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.

ASIACCS'07, March 20-22, 2007, Singapore.

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 [1] 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.

[1]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 are released to the public.

Advisory	CVE#	Vulnerability Phased	Patch Released	Interval (days)
MS06-054	CVE-2006-0001	11-09-2005	09-12-2006	307
MS06-053	CVE-2006-0032	11-30-2005	09-12-2006	286
MS06-052	CVE-2006-3442	07-07-2006	09-12-2006	67
MS06-051	CVE-2006-3443	07-07-2006	08-08-2006	32
	CVE-2006-3648	07-17-2006	08-08-2006	22
MS06-050	CVE-2006-3086	06-19-2006	08-08-2006	50
	CVE-2006-3438	07-07-2006	08-08-2006	32
MS06-049	CVE-2006-3444	07-07-2006	08-08-2006	32
MS06-048	CVE-2006-3590	07-14-2006	08-08-2006	25
	CVE-2006-3449	07-07-2006	08-08-2006	32
MS06-047	CVE-2006-3649	07-17-2006	08-08-2006	22
MS06-046	CVE-2006-3357	07-06-2006	08-08-2006	33
MS06-045	CVE-2006-3281	06-28-2006	08-08-2006	41

Table 1: The time-lines of 10 recent Microsoft patches (MS06-045 to MS06-054) that are released between August and September 2006.

If we examine the life-cycle of a software vulnerability, it can be roughly divided into three main phases: vulnerability discovery, patch generation, and patch installation. Among the three phases, software patch generation is critical as it provides the ultimate fix for a discovered vulnerability. The longer it takes to generate a patch, the more 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, especially in the face of the above emerging threats. Table 1 shows the time-lines of 10 recent Microsoft patches (MS06-045 to MS06-054) [10]: after a vulnerability is identified and reported, it took a month or even longer (75.46 days on average for the examined 10 patches) to generate and release the patch.

The long delay in generating and releasing software patches is partially due to the current manual patch generation process and significant challenges for patch writers. In addition to the stringent requirement of being intimately familiar with the discovered vulnerability and possible exploitation means, an authorized patch writer needs to laboriously go through related source code, precisely identify and correct those vulnerable statements, and then derive an efficient patch. After that, the patch should also go through a rigorous regression test phase to evaluate its robustness and compatibility before finally releasing it to public.

A number of systems (e.g., [18, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 44, 46, 47, 48]) have been built to mitigate the risks introduced by the long delay. Among the most notable, Sidiroglou and Keromytis et al. [40] proposed the notion of automatic patch generation and explored the feasibility in their follow-up works (e.g., DYBOC [41], STEM [42], and Application Communities [45]) to automatically derive a software patch on detected exploits by instrumenting (vulnerable) programs. In particular, considering every function execution as a transaction (in a manner similar to a sequence of operations in database), these systems will take a snapshot of current state of the program execution when a new transaction begins. Later, if an ongoing exploit (e.g., a buffer overflow attack) is detected, they will immediately abort the current transaction and roll back the execution to its enclosing transaction, hence "patching" the defected execution to an uninfected state. As a result, these systems provide run-time patches that are essentially based on a number of execution context snapshots taken whenever a potential vulnerable transaction is invoked.

In this paper, we explore another alternative: instead of generating run-time patches, we aim to derive source code level software patches. There are at least two major differences between a source patch and a run-time patch: (1) The run-time patch is a temporary fix to a software vulnerability. To ultimately eliminate a vulnerability, a final source patch is still necessary to generate an official patch. (2) To enable the run-time patch, it is necessary to snapshot current program context whenever a new transaction is started. In other words, every potential vulnerable function invocation could result in a new snapshot being taken, which introduces relatively high performance overhead (e.g., 30% for STEM). For the source patch, once it is installed, it only incurs very limited or even no performance degradation.

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 evaluated 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-ftpd). 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

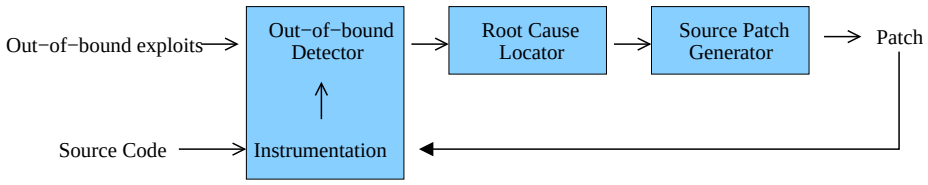


Figure 1: An overview of AutoPaG.

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: $s_0Set \neq \emptyset$
{Initializing $eSet$ with the detected out-of-bound variable e_0}
1: $eSet \Leftarrow \{e_0 | S(e_0) \in s_0Set \wedge OutOfBound(e_0)\}$.
{Initializing $sSet$ with the detected direct out-of-bound statement s_0Set}
2: $sSet \Leftarrow s_0Set$.
3: **while** $sSet$ keeps adding **do**
4: Visiting Stmt (S)
5: **match** S **with**
{If it is a declaration statement, add it to $sSet$ if it contains a tainted variable e_i.}
6: DECLARE(e_i): $e_i \in eSet \rightarrow sSet \Leftarrow sSet \cup \{S\}$.
{If it is an assignment operation $e_i = e_j$ and e_i is a tainted variable, add it to $sSet$ and update $eSet$ with e_j.}
7: ASSIGN(e_i,e_j): $e_i \in eSet \rightarrow eSet \Leftarrow eSet \cup \{e_j\}$, $sSet \Leftarrow sSet \cup \{S\}$.
{If it is a unary operation, add it to $sSet$ if it references/dereferences a tainted variable e_i.}
8: UNOP(e_i): $(deref(e_i) \vee ref(e_i)) \wedge e_i \in eSet \rightarrow sSet \Leftarrow sSet \cup \{S\}$.
{If it is a binary operation, add it to $sSet$ if it references/dereferences a tainted variable e_i.}
9: BINOP(e_i,e_j): $((deref(e_i) \vee ref(e_i)) \wedge e_i \in eSet) \vee ((deref(e_i) \vee ref(e_i)) \wedge e_j \in eSet) \rightarrow sSet \Leftarrow sSet \cup \{S\}$.
{If it is a function call, add it to $sSet$ if the parameter is passed by reference to a tainted variable e_i. Visit the function body if needed}
10: CALL(f, e_i): $CallByRef(e_i) \wedge e_i \in eSet \rightarrow sSet \Leftarrow sSet \cup \{S\}$, $notCStdFun(f) \wedge CallByRef(e_i) \wedge e_i \in eSet \rightarrow$ 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 *lvalue* (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_i) 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 that is 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 `argv[1]` is tainted since `strcpy` equals to the ASSIGN operation. Next, because the pointer `p` is the alias of `buf` (`p = buf`), the variable `buf` is also considered as tainted. Lastly, it finds these two variables' declarations (`char buf[4], char *p`) which are also tainted. The calculated results of $sSet$ and $eSet$ for this example are presented in Table 2.

s_0Set	`strcpy(p, argv[1])`

$sSet$	`3: char buf[4]`
	`4: char *p`
	`5: p = buf`
	`6: strcpy(p, argv[1])`

$eSet$	`main: p`
	`main: buf`
	`main: argv[1]`

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

In general, **Algorithm 1** needs to scan the source code (in terms of its intermediate representation) a few passes to calculate $sSet$ and $eSet$ (the last pass is used to determine whether there is an addition to $sSet$). The $sSet$ 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, $sSet$ might contain every statement.

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

After identifying the initial access statement (s_0Set) as well as the relevant statements ($sSet$), 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 `strcpy` with `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 *generator* will replace `strcpy(p, argv[1])` as `strncpy(p,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 *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: SAFE pointers (for pointers without casts or pointer arithmetic), SEQ/FSEQ pointers (for pointers involved in pointer arithmetic but not in pointer casts), and 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 SEQ/FSEQ pointers never go out of bound and 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 `ccuredcheck.h` and `ccuredlib.c`) and the associated wrapper functions (e.g., wrappers for `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.,"`aaaaaaaaaa`"), our *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 `ccured_fail_str` to the outer wrapper function `strcpy_wrapper_sff`, which is called by the `main` function. Note that the out-of-bound write is taking place in the `strcpy_wrapper_sff` function and our *detector* then further infers that the statement `test.c:6 strcpy(p, 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 *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 *detector* works as follows: (i) First, with the input (i.e., s_0Set, e_0Set) provided by our *detector*, the *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

```
#0  0x0804b0fb in ccured_fail_str (str=0x805cc73 "Ubound", file=0x805cc12 "lib/ccuredlib.c", line=3941,
        function=0x805daa5 "__read_at_least_f") at lib/ccuredlib.c:909
#1  0x0804b15d in ccured_fail (msgId=3, file=0x805cc12 "lib/ccuredlib.c", line=3941, function=0x805daa5
        "__read_at_least_f") at lib/ccuredlib.c:923
#2  0x0804fa0f in __read_at_least_f (ptr={_p = 0xbfaa9f90, _e = 0xbfaa9f94}, n=11) at lib/ccuredlib.c:3941
#3  0x0804fa75 in __copytags_ff (dest={_p = 0xbfaa9f90, _e = 0xbfaa9f94}, src={_p = 0xbfaabed2, _e =
        0xbfaabedd}, n=11) t lib/ccuredlib.c:3947
#4  0x0804a0dc in strcpy_wrapper_sff (dest=0xbfaa9f90 "", dest_e=0xbfaa9f94, src=0xbfaabed2 "aaaaaaaaaa",
        src_e=0xbfaabedd) at string_wrappers.h:79
#5  0x0804a006 in main (argc=2, __argv_input=0xbfaaa014) at test.c:6
```

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

```
373 void vuln_bss_return_addr(int choice)  { /* Attack form 4(a)*/
374   static char propolice_dummy_2[10];
375   static long bss_buffer[BUFSIZE];
376   static long *bss_pointer;
377   char propolice_dummy_1[10];
378   int overflow;
379
380   void * addr = &choice;
381
382 if ((choice == 11) &&
383     ((long)&bss_pointer > (long)&propolice_dummy_2)) {
384     /* First set up overflow_buffer with the address of the
385     shellcode, a few 'A's and a pointer to the return address */
386     overflow = (int)((long)&bss_pointer - (long)&bss_buffer) + 4;
387     overflow_buffer[0] = (long)&shellcode;
388     memset(overflow_buffer+1, 'A', overflow-8);
389     overflow_buffer[overflow/4-1] = (long)(addr-1);
390
391     /* Then overflow bss_buffer with overflow_buffer */
392     memcpy(bss_buffer, overflow_buffer, overflow);
393
394     /* Overwritten data from bss_buffer is copied to where
395        the bss_pointer is pointing */
396     *bss_pointer = bss_buffer[0];
397   }
398   else printf("Attack form not possible\n");
399   return;
400 }
```

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

s_0Set	392: memcpy(bss_buffer, overflow_buffer, overflow)
$sSet$	45: long overflow_buffer[OVERFLOWSIZE] 46: char shellcode[] = ''\xeb\x1f\...\xff/bin/sh'' 375: static long bss_buffer[BUFSIZE] 376: static long *bss_pointer 378: int overflow 380: void * addr = &choice 386: overflow = (int)((long)&bss_pointer - (long)&bss_buffer) + 4 387: overflow_buffer[0] = (long)&shellcode 388: memset(overflow_buffer+1, 'A', overflow-8) 389: overflow_buffer[overflow/4-1] = (long)(addr-1) 392: memcpy(bss_buffer, overflow_buffer, overflow) 396: *bss_pointer = bss_buffer[0]
$eSet$	global_variable: overflow_buffer global_variable: shellcode vuln_bss_return_addr: bss_buffer vuln_bss_return_addr: bss_pointer vuln_bss_return_addr: overflow vuln_bss_return_addr: addr vuln_bss_return_addr: choice

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

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 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-

Attack Type	Attack Targets	*Detector* (Detected?)	*Locator* (#LOCs)	*Generator* (#LOCs)	*Generator* (Prevented?)
Buffer overflow on stack	Return address	√	8	21	√
	Old base pointer	√	10	22	√
	Function pointer as local variable	√	11	24	√
	Function pointer as parameter	√	12	25	√
	Longjmp buffer as local variable	√	10	28	√
	Longjmp buffer as function parameter	√	15	32	√
Buffer overflow on heap/bss	Function pointer	√	13	26	√
	Longjmp buffer	√	13	29	√
Buffer overflow of pointers on stack	Return address	√	14	27	√
	Old base pointer	√	15	30	√
	Function pointer as variable	√	14	29	√
	Function pointer as function parameter	√	15	36	√
	Longjmp buffer as variable	√	13	31	√
	Longjmp buffer as function parameter	√	15	30	√
Buffer overflow of pointers on heap/bss	Return address	√	12	36	√
	Old base pointer	√	14	38	√
	Function pointer as variable	√	14	29	√
	Longjmp buffer as variable	√	16	28	√

Table 4: The effectiveness of AutoPaG with Wilander's benchmark test-suite.

dition to break out current `while()` loop. As a result, before instrumenting the read redirection, we need to first check the source code to determine whether the redirected value is used as a loop condition. If so, we will choose another value to avoid resulting in a dead loop. However, this value for the redirected read might introduce undesirable side-effects to current running program. As such, the automatically generated source patch will contain a side note, which needs to be manually resolved by the authorized patch writer. We point out that our evaluations so far (with 23 different vulnerability tests) have not encountered this issue.

- *Fixing out-of-bound writes* Similar to the above approach in fixing out-of-bound reads, fixing out-of-bound writes requires the additional bounds fixing code to discard those writes. More specifically, for an out-of-bound write, when the bound checking code finds it to be an out-of-bound write, the instrumentation code will either truncate the out-of-bound write, such as replacing `strcpy` with `strncpy`, `strcat` with `strncat`, etc., or silently do nothing (it is essentially equal to the truncation). For example, when executing an statement `*(p+i)=variable`, if `p` has already pointed to outside of its destination buffer boundary, the instrumented code can simply skip this statement. We also point out that there might exist a need to write a NULL value to the end of the pointed buffer if current out-of-bound write is related to a string type (similar to existing library functions for handling C strings).

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 [2]. 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

[2]Note the paper [30] presenting the test-suite described 20 different attacks while the publicly available program actually contains 18 of them.

CVE#	Program	Vulnerability Description
CVE-2002-1549	Lhttpd 0.1	Buffer overflow in `Log` function in `util.c`
CVE-2002-1816	ATPhttpd 0.4b	Buffer overflow in the `sock_gets` function in `sockhelp.c`
CVE-2002-1904	GazTek ghttpd 1.4	Buffer overflow in `Log` function in `util.c`
CVE-2003-1228	Mathopd 1.4p2	Buffer overflow in the `prepare_reply` function in `request.c`
CVE-2003-0466	Wu-ftpd 2.6.2	Buffer overflow in the `fb_realpath` function in `realpath.c`

Table 5: Evaluating AutoPaG with real-world software and their vulnerabilities.

Program	Total(#LOCs)	*Detector* (Detected?)	*Locator* (#LOCs)	*Generator* (#LOCs)	*Generator* (Prevented?)	*Locator* (#LOCs) / Total (#LOCs)
Lhttpd 0.1	893	√	14	87	√	1.568%
ATPhttpd 0.4b	1214	√	9	59	√	0.741%
GazTek ghttpd 1.4	837	√	14	87	√	1.673%
Mathopd 1.4p2	5027	√	19	320	√	0.380%
Wu-ftpd 2.6.2	19949	√	54	461	√	0.271%

Table 6: The effectiveness of AutoPaG with real world programs and their vulnerabilities.

responsible for the vulnerability exploited by the detected attack. With this set, a patch writer can significantly narrow down the source code he/she needs to examine and correct, reducing the time and efforts needed to generate a patch. As shown in the 4th column of Table 4, AutoPaG correlates each detected exploit to the set (in a size from 8 to 16) of related source statement. We manually examine the source code in the benchmark and the results confirms with the automated output from AutoPaG.

Table 4 (the 5th column) also shows the number of lines of code (LOCs) in the generated source patch by AutoPaG. To evaluate their effectiveness, we apply these source patches, compile them with the original programs, and repeat the same set of experiments. The results are encouraging: these automatically-generated patches are able to prevent all of these attacks!

Meanwhile, it is interesting to point out that the patched benchmark process will be unexpectedly terminated in 11 of these test cases. A detailed investigation shows that these unexpected terminations reveal a bug in the original benchmark implementation that will invoke an uninitialized function pointer that is supposed to be overwritten by the attack. As an example, our generated patch terminated the benchmark process when testing the vulnerability shown in Figure 4. The reason for the termination is due to the NULL dereference by the pointer variable `bss_pointer`. Within the `vuln_bss_return_addr` function, the `bss_pointer` variable is defined as an uninitialized static variable. As a result, it is considered as a bss-based variable, and will be initialized as zero by default. In the original benchmark implementation, the `vuln_bss_return_addr` function will wait for the `memcpy(bss_buffer, overflow_buffer, overflow)` function to overwrite this bss variable. However, due to the fact that our patch successfully prevents the overwriting attempt and hence the `bss_pointer` will remain as zero (a NULL pointer). Once it is being dereferenced, it will immediately cause the termination of the benchmark process. After we initialize this variable to a dummy function, the patched benchmark process will run normally and all of these attacks are successfully prevented.

4.1.2 Real-World Buffer Overflow Attacks

In this subsection, we further evaluate the effectiveness of AutoPaG with real-world buffer overflow attacks. We choose 5 of them , primarily because the source code as well as the attack code are publicly available from the Internet [11-15]. These vulnerable programs and the tested vulnerabilities are described in Table 5.

We run these vulnerable programs under AutoPaG and all of those exploitation attempts are successfully captured by our bounds checking based *detector*. For these applications, we find no false positives and false negatives, demonstrating the effectiveness of the bounds checking-based approach. To evaluate the effectiveness of our *locator*, we again use the size LOCs of the related source statement ($sSet$) as the metrics and the results are shown in Table 6.

By showing the total LOCs of the original program in Table 6, we can more easily compare the set size of the related source statement with the total size. Particularly, based on these numbers, it is likely that many software vulnerabilities might only involve a small piece of code. Consequently, if this small piece of code can be accurately identified, the patch writer will be relieved from the burden of examining other unrelated source code and then derive a patch in a more timely fashion.

Table 6 also contains the LOCs (the 5th column) of the source patches generated by AutoPaG. We would like to emphasize that the AutoPaG patch is not intended to serve as the official patch. Instead, it is our goal that the identification of only those related source code statements and the generation of a temporary source patch can provide convenient, timely hints for a patch writer to derive the official patch, hence reducing the time for patch generation and release.

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-

Program	Our System (seconds)			Compilation by `gcc` (seconds)	Ratio
	Locator & Generator	Recompilation by `gcc`	Total		
Lhttpd 0.1	2.381	0.761	3.142	0.711	4.4X
ATPhttpd 0.4b	2.154	0.884	3.038	0.847	3.6X
GazTek ghttpd 1.4	2.313	0.723	3.036	0.697	4.4X
Mathopd 1.4p2	6.157	2.372	8.529	2.326	3.8X
Wu-ftpd 2.6.2	23.096	7.726	30.822	7.514	4.1X

Table 7: Comparison of time cost between our system and gcc.

lation (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 ATPhttpd. Note that all of these software are instrumented with our *detector* to detect possible out-of-bound attacks.

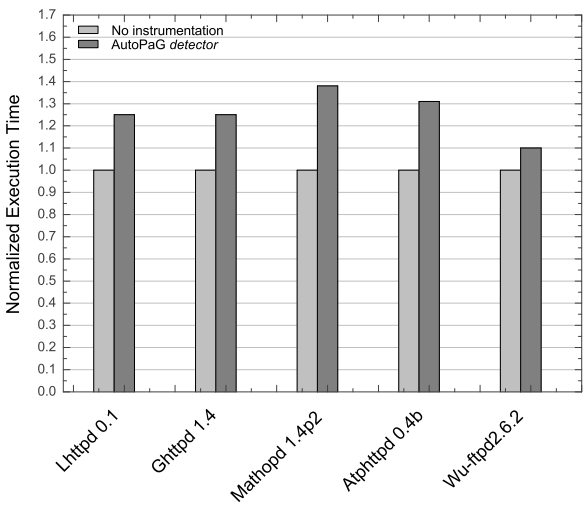


Figure 5: Performance evaluation of our detector.

Figure 5 shows the measurement results. We observe that our *detector*, which is primarily based on CCured, did degrade the performance from 1.1X to 1.4X for these tested 5 programs. However, we consider the slowdown acceptable as we only use them for attack detection and patch generation purpose, not in high-demanding production environments.

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_realpath` function – incurs 25% slowdown.

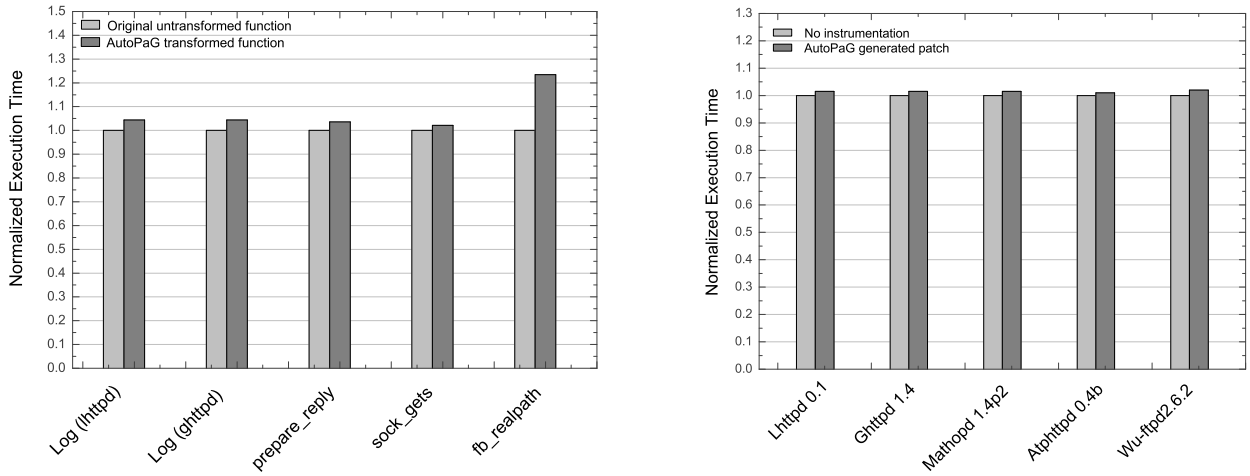
Macro-evaluation We also measure the performance impact of our generated patch on the application as a whole. Existing bounds checking systems [25, 27] usually impose significant performance overhead due to the need of extensively checking every related function. However, since our patch only checks 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



(a) Performance impact of the transformed functions

(b) Performance impact of our generated patch

Figure 6: Performance evaluation of the generated patches.

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 recovery from the attack. Sidiroglou 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 Wilander'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-0546173).

9. REFERENCES

[1] *http://www.microsoft.com/resources/design/windows.html*
[2] The CERT Coordination Center, *http://www.cert.org/*
[3] MSBlast Worms, *http://www.cert.org/advisories/CA-2003-20.html*.
[4] Sasser Worms, *http://www.microsoft.com/security/incident/sasser.asp*, May 2004.
[5] Witty Worms, *http://www.symantec.com/security_response/writeup.jsp?docid=2004-032009-1441-99*, March 2004.
[6] Windows WMF Zero-Day Attack, *http://www.counterpane.com/alert-cis-ra-0030-01.html*, December 2005
[7] Windows Word Zero-Day Attack, *http://www.eweek.com/article2/0,1895,1965042,00.asp*, May 2006
[8] Windows Excel Zero-Day Attack, *http://www.eweek.com/article2/0,1895,1978835,00.asp*, June 2006
[9] Windows PowerPoint Zero-Day Attack, *http://www.eweek.com/article2/0,1895,1988874,00.asp*, July 2006
[10] Microsoft Security Bulletin Search. *http://www.microsoft.com/technet/security/current.aspx*
[11] ATP httpd Single Byte Buffer Overflow Vulnerability. *http://www.securityfocus.com/bid/5956*
[12] Light HTTPD GET Request Buffer Overflow Vulnerability. *http://www.securityfocus.com/bid/6162*
[13] Ghttpd Log Function Buffer Overflow Vulnerability. *http://www.securityfocus.com/bid/5960*
[14] MathoPD Remote Buffer Overflow Vulnerability. *http://www.securityfocus.com/bid/9871*
[15] Multiple Vendor C Library realpath() Buffer Overflow Vulnerability. *http://www.securityfocus.com/bid/8315*
[16] D. Jackson and E. Rollins. Chopping: A generalization of slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994.
[17] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.
[18] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Oct. 2005.
[19] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks . In *Proceedings of the 10th ACM Conference On Computer and Communication Security*. 2003.
[20] G. Kc, A. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference On Computer and Communication Security.* 2003.
[21] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd Symposium on Reliable and Distributed Systems*, Florence, Italy, Oct. 2003.
[22] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium,* Washington, DC. 2003.
[23] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University. 2002
[24] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. N. Tuong, and J. Hiser. N-Variant Systems A Secretless Framework for Security through

Diversity. In *Proceedings of the 15th USENIX Security Symposium*, Aug. 2006.

[25] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*. 1997.

[26] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. *http://www.research.ibm.com/trl/projects/security/ssp/*.

[27] O. Ruwase and M. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04)*, San Diego, CA. Feb. 2004.

[28] D. Dhurjati and V. Adve. Backwards Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'06)*, May 2006.

[29] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.

[30] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*, Feb. 2003.

[31] Michael F. Ringenburg, and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of ACM Conference on Computer and Communications Security (CCS'05)*, Nov. 2005.

[32] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM SIGCOMM'04*, Aug. 2004.

[33] S. Singh, C.Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI'04)*, Dec. 2004.

[34] H. A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 13th Usenix Security Symposium*, Aug. 2004.

[35] J. Newsome, B. Karp and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[36] Z. Li, M. Sanghi, B. Chavez, Y. Chen and M. Kao. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2006.

[37] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*. Feb. 2005.

[38] J. R. Crandall, Z. Su, and S. F. Wu. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, Nov. 2005.

[39] J. Newsome, D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the Symposium on Annual Network and Distributed Systems Security (NDSS'06)*, Feb. 2006.

[40] S. Sidiroglou and A. D. Keromytis. Countering Network Worms Through Automatic Patch Generation. *IEEE Security and Privacy*, Volume:3 Issue 6, Nov. 2005.

[41] S. Sidiroglou, G. Giovanidis and A. D. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *Proceedings of the 8th Information Security Conference (ISC'05)*, Sept. 2005.

[42] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building A Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, Apr. 2005.

[43] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, Feb. 2005.

[44] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Oct. 2005.

[45] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS'06)*, Feb. 2006.

[46] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'05)*, Nov. 2005.

[47] Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'05)*, Nov. 2005.

[48] G. Portokalidis, A. Slowinska and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of ACM SIGOPS EUROSYS 2006*, Apr. 2006.

[49] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004

[50] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of International Conference on Compiler Construction (CC'02)*, Mar. 2002.

[51] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. In *ACM Transactions on Programming Languages and Systems*, Vol 27, No 3, May 2005.