Computer Security Analysis through Decompilation and High-Level Debugging*

Cristina Cifuentes
Sun Microsystems Labs
901 San Antonio Rd
Palo Alto CA 94303
USA
cristina.cifuentes@sun.com

Trent Waddington
University of Queensland
Dept Comp Sci and Elec Eng
Brisbane QLD 4072
Australia
trent@csee.uq.edu.au

Mike Van Emmerik
University of Queensland
Dept Comp Sci and Elec Eng
Brisbane QLD 4072
Australia
emmerik@csee.uq.edu.au

Abstract

The extensive use of computers and networks worldwide has raised the awareness of the need for tools and techniques to aid in computer security analysis of binary code, such as the understanding of viruses, trojans, worms, backdoors and general security flaws, in order to provide immediate solutions with or without the aid of software vendors.

This paper is a proposal for a high-level debugging tool to be used by computer security experts, which will reduce the amount of time needed in order to solve security-related problems in executable programs. The current state of the art involves the tracing of thousands of lines of assembly code using a standard debugger.

A high-level debugger would be capable of displaying a high-level representation of an executable program in the C language, hence reducing the number of lines that need to be inspected by an order of magnitude (i.e. hundreds instead of thousands of lines). Effectively, these techniques will help in reducing the amount of time needed to trace a security flaw in an executable program, as well as reducing the costs of acquiring or training skilled assembler engineers.

1. Motivation

Computer viruses are instructions in a program that use the facilities of an operating environment (such as an operating system or a word processor) to propagate themselves into files on a computer system, and often perform some malicious action (for example, the CIH/Chernobyl virus [4]). This type of virus has caused thousands of dollars of lost revenue due to infection of many large organisations, including Intel, AT&T, Compaq and Boeing, and tens of thousands of computers worldwide. The extent of disruption has been so serious that even the FBI has been involved in investigating who is responsible for such malicious viruses.

A computer worm is a program that uses network facilities, such as email, to propagate itself from computer to computer. The recent appearance of the Melissa virus and other Visual Basic scripting viruses, which both infect files and propagate themselves via email, has blurred the distinction between virus and worm.

Microsoft's Internet Information Server (IIS) is the most common Web server used under WindowsNT. It is estimated that IIS runs on 22.3% of the web servers on the Internet. In early June 1999, a security flaw in IIS was found, and an *exploit* made available, which allowed people with modest programming skills to gain complete control over a web server running IIS [12]. Clearly, this type of security vulnerability on a mission-critical application used by millions of businesses every day can jeopardize corporate data that resides on servers and cost thousands of dollars to such companies. In comparison, Apache runs on an estimated 60% of web servers on the Internet and is protected from such wide spread attacks by a proactive security community, due primarily to its publicly available source code.

Computer viruses and web server flaws are just two examples of numerous computer security related vulnerabilities, commonly known as "malware." Malware is a generic term that describes the malicious side effects caused by computer programs, whether caused by intentionally introduced malicious code or by the malicious exploitation of benign code to the detriment of the user [13]. Worldwide, computer network security teams have been created in order to provide for timely information and solutions related to malware. Examples include AusCERT, Australia's CERT (computer emergency response team) team hosted at The University of Queensland, and CERT, its US counterpart, hosted at Carnegie Mellon University. The current technique used to study malware is the use of a debugger to step

* This research was sponsored by the Australian Research Council under grant No. 00/ARCS252, and was first conceived while the first author was at the University of Queensland.

the executable program one assembly line at a time until the problem is found—it is then possible to reconstruct that part of the traced program in order to provide a solution for it. This method requires an expert engineer that understands assembly code—a skill that is disappearing as years go by, due to the increasing use of higher-level languages such as C++ and Java.

This paper is our proposal for the development of techniques and tools to support debugging at a higher level of abstraction. Rather than the security expert having to trace thousands of lines of assembly code, an order of magnitude less would be provided (i.e. hundreds of lines of high-level code) through the proposed environment. By reducing the amount of code that the engineer has to process, and by presenting the engineer with a higher level of abstraction, fewer man-hours will be needed in order to understand the program's code. Understanding the code that is part of the virus or that provides access to a backdoor in a program allows the engineer to develop a solution to the problem, with or without the aid of software vendors. Clearly, the techniques proposed in this project will provide computer security organisations with the ability to provide solutions to Internet security vulnerabilities in a more timely manner, hence reducing the amount of losses to companies that use Internet mission-critical applications in one way or another (for example, web servers and email). Further, these techniques will allow engineers who are not necessarily experts in assembly code, to aid in the understanding of code, hence reducing the additional skills and training required for professionals working in network security teams.

2. State of the Art

Security vulnerabilities in programs (e.g. backdoors or viruses) are commonly found in one of two ways: it is reported by users of programs or by network security teams actually testing programs for security flaws. In either case, network security teams worldwide use the current state-of-the-art to determine the code of the executable program that causes the flaw, and then find a way to patch the program to provide a solution to it. Current tools involve the use of NuMega's SoftIce, a Pentium/Windows advanced debugger, gdb, a Unix debugger, or Data Rescue's IDAPro disassembler [19].

Debugging tools were originally created to trace the paths that a program follows at execution time, in order to find where a program "crashes" due to an error in the program. Debuggers are normally written for particular high-level languages, such as C++, C and Java, and require a software developer to compile the program with a debugging option, which stores extra debugging information into the executable program. In this way, when the program runs, it can be debugged by associating lines of assembly code to their high-level counterpart lines (for each line of high-level code, there are n lines of assembly code), given that both, the source code and the executable code version are available to the developer. Debugging information makes executable files larger, hence, this information is generally not included in the final release version of a program. Tools like *SoftIce and* gdb assume that this type of information exists in the file to provide valuable debugging information to a user. In the event that the information is not there, these tools can only allow a user to follow paths of a program by looking at the assembly code in the program, without any knowledge of its equivalent high-level code; this "raw" assembler view is the one commonly used in the debugging of malware code, as such programs are developed by teams external to the debugging team.

```
pushl $.LC0
call getenv
addl $4,%esp
movl %eax,%eax
pushl %eax
leal -512(%ebp),%eax
pushl %eax
call strcpy
addl $8,%esp
```

Figure 1. Example Pentium Assembly Code for Security-related Problem

Figure 1 gives an example of a simple piece of assembly code that is commonly found in security-related problems. In this code, the program calls the get environment function (`getenv`) with the string `TERM` (for terminal; located at label `.LC0`). The program then copies the string in the `TERM` variable to a local string array variable on the stack, of size 512 bytes. This type of code is commonly used to overflow an array as the `TERM` variable can be set to be any string, including strings longer than 512 bytes. Once the program performs a buffer overflow, the program's stack is overwritten by the excess bytes in the `TERM` string, and such bytes can include instructions that the computer is coerced to execute—such instructions may constitute a security issue.

2.1. Life Cycle of a Security Flaw

We describe a typical life cycle of a security flaw to illustrate the process used today and why a better/faster solution is needed.

- A security team sets out to find a flaw in some software product.
- The team finds possible security flaws and posts their suspicions to a security mailing list like `bugtraq`,

calling for the vendor to fix the possible bug.

- The vendor either fixes the bug, does not even see the report or refuses to fix the bug because it is "not an immediate threat".
- The security team (or a different one), writes an *exploit* that demonstrates that the bug is indeed a security flaw and should be fixed by the vendor.
- The security team contacts the vendor and demonstrates that the flaw is serious.
- At this point, the vendor may decide to collaborate:
  - The vendor makes binary patches to their product, essentially by changing the source code, recompiling it, and providing the new executable file or the single library that the flaw is contained in.
  - The security team posts the patches and gets the credit for fixing the bug.
  - CERT, AusCERT and similar companies pick up the patches, ignore the exploit, and create an *advisory* which tells normal users what to do in order to install the patch and why it is needed.

  Alternatively, the vendor does not collaborate:
  - The software vendor does not believe the security team, does not think it is profitable to fix the bug, or otherwise does not care about the bug.
  - The security team alerts others about the problem by releasing the exploit on their web site (taking the moral high ground of "full disclosure") and stating that the software vendor does not care to fix the bug.
  - The software vendor gets embarrased and releases a patch to fix the bug, or gets stubborn and argues about whether the flaw is "theoretical" or not.

The time between the first posting on a security mailing list like `bugtraq` and the CERT advisory to the general community is very long; it can take months before a solution is widely advertised. In the mean time, hackers, who commonly read security mailing lists, take advantage of these bugs by writing their own exploits.

Instead, we propose an alternative to this life cycle, by providing security teams with better debugging tools, once a bug has been identified in a system, the team could provide a patch to it and go to the vendor with that patch, so that the vendor either releases its own patch, reuses the one provided by the security team, or decides to do nothing. In all cases, the security team can release a patch in a *timely* manner for the general community to use straight away.

3. A High-Level Debugger of Executable Code

The proposed high-level debugger would operate with existing executable files (that do not have debugging information stored in them), without assuming any access to the source code for that program. The debugger would allow users to debug executables for a variety of CISC and RISC machines, and would produce high-level code in the C language as output, as well as providing links to the associated assembly representation for that C code. The quality of the C code produced would be comparable to that written by a structured software developer, rather than assembly code written in C (i.e. an order of magnitude less lines of code). The C language was chosen because it is commonly used in the security area and many security-related flaws are originally written in C. Recovery from hand-written assembly code may not always be expressable in high-level code, in which case, inline assembly code can be used.

In more detail, the aims of this project are:

- To design high-level debugging techniques by dynamic decompilation of machine code into assembly code and then to C code. Dynamic techniques work online, at runtime, while the user debugs (i.e. examines) the program.
- To develop a graphical user interface that can display the recovered C code and associate it with the underlying assembly code.
- To support a variety of CISC and RISC machines in a machine-independent way.

The following sections describe how these aims can be achieved.

3.1. Dynamic Decompilation of Executable Code

The feasibility of static decompilation techniques, to recover some high-level form of a program, has been shown in the literature; examples include, Cifuentes' dcc decompiler [9, 5, 6], SourceAgain [1], the 8086 C decompiler [14, 15], O'Gorman's work [21] and more. These techniques have been developed over a period of over 30 years, with Halstead [16, 17, 18] making the original contributions to this area, in the context of the NELIAC language, as well as Caudle [3].

The techniques described by Cifuentes were implemented in the dcc decompiler, a prototype static 80286 decompiler of MS-DOS executable files. The quality of the recovered code approached that written by a structured software developer, in terms of the control flow abstractions used in the program and the instructions recovered, but did not include high-level types or meaningful variable names

(as these are not stored in the executable). These techniques were the basis for the SourceAgain Java decompiler. In the case of Java, typing information is contained in the "executable" class file, which gets interpreted or compiled at runtime. However, in the case of machine code executable programs, type information is *not* stored in the file, therefore the need for type recovery. Mycroft's work [20] on type recovery analysis based on RTL (register transfer) representations can provide code that more closely resembles typed high-level source code.

Nevertheless, one limitation exists with static techniques, which is inherent to the nature of current von Neumann machines. In von Neumann machines, code and data are represented in the same way, hence making it impossible to completely distinguish code from data *statically*. This means that a complete decompilation of a program may not always be possible.

The limitations of static decompilation are overcomed in a dynamic environment, where paths in a program are followed and decompiled "on the fly", as the program is debugged. When the program reaches an indirect transfer of control, only one value will be possible at that point in the program, and it will be known at runtime (unlike during static translation). Although this technique may sound like a natural extension of debugging techniques, an extensive search on tools used in this area reveals that this technique has not been used in debugging because debuggers are normally written for checking/debugging while you write a program (i.e. when you have the source code), rather than after the program has been shipped and the source code is unavailable.

3.2. High-Level Debugger Interface

Figure 2 shows the graphical view of the proposed high-level debugger. On the left-hand side, the assembly code is displayed, and on the right-hand side the equivalent C code is displayed. Note that the 14 assembly instructions on the left of the first window are equivalent to the one high-level C instruction on the right. As can be seen, parameters to both procedures have been recovered and placed in the actual parameter list. Further, since there is a local variable of 512 bytes used in the call to `strcpy`, that variable is given a name (for example, `loc1`) as the name is not stored in the executable file being debugged. The user of this tool can then change names of local variables as needed (e.g. change `loc1` for `buf` as in this example).

The development of dynamic decompilation techniques needs to trade off quality of recovered code for performance, as the dynamic debugger would recover high-level code as the paths in the program are traced and the user is waiting to see the results. This implies that efficient techniques to perform data flow, control flow and type analyses are needed, without being able to spend too much time on performing such analyses. Therefore, to lift the level of representation, incremental analyses can be performed so that code that is in paths constantly executed will be "optimised", that is, retranslated using more expensive analyses, so that better quality code can be created. This incremental improvement of the code is possible because more information is known about a program once more paths of the program have been executed. Further, idle time in the program can be used by a background analyser to improve the quality of the decompiled code.

In Figure 2, two techniques were used to recover the one high-level C statement—data flow analysis and parameter analysis. Existing data flow techniques can be easily adapted to dynamic translation. The recovery of parameters is more involved and will require more context information in some cases, hence requiring optimisation. In the example, the recovery of parameters was straightforward. The example does not show code that requires control flow analysis techniques to be used. These well developed techniques recover loops and conditional transfers of control, hence they require more context information, which can be achieved by retranslating pieces of code once more paths of the program have been executed; i.e. performing optimisations on the translated code.

3.3. Retargetability

A system is said to be retargetable if it can easily and quickly support a new *target* machine, whether it is for code generation purposes (e.g. in a compiler or an optimizer) or for decoding of an executable file (e.g. in a binary translator's frontend).

A key technique in a retargetable system is the representation of the semantics of the machine's instructions for analysis purposes. In many cases, a register transfer notation is used, such as that used by the optimizer VPO [2], GNU's gcc suite [23] and the binary translation framework UQBT [8, 7].

Our current work on retargetable binary translation [8, 7] has shown that the machine-dependent aspects of CISC and RISC machines can be specified in a variety of languages so that an intermediate representation of such instructions in the form of register transfers (RTLs) can be created for the purpose of analysis. These techniques have shown that the RTL representation is suitable for analysis of RISC and CISC machine code, and hence would be appropriate for decompilation analyses. In the case of binary translation, the recovery of machine code is performed to a pseudo high-level representation called HRTL, which is then brought down to machine code level for a target machine. The UQBT framework is illustrated in Figure 3.

For decompilation, only the first half of the system is

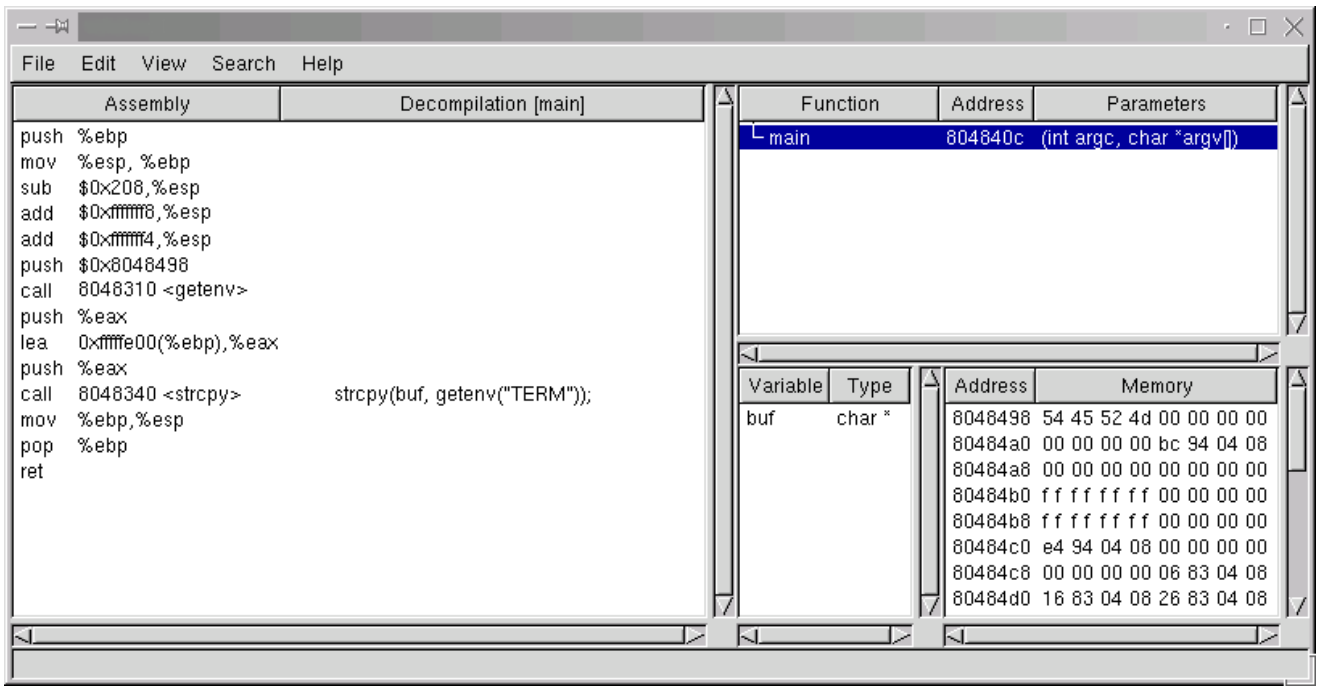

Figure 2. Graphical View of a High-Level Debugger

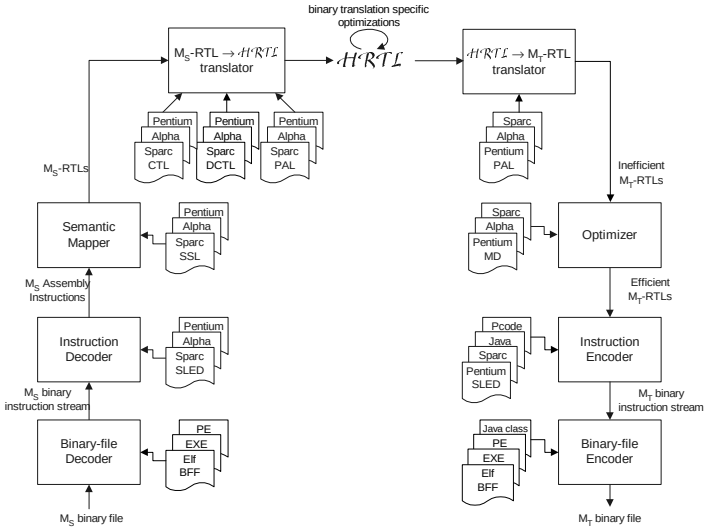

Figure 3. The UQBT Binary Translation Framework

needed; namely, the part that transforms the source/input binary file to the HRTL representation. A different backend then needs to be coupled to produce a higher level representation that performs high-level control flow recovery and performs type analysis.

Retargetability is supported in the UQBT framework by means of specification languages, APIs and pluggable modules; these are seen in Figure 3 and are as follows:

- APIs: UQBT provides the binary-file format and the control transfer APIs. In the case of binary (executable) files, its internal representation varies from one format to another (e.g. Elf, PE, PRC), but all representations have a way of extracting the code and data for the executable program—this type of information is the one obtained through the API. For control transfers, the API requires the developer to identify conditional branches from unconditional branches, as well as calls and returns. This is because all these instructions look like jumps but have slightly different semantics.

- Specification languages: UQBT uses 3 specification languages to describe different concepts of a machine or conventions used by the OS:

  – SLED (Specification Language for Encoding and Decoding): This language is part of the New Jersey Machine Code toolkit [22] and allows users to specify the *syntax* of machine instructions.

  – SSL (Semantic Specification Language): This UQBT language allows users to specify the meaning of assembly instructions by means of register transfers [10].

  – PAL (Procedural Abstraction Language): This UQBT language allows users to specify the ABI

conventions used by the OS for calling procedures, passing parameters and returning values, as well as locations in the stack frame where local variables and other information is placed [11].

- Machine-specific analyses: these are extra analyses that may need to be introduced in the translation in order to completely transform M_s-RTL code into HRTL, in such a way that machine-dependencies are successfully removed. Examples of such analyses include, removal of delayed transfers of control on SPARC, and the transformation of stack-based floating point code into register-based code on Pentium.

4. Summary

In this paper we have described the current state of the art to uncover security flaws in programs, and we have proposed an alternative *high-level debugger* to aid in the understanding of security flaws in programs.

The proposed high-level debugger incorporates decompilation techniques in order to more readily provide to the security expert fewer number of lines of code to be traced and understood.

The techniques described in this paper work in the context of no source code or debugging information being available, which is normally the case when a security problem is identified in software.

As part of the implementation for this proposed high-level debugger, the frontend of the retargetable UQBT binary translation project can be reused, in effect only leaving the implementation of control and type analyses in order to achieve high-level code that resembles that written in a high-level language such a C.

References

[1] Ahpah Software, SourceAgain decompiler. `http://www.ahpah.com`, 1998.

[2] M. Benitez and J. Davidson. A portable global optimizer and linker. In *Proceedings of the Conference on Programming Languages, Design and Implementation*, pages 329–338. ACM Press, July 1988.

[3] B. Caudle. On the inverse of compiling. `http://www.csee.uq.edu.au/csm/decompilation/ic80.htm`, 1980.

[4] CERT advisory CA-99-04-melissa-macro-virus. `http://www.cert.org/advisories/CA-99-04-Melissa-Macro-Virus.html`, Mar. 1999.

[5] C. Cifuentes. Interprocedural dataflow decompilation. *Journal of Programming Languages*, 4(2):77–99, June 1996.

[6] C. Cifuentes. Structuring decompiled graphs. In T. Gyimóthy, editor, *Proceedings of the International Conference on Compiler Construction*, Lecture Notes in Computer Science 1060, pages 91–105, Linköping, Sweden, 24-26 April 1996. Springer Verlag.

[7] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, Mar. 2000.

[8] C. Cifuentes, M. V. Emmerik, and N. Ramsey. The design of a resourceable and retargetable binary translator. In *Proceedings of the Working Conference on Reverse Engineering*, pages 280–291, Atlanta, USA, 6-8 October 1999. IEEE CS Press.

[9] C. Cifuentes and K. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.

[10] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *Proceedings of the International Workshop on Program Comprehension*, pages 126–133, Ischia, Italy, 24–26 June 1998. IEEE CS Press.

[11] C. Cifuentes and D. Simon. Procedure abstraction recovery from binary code. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 55–64, Zurich, Switzerland, Mar. 2000. IEEE CS Press.

[12] eEye. eEye digital security. `http://www.eEye.com/`, June 1999.

[13] European institute for computer anti-virus research. `http://www.eicar.org/`, 1999.

[14] C. Fuan and L. Zongtian. C function recognition technique and its implementation in 8086 C decompiling system. *Mini-Micro Systems*, 12(11):33–40,47, 1991.

[15] C. Fuan, L. Zongtian, and L. Li. Design and implementation techniques of the 8086 C decompiling system. *Mini-Micro Systems*, 14(4):10–18,31, 1993.

[16] M. Halstead. *Machine-independent computer programming*, chapter 11, pages 143–150. Spartan Books, 1962.

[17] M. Halstead. Machine independence and third generation computers. In *Proceedings SJCC (Sprint Joint Computer Conference)*, pages 587–592, 1967.

[18] M. Halstead. Using the computer for program conversion. *Datamation*, pages 125–129, May 1970.

[19] Ida PRO disassembler, Data Rescue. `http://www.datarescue.com/ida.htm`, 1997.

[20] A. Mycroft. Type-based decompilation. In *Proceedings of the European Symposium on Programming (ESOP)*, volume 1576 of *LNCS*. Springer-Verlag, 1999.

[21] J. O'Gorman. *Systematic Decompilation*. PhD dissertation, University of Limerick, Department of Computer Science and Information Systems, 1991. Technical Report UL-CSIS-91-12.

[22] N. Ramsey and M. Fernández. Specifying representations of machine instructions. *ACM Transactions of Programming Languages and Systems*, 19(3):492–524, 1997.

[23] R. Stallman. *Using and Porting GNU CC*, version 2.5 edition, Oct. 1993.