String Analysis of x86 Binaries

Mihai Christodorescu
mihai@cs.wisc.edu

Wen-Han Goh
wen-han@cs.wisc.edu

Nicholas Kidd
kidd@cs.wisc.edu

May 12, 2005

This is a project report for Professor Ben Liblit's *Graduate Seminar on Software Quality: Tools and Techniques for Verification, Testing, and Debugging*, Computer Sciences Department course 703 section 1, Spring 2005, at University of Wisconsin, Madison.

Abstract

1 Introduction

The strings used and generated by a program during execution contain significant high-level information about the program and its communication with the runtime environment (other processes on the same machine, processes on remote hosts, etc.). For example, an analyst trying to understand undocumented functionality can use the possible strings output at a particular line in a program as hints. In other cases, if we have information about the strings used to communicate with remote systems (for example, SQL commands sent to a database management system or SMTP commands sent to an email server), we can check the validity of such output, ensuring the program integrates properly into larger, distributed systems. Thus, we found the need for a tool to automatically analyze and produce the set of strings possibly generated at program points of interest.

Such tools have already been developed for particular situations and have been used with great success as building blocks in some program verification problems. Christensen, Møller, and Schwartzbach incorporated a string analysis engine into a high-level language, JWIG [1], designed for Web service programming. The JWIG compiler verifies that programs written in JWIG generated valid XML and XHTML documents. Gould, Su, and Devanbu [5] applied string analysis to construct a sound, static analysis for verifying the type correctness of dynamically generated strings and created *JDBC Checker* [4], a tool implementing this analysis. Kirkegaard, Møller, and Schwartzbach [6] incorporated XML documents into Java as first-class data types, and enhanced the Java compiler to perform type checking on the XML documents generated by the program. Their type checking algorithm for XML data makes uses of string analysis to determine the possible values of XML-typed variables.

All these applications of string analysis have in common the existence of a high-level language (Java) with rich types that help the string analysis task. Even when analyzing compiled Java code (in the form of `.class` bytecode files), the large amount of information preserved from the original Java source code aids the static analysis. Unfortunately, there are many analysis scenarios where high-level information about the program is not present. Reverse engineering of binary code is one such scenario: a program in native, executable format retains little detail beyond the actual semantics of the original source code. Renovating, upgrading, or replacing legacy code is one application of reverse engineering, where neither source code nor documentation are available. Forensic analysis of malicious code is another application of reverse engineering: in such situations, source code is simply not available or cannot be trusted. Both of these areas can benefit from string analysis techniques that recover information necessary to understand the semantics of a program.

In this paper, we present a string analysis technique that recovers string values at points of interest in a binary program. We do not assume the presence of any source code, debugging information (e.g. symbol data), or other high-level artifacts, making this analysis suitable for both benign and malicious programs. We build our analysis on

top of existing work, specifically leveraging to Java String Analyzer (JSA) infrastructure introduced by Christensen, Møller, and Schwartzbach [2]. The major effort in our work is the recovery of sufficient information from the binary in order to identify the string manipulated by the program and the operations applied to them. Using this information, we construct a *string flow graph*, a data structure that the JSA infrastructure can analyze to produce possible string values.

We make the following contributions described in the remainder of this paper:

• **A static analysis for recovering strings in binary code.** We successfully bridge the gap between x86 machine code, a low-level language, and the JSA string flow graph, a high-level construct (described in Section 2). Towards this end we make novel use of several static analysis techniques to recover semantic information missing from the binary file. We present our static analysis techniques in Section 3 (for the intraprocedural case) and Section 4 (for the interprocedural case).

• **A practical implementation (*x86sa*) to analyze x86 binaries.** We develop a tool allowing the user to query for the possible string values at a point of interest in the program. The *x86sa* tool builds on top of the popular disassembler IDA Pro [3], making it easy to use.

• **An experimental evaluation showing that our approach is feasible.** We tested the *x86sa* tool on both benign and malicious programs, to evaluate the strengths and limitations of our static analysis. As we show in Section 5, our *x86sa* tool performs equally well on all test cases, within the analytically predicted accuracy limits.

2 Modifications to Java String Analyzer

The Java String Analyzer (JSA) [2] is a tool developed at the University of Aarhus for the puposes of analyzing string expressions in Java programs. In this section, we will briefly describe the overall architecture of the JSA and the modifications made to allow it to parse flowgraph files produced by x86sa during the analysis of x86 binaries.See Figure 1.

2.1 Overview of JSA

The JSA is built on top of prior work on parsing Java bytecode [8] and approximating context free grammars with regular expressions [7]. It takes Java bytecode as input and transforms this into a flowgraph. A flowgraph will contain Init nodes for each string constant that appears in the Java program, Concat nodes for each string concatenation, UnaryOp nodes for unary operations such as setCharAt, and BinaryOp nodes for binary operations such as replace. Of all string operations, concatenations are modeled most precisely by the tool because it is context free, while other operations are not. Once the flowgraph has been built, it is then transformed into a context free grammar via a simple transformation; for each node that is not an Init node, a new production is added where the left hand side is a freshly generated non-terminal and a terminal or non-terminal is added for each node that is a predecessor of the current node in the flowgraph. The Mohri-Nederhof algorithm [7] is then applied to the CFG to produce a strongly-regular CFG, i.e. a CFG that can be accurately modeled by regular expressions. For each hotspot in the CFG, a finite state automaton is generated. Hotspots are user-defined locations in the code where the user would like to know the set of possible string expressions that may reach that program point. Examples of hotspots include print statements, SQL queries, and system calls. The final output of JSA is for each hotspot, a FSA accepting a superset of the strings that reach the hotspot.

2.2 Flowgraph files

As previously mentioned, we modified the JSA to accept flowgraph files instead of Java bytecode. Each line in the file represents a node in the flowgraph and contains information such as type of node, node name, incoming edges and string constants. To illustrate our flowgraph specification language, we include several examples here.

```
init a "a string constant"
```

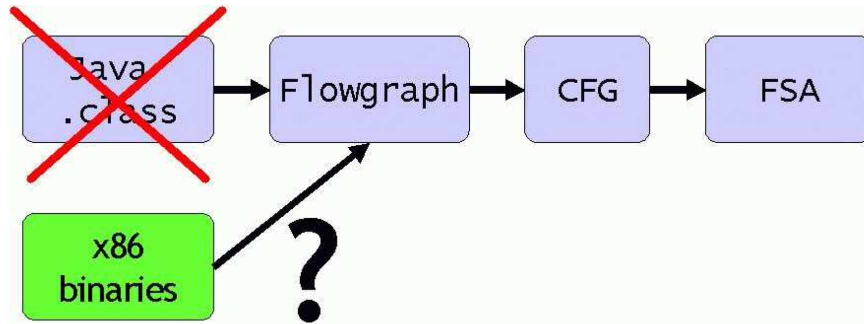

Figure 1: *Architecture of modified JSA* The blue boxes indicate the original architecture of JSA. We replaced the Java analyzer frontend with our x86 analyzer, as indicated by the question mark (Section 3). The x86 analyzer produces a flowgraph file which is later parsed by JSA.

This creates an Init node in the flowgraph with the name `a` and string constant `"a string constant"`. Init nodes do not have incoming edges and are turned into right hand side terminals during transformation.

```
assign d (a b c)
```

This creates an Assign node in the flowgraph with the name `d` and incoming edges from nodes `a`, `b` and `c`. Assign nodes in the flowgraph represent points where any one of the string expressions coming from the incoming edges may be taken. Assign nodes may have any number of incoming edges. However, a node with less than 2 incoming edges does not serve any purpose in the flowgraph.

```
concat i (a h)
```

This creates a Concat node in the flowgraph with name `i` and incoming edges from nodes `a` and `h`. Concat nodes must have exactly 2 incoming edges representing the string expressions being concatenated. Additionally, concat nodes and assign nodes can be qualified by "`hotspot`". JSA will produce a FSA for each hotspot. If no hotspots are defined, JSA will not produce any output.

3 Intraprocedural Analyses

In order to replace the JSA frontend with our x86sa frontend, two main differences between Java binaries and Intel x86 binaries need to be addressed. First is the lack of type information in the x86 binary and second is the differing semantics with respect to strings between the Java binary and the Intel x86 binary.

Java binaries retain much of the high level information present in a Java program. One such piece of information is the type of both objects and methods. In an Intel x86 executable, such information is not available. This poses an initial problem when one desires to perform a string analysis of an Intel x86 binary. To overcome this lack of information, x86sa performs two intraprocedural analyses, namely a string inference analysis (Section 3.1) and a stack height analysis (Section 3.2).

The Java programming language defines the String object as being immutable. This requires all string operations that modify a Java String object to create a new String object. Intel x86 programs do not adhere to such strict semantics (there isn't actually a notion of a String object). Assuming the Intel x86 binary uses c-style strings, many `libc` string functions directly modify the destination string's memory contents (e.g. `strcat`). To overcome this difference, x86sa performs an alias analyis to determine the set of strings that a `libc` string function may modify (Section 3.3). The alias analysis is used to create the *string flow graph* (SFG) which is then fed into the JSA backend (Section 3.4).

```
            mov  ecx, [ebp+4]
firstpush:  push ecx
            mov  ebx, [ebp+8]
secondpush: push ebx
            call _strcat
cleanup:    add  esp, 8
```

Figure 2: Intel x86 assembly code for a call to the libc strcat function. This is typical of the c code strcat(dest,src) where dest is pointed to by register ebx and src by register ecx.

3.1 String Inference

To locate possible c-style strings in an Intel x86 executable, x86sa performs an intraprocedural string inference analysis. The intuition behind the analysis is that the common `libc` string functions have a known interface and therefore a known type signature. This information is used to discover which registers and memory locations may reference c-style strings. In order for the analysis to be safe, x86sa assumes all strings in the program are manipulated through said `libc` string functions. Take for example the call to strcat in Figure 2. Using `strcat`'s method signature, the registers `ebx` and `ecx` reference c-style strings before the call. After `strcat` completes, the register `eax` references the same string as `ebx`. The memory contents referenced by `eax` and `ebx` contain the contatenation of the strings pointed to by `ebx` and `ecx` before the call to `strcat`. Using the `strcat` method signature as well as the rest of the `libc` string functions, a dataflow analysis for infering string variables can be performed.

The dataflow analysis performs a backwards traversal of the function's control flow graph[1] (CFG) propagating string (type) information in a gen-kill fashion. Each call to a `libc` string function generates the appropriate string information. The destination register for an x86 instruction will kill string information for that register. If the destination register may reference a string after the instruction, then the source register may reference a string before the instruction. This propagates string information through registers and memory locations. The string information associated with the CFG's enter node is the set of memory locations that are either string constants or function parameters.

A limitation of the analysis is the way in which memory is modeled. Currently, string information for memory locations is only tracked if the memory location is a constant offset from the frame pointer (register `ebp`). This is a common mechanism for accessing local variables of a higher level language that has been compiled to an Intel x86 executable. This limitation is addressed in the future work section.

3.2 Modeling the x86 stack

In order to generate string information for the `libc` string functions, the arguments to the function call must be known. However, the Intel x86 ISA does not explicitly associate arguments with their respective function call (see Figure 2). Function arguments can be pushed onto the stack or passed through registers. Because the analysis is intraprocedural, it assumes the binary follows the "`_cdecl` calling convention. This calling convention mandates that arguments are pushed on the stack and popped off the stack by the caller of the function. With this assumption, x86sa performs a stack height analysis that is used to associate push instructions with their function calls. The analysis is broken down into two parts.

First, a forwards dataflow analysis on the CFG is used to locate the "cleanup" instruction for a function call. A "cleanup" instruction is defined as the first instruction that increments the stack pointer after a function call. In Figure 2, the cleanup instruction is labeled `cleanup`. Standard compilers adhering to the "`_cdecl`" calling convention typically use an `add esp,C` instruction where `C` is a constant. Dividing `C` by the wordsize of the x86 architecture (4 bytes) provides the number of arguments passed to the function call.

Second, a forwards dataflow analysis on the CFG is used to model the x86 stack. This analysis associates with each CFG node a set of possible stack configurations that may reach that node. Each CFG node is assigned a transformer that models the CFG node's instruction's effect on the stack. For example, a `push` instruction will push a reference to the instruction (its effective address) onto its set of stacks that reach the instruction (a union of the set of stacks associated with the node's predecessors).

[1] Control flow graphs for each function in the x86 binary are created by the Connector [?], a plugin for the IDA Pro dissassembler used to analyze x86 binaries.

Using the number of arguments pushed at a function callsite, the stack model for the callsite CFG node is queried to get the instructions that setup its arguments. In Figure 2, the two push instructions (labeled `pushecx` and `pushebx`) are associated with the call to `strcat`. This information is used to create the gen-transformer for the `strcat`'s CFG node during string inference.

3.3 Alias Analysis

Due to the differing semantics between Java binaries and Intel x86 binaries, a string operation may effect the memory contents of multiple string pointers. To overcome this hurdle, an alias analysis determines the set of string variables that may be modified by a string operation. As mentioned earlier (Section 3.1), it is assumed that strings are only modified by `libc` string functions. Therefore it is only necessary to track the aliases between registers (and constant offsets from the frame pointer).

Because the analyis is being performed on an Intel x86 binary and the assumptions made, alias analysis can be described as a set of variable and "string creation point" pairs. A "string creation point" is a point in the program that creates or modifies a string. "String creation points" are associated with the CFG's enter node (constant string or function argument), `libc` memory allocation functions (e.g. `malloc`), and with calls to `libc` string function (e.g. `strcat`).

Alias analysis is a forwards dataflow analysis on the CFG. Each CFG node has a transformer assigned to it that reflects the nodes effects on the alias information. For example, the `mov` and `lea` instructions create aliases between registers (or a register and memory location). Each `libc` string function generates a new "string creation point" and its effects on the alias relation is modeled accordingly. **??** shows how the transformer for the `strcat` function operates. As shown, `strcat` generates a new "string creation point" and sets up aliases to that point for the registers `eax` and `ebx`. To increase the precision, the alias analysis uses may-must relations. The may and must sets are disjoint sets of variable and "string creation point" pairs. The sets must be disjoint as a variable (register or memory location) in the must set can have only one alias whereas variables in the may set can have multiple aliases. Join points in the CFG require the intersection of the *must* alias sets and the union of the *may* alias sets. Any alias information not contained in both must sets is migrated to the *may* alias set.

3.4 String Flow Graph

After alias analysis, the alias information associated with each "string creation point" is an encoding of the *string flow graph* (SFG) (Section 2.2). A forwards traversal of the CFG is used to translate the encoding into a SFG. At each "string creation point" (i.e. `malloc`, `strcat`, etc.) an appropiate SFG node is created (i.e. `init`, `concat`, etc.). If a source operand to a `libc` string function has multiple aliases (i.e. is in the may alias set), an `assign` SFG node is created first and the `assign` node is used as the predecessor to the "string creation point". For example, the `libc strcat` function correlates to an SFG `concat` node. The alias information of the operands represents the predecessors of the SFG node. If there are multiple aliases, then an `assign` node must first be created as a `concat` node can have only two predecessors. An example transformation is presented by the SFG shown in Figure 4. If the Intel x86 binary being analyzed contains only one function, then the created SFG can be fed directly into the JSA backend. Otherwise an interprocedural analysis must be performed.

4 Interprocedural Analysis

When the binary being analyzed contains multiple functions, the SFG for each function must be linked together before being fed into the JSA backend. The current implementation does not perform this linking but three techniques could be used.

First, the SFG of a called function can be inlined at the function callsite. This provides a precise analysis but cannot handle recursion. An alternative approach is to create a Super SFG. This is the string equivalent to a Control Flow Super Graph in the PL literature. A Super SFG has an `assign` node for each *formal* parameter of a function. The `assign` node's predecessors are the corresponding *actual* arguments at each callsite to that function. Finally, a polyvariant analysis (as described in [2]) could be used to gain precision through context sensitivity. It is left as future work to implement the interprocedural analysis.

```
int main() {
    char * c="c";
    char * s = malloc(101);
    s[0] = '\0';
    for( int i=0; i<100;i++)
        strcat(s,c);
    printf(s);
}
```

Figure 3: C program that prints the string c^{100}.

```
init    1 "c"
init    2 ""
concat  3 (1 4)
hotspot assign 4 (2 3)
```

Figure 4: *String flow graph* generated by x86sa for the C program in Figure 3

5 Evaluation

In this section we present the analysis of a simple C program that has been compiled into an Intel x86 binary and of a real binary taken from a Linux rootkit (the Lion worm).

5.1 C program

The C program in Figure 3 gernerates a string containing 100 'c's (or c^{100}). After x86sa runs its analysis, the SFG (Figure 4) for the main function is fed into the JSA backend. The JSA backend returns the regular expression c^*. This is a safe overapproximation of the actual string value that can arise at the call to `printf`. The analysis's overapproximation of the actual value is due to not interpreting the loop condition. This can also arise at join points in the program.

5.2 Lion worm

One of the goals of x86sa was to be able to analyze Intel x86 viruses and worms. To validate our approach we ran x86sa on one of the binaries propagated by the Lion worm. The Lion worm is a Linux rootkit that replaces critical system files. One such file replaced is the Unix `password` binary. The modified binary looks, from the user's perspective, to change the user's password. However; when run, the trojaned binary sends the Linux computer's network information to an email address. The source code for the binary is not available and the SFG too large for this paper so they are omitted. Figure 5 shows the regular expression output after running x86sa. This is the exact command the trojaned binary runs to send the network information to the attacker.

6 Related work

7 Conclusions

```
/sbin/ifconfig -a |/bin/mail angelz1578@usa.net
```

Figure 5: The output of x86sa on the trojaned binary propagated by the Lion worm.

References

[1] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, Nov. 2003. 1

[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium (SAS '03)*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, June 2003. 1, 2, 4

[3] DataRescue sa/nv. IDA Pro – interactive disassembler. Published online at `http://www.datarescue.com/idabase/`. Last accessed on 3 Feb. 2003. 1

[4] C. Gould, Z. Su, and P. Devanbu. JDBC checker: A static analysis tool for SQL/JDBC applications. Formal research demo presented at the 26th International Conference on Software Engineering (ICSE '04), May 2004. 1

[5] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 645–654, Edinburgh, Scotland, UK, May 2004. IEEE Computer Society. 1

[6] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, Mar. 2004. 1

[7] M. Mohri and M.-J. Nederhof. Regular approximation of context-free grammars through transformation. In J.-C. Junqua and G. van Noord, editors, *Robustness in Language and Speech Technology*, pages 153–163. Kluwer Academic Publishers, 2001. 2.1

[8] R. Vall, e Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay. Soot - a java bytecode optimization framework, 1999. 2.1