String Analysis for x86 Binaries*

Mihai Christodorescu†
mihai@cs.wisc.edu

Nicholas Kidd‡
kidd@cs.wisc.edu

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

Department of Computer Sciences
University of Wisconsin–Madison
Madison, WI, USA

ABSTRACT

Information about string values at key points in a program can help program understanding, reverse engineering, and forensics. We present a static-analysis technique for recovering possible string values in an executable program, when no debug information or source code is available. The result of our analysis is a regular language that describes a superset of the string values possible at a given program point. We also impart some of the lessons learned in the process of implementing our analysis as a tool for recovering C-style strings in x86 executables.

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 can use the possible strings output at a particular line in a program as hints towards understanding undocumented functionality. In other cases, if we have information about the language 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 could check the validity of such output, ensuring that the program integrates properly into larger, distributed systems. Thus, there are several reasons why it would be desirable to have a tool to automatically analyze a program and produce the set of strings possibly generated at 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 [3], designed for Web service programming. The JWIG compiler verifies that programs written in JWIG generate valid XML and XHTML documents. Gould, Su, and Devanbu [12] applied string analysis to construct a sound, static analysis for verifying the type correctness of dynamically generated strings, and created *JDBC Checker* [11], a tool that implements this analysis. Kirkegaard, Møller, and Schwartzbach [13] 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 executable code is one such case: a program in executable format retains little detail beyond the actual semantics of the original source code. Renovating, upgrading, or replacing legacy code where neither source code nor documentation are available is one possible reverse-engineering scenario. Forensic analysis of malicious code is another application of reverse engineering: in such situations, source code cannot be trusted or is simply not available. Both of these areas can benefit from string-analysis techniques that recover information to aid one's understanding of a program.

In this paper, we present a string-analysis technique that recovers string values at points of interest in an executable 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 the Java String Analyzer (JSA) infrastructure recently introduced by Christensen, Møller, and Schwartzbach [4]. The major effort in our work is the recovery of sufficient information from the executable to identify the strings manipulated by the pro-

*The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

†Mihai Christodorescu was supported in part by the Office of Naval Research under contracts N00014-01-1-0796 and N00014-01-1-0708.

‡Nicholas Kidd was supported in part by the National Science Foundation under grant CCR-9986308.

PASTE '05 Lisbon, Portugal

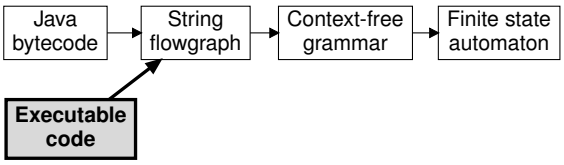

Figure 1: Architecture of modified JSA. The top boxes indicate the original architecture of JSA.

gram 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 at points of interest.

Our work makes the following contributions:

- **A static analysis for recovering strings in executable code.** We successfully bridge the gap between native machine code, a low-level language, and the JSA string flow graph, a high-level construct (described in Section 2). Toward this end we make novel use of several static-analysis techniques to recover semantic information missing from the executable 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 developed a tool that allows the user to query for the possible string values at points of interest in the program. The *x86sa* tool builds on top of the popular disassembler IDA Pro [7], making it easy to use.
- **An experimental evaluation showing the feasibility of our approach.** We evaluate the *x86sa* tool on both benign and malicious programs to illustrate the strengths and limitations of our static analysis. As we show in Section 5, *x86sa* performs well on all test cases, within the analytically predicted accuracy limits.

2. STRING-ANALYSIS OVERVIEW

We model the flow of string values through the program using the *string flow graph* structure introduced by Christensen, Møller, and Schwartzbach [4]. The construction that we use maps string constants and string operations in the program to nodes in the string flow graph, while assignment statements and call site arguments are represented as edges. The Java String Analyzer (JSA) [4] is a tool developed at the University of Aarhus for the puposes of analyzing string expressions in Java programs. In this section, we briefly describe the overall architecture of the JSA and the modifications made to allow it to parse flow graph files produced by *x86sa* during the analysis of an executable program. Figure 1 highlights the main components of JSA. In our implementation, we replace the Java analyzer front end with our x86 analyzer, which produces a flow graph file that is later parsed by JSA. The focus of our work is on performing the translation from the executable program to the string flow graph.

2.1 String Analysis for Java Programs

The JSA is built on top of the Soot toolkit for parsing Java bytecode [20] and uses the Mohri-Nederhof algorithm for

```
1 String x;
2 if( ... )
3    x = "abc";
4 else
5    x = "def";
6 String y = x + " " + x;
7 System.out.println( y );
```

Figure 2: Java program fragment using string operations.

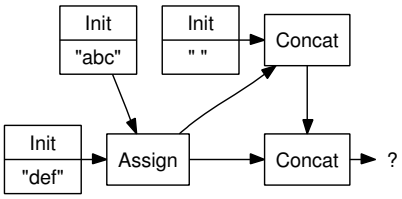

Figure 3: String flow graph for the code in Figure 2.

approximating context free grammars with regular expressions [16]. The JSA front end takes Java bytecode as input and transforms it into a string flow graph. A string flow graph contains the following types of nodes: INIT nodes for each string constant that appears in the Java program, ASSIGN nodes for assignments and control join points, CONCAT nodes for each string concatenation, UNARYOP nodes for unary operations such as `setCharAt`, and BINARYOP nodes for binary operations such as `replace`.

JSA models operations on objects of the `String` and `StringBuffer` class types, while everything else is abstracted away. String concatenation is modeled most precisely by the tool because concatenation is an inherent construct in a context-free grammar; other operations (either unary or binary) are only approximated using automaton operations or character sets.

The string flow graph is then transformed into a *context-free grammar* in a straightforward manner: for each node that is not an INIT node, a new production is added whose left-hand side is a new non-terminal and whose right-hand side is a list of terminals and non-terminals, one per flow-graph predecessor. The Mohri-Nederhof algorithm [16] is then applied to produce an approximation in the form of a *strongly-regular context-free grammar* that, in turn, can be accurately modeled by a regular expression.

Points of interest in the program are specified by the user as *hotspots*: for each hotspot in the resulting strongly-regular context-free grammar, JSA generates a finite state automaton (FSA) that describes a superset of the strings that reach the hotspot. Examples of hotspots include, but are not limited to, `print` statements, SQL queries, and system calls.

Consider the code in Figure 2. (For readability the example is given in Java, whereas JSA actually operates at the bytecode level.) Lines 3 and 5 become INIT nodes in the string flow graph, line 6 is expanded into a series of CONCAT nodes, and line 7 is the hotspot indicated by the user. The string flow graph derived from the Java code is shown in Figure 3, and the result of the JSA algorithm is the set of strings {`"abc abc"`, `"def def"`}.

By constructing a meaningful string flow graph from an executable, we can use the modified JSA tool to analyze the string flow graph and recover string values from the executable. By these means, we reduce the problem of string

```
1  if( ... )
2     x = "abc";
3  else
4     x = "def";
5  tmp = strcat( x, " " );
6  y = strcat( tmp, x );
7  printf( y );
```

Figure 4: C-like program fragment using string operations, similar to the Java program in Figure 2.

```
1  mov  ecx, [ebp+4]
2  push ecx
3  mov  ebx, [ebp+8]
4  push ebx
5  call _strcat
6  add  esp, 8
```

Figure 5: x86 assembly code for a call to the `libc` `strcat` function from line 6 of Figure 4. Here, the C variables `tmp` and `x` are stored in memory at addresses `ebp+4` and `ebp+8`, respectively.

analysis for binaries to the problem of string flow graph construction for binaries.

2.2 Transitioning from Java Bytecode to Native Code

Unfortunately, there are significant differences between the Java language and any native code (such as IA-32 x86). These differences, detailed below, require additional analyses before a string flow graph can be constructed.

The first problem any analysis for binaries runs into is the lack of type information. Java has a rich, well-defined type system that is present even in the compiled bytecode. This lack of type information in executable code makes harder the identification of strings and of functions that manipulate strings (compare Figures 2 and 4 with Figure 5, which shows a C-language call translated to x86 assembly code). Compounding this problem is the absence of local variables in executable code: all operations are performed using a fixed set of global variables (the registers and memory). This means the same register can hold values of different types at different points in the program.

A second problem is caused by the syntactic restrictions of executable code: a call to a function does not have the actual arguments explicitly represented (see Figure 5). A number of different calling conventions exist, which have different rules for how actual parameters are stored and whether the caller or the callee deallocates the parameters. The same problem exists in Java bytecode, but it is alleviated by the fact that methods are typed (making it easier to recover actual parameters) and by the restriction imposed on the stack (at join points, the different stacks must have the same height and the same type information).

Finally, the `String` type in Java is peculiar in that it is immutable (i.e., all values are constant). Any operation that returns a `String` automatically creates a new object. This is in contrast with the `StringBuffer` type and the string types in other languages (such as C), which are mutable. The immutability of the `String` type is beneficial for analysis, since any update to a string automatically kills any aliases to that value. In contrast, the fact that all operations modify strings in executable code significantly raises the complexity of building a string flow graph.

3. STRING ANALYSIS FOR X86 EXECUTABLES

We now consider the problem of analyzing a single function in a program and constructing a string flow graph for C-style strings in x86 executables. We extend the analysis to the interprocedural case in Section 4, and discuss how to adapt this analysis for other string libraries in Section 6.

We apply two analyses, a *string-inference analysis* (Section 3.1) and a *stack-height analysis* (Section 3.2), to recover a limited amount of type information for the executable. To overcome the mutability of C-style strings, *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, which is then passed to the JSA backend (Section 3.4).

All the analyses presented in this section operate over pointers to strings. Pointers to strings are stored in "machine variables," which are either a register or a memory location relative to the active stack frame (i.e., corresponding to a local variable in a high-level language) at a given program point (e.g., `eax@0x401450` represents the value of x86 register `eax` at program point `0x401450`). For simplicity of exposition, we use the terms "pointer to string" and "string" interchangeably in the rest of the paper.

3.1 String Inference for x86 Executables

To locate possible C-style strings in an x86 executable, *x86sa* performs a string-inference analysis. The intuition behind the analysis is that common `libc` string functions have a known interface and therefore a known type signature. This information is then used to discover which registers and memory locations may reference C-style strings. String inference is essentially type inference with a trivial type system, namely a type system of only strings and the unknown type. *x86sa* assumes that all strings in the program are manipulated through the `libc` string functions.

ASSUMPTION 1. *Strings in the x86 executable are represented using the C-style (null-terminated) encoding.*

ASSUMPTION 2. *All operations on strings use the* `libc` *string functions.*

For example, consider the call to `strcat` in Figure 5. From the `strcat` signature, we can infer that registers `ebx` and `ecx` reference C-style strings before the call. After `strcat` completes, the register `eax` contains the return value and references the same string as `ebx`. The memory contents referenced by `eax` and `ebx` contain the concatenation 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.

Formally, string inference is a backward dataflow analysis on the function's control flow graph (CFG), propagating string type information. Each dataflow fact $d \in D$ contains a set of variables that *may* reference a C-style string. The lattice is ordered by the superset relation, $\supset$, and the meet operator is defined as set union, $\cup$.

Each call to a `libc` string function transforms the string information in an appropriate fashion. For example, the transformer for `strcat(A,B)` is defined as:

$$\lambda d.d \setminus \{\texttt{eax}\} \cup \{\texttt{A}, \texttt{B}\} \qquad \text{where } d \in D$$

Note that `strcat` removes any knowledge of the register `eax`, as `eax` is, by convention, used for the function return value. In general, the destination register for an x86 instruction kills the 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.

String inference runs in $O(nk)$ time, where n is the number of variables being tracked and k is the number of edges in the CFG.

3.2 Stack-Height Analysis for x86 Executables

To generate transformers for the `libc` string functions, the instructions that carry out the passing of actual parameters from the caller to the callee (i.e., a `libc` entry point) must be known. However, the x86 ISA does not explicitly associate arguments with their respective function call (see Figure 5). Function arguments can be pushed onto the stack or passed through registers. The analysis assumes that the executable follows the `_cdecl` calling convention, with all the function arguments passed through the stack.

ASSUMPTION 3. *The x86 executable uses the* `_cdecl` *calling convention.*

ASSUMPTION 4. *Function arguments are passed from the caller to the callee using the stack.*

This calling convention mandates that the caller pushes the arguments on the stack and pops them off the stack. With this assumption, *x86sa* performs a stack-height analysis that is used to associate push instructions with their function calls. The analysis is split into two parts.

First, a forward 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 5, the instruction on line 6 is a cleanup instruction. Standard compilers adhering to the `_cdecl` calling convention typically use an `add esp,C` instruction where `C` is a constant. Dividing `C` by the word size of the x86 architecture (4 bytes) provides the number of arguments passed to the function call. This heuristic algorithm, although it fails for arguments with sizes different from the native word size, has performed well in our experiments (where most of the arguments passed were integer or pointer values).

Second, a forward 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 the node. A stack configuration is a vector of effective addresses of instructions pushing data on the program's stack. Each CFG node is assigned a transformer that models the CFG node's instruction's effect on the program's stack. For example, a `push` instruction pushes a reference to the instruction (its effective address) onto the set of (abstract) stacks that reach the instruction (the union of the set of abstract stacks associated with the node's predecessors). Likewise, a `pop` instruction pops an effective address of each stack in the node's stack configuration.

Using the number of arguments pushed at a function call site, the stack model for the call site CFG node is queried to identify the instructions that set up its arguments. In Figure 5, the two push instructions (lines 2 and 4) are associated with the call to `strcat`. This information is used to create the transformer for the `strcat`'s CFG node during string inference.

3.3 Alias Analysis for x86 Executables

A string operation in an x86 program may affect the memory contents aliased by multiple string pointers (in contrast to Java strings, which are immutable). We use alias analysis to determine 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 "machine variables".

Alias analysis is a forward dataflow analysis on the CFG. It associates with each CFG node a set of $\langle$variable, *string-creation point*$\rangle$ pairs. A string-creation point is a point in the program that creates or modifies a string, such as a CFG enter node (where constant strings and function arguments are "created"), a call to `libc` memory allocation functions (e.g., `malloc`), or a call to `libc` string functions (for example, `strcat`).

To increase precision, alias analysis uses *may-must relations* [1] that allow for strong updates in the must-alias set. Each dataflow fact $d \in D$ contains two sets, a must-alias set and a may-alias set, of string-creation point pairs. The lattice is ordered by the superset relation, $\supset$, and the meet operator is defined as follows:

$$\begin{aligned} d_1 \sqcap d_2 = \langle\ & must(d_1) \cap must(d_2), \\ & may(d_1) \cup may(d_2) \cup \\ & \quad (must(d_1) \triangle must(d_2))\ \rangle \end{aligned}$$

The dataflow transformers are defined for each `libc` function. For example, the transformer for `strcat(A,B)` is defined for the corresponding string-creation point $scp_{\texttt{strcat}}$ as follows:

$$\begin{aligned} \lambda d.\langle\ & (must(d) \setminus \{(\texttt{A}, *), (\texttt{eax}, *)\}) \cup \\ & \quad \{(\texttt{A}, scp_{\texttt{strcat}}), (\texttt{eax}, \texttt{A})\}, \\ & may(d) \setminus \{(\texttt{A}, *), (\texttt{eax}, *)\}\ \rangle \end{aligned}$$

Aliases are created by register assignments and writes to a string. For example, the `mov` and `lea` instructions create an alias between a register (or memory location) and a string-creation point. Each `libc` string function generates a new string-creation point and its effects on the alias relation are modeled accordingly. Figure 6 shows how the transformer for the `strcat` function operates. Before the call, there are three string-creation points, which are represented by the different shaded figures on the left. These string-creation points are referenced by the registers `ecx`, `ebx`, and `esi`. Registers `ebx` and `esi` are aliases to each other because they reference the same string-creation point. All aliases before the call to `strcat` are in the *may-alias* set (represented

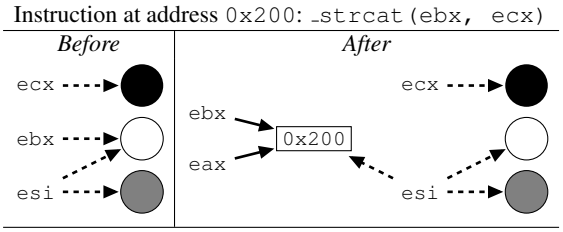

Figure 6: Alias analysis applied to a `strcat` call in x86 code at address `0x200`. The alias relations before the call are on the left. Dotted lines represent *may-alias* relations, solid lines represent *must-alias* relations.

by the dashed arrows). After the call, there are four string-creation points, represented by the different shaded figures on the right. `strcat` generates a new string-creation point (`0x200`). After the call, the registers `eax` and `ebx` are added to the *must-alias* set (represented by solid arrows). A may-alias between register `esi` and the string-creation point `0x200` is added because `esi` was a may-alias of the string-creation point for register `ebx` before the call.

Alias analysis runs in $O(nmk^2)$ time, where n is the number of variables being tracked, k is the number of edges in the CFG, and m is the maximum number of variables that may be assigned to at any program point.

3.4 Construction of String Flow Graphs from x86 Executables

After alias analysis, the alias information associated with each string-creation point is an encoding of the string flow graph. A forward traversal of the CFG is used to translate the encoding into a string flow graph. At each string-creation point (i.e. `malloc`, `strcat`, etc.) an appropiate flow graph 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 node is created first and the ASSIGN node is used as the predecessor to the string-creation point. For example, the `libc strcat` function corresponds to a CONCAT node. The alias information of the operands represents the predecessors of the flow graph node. If there are multiple aliases, then an ASSIGN node must first be created (because a CONCAT node can have only two predecessors). A string flow graph for the C program in Figure 7 is shown in Figure 8.

4. THE INTERPROCEDURAL CASE

When the executable being analyzed contains multiple functions, the string flow graphs for all functions 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 string flow graph of a called function could be inlined at the function call site. This provides a precise analysis, but cannot handle recursion.

Second, the string flow graph for each function could be linked together to form a super string flow graph. A limitation of this approach is that invalid paths can pollute the analysis; however, this is the approach used by the JSA. The general algorithm associates with each flow graph an ASSIGN node for each of its *formal* parameters. Each *actual* parameter at a call site flows into the callee's flow graph.

```
1  int main() {
2     char * c= "c";
3     char * s = malloc(101);
4     s[0] = '\0';
5     for(int i=0; i<100;i++)
6        strcat(s,c);
7     printf(s);  // ← hotspot
8  }
```

Figure 7: C program that prints the string c^{100}.

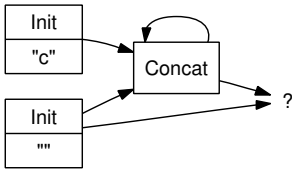

Figure 8: String flow graph generated by *x86sa* for the C program in Figure 7.

The return value of the called function flows back to each call site. One subtle difference that arises in x86 binaries is that all strings are mutable. Therefore, each *formal* parameter of a function must flow back to the caller's *actual* parameters. This ensures modifications to strings by the callee are witnessed in the caller.

Third, a context-sensitive analysis can be used to eliminate the invalid paths. We are currently investigating the use of Weighted Pushdown Systems [18] to provide a context-sensitive analysis.

5. EVALUATION

In this section we present *x86sa* results for two benign C programs (compiled into Intel x86 executables) and a malicious program taken from a Linux rootkit (propagated by the Lion worm).

5.1 A Simple C Program

The C program in Figure 7 generates the string c^{100}. After *x86sa* runs its analysis, the string flow graph (Figure 8) 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 overapproximation of the actual value is due to not interpreting the loop condition.

5.2 A Tricky C Program

In order to compare the *x86sa* implementation against the original JSA implementation, we converted the Java-based `Tricky` example from the JSA paper [4] into an equivalent C program. As the current *x86sa* implementation has no support for interprocedural analysis, we inlined calls from the function `foo` to the recursive function `bar` in the form of loops. The resulting C program, shown in Figure 9, generates and prints strings of the form:

```
((((((6*5)*4)+3)+2)+1)+0)
```

The point of interest (i.e. the hotspot) is the call to `printf` on line 50. Since the language of all string values possible at that hotspot is not regular, the result is necessarily an overapproximation of the correct context-free language.

The result of the string analysis, applied to a binary compiled from the source code in Figure 9, is the following reg-

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

char* foo( int n ) {
    char *lp = "(";
    char *rp = ")";
    char *add = "+";
    char *mul = "*";
    int i;
    int mullim = n/2-1;
    int addlim = n - mullim;

    char *buf = malloc(1024);
    buf[0] = 0;

    if( n < 2 ) {
        strcat(buf,lp);
    }

    for( i=0; i < n ; i++ ) {
        strcat(buf,lp);
    }

    {
      char tmp[10];
      sprintf(tmp,"%d",n);
      strcat(buf,tmp);
    }

    // First inlined call to bar()
    for( i=0; i < mullim ; i++ ) {
        char tmp[10];
        sprintf(tmp,"%d",n-1-i);
        strcat(buf,mul);
        strcat(buf,tmp);
        strcat(buf,rp);
    }

    // Second inlined call to bar()
    for( i=0; i < addlim; i++ ) {
        char tmp[10];
        sprintf(tmp,"%d",n-n/2-i);
        strcat(buf,add);
        strcat(buf,tmp);
        strcat(buf,rp);
    }

    printf("%s\n",buf);    // ← hotspot
    return buf;
}

int main() {
    char *b;
    srand(time(0));

    b = foo(rand()%100);

    free(b);
    return 0;
}
```

Figure 9: The C-version of the `Tricky` example.

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

Figure 10: The output of *x86sa* on a rootkit program propagated by the Lion worm. Here, the string is split to fit in one column.

ular expression (in POSIX 1003.2 notation):

$$(*\langle int\rangle\ (\backslash *\langle int\rangle\backslash)) * (\backslash +\langle int\rangle\backslash)) *$$

where $\langle int\rangle$ is shorthand for the regular expression:

$$0|[1\text{-}9][0\text{-}9]*$$

This regular expression is similar to the one obtained by JSA when applied to the Java-bytecode implementation of `Tricky`. Comparing the two regular expressions, we note that we obtain a more precise result due to the manual inlining step performed during the conversion from Java to C. As a result of the inlining, some of the infeasible paths are eliminated and the *x86sa* string analysis yields a more precise regular expression.

5.3 Lion Worm

One of the goals of *x86sa* is to be able to analyze x86 viruses and worms. To test our approach, we ran *x86sa* on one of the binaries propagated by the Lion worm. The Lion worm uses a Linux rootkit (called `t0rn`) to replace critical system files.

One of the `t0rn` toolkit components is an executable that appears to be a wrapper around the `crypt` function. When *x86sa* is run on this component, an unexpected string appears to be built right before a call to `execve`. The source code for this executable is not available and the string flow graph too large for this paper, so we have omitted them. Figure 10 shows the regular expression that results from running *x86sa*. This string expression constitutes a command that, when run, sends the computer's network configuration to an outside email address.

A simple examination of the executable file (e.g., using the Unix tool `strings`) does not reveal any information, only a list of strings of length 1 (e.g., `"s"`, `"i"`, etc.). This is because the program builds the string at runtime, character by character, using `strcat` (i.e., the string `"/sbin"` is created using 5 calls to `strcat`). Our analysis defeats this obfuscation attempt and provides a forensic expert with better information.

6. FUTURE WORK

The current *x86sa* prototype is able to analyze one function in a x86 executable at a time. This section lists the future work necessary for *x86sa* to move from a prototype to a fully featured x86 executable analyzer.

The primary limitation of our static-analysis tool is the assumption that all strings in the x86 executable are C-style strings. This allowed us to model only the `libc` string functions. However, other string encodings exist in common software packages (e.g., Unicode). Relaxing this assumption requires modeling the APIs associated with the other string encodings, such as the `libc` functions for handling the wide (Unicode) characters.

A second limitation is the assumption that all strings are only modified by the `libc` string functions. Value Set Anal-

ysis [2], or VSA, is a recently developed method for analyzing memory accesses in and recovering variables from x86 executables. This analysis provides an abstraction called *abstract locations* that models a superset of the values contained within a memory region. We plan to use VSA to relax *Assumption 2*.

A third limitation is that *x86sa* does not perform interprocedural analysis. Section 4 discusses three possible solutions for implementing interprocedural analysis between the SFGs. To completely relax *Assumption 3*, the stack-height analysis needs to be extended to perform an interprocedural analysis. This may be necessary to correctly analyze malicious code that has been obfuscated, as described in [14].

7. RELATED WORK

There is a vast body of work on static analysis of programs for verification, reverse engineering, and understanding purposes. We highlight some of the research relevant to the goals and solutions presented in this paper. Christensen, Møller, and Schwartzbach presented a technique for discovering the possible values of string expressions in Java programs [4]. The Java String Analyzer (JSA), described in detail in Section 2, arose from this research and constitutes the foundation upon which we created *x86sa*.

The possibility of buffer overruns in untyped or weakly typed languages (such as C), combined with the increasing number of online services, has created significant security problems. As a result, numerous approaches have been proposed for statically detecting buffer overflows: automatic checking of user-supplied annotations of buffer operations [9, 15], symbolic evaluation [19, 22], and constraint solving [10, 21]. Many other techniques have been suggested for dynamic detection of buffer overflows, but they are outside the scope of this paper. What is common to all of these static analysis techniques is the domain over which they operate: the goal is to determine whether a buffer index or a pointer into a buffer can ever step outside the bounds of the buffer. In contrast, we do not concern ourselves with the lengths of buffers, focusing instead on the contents of string buffers. Our analysis can be used to enhance the reports generated by a buffer overrun detector, for example, by providing a sample string value that overflows the buffer.

The string-inference algorithm is part of the general area of type inference, where descriptive types are added to untyped operands based on usage patterns. A related approach is the type-based decompilation work of Mycroft [17] that associates bit-vector types with the operands of machine instructions. In comparison, our work focuses on two high-level types: strings (i.e. arrays of characters) and pointers to strings. The Value Set Analysis (VSA) of Balakrishnan and Reps [2] aims to recover data abstractions from x86 executable programs, also based on usage patterns. We plan to use VSA as an additional analysis supplementing our own string-inference methods, as discussed in Section 6. The stack-height analysis we use to reconstruct information about call site arguments is related to the analysis introduced by Lakhotia and Khumar for detecting obfuscated calls in binaries [14]. The alias-analysis algorithm we use is fairly standard. Debray, Muth, and Weippert [8] introduce an alias-analysis algorithm for executables, with the goal of over-approximating the values each register can hold at each program point. In contrast, our alias analysis is aimed at recovering information about string pointers.

Finally, the fields of reverse engineering, decompilation, and binary translation are also related to our goals. However, past work in this area [5, 6] has been targeted towards the recovery of high-level control flow constructs (loops, recursion, structure control flow, etc.), while we target string values.

8. CONCLUSIONS

We have presented a string analysis for executable programs, built on top of the Java String Analyzer toolkit. To bridge the semantic and syntactic chasms between Java (a high-level, strongly typed language) and x86 (a machine-level, untyped language), we created a static analysis to recover information about string usage in x86 executables. In our experiments, string values were recovered successfully, within the limitations of our assumptions. As outlined in Section 6, we plan to ease some of these assumptions in our future work.

9. ACKNOWLEDGMENTS

We thank Ben Liblit, Somesh Jha, and Thomas Reps for their guidance, support, and feedback throughout the development of this work. We also thank the anonymous referees for their useful comments.

10. REFERENCES

[1] R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 74–84, New York, NY, USA, 1995. ACM Press.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction (CC'04)*, pages 5–23, 2004.

[3] 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.

[4] 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.

[5] C. Cifuentes and A. Fraboulet. Interprocedural dataflow recovery of high-level language code from assembly. Technical report, University of Queensland, 1997.

[6] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, pages 228–237, 1998.

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

[8] S. K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proceedings of the 1998 Conference on Principles of Programming Languages (POPL'98)*, pages 12–24, 1998.

[9] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (SIGPLAN'03)*, pages 155–167, San Diego, CA, USA, June 2003. ACM Press.

[10] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 345–354, Washington D.C., USA, Oct. 2003. ACM Press.

[11] 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.

[12] 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.

[13] 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.

[14] A. Lakhotia and E. U. Kumar. Abstract stack graph to detect obfuscated calls in binaries. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 17–26, Chicago, IL, USA, Sept. 2004. IEEE Computer Society.

[15] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, 2001.

[16] 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.

[17] A. Mycroft. Type-based decompilation. In *Proceedings of the European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer-Verlag, 1999.

[18] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 2005. to appear.

[19] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 182–195, Vancouver, BC, Canada, June 2000. ACM Press.

[20] R. Vall, E. Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay. Soot – a Java bytecode optimization framework, 1999.

[21] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Networking and Distributed System Security Symposium (NDSS'00)*, San Diego, California, Feb. 2000.

[22] Y. Xie, A. Chou, and D. Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes*, 28(5):327–336, Sept. 2003.