# Static security analysis based on input-related software faults

Csaba Nagy
Department of Software Engineering
University of Szeged
Szeged, Hungary
ncsaba@inf.u-szeged.hu

Spiros Mancoridis
Department of Computer Science
Drexel University
Philadelphia, USA
spiros@drexel.edu

## Abstract

*It is important to focus on security aspects during the development cycle to deliver reliable software. However, locating security faults in complex systems is difficult and there are only a few effective automatic tools available to help developers. In this paper we present an approach to help developers locate vulnerabilities by marking parts of the source code that involve user input. We focus on input-related code, since an attacker can usually take advantage of vulnerabilities by passing malformed input to the application. The main contributions of this work are two metrics to help locate faults during a code review, and algorithms to locate buffer overflow and format string vulnerabilities in C source code. We implemented our approach as a plugin to the Grammatech CodeSurfer tool. We tested and validated our technique on open source projects and we found faults in software that includes Pidgin and cyrus-imapd.*

## 1. Introduction

As IT solutions become more common, the security of software systems becomes an increasingly important consideration. Companies are paying close attention to software security, as evidenced by their increased focus on security aspects during the development cycle. For instance, Microsoft has published its *Security Development Lifecycle* standard, which is a Microsoft-wide initiative and mandatory policy since 2004 [25]. One important activity of SDL, and many other software development process models, is the use of static code-scanning tools during the implementation phase [25, 9]. These code-scanning tools perform a static analysis of the source code, without executing the programs built from the code, which is the domain of dynamic analysis. Locating security faults in software is still difficult to do even with the help of static analyzers. Moreover, there are only few effective automatic tools available that can help developers locate security faults. Even if an effective code-scanning tool is available, it is difficult to locate security faults during a code review: in general, a good code review can uncover around 50% of the security problems [9].

This paper presents an approach to helping developers locate faults that are related to security by identifying parts of the source code that involve user input. The focus is on the input-related parts of the source code, since attackers commonly exploit security vulnerabilities by passing malformed input data to applications. Mishandling input data can be a source of common security faults in many languages that support pointer arithmetic such as C and C++. Examples of security faults are *buffer overflows*, *format string vulnerabilities*, and *integer overflows* [19]. The best known and, arguably, the most dangerous security faults are caused by buffer overflows, which are described in an article published in 1996 [1], and appear in the literature as far back as 1988 [12]. This type of vulnerability is still common in software systems and is difficult to locate either automatically or by a manual code review. Recent research has shown that code defects related to buffer overflows are still frequent in open source projects [10].

The main contributions of this paper are:

- two metrics (*input coverage*, *input distance*), which can help developers during a code review to locate functions that likely contain security faults,
- two algorithms, one published previously [26] and a new algorithm we created, to locate buffer overflow and format string vulnerabilities in C source code,
- a demonstration of the effectiveness of these metrics and algorithms as they are applied to open source software projects.

Our technique is implemented as a plugin to the CodeSurfer product of GrammaTech, Inc. The technique was validated on open source projects and successfully identified several faults in software including in Pidgin and Cyrus Imapd.

The remainder of this paper is organized as follows: Section 2 is an overview of our analysis technique. Section 3 describes the metrics and algorithms underlying our technique. The results of applying our technique on security-critical open source projects are presented in Section 4. Section 5 presents related work. Lastly, the paper concludes and outlines plans for future work in Section 6.

## 2. Overview

This section presents an overview of the technique employed for a static security analysis based on input-related faults.

### 2.1. Technique

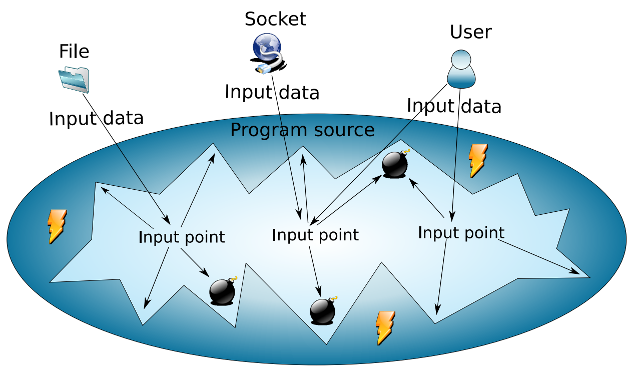


**Figure 1. Illustration of input-related security faults.** Faults related to user input are marked with "bombs" indicating vulnerabilities.

The main idea behind our approach is to focus on the input-related parts of the source code, since an attacker can usually take advantage of a security vulnerability by passing malformed input data to the application. If this data is not handled correctly it can cause unexpected behavior while the program is running. The path which the data travels through can be tracked using *dataflow analysis* [20] to determine the parts of the source code that involve user input. Software faults, can appear anywhere in the source code, but if a fault is somewhere along the path of input data it can act as a "land mine" of a security vulnerability. (An illustration can be seen on Figure 1).

The main steps of our approach (Figure 2) are the following:

1. find locations in the source code where data is read using a system call of an I/O operation. These calls are marked as *input points*,
2. get the set of program points involved in user input,
3. get a list of dangerous functions using metrics,
4. perform automatic fault detection to find vulnerabilities.

**2.1.1. Locate I/O points.** Input data can come from many different sources, not only from the standard input. It can come from input character devices, Internet sockets, files in the file system, *et cetera*. In general, *input points* are statements used to read input data from an external source by calling a system function to perform an I/O operation. The input data is often a string that is stored in a buffer that has been allocated on the stack or the heap.

**2.1.2. Extract input-related program points.** After locating the input points in the source code, it is possible to determine how the input data travels from one statement to another statement. This can be done using *dataflow analysis*, a technique for gathering information about the possible set of values calculated at various points in a program. Once we have the path for all input points, we can determine which parts of the source code involve user input by computing the union of these paths.

To perform dataflow analysis on C/C++ code we use the CodeSurfer tool of GrammaTech, Inc.

**2.1.3. Get the list of dangerous functions.** We can obtain a list of functions that warrant an increased scrutiny by determining which parts of the source code involve user input. We call the list of such functions *dangerous functions*.

To give developers more information about a dangerous function we measure its *coverage* as the percentage of its source code statements that are tainted by user input. We also measure the *distance* in the dataflow graph between the entry point of the function and the origin of the input data (*i.e.,* the statement where input occurs). These metrics are used to rank the functions in order to identify the functions that are the most tainted by user input.

**2.1.4. Automatic fault detection.** Automatic fault detection is performed by our technique to detect security problems in dangerous functions. These fault detections are based on algorithms that are applied to the code's corresponding data dependence graph and can point to buffer overflow or format string vulnerabilities.

### 2.2. CodeSurfer

Our technique is implemented as a CodeSurfer[1] plugin. CodeSurfer is a powerful static-analysis tool for C/C++ programs. This tool was chosen because it is able to create a wide range of intermediate representations [2] for a

[1]`http://www.grammatech.com`

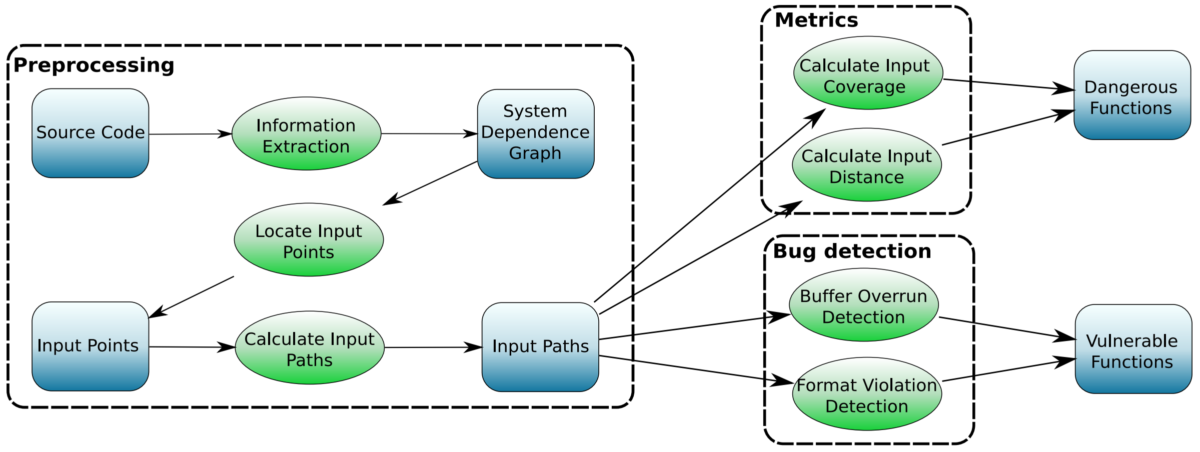


**Figure 2. The overview of our system.**

given program including: Abstract Syntax Tree (AST), Call Graph, Interprocedural Control-Flow Graph (CFG), Points-to Graph, set of variables used and modified for each function, Control Dependence Graph, and Data Dependence Graph. The CodeSurfer tool can be extended with plugins using its internal scripting language or its C/C++ API.

The most important feature of CodeSurfer, for the purposes of this work, is that, after a whole-program analysis is performed, CodeSurfer can build a precise *system dependence graph* [18] due to its *pointer-analysis* [3] capability.

## 2.3. System dependence graph

Depending on the the application there are different definitions for *program dependence graph* (PDG) [18, 14, 22]. PDG is a directed graph representation ($G_P$) of a program ($P$), where vertices represent program points (*e.g.,* assignment statements, call-sites, variables, control predicates) that occur in $P$ and edges represent different kinds of control or data dependencies. There is a data dependence edge between two vertices if the first program point may assign a value to a variable that may be used by the second point. There is a control dependence edge between two vertices if the result of executing the first program point controls whether the second point will be executed or not.

A *system dependence graph* (SDG) [18] is the interprocedural extension of the PDG. It consists of interconnected PDGs (one per procedure in the program) and extends the control and data dependencies with interprocedural dependencies. An interprocedural control-dependence edge connects procedure call sites to the entry points of the called procedure and an interprocedural data-dependence edge represents the flow of data between actual parameters and formal parameters (and return values). Globals and other non-local variables such as file statics, and variables accessed indirectly through pointers are treated as additional parameters to procedures.

A system dependence graph can be used for many purposes such as code optimization [14], reverse engineering, program testing [6], program slicing [18], software quality assurance [17], and software safety analysis [29].

This work employs SDG, and the extracted dataflow information stored in this representation, to determine the paths on which user-related input travels from its input point.

## 3. Technique

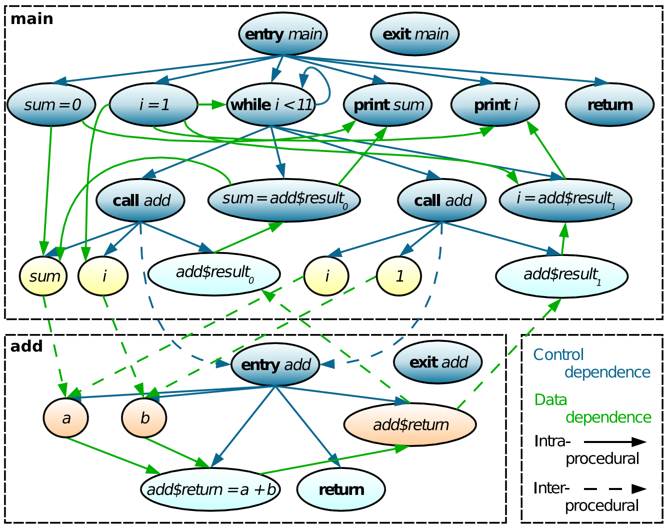
This section describes our technique and presents details about how metrics and algorithms are used to locate buffer overflow and format string vulnerabilities.

### 3.1. Locating input points

Input points are statements in the source code that perform I/O operations to read data from standard input, network sockets, or files. To locate these statements we look for invocations of I/O functions that are declared in the header files of the *C standard library*. Examples of these function calls are: `fscanf`, `scanf`, `getc`, `gets`, and `read`. We handle 28 function calls as input functions declared in header files such as: `stdio.h`, `stdlib.h`, `unistd.h`, and `pwd.h`.

The `argc` and `argv` parameters of a program's `main` function are also considered input points, since these parameters relate to user input.

If an input point is a function call, its call-site vertex usually does not have any forward data dependencies in the SDG, since the returned value of the function has a separate node because of interprocedural dependencies. To handle this, in our representation, each input point has a *generate*



```
int add(int a, int b) {
  return a + b;
}

void main() {
  int sum, i;
  sum = 0;
  i = 1;
  while (i<11) {
    sum = add(sum, i);
    i = add(i,1);
  }
  printf(”%d\n”, sum);
  printf(”%d\n”, i);
}
```

**Figure 3. CodeSurfer's System Dependence Graph of an example source code [4].** The SDG represents the source code on the right side. Nodes are program points such as call-sites, assignments, return statements, etc. and edges are inter/intra procedural data/control dependencies.

*set* containing the nodes that are not connected to the input call-site with data dependence edges, but are directly affected by the I/O operation. For instance, a generate set of a `scanf` call contains the parameter variables of the call site. Generate sets can be used to track the points in the SDG where the content of an input-related buffer is copied into another buffer with standard library functions such as `strcpy` or `strcat`. Since standard library functions are not defined in the user's program code, these functions require special attention for static analyzers. CodeSurfer offers a sophisticated library model to handle common used library functions, however to keep our algorithms general we follow these operations manually.

## 3.2. Metrics

### 3.2.1. Input coverage.

**3.2.1. Input coverage.** *Input coverage* is used to describe the percentage of statements in a function that are tainted by user input. The formal definition is the following:

$$Coverage(f_j) = \frac{|\bigcup_{i=1}^{n} L_{IO}(p_i, f_j)|}{|L(f_j)|}$$

where $p_i$ as a node of the SDG is one of the $n$ input points, $f_j$ is a function of total $m$ functions, $L_{IO}(p_i, f_j)$ is the set of statements in $f_j$ along the forward data dependence chain of $p_i$ input point and $L(f_j)$ is the set of all statements in $f_j$.

The definition can be extended to cover the full source code of a program:

$$Coverage = \frac{\sum_{j=1}^{m}\left(|\bigcup_{i=1}^{n} L_{IO}(p_i, f_j)|\right)}{\sum_{j=1}^{m}|L(f_j)|}$$

It is important to notice that CodeSurfer's SDG contains many additional nodes (like pseudo variables because of global variables, or splitted statements because of AST normalization). Additional nodes may be particularly relevant in case of global variables as described in [7] where the authors state that number of nodes per line of code may vary dramatically because of pseudo variables. Therefore, using the conventional definition of statement coverage would result in false measurements, so instead of calculating statement coverage, we measure line coverage. The definition of line coverage is the same as that for statement coverage except that $L_{IO}(p_i, f_j)$ stands for the set of lines containing statements in $f_j$ along the path of $p_i$ input point and $L(f_j)$ stands for the lines of $f_j$.

**3.2.2. Input distance.** While input data travels from statement to statement in the control or data flow graphs, the data might be modified and reassigned to new variables. If input data or a variable is modified many times after reading it, developers may handle it less carefully or they may even forget the origin of the actual data. Using dataflow analysis it is possible to tell how many times the input data is modified or gets re-assigned to another variable. Thus, it is possible to compute the distance between an input point and the entry point of a function along the data dependence chains of the input in the SDG. The formal definition is:

$Distance(p_i, f_j)$: number of SDG nodes on the shortest path (for only data dependence edges) from $p_i$ input point to the entry point of $f_j$ function.

Input data may travel on different paths from its input point to a destination point. Selection statements and loops may cause branches along the path of the input data undergoing scrutiny. Inside a loop statement the variable that stores the input data may be modified several times and static analyzers cannot determine how many times the loop body will be executed at runtime. To eliminate the effect of loops and branches, we measure the length only for the shortest path in the SDG. An illustration can be seen in Figure 4.

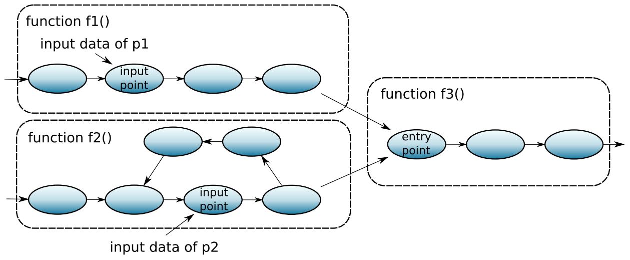


**Figure 4. Illustration of the distance metric.** The graph is a portion of an SDG showing only data dependence edges. $p_1$ is an input point in function $f_1$ and $p_2$ is an input point in $f_2$. $Distance(p_1, f_3) = 3$, $Distance(p_2, f_3) = 2$. Inside function $f_2$ there is a loop, but for calculating distance we count nodes only along the shortest path

This metric can be used to answer two questions:

⇒ How far has a datum travelled from its original input point?

⇐ From how far does an input-related function get its input?

## 3.3. Fault detection

Using our metrics we can determine a software system's critical (dangerous) functions. These functions, which must be handled more carefully during code inspection and testing, are more likely to contain faults that threaten a system's reliability or security. Once these functions are determined, we apply automatic fault detection algorithms to them.

**3.3.1. Buffer overflow detection with path sensitivity and pattern matching.** Livshits et al. [26] published a technique to locate buffer overflow and format string vulnerabilities in C programs by tracking pointers with path and context sensitivity. The novel approach in their method was their precise and fast pointer analysis, which made it possible to analyze large projects (the largest project they analyzed was `pcre` with about 13,000 LOC) quickly. With this technique they could track the path of input data and warn when the data was written into a buffer with a statically declared size.

Our technique implements a similar fault detection method that uses the SDG extracted from the source code to track the path of input data and, by simple pattern matching, locates `strcpy`-kind functions along the paths that allocate buffers on the stack. These functions, such as `strcpy`, `strcat` and `sprintf` write the contents of the input data into a buffer without first performing bounds checking.

**3.3.2. Format string vulnerability detection.** The recognition of format string vulnerabilities can be similar to buffer overflow faults even if the two different types of vulnerabilities have different technical backgrounds. In case of format string fault a mishandled buffer (related to user input) is used as format string of a function from the `printf` family. If an attacker can insert special format characters into the format string of a vulnerable function, he may even execute malicious code during program execution. The same technique used to locate buffer overrun errors can be used to locate format string faults. As opposed to looking for function calls of the `strcpy` family, our technique looks for call-sites with system functions of the `printf` family. If the format argument of such a function is related to the user input, it is a potential format string fault, unless the content of the variable was checked.

**3.3.3. Buffer overflow detection with taintedness checking.** After implementing our version of the buffer overflow detection technique, which was based on the algorithm described in [26] (Section 3.3.1), we realized that many of the reported faults were false positives. The most common reason for these false positives was that, before the problematic statement there were conditions that checked the size of the buffer.

To eliminate these false positives from set of reported warnings, we extended the algorithm with a technique usually referred as *taint checking* [21] or *taintedness detection* [32]. The main idea of this technique is to mark the data that may cause errors as *tainted* and follow its state during execution or in the program flow.

For instance, suppose that we mark the variables or buffers that store data from a user's input as 'tainted'. Whenever the value of the variables or buffers are assigned to a new variable or copied to a new buffer, the new variable or buffer is also marked tainted. Writing the data from a tainted buffer to another buffer without bounds checking is a dangerous operation, which generates a warning. However, if there is a selection statement (*e.g.* an `if` statement) which checks the size of the tainted buffer before it is copied to a new buffer, this selection statement untaints the copy operation and the new buffer. Tracking these selection statements which untaint further variables and string operations can be realised using control dependcies of the SDG in addition to data depencies.

**Algorithm 1** Buffer overflow detection with taintedness checking. (Described in Section 3.3.3)

```
procedure $DetectBufferOverflow(IP, V)$
Input: $IP$ set of input points
Output: $V$ set of potential buffer overflow statements

1: $V \leftarrow \emptyset$
2: $InputRelated \leftarrow GetDataDeps(IP)$ {Get the set of input related vertices}
3: $StrLens \leftarrow StrLenCalls(IP)$ {Get the set of strlen calls along the SDG paths starting from points in $IP$ and following forward data dependencies}
4: $StrLenVars \leftarrow GetDirectDataDeps(StrLens)$ {Get the variables directly depending on the return expressions of calls in $StrLens$ }
5: $NextNodes \leftarrow IP$ {Ordered list of next nodes to visit}
6: $AlreadyVisited \leftarrow \emptyset$ {Start a preorder traversal on SDG following forward control dependencies}
7: while not $Empty(NextNodes)$ do
8:   $n \leftarrow First(NextNodes)$
9:   $Remove(NextNodes, n)$
10:  if $n \in AlreadyVisited$ then
11:    {this node was already visited, simply remove it from $NextNodes$}
12:  else
13:    if $IsControlStatement(n)$ AND
       $UsesVarIn(StrLenVars)$ AND
       $n \in InputRelated$ then
14:      {this node dissolves tainted state of data, skip its subtree}
15:    else
16:      if $IsStrcpyCall(n)$ AND $n \in InputRelated$ then
17:        $V \leftarrow V \bigcup n$ {This is a potential Buffer Overflow!}
18:      end if
19:      $PutFirst(NextNodes, GetDirectControlDeps(n)$
         {Get the next nodes in SDG following forward control dependencies}
20:    end if
21:    $AlreadyVisited \leftarrow AlreadyVisited \bigcup n$
22:  end if
23: end while
```

Algorithm 1 gives a formal description of our algorithm, which works as follows: We first get the set of input-related vertices from the SDG and then traverse over the data dependence edges to locate `strlen` calls after the input statements. A `strlen` call is usually not used directly in a condition but its return value is stored in a variable that is used later. To handle these cases, we get the list of variables that depend on the return value of the `strlen` calls and use this set of vertices for taintedness checking. After calculating these sets, we start a pre-order traversal to walk over the control dependencies of the input statements. When we visit a control statement (this is usually an `if` statement, but it can be any kind of selection statement) and the condition of the control statement uses a variable that depends on a `strlen` call, we skip walking over the subtree of this control statement. However, when we visit a call site of a `strcpy`-kind function, and this function was already marked as input-related, then we mark this node as a place of potential buffer overflow.

## 4. Results

In this section we present results of our metrics and algorithms on open source software. We analyzed 12 security-critical products and we scanned a total 811,072 lines of C source code. The largest project we analyzed was `pidgin` with 229,825 lines of code. The full list of projects can be seen in Table 1.

The analyzed projects are security critical in the sense that they are common targets of attackers, since most of the projects are ftp, imap or irc daemons. These daemons usually have a user management and sometimes even anonymous users can log in and send input data remotely to the daemon running on a server. If the daemon has a security vulnerability, a malicious user may easily get access to the server.

Our buffer overflow detection algorithm extended with taintedness checking produced 10 warnings for the analyzed systems (8 times for Cyrus Imapd, 1 for Eggdrop and 1 for Pidgin) and 6 of these warnings belonged to the same statement in a function along different input paths of Cyrus Imapd. We have manually evaluated all of these warnings and we have found 3 warnings representing real buffer overflow faults: 2 faults in the code of Cyrus Imapd and another one in Pidgin. These faults were not critical security threats as they were not exploitable by malicious remote users, but they could cause unexpected behaviour during program execution. In addition to the warnings of deployed algorithms we used our metrics for further inspections as we describe it later in a case study (Section 4.1).

We note that because these open source projects are common targets of attacks they are also often the subjects of security research and analyses. As a result, common mistakes and errors are discovered as soon as a new version is

| project | description | LOC |
|---|---|---|
| pidgin-2.4.1 | chat client | 229825 |
| cyrus-imapd-2.3.12p2 | imap daemon | 170875 |
| irssi-0.8.12 | irc client | 71253 |
| openssh-5.0p1 | ssh daemon/cl. | 62251 |
| Unreal3.2.7 | irc daemon | 59196 |
| eggdrop1.6.19 | irc robot | 58468 |
| ircd-hybrid-7.2.3 | irc daemon | 53427 |
| proftpd-1.3.2rc1 | ftp daemon | 39122 |
| wzdftp-0.8.3 | ftp daemon | 34897 |
| pure-ftpd-1.0.21 | ftp daemon | 14798 |
| vsftpd-2.0.6 | ftp daemon | 12378 |
| bftpd | ftp daemon | 4582 |

**Table 1. List of analyzed open source projects.**

released. Widely used tools such as hybrid ircd or proftpd are also well tested. These projects are good subjects to locate vulnerabilities that are hard to find for static analyzers because the new vulnerabilities detected in this software are faults that were probably not reported by other analysis tools.

## 4.1. Pidgin Case Study

In this case study we detail the steps of our analysis (metrics, and fault detection) and demonstrate the effectiveness of our approach by applying our technique to a selected open source software, Pidgin.

**4.1.1. Overview of Pidgin and the analyzer system.** From the projects listed in Table 1 we have chosen Pidgin as the subject of a case study because of its size and popularity. Pidgin is the largest project we analyzed with 7,173 functions and 229,825 lines of C source code. CodeSurfer requires that it compiles the full source code before it analyzes it so it can build a proper ASG of the full project. We compiled Pidgin with the default configure parameters and it took CodeSurfer and GCC 31 minutes to compile and analyze the source code of Pidgin. In addition to this time, our plugin required another 1 minute to compute the metrics and perform fault detections. Compiling Pidgin simply with GCC using the same configuration options took around 8 minutes for the same system. Our analysis was conducted on an Intel Dual Xeon 2.8Ghz system with 3G memory and Ubuntu 8.04 (Hardy Heron) installed on it. CodeSurfer's settings for the analysis were the default settings, but we used `-cfg-edges both` and `-basic-blocks yes` to have additional detailed control flow information for further manual inspection.

**4.1.2. Input points.** In Pidgin we have found 99 input points in total (Table 2). Most of these are `read` calls used to read a buffer from a file descriptor, but there are many `fread` and `fgets` functions as well. We did not find dangerous input functions like `gets` or `scanf(''%s'', str)`.

| name | occurrences |
|---|---|
| read() | 55 |
| fread() | 12 |
| fgets() | 10 |
| gg_read() | 9 |
| gethostname() | 6 |
| getpwuid() | 2 |
| fscanf() | 1 |
| getenv() | 1 |
| getpass() | 1 |
| char *argv[] | 1 |
| int argc | 1 |

**Table 2. Input points in Pidgin.** First column is the name of input statement and the second column shows the number of occurrences of the actual statement. (`gg_read()` is an internal function to read data from SSL sockets.)

**4.1.3. Metrics.** The total input coverage of the source code is 10.56% while the mean value of the input coverage for all functions is 9.67%. The function with the highest coverage has 84.62% input coverage while 2,728 functions involve user input. The list of functions with top 10 coverage values are shown in Table 3.

After manual inspection we noticed that most of these functions were used to clean up memory after using the input related buffers (their naming conventions show this: `*_free`, `*_destroy`) and we did not find faults in these functions. However, we note that besides the top functions there are 379 functions with at least 50% coverage and only 65 of them have more than 20 lines of code. We did not inspected all of these functions, but we noticed that with our technique we could lower the number of functions that required inspection during a code review to a small value.

The longest distance an input travels from its input point through the dataflow is 100 vertices, which is high compared to the other analyzed projects (the average value of longest distances for all analyzed projects was 44.08 and 9 projects of 12 were below 50). In this particular case, the input statement was a `fgets` system call that read a buffer with a limited size from an input file. Along the path of the same input there is a total of 365 functions involved. This distance is high even inside the project itself, as the

| function | lines | coverage |
|---|---|---|
| yahoo_roomlist_destroy | 12 | 83.33 |
| aim_info_free | 13 | 84.62 |
| s5_sendconnect | 22 | 77.27 |
| purple_ntlm_gen_type1 | 35 | 77.14 |
| gtk_imhtml_is_tag | 91 | 76.92 |
| jabber_buddy_resource_free | 25 | 72.00 |
| peer_oft_checksum_destroy | 8 | 75.00 |
| qq_get_conn_info | 12 | 75.00 |
| _copy_field | 8 | 75.00 |
| qq_group_free | 8 | 75.00 |

**Table 3. List of top 10 input coverage values of functions in Pidgin.**

average of longest distances for different input points (average of $max\{Distance(p_i, f_1), \ldots, Distance(p_i, f_n)\}$ values for all $p_i$) is only 12.98 in Pidgin.

There is another interesting top value in Pidgin related to the function that is involved in input data of the most input points (it can be calculated by counting $p_i$ points with $Distance(p_i, f_j) > 0$ values for all $f_j$ functions and taking the maximum of these values). In Pidgin we found a function that works with input data coming from 31 different input points in the source code. This function is called `gg_debug` and is used for internal debugging purposes, which explains the high number of related input points since the function is called with string parameters in many different contexts.

**4.1.4. Fault detection.** We analyzed Pidgin with the format string detection (Section 3.3.2) and the buffer overflow detection with taintedness checking algorithms (Section 3.3.3). The evaluated algorithms produced only one warning on file `libpurple/protocols/zephyr/ZVariables.c` for a `strcpy` function call. After a manual inspection we found that this call is a fault that is related to a buffer overflow. The fault is detailed in Figure 5.

```
25: char *ZGetVariable(var)
26:     char *var;
27: {
28:     char varfile[128], *ret;
29:
30:     if (get_localvarfile(varfile))
31:         return ((char *)0);
...
42: }
...
114: static int get_localvarfile(bfr)
115:     char *bfr;
116: {
117:     const char *envptr;
118: #ifndef WIN32
119:     struct passwd *pwd;
120:     envptr = purple_home_dir();
121: #else
...
127: #endif
128:     if (envptr)
129:         (void) strcpy(bfr, envptr);
130:     else {
131: #ifndef WIN32
132:         if (!(pwd = getpwuid((int) getuid()))) {
133:             fprintf(stderr, "Zephyr␣....");
134:             return (1);
135:         }
136:         (void) strcpy(bfr, pwd->pw_dir);
137: #endif
138:     }
...
143: }
```

**Figure 5. A buffer overflow fault in Pidgin.** Vulnerable `strcpy` is in line 136 of file libpurple/protocols/zephyr/ZVariables.c

In the source code snippet in Figure 5 our fault detection produced a warning for the `strcpy` call in line 136 of function `get_localvarfile`. This function copies the content of the buffer `pwd->pw_dir` into the destination buffer pointed by `bfr` without performing bounds checking. `bfr` is a pointer parameter of this function, but when the function is called from `ZGetVariable`, `bfr` points to `varfile` which is a statically allocated string buffer with maximum of 128 characters (line 28). The content of `pwd->pw_dir` is set in line 132, and it contains the name of the home directory of the current user. If the length of this directory name exceeds 128 characters, the `strcpy` call produces a *segmentation fault*.

## 5. Related Work

Security analysis is an important area in research. Many static analysis tools appeared recently to help companies develop more reliable and safe systems by automating testing, code review, and source code auditing processes during the development cycle. There are different solutions for different languages like CodeSonar tool of GrammaTech and PCLint[2] for C/C++, CheckStyle[3] or PMD[4] for Java or FXCop developed by Microsoft for C# and there are multi front-end solutions like Columbus developed by FrontEndART Ltd. Many of these tools are able to find certain rule violations and they can show potential faults to developers, but only some of them are able to locate security errors. Chess et al. published a brief overview of security analysis tools comparing their benefits in a paper [8] and Tevis et al. published a similar comparison of security tools [30].

[2]http://www.gimpel.com/
[3]http://checkstyle.sourceforge.net/
[4]http://pmd.sourceforge.net/

Since Aleph1 published his exploiting technique [1] against buffer overflow vulnerabilities, researchers published many methods to detect these kind of vulnerabilities. Most of these approaches work with dynamic bounds checking techniques [27, 23, 33, 11]. Static techniques were also published based on integer-range analysis [31], annotation-assisted static analysis technique [13] or on pointer analysis techniques [26, 5]. However tests and comparisons of these techniques and their related tools show that it is still hard to locate these kind of errors and usually these techniques still have many false positive warnings [33]. A comparison of available exploiting, defending and detecting techniques was published by Lhee et al. in [24].

Focusing on user-related input is usually an important idea behind static security analyzers and it is also common to work with a graph representation that can be used to track control and data dependcies. Scholz et al. describe an approach in [28] to identify security vulnerabilities via user-input dependence analysis. In their technique they map user-input dependency test to a graph reachability problem which can be solved with simple graph traversal algorithms. Hammer et al. presents another notable technique in [15] that is closely related to our work since they perform security analysis based on PDGs. Their work is about information flow control, which is a technique for discovering security leaks in software. This technique is closely related to tainted variable analysis.

Our approach uses an input coverage metric to show developers which functions involve user input. Using coverage metrics is common in testing and there are tools that can measure this value at run time. However our approach computes this metric statically from the SDG of the program. Our coverage metric can be also described as a coupling metric, which measures the coupling of functions to the input variables in the source code. A similar idea has been presented by Harman et al. [16], where the authors propose a coupling metric to measure how information flows between functions.

## 6. Conclusions and Future Work

Locating security faults in complex systems is difficult and there are only few effective automatic tools available to help developers. In this paper we presented an approach to automatically locate input-related security faults (buffer overflow and format string vulnerabilities) and help developers locate security vulnerabilities by marking parts of the source code that involve user input. Our technique has three main steps: (1) locate input points, (2) calculate metrics to determine a set of dangerous functions, (3) perform automatic fault detection to identify security faults.

We presented the results of applying our technique on open source software and presented a case study on Pidgin as the largest and most popular software we analyzed. We found security faults in Pidgin and in other analyzed software. Our fault detection techniques focused on buffer overflows and format string vulnerabilities, which are the most common input-related faults in C source code. Our approach is novel as it uses input coverage and distance metrics to show developers the list of functions that are the most likely to contain potential security faults. The Pidgin case study demonstrates the effectiveness of our metrics. Pidgin has a total number of 7,173 functions and 229,825 LOCs. According to our measurements, only 10.56% of the code is related directly to user input and 2,728 functions work with input data. Limiting the number of dangerous functions and code that is affected by user input is important in reducing the effort of a code review.

As future work we plan to improve our current fault detection methods and implement new ones to find more vulnerabilities in the source code automatically. In addition to the improvement of fault detections, another important way to continue our work is to implement a ranking technique that operates on the list of dangerous functions so developers can focus on the top ranked functions during a manual code inspection.

Further possibilities are to extend our technique to additional languages where input-related security faults are common reasons of security vulnerabilities (*e.g.,* C++, Java).

## 7. Acknowledgements

Author Csaba Nagy was sponsored by HAESF[5] (Hungarian American Enterprise Scholarship Found) and CIEE[6] (Council on International Educational Exchange) during this research at the Software Engineering Research Group[7] of Drexel University. We are pleased to thank the support of HAESF, CIEE and the help of all the members of SERG in this work.

This research was supported in part by the Hungarian national grants RET-07/2005, OTKA K-73688, and TECH_08-A2/2-2008-0089.

## References

[1] E. L. (Aleph1). Smashing the stack for fun and profit. *Phrack*, 49, November 1996.

[2] P. Anderson. Codesurfer/path inspector. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, page 508, Washington, DC, USA, 2004. IEEE Computer Society.

[5]http://www.haesf.org
[6]http://www.ciee.org
[7]http://serg.cs.drexel.edu

[3] P. Anderson, D. Binkley, G. Rosay, and T. Teitelbaum. Flow insensitive points-to sets. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:0081, 2001.

[4] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering (CAV 2001)*, Paris, France, July 2001.

[5] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a c pointer analysis. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 332–341, New York, NY, USA, 2005. ACM.

[6] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, New York, NY, USA, 1993. ACM.

[7] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2):8, 2007.

[8] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[9] B. Chess and J. West. *Secure programming with static analysis*. Addison-Wesley Professional, 2007.

[10] Coverity, Inc. Opensource report, 2008.

[11] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 162–171, New York, NY, USA, 2006. ACM.

[12] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: an analysis of the internet virus of november 1988. In *IEEE Computer Society Symposium on Security and Privacy*, pages 326–343. IEEE Computer Society Press, 1989.

[13] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19(5):87–96, 1994.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[15] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *ISOLA '06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 119–128, Washington, DC, USA, 2006. IEEE Computer Society.

[16] M. Harman, M. Okulawon, B. Sivagurunathan, and S. Danicic. Slice-based measurement of function coupling. *IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution ( PMESSE'97)*, pages 28–32, May 1997.

[17] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, New York, NY, USA, 1992. ACM.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[19] M. Howard, D. LeBlanc, and J. Viega. *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, 2005.

[20] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[21] J. Kong, C. C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24, New York, NY, USA, 2006. ACM.

[22] D. J. Kuck, Y. Muraoka, and S.-C. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Trans. Comput.*, 21(12):1293–1310, 1972.

[23] E. Larson and T. Austin. High coverage detection of input-related security facults. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[24] K.-S. Lhee and S. J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33(5):423–460, 2003.

[25] S. Lipner. The trustworthy computing security development lifecycle. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[26] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. *SIGSOFT Softw. Eng. Notes*, 28(5):317–326, 2003.

[27] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[28] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 25–34, Beijing, September 2008.

[29] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[30] J.-E. J. Tevis and J. A. Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 197–202, New York, NY, USA, 2004. ACM.

[31] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[32] J. Xu and N. Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[33] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.