AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair

Zhiqiang Lin†, Xuxian Jiang‡, Dongyan Xu§, Bing Mao†, and Li Xie†

†Dept. of Computer Science
Nanjing University, China
linzq@dislab.nju.edu.cn

‡Dept. of Information and Software Engineering
George Mason University, USA
xjiang@gmu.edu

§CERIAS and Dept. of Computer Science
Purdue University, USA
dxu@cs.purdue.edu

ABSTRACT
Software patch generation is a critical phase in the life-cycle of a software vulnerability. The longer it takes to generate a patch, the higher the risk a vulnerable system needs to take to avoid from being compromised. However, in practice, it is a rather lengthy process to generate and release software patches. For example, the analysis on 10 recent Microsoft patches (MS06-045 to MS06-054) shows that, for an identified vulnerability, it took 75 days on average to generate and release the patch.

In this paper, we present the design, implementation, and evaluation of AutoPaG, a system that aims at reducing the time needed for software patch generation. In our current work, we mainly focus on a common and serious type of software vulnerability: the out-of-bound vulnerability which includes buffer overflows and general boundary condition errors. Given a working out-of-bound exploit which may be previously unknown, AutoPaG is able to catch on the fly the out-of-bound violation, and then, based on data flow analysis, automatically analyzes the program source code and identifies the root cause – vulnerable source-level program statements. Furthermore, within seconds, AutoPaG generates a fine-grained source code patch to temporarily fix it without any human intervention. We have built a proof-of-concept system in Linux and the preliminary results are promising: AutoPaG is able to successfully identify the root cause and generate a source code patch within seconds for every vulnerability test in the Wilander’s buffer overflow benchmark test-suite. In addition, the evaluation with a number of real-world out-of-bound exploits also demonstrates its effectiveness and practicality in automatically identifying (vulnerable) source code root causes and generating corresponding patches.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution and Maintenance; D.4.6 [Operating Systems]: Security and Protection

General Terms
Security, Management

Keywords
Out-of-bound Vulnerability, Data Flow Analysis, Automated Patch Generation, Software Security

1. INTRODUCTION
Software today is getting increasingly complicated. For instance, the Windows XP released in 2001 contains more than 45 million lines of code and the Windows Server 2003 has over 50 million lines of code [1]. Such complexity inevitably introduces software vulnerabilities, evidenced by the fact that CERT [2] published 5,990 new vulnerabilities in year 2005, about 1.5 times of the number published in year 2004. Moreover, we have witnessed an alarmingly decreased time window between the release of vulnerability information and the appearance of attack code exploiting that vulnerability. The Blaster worm [3] (August 2003) attacks a Microsoft security flaw which was announced nearly 1 month earlier. The Sasser worm [4] (May 2004) exploits another Microsoft security flaw for which Microsoft issued a patch less than 3 weeks ago. The Witty worm [5] (March 2004) targets a buffer overflow vulnerability in several Internet Security Systems (ISS) intrusion detection software, only 1 day after the patch was released. Even worse, most recently, there have been a flurry of zero-day exploits that attack a variety of software, including the Windows Graphics Rendering Engine [6] (December 2005), Windows Word [7] (May 2006), Excel [8] (June 2006), and PowerPoint [9] (July 2006). Note that these zero-day exploits are disclosed before the corresponding patches are made available.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission granted to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

In this paper, zero-day exploits are defined as those exploits that are released before or on the same day when the vulnerability or the vendor patch is released to the public.
A number of execution context snapshots taken whenever a systems provide run-time patches that are essentially based on execution to an uninfected state. As a result, these systems abort the current transaction and roll back the execution if a snapshot of current state of the program execution when a buffer overflow attack) is detected, they will immediately derive a software patch on detected exploits by instrumenting means, an authorized patch writer needs to laboriously familiar with the discovered vulnerability and possible exploits further confirms its effectiveness and practicality in automatically identifying (vulnerable) source code level and automatically generate a source patch within seconds. Given a working out-of-bound exploit which may be previously unknown, AutoPaG is able to catch on the fly the out-of-bound violation, and then, based on data flow analysis, automatically analyzes the program source code and identifies the root cause - vulnerable (source-level) program statements. Furthermore, within seconds, AutoPaG generates a fine-grained source patch to temporarily fix it without any human intervention. We point out that the root cause identification at the source code level and the generation of a temporary patch could greatly help an authorized patch writer to generate the official patch.

We have created a proof-of-concept system in Linux called AutoPaG that aims at significantly reducing the time needed for source patch generation. In our current work, we mainly focus on a common and serious category of software vulnerability: the out-of-bound vulnerability, including buffer overflows and general boundary condition errors. Given a working out-of-bound exploit which may be previously unknown, AutoPaG is able to catch on the fly the out-of-bound violation, and then, based on data flow analysis, automatically analyzes the program source code and identifies the root cause - vulnerable (source-level) program statements. Furthermore, within seconds, AutoPaG generates a fine-grained source patch to temporarily fix it without any human intervention. We point out that the root cause identification at the source code level and the generation of a temporary patch could greatly help an authorized patch writer to generate the official patch.

We have developed AutoPaG with the Wilander’s buffer overflow benchmark test-suite [30] as well as a number of real-world out-of-bound exploits against widely deployed software (e.g., wu-ftp). The results are encouraging: for every vulnerability test in the Wilander’s test-suite, AutoPaG is able to successfully identify the root cause at the source code level and automatically generate a source patch within seconds. Also, the evaluation with five real-world out-of-bound exploits [11-15] further confirms its effectiveness and practicality in automatically identifying (vulnerable) source code root cause and generating source patches.

The rest of this paper is organized as follows: Sections 2

<table>
<thead>
<tr>
<th>Advisory</th>
<th>CVE.#</th>
<th>Vulnerability Phased</th>
<th>Patch Released</th>
<th>Interval (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS06-054</td>
<td>CVE-2006-0001</td>
<td>11-09-2005</td>
<td>09-12-2006</td>
<td>307</td>
</tr>
<tr>
<td>MS06-053</td>
<td>CVE-2006-0032</td>
<td>11-30-2005</td>
<td>09-12-2006</td>
<td>286</td>
</tr>
<tr>
<td>MS06-052</td>
<td>CVE-2006-3412</td>
<td>07-07-2006</td>
<td>09-12-2006</td>
<td>67</td>
</tr>
<tr>
<td>MS06-051</td>
<td>CVE-2006-3648</td>
<td>07-17-2006</td>
<td>08-08-2006</td>
<td>22</td>
</tr>
<tr>
<td>MS06-050</td>
<td>CVE-2006-3086</td>
<td>06-19-2006</td>
<td>08-08-2006</td>
<td>50</td>
</tr>
<tr>
<td>MS06-049</td>
<td>CVE-2006-3444</td>
<td>07-07-2006</td>
<td>08-08-2006</td>
<td>32</td>
</tr>
<tr>
<td>MS06-048</td>
<td>CVE-2006-3590</td>
<td>07-14-2006</td>
<td>08-08-2006</td>
<td>25</td>
</tr>
<tr>
<td>MS06-047</td>
<td>CVE-2006-3449</td>
<td>07-17-2006</td>
<td>08-08-2006</td>
<td>32</td>
</tr>
<tr>
<td>MS06-046</td>
<td>CVE-2006-3357</td>
<td>07-06-2006</td>
<td>08-08-2006</td>
<td>33</td>
</tr>
<tr>
<td>MS06-045</td>
<td>CVE-2006-3281</td>
<td>06-28-2006</td>
<td>08-08-2006</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 1: The time-lines of 10 recent Microsoft patches (MS06-045 to MS06-054) that are released between August and September 2006.
presents an overview of AutoPaG while the detailed design is described in Section 3. Section 4 provides the evaluation based on the Wilander's buffer overflow benchmark as well as five real-world out-of-bound exploits. Section 5 examines its limitations and possible countermeasures, followed by a discussion of related work in Section 6. Finally, Section 7 concludes.

2. AUTOPAG OVERVIEW

Figure 1 shows the overall architecture of AutoPaG that consists of three main components: (1) The out-of-bound detector instruments the source code of a vulnerable program and will capture ongoing out-of-bound exploits; (2) The root cause locator, upon the detection of an exploit, will examine the faulty instructions as well as related context information to identify the root cause – the vulnerable source code statements; (3) The source patch generator will transform the vulnerable source code by generating a vulnerability-specific source code patch, hence preventing the same vulnerability from being exploited. The patch is then integrated back to the detector so that the same vulnerability will not be reported again.

In the following subsections, we describe in detail the techniques used by these three components: Section 2.1 describes the detection approach used by the detector to capture out-of-bound exploits. The algorithm used by the locator to accurately identify the responsible vulnerable statements will be discussed in Section 2.2. Finally, Section 2.3 presents how a vulnerability-fixing patch is generated.

2.1 Detecting Out-of-Bound Exploits with Bounds Checking

The detector is responsible for capturing ongoing exploits. Note that there already exist a number of approaches that can be potentially used for exploit detection, such as system randomization (e.g., address space layout randomization (ASLR) [21, 22], instruction set randomization [19, 20], system call randomization [23], and N-variant systems [24]), taint analysis (e.g., TaintCheck [37], Vigilante [44], and Argos [48]), and bounds checking [25, 27, 28, 29, 51]. These approaches have different pros and cons. For example, the ASLR randomizes the memory layout of a running process, which makes it hard for an exploit to accurately locate the injected attack code and existing program code (e.g., libc functions), hence preventing the attack from successfully hijacking the control flow. Unfortunately, it will also crash the running program in the presence of an attack, and fails to provide sufficient information to trace back to the vulnerable instruction(s). For example, the context information, including the call stack, can be lost or completely destroyed by the attack [46, 47]. The instruction-level taint analysis such as TaintCheck [37] can identify the vulnerable instruction at the machine instruction level. However, it cannot provide semantic-level information related to the attack. Note that the goal of our detector is to provide useful information at the source code level for a detected attack. For this reason, existing bounds check systems [25, 51] can be used as the basis for our detector.

```
1 #include <string.h>
2 int main(int argc, char **argv) {
3     char buf[4];
4     char *p;
5     p = buf;
6     strcpy(p, argv[1]);
7     return 0;
8 }
```

Figure 2: An example code containing an out-of-bound vulnerability.

However, the original bounds checking approach is still not sufficient for our purpose. Particularly, when an out-of-bound access is detected, we would like to obtain further information about the attack, e.g., which variable or data is overflowed? What are those statements making the out-of-bound access possible? Note that such information is crucial to the subsequent automated root cause (source code level) identification. As an example, Figure 2 shows a classic piece of code that contains an out-of-bound vulnerability or, more specifically, a buffer overflow vulnerability. Suppose this vulnerable program is executed with a long string argument (more than 4 bytes) that will overflow the local variable buf, our detector needs to return the access statement directly causing the out-of-bound violation, i.e., line 6 strcpy(p, argv[1]), the local pointer variable p, as well as buf for our later use. We will describe its design in detail in Section 3.1.

2.2 Identifying Root Cause with Data Flow Analysis

After an out-of-bound violation is detected, we first determine the variable that is overflowed by this violation. By leveraging this variable, we further reason about the root cause behind the out-of-bound violation. In particular, our system finds out those program source code statements that contribute to the computation of the overflowed variable. For instance, if the overflowed variable is a pointer, we need to find out its declaration statement, its scope and aliases, as well as possible references and dereferences. These statements need to be examined to eliminate the detected out-of-bound vulnerability as they provide important information on where the overflowed variable comes from (the declaration statement) and how it is processed (e.g., its references, dereferences, and aliases) etc. In other words, as these statements contribute to the vulnerability behind the violation, they should be revised and patched. To this end, we propose
Algorithm 1 Calculate $sSet, eSet$

Require: $sSet \neq \emptyset$

1: $eSet \leftarrow \{e_0 | S(e_0) \in sSet \lor OutOfBound(e_0)\}$
2: $sSet \leftarrow s_0Set$
3: while $sSet$ keeps adding do
4: Visiting Stmt $S(S)$
5: match $S$ with
6: DECLARE$(e_i) \leftarrow e_i \in cSet \rightarrow sSet \leftarrow sSet \cup \{S\}$
7: ASSIGN$(e_i,e_j) \leftarrow e_i \in cSet \rightarrow eSet \leftarrow eSet \cup \{e_j\}$, $sSet \leftarrow sSet \cup \{S\}$
8: UNOP$(e_i) \leftarrow (deref(e_i) \lor ref(e_i)) \land e_i \in eSet \rightarrow sSet \leftarrow sSet \cup \{S\}$
9: BINOP$(e_i,e_j) \leftarrow ((deref(e_i) \lor ref(e_i)) \land e_i \in eSet) \lor ((deref(e_i) \lor ref(e_i)) \land e_j \in eSet) \rightarrow sSet \leftarrow sSet \cup \{S\}$
10: CALL$(f,e)$: CallByRef$(e_i) \land e_i \in eSet \rightarrow sSet \leftarrow sSet \cup \{S\}$, notCStdFun$(f) \land CallByRef$(e_i) \land e_i \in eSet \rightarrow Visiting Stmt (f.body)$
11: end match
12: end while
13: output $sSet, eSet$

A data flow analysis algorithm outlined in Algorithm 1.

The goal of the algorithm is to calculate the vulnerability-relevant statements (as the tainted set). It starts from the overflowed variable as well as the initial access statement causing the out-of-bound violation (provided by our detector). In our notation, $sSet$ stands for a set of tainted statements contributing to the vulnerability (e.g., the initial access statement causing the out-of-bound violation), as well as other statements that contain the references/dereferences of the tainted variables. $eSet$ is a set of tainted variables including the overflowed variable and its aliases etc; $s_0Set$ is the initially identified access statement causing the out-of-bound violation; $S$ represents a source code statement while $e_i$ or $e_j$ refers to a variable that is included in the statement $S$. For the convenience of analysis, each tainted statement contains the corresponding location information while each tainted variable contains its scope information. Algorithm 1 repeatedly examines every statement in the source code until no additional tainted statement is added to $sSet$. Specifically, our algorithm differentiates different types of program statements.

- **Declaration Statement**: If the variable declaration statement (DECLARE$(e_i)$) contains a tainted variable ($e_i \in eSet$), then this statement is included ($sSet \leftarrow sSet \cup \{S\}$) for further analysis. The reason is that from the declaration statement, we can infer the allocated buffer size of the declared variable.

- **Assignment Statement**: If the $bvalue (e_i)$ of assignment operation (ASSIGN$(e_i,e_j)$) is tainted, then its original source $e_j$ as well as the corresponding statement are also tainted ($eSet \leftarrow eSet \cup \{e_j\}$, $sSet \leftarrow sSet \cup \{S\}$). Essentially, this match operation is used to capture the sources of the tainted pointers (from aliases). Note that the string handling glibc functions (e.g., strcpy, strcat, and memcpy) are considered as the assignment operations; therefore, when the destination parameter of these functions is tainted, its source and other corresponding arguments are also tainted.

- **Unary/Binary Operation Statement**: If a tainted variable (e.g., a global variable) is used in different statements, we need to identify all of their uses because these statements may contribute to the propagation of the tainted variable. This is achieved by checking all of unary operations (UNOP$(e_i)$): if the associated operation is a pointer reference ($ref(e_i)$) or dereference ($deref(e_i)$)), then the statement is tainted ($sSet \leftarrow sSet \cup \{S\}$). Similarly, for every binary operation (BINOP$(e_i,e_j)$), if $e_i$ or $e_j$ is tainted, this statement is also tainted.

- **Function Call Statement**: If the tainted pointer variable is passed to a function, we also examine this function. Particularly, if we check the function invocation statement CALL$(f,e)$ and find one of its arguments is tainted ($e_i \in eSet$) and called by reference (CallByRef$(e_i)$), then the function invocation statement is tainted ($sSet \leftarrow sSet \cup \{S\}$). In addition, if the function called is not a standard C library function (notCStdFun$(f)$), then the called function body will be included for further analysis (Visiting Stmt $(f.body)$). We exclude the C standard library function mainly because it is only their unsafe usage (e.g., no argument bounds checking for strcpy) that leads to the security vulnerabilities, and the functions themselves are considered as safe (containing no bug in them).

To better understand the algorithm, we use the code shown in Figure 2 as a simple illustration. As described in Section 2.1, when we run this program with an argument of a long string (more than 4 bytes), the detector will report an out-of-bound violation caused by: line 6 strcpy$(\ p, \ argv[1])$. After that, our data flow analysis is invoked to calculate $sSet$ and $eSet$. Initially, the pointer $p$
is tainted since the out-of-bound write takes place at the address to which pointer \( p \) points, and \( \text{argv}[1] \) is tainted since \texttt{strcpy} equals to the \texttt{ASSIGN} operation. Next, because the pointer \( p \) is the alias of \( \text{buf} \), the variable \( \text{buf} \) is also considered as tainted. Lastly, it finds these two variables' declarations (char \( \text{buf}[4] \), char \( *p \)) which are also tainted. The calculated results of \( s\text{Set} \) and \( e\text{Set} \) for this example are presented in Table 2.

Table 2: Root cause identification for the out-of-bound vulnerability in Figure 2.

<table>
<thead>
<tr>
<th>( s\text{Set} )</th>
<th>\texttt{strcpy(( p, \text{argv}[1] ))}</th>
</tr>
</thead>
<tbody>
<tr>
<td>3: char ( \text{buf}[4] )</td>
<td></td>
</tr>
<tr>
<td>4: char ( *p )</td>
<td></td>
</tr>
<tr>
<td>5: ( p = \text{buf} )</td>
<td></td>
</tr>
<tr>
<td>6: \texttt{strcpy(( p, \text{argv}[1] ))}</td>
<td></td>
</tr>
<tr>
<td>( e\text{Set} )</td>
<td>main: ( p )</td>
</tr>
<tr>
<td></td>
<td>main: ( \text{buf} )</td>
</tr>
<tr>
<td></td>
<td>main: \text{argv}[1]</td>
</tr>
</tbody>
</table>

In general, Algorithm 1 needs to scan the source code (in terms of its intermediate representation) a few passes to calculate \( s\text{Set} \) and \( e\text{Set} \) (the last pass is used to determine whether there is an addition to \( s\text{Set} \)). The \( s\text{Set} \) contains only a few statements partially due to the observation that related statements tend to be grouped together (e.g., inside a function), which is confirmed by our experiments with real-world programs. However, in the worst case, \( s\text{Set} \) might contain every statement.

2.3 Preventing Out-of-Bound Exploits with Automated Source Patch Generation

After identifying the initial access statement (\( s_0\text{Set} \)) as well as the relevant statements (\( s\text{Set} \)), our system will automatically derive a source patch that prevents the identified vulnerability from being exploited.

Consider how an official patch is manually developed for an out-of-bound vulnerability: The patch writer will first identify the exact location of the vulnerability, then determine the size of the vulnerable buffer, and finally rewrite some part of the program (e.g., replace \texttt{strcpy} with \texttt{strncpy}) to eliminate the out-of-bound error. Our source patch generation is motivated by this manual process and is developed to automate it without human intervention. In particular, the source patch will truncate any (illegal) out-of-bound writes, and ensure the out-of-bound reads to be within the bound. With the identified out-of-bound vulnerability in Figure 2 as an example, our source patch \texttt{generator} will replace \texttt{strcpy(\( p, \text{argv}[1] \))} as \texttt{strncpy(\( p, \text{argv}[1,4] \)). The detailed design on how to achieve this will be described in Section 3.3.

3. DETAILED DESIGN

3.1 Out-of-Bound Detector

Our \texttt{detector} component captures out-of-bound violation by instrumenting the program source code with necessary run-time bounds checking. Specifically, we leverage the CCured memory safety system [51], which infers and divides all program pointers into three main categories: \texttt{SAFE} pointers (for pointers without casts or pointer arithmetic), \texttt{SEQ}/\texttt{FSEQ} pointers (for pointers involved in pointer arithmetic but not in pointer casts), and \texttt{WILD} pointers (for pointers involved in pointer casts, in particular the arbitrary casts). Note that the bounds checking code can obtain the related meta-data information, e.g., the size information of these pointers [51]. At runtime, the instrumented code ensures that \texttt{SEQ}/\texttt{FSEQ} pointers never go out of bound and \texttt{WILD} pointers do not clobber the memory of other objects. If the original program contains an out-of-bound vulnerability, the instrumented code will detect the out-of-bound access, report an access violation, and abort or stop current process execution.

However, the basic bounds checking capability is still insufficient for our purpose. Particularly, we require other information related to a detected access violation, including which statement causes the violation and which variable or data is overflowed by the violation. We point out that CCured will report the occurrence of a violation and the detected location in the source code. However, it does not pinpoint the overflowed data and the identified location might not be accurate. For example, for the vulnerability shown in Figure 2, it will report that “Failure UBOUND at lib/ccuredlib.c:3941: _read_at_least_f()”, and it is located at “lib/ccuredlib.c:3941”. Therefore, we provide our own instrumentation code to obtain the overflowed variable/data and derive the accurate location information. Specifically, we replace the original memory checking library functions (in \texttt{ccuredcheck.h} and \texttt{ccuredlib.c}) and the associated wrapper functions (e.g., wrappers for \texttt{strcpy}) with our own, such that, if an out-of-bound violation is caused by the current program implementation and the initially identified location is located in an external library function, our implementation will further traverse the call-stack to accurately locate the calling location of the invoked external function in the program source code.

As a simple illustration, we again use the example code shown in Figure 2. When we test this program with a malicious parameter (e.g., “aaaaaaaaaaaaa”), our \texttt{detector} will successfully intercept the out-of-bound write, and it then traverses the runtime call-stack (shown in Figure 3) from the most-inner helper function \texttt{ccured_fail_str} to the outer wrapper function \texttt{strcpy_wrapper_sff}, which is called by the \texttt{main} function. Note that the out-of-bound write is taking place in the \texttt{strcpy_wrapper_sff} function and our \texttt{detector} then further infers that the statement \texttt{test.c:6 strcpy(\( p, \text{argv}[1] \))} is the one that directly triggers the out-of-bound violation.

3.2 Root Cause Locator

After the overflowed variable and the access statement causing the out-of-bound violation are identified, the root cause \texttt{locator} will use them to determine those tainted sets of statements and variables. Based on the data flow analysis algorithm (shown in Algorithm 1), we need to scan the program’s source code a number of rounds until there is no additional statement that will be considered as tainted.

Specifically, given an intermediate (compiler-generated) representation of a program, our \texttt{detector} works as follows: (i) First, with the input (i.e., \( s_0\text{Set}, e\text{Set} \)) provided by our \texttt{detector}, the \texttt{locator} examines the high level syntax tree and visits every statement. (ii) Second, on the basis of the data flow analysis algorithm (Algorithm 1), it taints visited statements and their variables if they are related to the detected vulnerability. (iii) Thirdly, it repeats the whole process until there is no additional tainted statement. Based on the observation that most of the identified statements...
## 3.3 Source Patch Generator

Once the tainted sets of statements and variables are identified, our source patch _generator_ will rewrite some of them and attempt to automatically repair the vulnerability. In the following, we describe how the _generator_ calculates the vulnerable buffer boundaries and deals with the out-of-bound accesses (i.e., read or write):

- **Determining vulnerable buffer boundaries** Since the buffer sizes pointed by many pointers cannot be statically determined, we need to instrument necessary code to dynamically determine them. Fortunately, our _detector_ already calculates the boundary information when detecting possible out-of-bound violation. We can simply re-use the same code and add bound-fixing instrumentation code to correct it. To be more specific, we extend the CCured’s implementation by associating related meta-data (e.g., the beginning addresses and ending addresses for SEQ/FSEQ pointers) to those pointers that need to be analyzed (i.e., in _eSet_). Based on these meta-data, we can then properly handle possible out-of-bound accesses.

- **Fixing out-of-bound reads** For an out-of-bound read, our current prototype redirects the read to a value located within the buffer boundary. For example, suppose p[i] (e.g., i = 10) is an out-of-bound read (it does not necessarily mean that every p[i] is unsafe!). We will redirect p[i] as p[i mod size], where the _size_ is calculated by (x,e−x.b) / sizeof(τ), where τ is the type being read, x.b (the beginning address) and x.e (the ending address) are the associated meta-data for the destination buffer x. Intuitively, any value can be used for the redirection. However, in practice, it is possible that the redirected value may be used to as a con-

---

**Figure 3:** The call stack information when detecting the out-of-bound violation in Figure 2.

**Table 3:** Root cause identification for the out-of-bound vulnerability in Figure 4.

<table>
<thead>
<tr>
<th>sSet</th>
<th>392: memcpy(bss_buffer, overflow_buffer, overflow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>390: void * addr = &amp;choice;</td>
<td></td>
</tr>
<tr>
<td>391: overflow = 4;</td>
<td></td>
</tr>
<tr>
<td>392: memcpy(bss_buffer, overflow_buffer, overflow)</td>
<td></td>
</tr>
<tr>
<td>393: *bss_pointer = bss_buffer[0];</td>
<td></td>
</tr>
<tr>
<td>394: /* Overwritten data from bss_buffer is copied to where</td>
<td></td>
</tr>
<tr>
<td>395: the bss_pointer is pointing */</td>
<td></td>
</tr>
<tr>
<td>396: *bss_pointer = bss_buffer[0];</td>
<td></td>
</tr>
<tr>
<td>397: }</td>
<td></td>
</tr>
<tr>
<td>398: else printf(&quot;Attack form not possible\n&quot;)</td>
<td></td>
</tr>
<tr>
<td>399: return;</td>
<td></td>
</tr>
</tbody>
</table>

**eSet**

| 375: static long bss_buffer[BUFSIZE];           |
| 376: static long *bss_pointer;                  |
| 377: char propolice_dummy_2[10];                |
| 378: int overflow;                              |
| 379: void * addr = &choice;                     |
| 380: if (((choice == 11) &
| 381: ((long)bss_pointer > (long)propolice_dummy_2)) |
| 382: /* First set up overflow buffer with the address of the |
| 383: shellcode, a few 'A's and a pointer to the return address*/ |
| 384: overflow = (int)((long)bss_pointer - (long)bss_buffer) + 4; |
| 385: memset(overflow_buffer+1, 'A', overflow-8); |
| 386: overflow_buffer[overflow/4-1] = (long)(addr-1); |
| 387: overflow_buffer[overflow/4-1] = (long)(addr-1); |
| 388: /* Then overflow bss_buffer with overflow_buffer */ |
| 389: memcpy(bss_buffer, overflow_buffer, overflow); |
| 390: *bss_pointer = bss_buffer[0];               |
| 391: }                                            |
| 392: else printf("Attack form not possible\n")   |
| 393: return;                                     |

---

The following code example illustrates the redirection process when p[i] is an out-of-bound read.

```c
void vuln_bss_return_addr(int choice) { /* Attack form 4(a)*/
    static char propolice_dummy_2[10];
    static long bss_buffer[BUFSIZE];
    static long *bss_pointer;
    char propolice_dummy_1[10];
    int overflow;
    void * addr = &choice;
    if (((choice == 11) &&
        ((long)bss_pointer > (long)propolice_dummy_2))
        /* First set up overflow buffer with the address of the shellcode, a few 'A's and a pointer to the return address*/
        overflow = (int)((long)bss_pointer - (long)bss_buffer) + 4;
        memset(overflow_buffer+1, 'A', overflow-8);
        overflow_buffer[overflow/4-1] = (long)(addr-1);
        /* Then overflow bss_buffer with overflow_buffer */
        memcpy(bss_buffer, overflow_buffer, overflow);
        *bss_pointer = bss_buffer[0];
    } else printf("Attack form not possible\n")
    return;
}
```

---

**Figure 4:** One of the buffer overflow vulnerabilities in Wilander’s test-suite.

![Figure 4](image-url)

belong to one or a few functions (the locality property), we have optimized our _locator_ by scanning within an identified function a few passes until there is no addition to the _sSet_ before examining other functions. Note our current prototype is implemented on the intermediate representation provided by CIL [50]. However, it is certainly applicable to other compiler systems including gcc.

To provide a detailed illustration of how our _locator_ works, we use a real world example – a vulnerable program in the Wilander’s benchmark test-suite. Figure 4 only shows those lines of code that are related to our discussion. More specifically, it contains an out-of-bound vulnerability (at line 392) that overflows the _bss_buffer_ variable. Once this line is identified, our _locator_ first taints the overflowed data _bss_buffer_ as well as the corresponding parameters _overflow_buffer_ and _overflow_. After that, the function _vuln_bss_return_addr_ is examined and the data flow analysis is repeatedly applied to locate those tainted statements and variables. Eventually, we identify 12 lines of code in _sSet_ and 7 tainted variables in _eSet_. The results of the tainted sets are shown in Table 3.

### 3.3 Source Patch Generator

Once the tainted sets of statements and variables are identified, our source patch _generator_ will rewrite some of them and attempt to automatically repair the vulnerability. In the
After having fixed the identified vulnerable statements, the generator will automatically compile the patch code together with other unaffected code to produce a new executable that is not vulnerable to the detected exploit. Note that our bounds fixing scheme can be directly applied to a program’s intermediate representation. If a statement is vulnerable, the intermediate representation can be directly repaired with our bound-fixing instrumentation code. If it is not vulnerable, its current representation will remain intact.

4. EVALUATION

We have created a proof-of-concept system in Linux. To verify the effectiveness and responsiveness of our system, we have deployed it in our lab and conducted a number of experiments. We used a buffer overflow benchmark test-suite developed by Wilander et al. [30], as well as five additional real-world exploits [11-15] in our evaluation. These experiments are performed in a machine with two 2.4G Pentium processors and 1G RAM running the Linux kernel 2.6.3 operating system. The vulnerable programs are transformed with CIL 1.3.5 and CCured 1.1.2 (with Ocaml 3.09.0) and compiled with gcc 4.0.

4.1 Effectiveness

4.1.1 Wilander’s Benchmark Test-Suite

There exists 18 different buffer overflow attacks in the publicly available Wilander’s Benchmark Test-Suite. Based on the overwritable buffer locations and exploitation techniques, these 18 test cases can be mainly classified into four categories: (1) The first category overflows a stack-based buffer all the way to an attack target (that can be either a return address, the old base pointer, a function pointer, or even a longjmp buffer); (2) The second category overwrites a heap/bss-based buffer all the way to an attack target; (3) The third category attacks a stack-based pointer so that it points to an attack target; (4) The forth category fills a heap/bss-based pointer with a location that points to an attack target. Table 4 reports these 18 test cases. In particular, it highlights the attack target addressed by each test case. Interested readers are referred to [30] for more details.

AutoPaG is able to successfully detect (the 3rd column of Table 4) all exploitation attempts introduced by the benchmark. Moreover, for each detected attack, AutoPaG automatically identifies a set of source-level statements that are

<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Attack Targets</th>
<th>Detector (Detected?)</th>
<th>Locator (#LOCs)</th>
<th>Generator (#LOCs)</th>
<th>Prevented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer overflow on stack</td>
<td>Return address</td>
<td>✓</td>
<td>8</td>
<td>21</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>‘Old base pointer’</td>
<td>✓</td>
<td>10</td>
<td>22</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Function pointer as local variable</td>
<td>✓</td>
<td>11</td>
<td>24</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Function pointer as parameter</td>
<td>✓</td>
<td>12</td>
<td>25</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Longjmp buffer as local variable</td>
<td>✓</td>
<td>10</td>
<td>28</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Longjmp buffer as function parameter</td>
<td>✓</td>
<td>15</td>
<td>32</td>
<td>✓</td>
</tr>
<tr>
<td>Buffer overflow on heap/bss</td>
<td>Function pointer</td>
<td>✓</td>
<td>13</td>
<td>26</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Longjmp buffer</td>
<td>✓</td>
<td>13</td>
<td>29</td>
<td>✓</td>
</tr>
<tr>
<td>Buffer overflow of pointers on stack</td>
<td>Return address</td>
<td>✓</td>
<td>14</td>
<td>27</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>‘Old base pointer’</td>
<td>✓</td>
<td>15</td>
<td>30</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Function pointer as variable</td>
<td>✓</td>
<td>14</td>
<td>29</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Function pointer as function parameter</td>
<td>✓</td>
<td>15</td>
<td>36</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Longjmp buffer as variable</td>
<td>✓</td>
<td>13</td>
<td>31</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Long jmp buffer as function parameter</td>
<td>✓</td>
<td>15</td>
<td>30</td>
<td>✓</td>
</tr>
<tr>
<td>Buffer overflow of pointers on heap/bss</td>
<td>Return address</td>
<td>✓</td>
<td>12</td>
<td>36</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>‘Old base pointer’</td>
<td>✓</td>
<td>14</td>
<td>38</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Function pointer as variable</td>
<td>✓</td>
<td>14</td>
<td>29</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Longjmp buffer as variable</td>
<td>✓</td>
<td>16</td>
<td>28</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4: The effectiveness of AutoPaG with Wilander’s benchmark test-suite.
4.2 Responsiveness

We measure the responsiveness of AutoPaG by counting the time needed for source code root cause identification (done by our detector and locator) and source patch generation (done by our generator). We also measure the time needed to recompile (by gcc) the patched program source code, and compare it with the time needed to compile the original unpatched source code. These results are presented in Table 7. We observe the whole process (the 4th column) only take tens of seconds or even seconds to complete. When compared with the time needed for normal program compi-
ation (the 5th column), the overall process including the identification of source code root cause, source patch generation time, and additional recompilation time only causes 4.1 times slowdown on average. Most importantly, the whole process can be automatically conducted without any human intervention. Meanwhile, we point out that our total response time could be further reduced if we keep the compiled objects of those unaffected files. The reason is that our generated patch only affects a very small number of files, hence significantly reducing the time for the recompilation.

4.3 Performance

4.3.1 Performance of the Detector

For the performance overhead of our detector, we use the set of software in Table 6 for our evaluation. As these software provide various network-oriented services, we measure the response time while requesting large files from them. For instance, we request a file using the ftp protocol from wu-ftpd, while requesting another file with the same size using the http protocol from Lhttpd, Ghttpd, Mathopd and ATPhp. Note that all of these software are instrumented with our detector to detect possible out-of-bound attacks.

4.3.2 Performance of the Generated Patch

Micro-evaluation We measure the performance slowdown for those affected functions that will be “patched” by the generated source patch. Figure 6(a) shows the affected functions in the experimented software and the corresponding performance degradation. We observe that our instrumented code only imposes small overhead: most of them incur the slowdown of less than 5% while the worst case – fb_re alph path function – incurs 25% slowdown.

Macro-evaluation We also measure the performance impact of our generated patch on the application as a whole. Existing bounds checking systems [25, 27] usually impose significant performance overhead due to the need of extensively checking every related function. However, since our patch only checks those identified vulnerable statements, high performance cost should not occur. The result of our measurement is described in Figure 6(b). As expected, our patch only imposes very small overhead – from 0% to 5%.

5. DISCUSSION

In the following, we examine the limitations of the AutoPaG prototype and discuss possible counter-measures.

- First, the current prototype only addresses one common and severe vulnerability – the out-of-bound vulnerability – and does not consider other types (e.g., format string bugs, integer overflows, and race condition errors). The development of next-generation AutoPaG should take these vulnerabilities into account. We point out that different vulnerabilities may require different mechanisms or even different methodologies for accurate detection and repair. For example, exploiting a format string vulnerability can be detected and prevented using a dynamic checking scheme [31], instead of the bound checking approach described in this paper. To this end, we can extend the AutoPaG prototype with a more modular architecture: the detector accommodates a number of vulnerability-specific sub-detectors, each of which is responsible for a particular type of vulnerabilities (e.g., out-of-bound vulnerabilities, format string bugs); the locator can then be further enhanced to identify the related source statement for all supported vulnerabilities; and the generator needs to take into consideration the specifics of detected vulnerabilities (possibly with additional context information) to generate effective vulnerability-specific counter-measures.

- Second, our current system requires access to a program’s source code. Consequently, it may not be possible for AutoPaG to generate source patch for other third-party proprietary software. However, we note
that one ultimate goal of this work is to develop and release AutoPaG so that others, including software vendors, can benefit from this work.

- Third, though the generated source patch has shown effectiveness in detecting and preventing the target vulnerability, it may require additional rigorous regression test before public release or deployment in production systems. Note that there exist a number of systems [40] that are capable of performing automated regression testing. However, the scope or coverage of regression testing may be highly specific to different applications and their deployment environments.

6. RELATED WORK

In recent years, extensive research work has been carried out on how to defend against fast vulnerability exploits. In this section, we do not intend to examine all of them. Instead, we mainly compare those that are most related to ours, and we divide these related work into three main categories: proactive source transformation, just-in-time execution filtering, and reactive runtime patching.

Proactive Source Transformation: This approach instruments the original program source code with additional attack-resilient code so that it can detect, mitigate, or even recovery from an ongoing attack. Failure-oblivious computing (FOC) [49], and DIRA [43] are two well-known examples.

FOC [49] leverages the CRED safe-C compiler [27] to instrument the program source code so that it can capture run-time memory errors. Note that the original CRED safe-C compiler will terminate the execution of the program once a memory error is detected. FOC extends it so that instead of terminating the execution, it discards illegal memory writes and returns a predetermined sequence of values for illegal memory reads. Its main purpose is to allow a program to continue its execution even in the presence of buffer overflow attacks. Similarly, DIRA [43], implemented as an extension to the gcc compiler, transforms the program source code so that it can maintain a memory update log for a running program. Based on the memory update log, if a control-hijacking attack [43] is detected, it can roll back the memory state so that the memory contaminated by the attack can be restored.

However, due to the need of heavily instrumenting source code for proactive detection of future attacks, both FOC and DIRA impose considerably high performance overhead (e.g., 1X-8X slowdown in FOC). Most importantly, they intend to dynamically recovery from an ongoing attack, not to investigate the vulnerability behind the attack or provide additional leads in deriving an ultimate patch to fix it, which is the main focus of our system.

Just-In-Time Execution Filtering: This approach typically keeps track of the propagation of tainted information (e.g., network input) at the machine instruction level and detects the presence of an attack if current execution (e.g., the EIP register) somehow points to the tainted data. Note that the associated taint analysis algorithm can be further extended to derive a vulnerability-specific signature for just-in-time execution filtering. A number of systems have been developed in this category, including TaintCheck [37], DACODA [38], VSEF [39], Vigilante [44], and Argos [48].

TaintCheck [37] performs a dynamic taint analysis at the instruction level so that it can follow the propagation of network input data (that is considered as tainted), and then raise an alert when the tainted data is directly or indirectly executed. Based on the tainted network input data that eventually leads to the alert, TaintCheck also derives a semantic-aware attack signature for later execution filtering. The follow-up work on VSEF [39] takes a step further by avoiding the need to monitor every instruction. Instead, it only monitors and instruments those instructions that are related to the exploited vulnerability.

DACODA [38], Vigilante [44], and Argos [48] also take a similar approach. DACODA monitors the execution flow of the whole system, and correlates the network input to control flow change that can be used to infer the existence of an attack. Vigilante tracks the flow of information from network inputs to data used in attacks, and further develops the notion of self-certifying alerts (SCAs) that can be
shared over the network without requiring recipients to trust each other. Argos uses dynamic taint analysis to detect exploits in the whole system, different from TaintCheck that is performed only for an application. Note that a major concern in these systems is the performance overhead due to the need for tracking every machine instruction without efficient hardware support.

AutoPaG takes a different approach from these systems. Instead of focusing on the detection and prevention of an attack at the machine instruction level, AutoPaG is more intended to automatically walk through the program source code and then identify and patch those relevant source statements that directly or indirectly “contribute” to the detected vulnerability.

**Reactive Runtime Patching:** Upon the detection of an ongoing attack, this approach can patch current program execution (e.g., instructions or states) so that it can recover from the attack. Sidiropoglou and Keromytis et al. [40] first proposed the notion of automatic patch generation and extensively explored its feasibility. For example, the DYBOC [41] system instruments parts of the application’s source code which may be vulnerable to buffer overflow attacks, and the instrumentation code will recovery from detected attacks via a so-called function call transaction mechanism. The STEM [42] system takes a step further by selectively emulating the identified vulnerable code segments. The emulation allows for a vulnerable program to restore or rollback the memory changes performed within the faulty functions.

AutoPaG has a different goal. Instead of patching current execution during runtime to recovery from an attack, AutoPaG focuses on the vulnerability exploited by the attack by locating those relevant source code statements and generating a patch at the source code level. Note that an existing software vulnerability will ultimately require a source patch to fix it, which is the intended goal of AutoPaG.

### 7. CONCLUSION

In this paper, we present the design, implementation, and evaluation of AutoPaG, a system proposed to reduce the long delay in software patch generation. Given a working out-of-bound exploit (e.g., a buffer overflow attack) which may be previously unknown, AutoPaG is able to catch on the fly the out-of-bound violation, and automatically walks through the program source code and identifies the root cause – vulnerable program source statements. Furthermore, within seconds, AutoPaG automatically generates a fine-grained source patch. The evaluation using the Wildaner’s buffer overflow benchmark as well as a number of real-world exploits successfully demonstrates its effectiveness and responsiveness.

### 8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive and helpful feedbacks and suggestions. This work was supported in part by the Chinese National Science Foundation (No.60373064) and the Chinese National 863 High-Tech Program (No.2003AA144010), and by a gift from Microsoft Research and grants from the U.S. National Science Foundation (OCI-0438246, OCI-0504261, CNS-054617).


