changes between the experiment runs to determine their effects in the outcomes
4. Control factor levels - specified settings for the control factors
5. Uncontrollable factors - parameters that can affect outcomes, and should be recorded
6. Controllable factors not included - identified parameters and their constant levels to be recorded
7. Experiment procedure – including test sequence, changes to control factors and data collection
8. Data analysis plans – including specific analyses and hypothesis tests to address the experiment objective and assumptions of analytical techniques

Proper treatment of the experiment design ensures that the objectives are addressed through well-designed analysis of the results based on gathering sufficient information.

The initial set of experiments is designed to assess feasibility and associated level of effort with specific reverse engineering tasks. The experiments are not designed to measure variability in the results due to different test subjects or levels of experience.
The top level response variables are:

- Whether a specific reverse engineering task was feasible, i.e., could it be completed without any additional guidance (a boolean observation).
- How much additional guidance was required to enable completion of the task – which of a set of hints were necessary to allow completing the task.
- How much time (effort) is required to complete a task.
- What kind of activities and techniques were used to complete the task.

The set of experiments were executed by a limited community of participants. An attempt was made to equalize the knowledge these participants have with regard to performing reverse engineering activities by providing some background material for them to study, by providing practice examples for them to train on, and by providing access to mentors who can assist them during the practice examples. The following factors (controlled and uncontrolled) were important to the experiment observations:

- Formulation of source code examples and executables were designed to test different levels of complexity of source code design and obfuscation. Experiments using these examples are repeatable, therefore making this factor controlled.
- Questions and hints were formulated to define or redefine a task. There was no reason a priori to believe that the tasks presented to the experimenters are intractable. The tasks may be difficult in terms of the amount of time and intellectual persistence they take. In order to obtain data regarding effort required to perform a task, a set of hints or task reformulations were developed. These hints are used when a subject decides that they cannot complete the task as posed. The amount of time spent between hints and the number of hints are collected. This factor is controlled.
- Selection of tools used in accomplishing tasks will not vary between experimenters, therefore making this factor controlled. One artifact from the experiments will be a determination of which tools are more effective.
- Personal skill at using reverse engineering tools and performing decompilation tasks is one of the uncontrolled parameters in the experiments. An attempt to reduce learning curve issues is made by providing training and consultation to the subjects.
- General intelligence and capability of subjects at cracking software executables is also an uncontrolled parameter in the experiments. Given the limited number of test subjects and the relative obscurity of the skills necessary to perform these tasks at an expert level we will not control for this factor.

3.1. Experimentation Procedure

Each subject conducted the following set of progressively more complex trials. Separate groups performed experiments using compiled Java programs and Java decompilation tools and compiled C/C++ programs and C/C++ decompilation tools.

3.1.1. Simple examples. To permit the subjects to get familiar with the tools’ environments and experimentation procedures some initial experiments were run. This gave the subjects an opportunity to train and familiarize themselves with the tools and tasks. The simple examples included the following examples:

- Trivial code – a simple compiled program to verify proper configuration of the tools environment. The decompilation task is to identify the call tree of the program.
- Simple code – a sort routine with a key parameter embedded in the sort. The decompilation task is to identify the value of the key parameter.
- Simple code – an I/O routine that makes use of 3rd party libraries. The decompilation task is to identify which 3rd party API routines are used and where.
- Algorithmic code – a complex computation routine with parameters (e.g., FFT algorithm). The decompilation task is to characterize the algorithm.

3.1.2. Measured isolated examples. The measured isolated examples tested specific reverse engineering techniques after subjects had achieved competence in the use of the tools and techniques. The complexity of the generated executables is increased by including 3rd party libraries (e.g., licensing check points), disabling features in the executable via internal logic, and utilizing a variety of obfuscation techniques, including name mangling and code control flow obfuscation.

The types of tasks executed in the measured isolated examples included:

- Finding the FlexLM licensing implementation points, thus supporting the theoretical goal of disabling the licensing scheme
- Finding code that disables software features, thus supporting the theoretical goal of turning those features back on
- Locating critical parameters and their values in the code
- Locating critical data structures in the code
- Locating critical algorithms in the code.

3.2. Data Collection

Test subjects were provided with the tools and the software to decompile. Each subject kept a journal of activity entries accurate to the second. The precise logging allowed the subjects to perform the experiments
in non-scheduled time, with the post analysis of the logs to reconstruct the total time spend on specific activities. This procedure is used successfully in Personal Software Process [28] activities. The data was collected by running a simple time-tally tool that was written to support the experiments. The time tally tool is written in Java and allows the experimenters to select the experiment that they are working on, the state or task which they are performing (i.e. executing experiment, reading docs, etc.), and enter a comment. Tally keeps track of the time spent on each experiment and task. Post-analysis of the log files can be performed for each experiment or for a selection of experiments.

The time journals were post-processed for aggregate and specific statistics including:

- Time spent in different categories of work
- Types of reverse engineering techniques that are being used
- Time spent from start to finish
- Special difficulties encountered
- Number of hints received.

The subjects operated using a think-aloud evaluation protocol. This protocol is often used to record users’ interactions with a system. The user is encouraged to speak while performing the tasks and his or her partner asks questions to elicit more information and record the process. The pair approach facilitated collection of the activities and thought processes during execution of the decompilation tasks. While the dominant conversation flow was from the tester to the recorder, the tester was permitted to receive input from the recorder. This is not a problem because a) information sharing is a plausible occurrence in a real-life reverse engineering situation and b) this activity does not control the amount of domain knowledge of the user (i.e. the combined insights of two people does not bias any measured variables).

4. Experiment Results

To compare binary reverse engineering tools for C++ and Java, focusing on language-specific differences in capabilities, we selected the SourceAgain Java decompiler and the IDA Pro interactive disassembler, which is the only tool currently capable of handling C++ code.

The following describes our experiences in applying these tools to the simple examples and two measured isolated examples within our test suite.

- Trivial code – the task of identifying the call tree of this program was readily performed using IDA Pro’s call graph generation facilities. In Java, the SourceAgain program easily regenerated the source code from the Java class files.
- Simple code – the algorithm used by this sort routine (quicksort) was inferred from the function names appearing in the disassembled code, since the code used dynamically linked built-in functions. The key parameter (the pivot element) was identified as being in an IDA Pro-generated variable name (var_20). In Java, the SourceAgain program had a problem decompiling the sort program, giving only partial code regeneration.

- io – the goal of identifying which third party API routines are used and where was assisted by IDA Pro’s call graph generation. Disassembling with IDA Pro revealed that image processing routines are being used from the tiff image manipulation library. A list of the routines was created by generating a call graph and filtering out those with the “tiff” prefix. While IDA Pro generates the call graph, it does not allow access to the internal data structure nor does it provide ways of automatically walking the graph to facilitate the search for routines with certain properties. So using IDA Pro for this task was tedious and time consuming. However, such graph walking/searching tools would not be difficult to write to speed up the process. In Java, the experimenters easily identified the Java packages used in this program, including the TIFFCodec. Experimenters used SourceAgain to quickly regenerate the source code and determine the functionality of the test: 1) load the tiff image from file, 2) convert the image to grayscale, and 3) save the image as a bitmap file.

- Algorithmic code – this code was found to contain an algorithm for computing the root mean square of input vectors of double-precision floating point numbers. This was determined by doing a data and control flow analysis of the routine. IDA Pro generated the control flow graph, but the data flow analysis was performed manually. IDA Pro does not provide much support for dataflow analysis; its capabilities are limited to string searching for variable names and some cross-referencing navigation showing where global variables are used. In Java, SourceAgain nicely regenerated the source code for this example. The experimenters found that the algorithm is a Daubechies D4 wavelet transform (D4 denotes four coefficients).

- Exp8 and 9 – Results from experiments 8 and 9 are presented in the graphs shown in Figures 1, 2, and 3. Experiment 9 was a stripped binary, while experiment 8 was not. They are executables from two different source programs of similar size, complexity, and application, but different algorithms.
These experiments had the goal of finding the licensing implementation points, which was achieved by identifying the branch instruction that controls whether the main licensed computational routine is performed or the exit code is executed. This is easily determined by finding branches to the error-producing exit code or by finding calls to the license checking function. Two ways of disabling the licensing scheme were performed: 1) in exp9, at run-time, GDB was used to set a breakpoint at the license check branch and alter its outcome, and 2) in exp8, the binary was modified to convert the conditional branch instruction to an unconditional branch that always jumps to the computational routine; this was done by changing the opcode of the branch instruction using the Hackman hex editor. In Java, the experimenters were able to make the executable run by short-circuiting the licensing options in the source code. They commented out the imports of the licensing classes and methods.

The actual absolute time taken to circumvent the license had a greater disparity. In general, it took longer to actually disable the license in the C++ binaries than in the Java binaries. For Java binaries, a decompiler can recover source code in which the licensing classes and methods can be easily commented out. For C++, the license-disabling changes must be performed at the assembly language level by finding a specific branch instruction whose outcome must be altered and then determining how to correctly affect the change (e.g., which way should the branch transfer control? which binary (or hexadecimal) strings to change and what to change them to?).

A surprising result is in the differences in absolute time taken for Exp 8 and 9 on the C++ binaries: it took approximately five times longer to find and disable the licensing checkpoints in the unstripped binary as it did in the stripped binary. This difference is due in part to the greater difficulty of statically modifying the binary to permanently disable the license (as was done in the unstripped binary) as opposed to dynamically disabling the license each time the code is run (as was done for the stripped binary). However, an examination of the activity logs for these experiments reveals additional reasons for the time disparity. Because the licensing checkpoints correspond to calls to third-party routines, their names are
not stripped out, so they tend to stick out like a sore thumb in the stripped binary. Additionally, the unstripped binary is richer than the stripped binary in details that can be examined and analyzed. In other words, the stripped version does not have as many details that can distract from the task of finding and circumventing the license.

5. Conclusions

This paper compared the state of the art of Java and C/C++ tools, based both on a literature survey and on empirical assessment of a select set of popular tools. To perform the empirical assessment, we developed an experimentation framework that includes a test suite of binary programs, organized by varying degree of software protection mechanisms, a set of representative, focused reverse engineering tasks, and a methodology. Results obtained so far in applying the experimentation procedure to a portion of the test suite were presented. We plan to continue experimentation with the rest of the test suite and assess additional tools in the future.

In general, the Java SourceAgain decompiler worked very well on classes that were not obfuscated. Java is based on a standard and fixed byte code layout that separates program from data. This supports the verification done by the Java byte code checker. The separation of program and data makes Java less susceptible to viruses as it is difficult to add rogue instructions to the program stream, making for relatively secure host-based programs, i.e., relatively secure host-based execution. This uniform design and specified bytecode format makes reverse engineering tasks for Java comparatively easier. This facility has both advantages and disadvantages depending on the purposes for which reverse engineering is being performed.

In contrast, C/C++ executables have a runtime layout in which program and data are mixed. Decompiling C/C++ programs is more difficult because program and data areas are not clearly separated. Data areas can be made to look like program areas to throw off analysis. Data areas can be jumped into thus permitting execution from these areas. This also makes C/C++ more susceptible to virus attacks. The standard buffer overflow attacks allow attackers to insert program instructions of their design into a program data area and jump to execution of these rogue instructions. In this sense, these programs are less host secure than the corresponding Java programs. This runtime insecurity leads to programs that are harder to analyze and, therefore can, in theory, be made less amenable to static analysis-based reverse engineering tasks. The difficulty in analysis by static means will make dynamic analysis increasingly important when performing these tasks.

Acknowledgements

This work is supported under the Air Force Research Laboratory’s Advanced Technologies for the Software Protection Initiative (ATSPI) program under contract #FA8650-02-C1298. We have benefited greatly from the technical guidance of Dr. Martin R. Stitz, Senior Research Scientist and Engineer, AFRL. We are also grateful for the contributions of Lewis Baumstark and Hongkyu Kim at Georgia Tech, ALPHATECH employees Mark Keaton, Sean Griffin, and Robert Flynn, and ALPHATECH summer interns Ryan Twomey and Costa Walcott. We would also like to acknowledge the work of Howard Reubenstein, formerly of ALPHATECH, in the early phases of this project.

References


