CloudER: A Framework for Automatic Software Vulnerability Location and Patching in the Cloud

Ping Chen†‡ Dongyan Xu† Bing Mao‡
‡State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
†Department of Computer Science, Purdue University

ABSTRACT

In a virtualization-based cloud infrastructure, customers of the cloud deploy virtual machines (VMs) with their own applications and customized runtime environments. The cloud provider supports the execution of these VMs without detailed knowledge of the guest applications and operating systems in the VMs. In addition to elastic resource provisioning for the VMs, a desirable "value-added" service the cloud provider can provide is the emergency response to runtime incidences of software bugs and vulnerabilities. The challenge is to facilitate the automatic runtime detection, location, and patching of the software vulnerability – outside the VMs and without the source code. In this paper, we present CloudER, a cloud "emergency room" architecture that automatically detect, locate, and patch software vulnerabilities in cloud application binaries at runtime. CloudER leverages an existing taint-based system (Demand Emulation) for runtime anomaly detection, employs new algorithms for software vulnerability location and patch generation, and adapts a virtual machine introspection system (XenAccess) for dynamic patching. Our preliminary evaluation experiments with a number of real-world server applications show that CloudER achieves timely response to runtime software faults or attacks from outside the VMs.

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

Buffer Overflow, Automated Patch Generation, Software Security, Cloud Computing

1. INTRODUCTION

Virtualization-based cloud computing platforms, such as Amazon EC2 [1] and Eucalyptus [11], support the hosting of customers' virtual machines (VMs) in the cloud infrastructure. The customers create the images of their VMs with their own applications and guest operating systems (OSs) and the cloud provider will then host the execution of the VMs with *value-added* functionalities such as autonomic resource provisioning, cloning, migration, and suspend/resume all without the customer's involvement.

In this paper, we propose another desirable self-management capability in the cloud: automatic response to runtime software faults and attacks, which arise from bugs and vulnerabilities in the application binaries. More specifically, we envision that the cloud should be able to detect a software fault or attack at runtime, locate the culprit code, generate a patch and apply it to the binary – all automatically without the customer's attention. Such capability is highly desirable considering the wide existence of software bugs and vulnerabilities in cloud applications, such as buffer overflow [2] and format string [18]. Automatic detection, location, and patching of software vulnerabilities is challenging in the cloud because the cloud provider does not have the source code of applications running in the customers' VMs. Moreover, as a cloud management function, the detection, location, and patching operations have to be performed from *outside* the production VMs.

In this paper, we propose CloudER, a cloud *emergency room* architecture, for automatic runtime location and patching of software vulnerabilities in response to software faults or attacks. More specifically, CloudER leverages *Demand Emulation* [6], an efficient and effective taint-based system for detecting software runtime anomalies from outside the VM being protected. Upon the detection of a taint-based anomaly, CloudER will locate the instructions directly responsible for the anomaly and further generate a binary patch to bypass those instructions yet maintaining the legitimate semantics of the application. The patch will be applied to the protected VM via enhanced virtual machine introspection technique capable of VM binary write. Finally, the VM's state will be restored to the one right before the anomaly detection so that the patched VM can resume execution properly. CloudER performs all the above steps without human administrator involvement. Moreover, except for the binary patching, CloudER does not modify the application and guest OS.

The main contributions of this paper are highlighted as follows:

- CloudER is an integrated architecture that improves the runtime reliability of cloud applications. It covers the full life cycle of exploit detection, culprit instruction location, patch generation and application, and execution state recording and reset – all performed from outside the protected VM and without the source code of the applications.
- While leveraging existing techniques for taint-based exploit detection, CloudER involves new methods for culprit instruction location and binary patch generation. The methods cover

ASIACCS '12, May 2–4, 2012, Seoul, Korea.

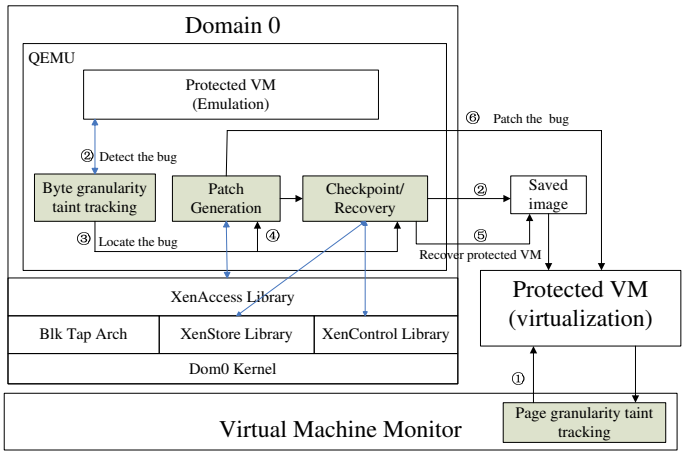

Figure 1: CloudER Overview

some of the most common types of software vulnerabilities and the patches generated are of small size (tens of bytes).

- CloudER incurs reasonable performance overhead to the application in comparison with running the application in an unprotected VM. The interruption to the production VM's execution (for culprit instruction location and patch generation) is less than half a minute in our experiments with real-world applications.

2. OVERVIEW

An overview of CloudER is presented in Figure 1. It consists of four components as indicated by the shaded boxes in the figure: page granularity taint tracking, byte granularity taint tracking, patch generation, and VM state recovery. The page and byte-granularity taint tracking components leverage *Demand Emulation* [6], which is a taint-based VM protection system via on-demand emulation. Under *Demand Emulation*, the page-granularity taint tracking is performed for the production VM in Xen; whereas the byte-granularity taint tracking is performed for the same VM but by the QEMU emulator in domain0. The byte granularity taint tracking is necessary to detect specific exploits/attacks and locate specific culprit instructions. The patch generation component then generates a binary patch for the corresponding vulnerability. The application of the patch and resumption of the production VM are performed by the recovery component (which in turn calls an adapted version of XenAccess library to apply the patch).

The work flow of CloudER's operations is also shown in Figure 1. In step ①, the protected VM on Xen in the virtualization mode is monitored by the page level taint tracking component. When the component detects that the system registers (e.g., CS, ESP, EIP) of the protected VM are tainted, it switches the protected VM to emulation mode executed on QEMU inside domain0. QEMU shares the memory space of the protected VM and leverages the byte-level taint analysis to monitor the protected VM (step ②). Meanwhile CloudER takes a live snapshot of the protected VM and saves the snapshot image. When QEMU detects that the application in the protected VM accesses a tainted security-sensitive object (e.g., return address, function pointer), it will locate those instructions that taint the security-sensitive object (step ③). Then QEMU triggers the patch generation module (④) which automatically generates a binary patch to bypass those culprit instructions. After binary patch is generated, the recovery module is triggered to recover the saved snapshot of the protected VM (step ⑤) and CloudER will apply the binary patch to the application's memory space (step ⑥) via virtual machine introspection.

Next we use an example to demonstrate the usage of CloudER. Figure 2(a) shows a sample source code that contains a buffer overflow vulnerability. Figure 2(b) is the corresponding disassemble code of the program. Using a malicious input string, the buffer overflow vulnerability can be exploited (line 9 in Figure2 (a)). The root cause of the bug is that the write operation lacks boundary check, and it may write to outside of the memory of *buf1* (e.g, overwriting the return address). CloudER will be able to detect the exploit when the program executes the `ret` instruction (line 33 in Figure 2(b)) with the tainted return value at the epilogue of function *main*. Furthermore, CloudER locates the vulnerable instructions (line 24-25 in Figure 2(b)). Then CloudER generates the binary patch, which will check whether the write instruction (line 25 in Figure 2(b)) writes to the safe area of the memory (i.e. from *buf1* to return address) and then skip those write operations that write to the security-sensitive area (object). Finally, CloudER applies the patch and resumes the execution of the protected VM.

```
1 int a;//global
2 int main()
3 {
4     char buf1[10];
5     char buf2[100];
6     scanf("%99s",buf2);
7     while(buf2[a])
8     {
9         buf1[a]=buf2[a];
10        a++;
11    }
12    buf1[a]='\0';
13    return 0;
14 }
```

(a) source code

```
1  0x8048364   push %ebp
2  0x8048355   mov  %esp, %ebp
3  0x8048367   sub  $0x98,%esp
4  0x804836d   and  $0xfffffff0, %esp
5  0x8048370   mov  $0x0, %eax
6  0x8048375   add  $0xf, %eax
7  0x8048378   add  $0xf, %eax
8  0x804837b   shr  $0x4, %eax
9  0x804837e   shl  $0x4, %eax
10 0x8048381   sub  %eax, %esp
11 0x8048383   lea  0xffffff78(%ebp), %eax
12 0x8048389   mov  %eax, 0x4(%esp)
13 0x804838d   movl $0x8048488, (%esp)
14 0x8048394   call 0x8048280
15 0x8048399   lea  0xffffff78(%ebp), %eax
16 0x804839f   add  0x804959c, %eax
17 0x80483a5   cmpb $0x0,(%eax)
18 0x80483a8   je   0x80483ce
19 0x80483aa   lea  0xffffffe8(%ebp), %eax
20 0x80483ad   mov  %eax, %edx
21 0x80483af   add  0x804959c, %edx
22 0x80483b5   lea  0xffffff78(%ebp), %eax
23 0x80483bb   add  0x804959c, %eax
24 0x80483c1   movzbl (%eax), %eax
25 0x80483c4   mov  %al,(%edx)
26 0x80483c6   incl 0x804959c
27 0x80483cc   jmp  0x8048399
28 0x80483ce   lea  0xffffffe8(%ebp), %eax
29 0x80483d1   add  0x804959c, %eax
30 0x80483d7   movb $0x0,(%eax)
31 0x80483da   mov  $0x0, %eax
32 0x80483df   leave
33 0x80483e0   ret
```

(b) disassembly code

Figure 2: An Illustrative Example

3. DETAILED DESIGN

3.1 Bug Location Approach

We leverage the on-demand taint-analysis system Demand Emulation [6]. When Demand Emulation detects an exploit, it does not perform any specific remedial actions of its own. Instead, triggered by the detection point, CloudER will proceed to locate the instructions to be patched.

CloudER records memory tainted information under the emulation mode. More precisely, CloudER records the memory written instructions which propagate the taint flag as well as the tainted memory in the form of (M, W). M is the memory which is tainted, and W is the memory write instruction which taints M. With the (M, W) records, CloudER locates the bug in the following steps. First, CloudER detects the attack, it regards M' as *error_address*. Then, it uses M' to find the write instruction W' which taints M' based on the (M, W) records. In order to replace the write instruction with our binary patch, we require that the size of the replaced

instructions to be no less than 5 bytes [1]. However, the write instruction is often 2-3 bytes long (e.g., `mov %al,(%edx)` is 2 bytes), thus we put the previous instructions into the *bug instructions* to guarantee the bug instructions be no less than 5 bytes.

Consider the example in Figure 2, suppose the attacker injects 100 bytes of data (the size of `buf2`). When QEMU executes the write instructions (line 24-25 in Figure 2(b)), which represent the statement in line 9 of Figure 2(a), it records the 100 records because this statement will be executed to write 100 bytes from the outside input into `buf1`. In the 100 records, all W turns out to be the same instruction (line 25 in Figure 2(b)), and M are the contiguous memory addresses (ranging from `0xbffff6e0` to `0xbffff744`). Note that `0xbffff6e0` is the start address of `buf1`. When the program executes the `ret` instruction, CloudER detects the return address (`0xbffff6fc`) is tainted, so the error address is `0xbffff6fc`. Then CloudER sets M' as `0xbffff6fc`, searches the 100 records and finds the record (`0x80483c4`,`0xbffff6fc`), and locates the write instruction W' at `0x80483c4`. As this instruction is less than 5 bytes, CloudER puts the instruction (line 24 in Figure2(b)) just before it into bug instruction set.

There are two problems we need to solve here. One is the location of the vulnerable library function (e.g.,strcpy,printf). If the W' belongs to a dynamic linked library function, it is reasonable to locate the function call point, and wrap the call instruction with our binary patch. We use dynamic linked library function identification technique to solve it. It is well-known that the destination address of library function call instruction is in the PLT section in ELF. During the program execution, we check whether the destination of call instruction is in the range. If it is true, we replace the write instruction W in the function with the address of call instruction.

Another problem is that the execution context of the bug instructions may vary, and the write instruction can be leveraged in the benign scenario. To solve this problem, we need to additionally record the start address of the destination object the bug instructions writes to, and we named the start address of the destination object as *start_address*. Take the example in Figure 2 for instance, the destination object of the maliciously written instruction in line 9 is `buf1`. And suppose that we get 100 (M,W) records during the program execution, and they have the same W with the different M ranging from `0xbffff6e0`(buf1) to `0xbffff744`. *start_address* is the smallest address `0xbffff6e0`. After bug location analysis, we record the *start_address*, *error_address*, and the bug instructions.

3.2 Patch Generation and Application

When we have located the bug, we get the following information: *start_address*, *error_address*, and the bug instructions. In current patch generation mechanism, we replace the bug instructions with *jmp* instruction, the destination address of the *jmp* instruction is the start address of our binary patch code, and we define the address as `mmp_start`. Note that `jmp` instruction is often 5 bytes long, if bug instructions are larger than 5 bytes, we insert `nop` instructions behind the `jmp` instruction to pad the rest code space. In order to insert binary patch code into the process space, we mmap the memory space for the patch in the process, and write the binary patch into it, and we define the start address of the binary patch as `mmp_start`. More detailed, when we insert the binary patch, we add the `mmap` system call by using the micro operations in the QEMU and then QEMU translates the micro operations and executes them to mmap a memory space into the process. The binary patch is used to skip the maliciously write instruction.

[1]In order to insert a jump to the patch.

```
1 int a;//global
2 int main()
3 {
4    char buf1[10];
5    char buf2[100];
6    scanf("%99s",buf2);
7    while(buf2[a])
8    {
9        buf1[a]=buf2[a];
10       a++;
11   }
12   buf1[a]='\0';
13   return 0;
14 }
```

```
0x80483c1:
      movzbl (%eax),   %eax
      mov  %al, (%edx)
0x80483c6 :
```

```
0x80483c1 :
      jmp mmp_start
0x80483c6 :
```

```
mmp_start:
      cmp  start_address, %edx
      jl Label1
      cmp  error_address, %edx
      jge Label1
      movzbl    (%eax),%eax
      mov   %al,(%edx)
Label1:
      jmp 0x80483c6
```

Figure 3: An Example of Binary Patching

As a simple illustration, we again use the example shown in Figure 2. Figure 3 shows our binary patch for this program. When we locate the bug instructions from `0x80483c1` to `0x80483c5`. Then we replace them with `jmp mmp_start`. In the binary patch, we compare the destination memory address with the *start_address* (`0xbffff6e0`)and *error_address*(`0xbffff6fc`). If the address is between them, we execute the instructions. Otherwise ,we skip the instructions. In the end of binary patch, there is a *jmp* instruction to jump back to the next instruction of the bug instructions.

There is another problem when we design our patch generation mechanism. As the same bug instructions may occur at several attack scenario, we may generate several patches for the same bug instructions. It is a problem that how we update the binary patch. To solve this problem, we first check whether the *start_address* exists. If so, we compare the *error_address* to the one in the existing patch, and replace the older one if *error_address* is smaller. If the *start_address* is not existing, we need to generate a new patch for it. And then we chain the patch together by *jmp* instruction.

When the binary patch is generated, we leverage the `mmap` system call to allocate the memory space for the binary patch, and get the base address of the `mmap` area. We write a tool which can replace the bug instructions with the binary patch. The tool leverages the virtual machine introspection system XenAccess-0.5 [13], the library is modified by us to be compatible with Xen-3.0.0. The work flow of the tools is as following. First, our tool maps the memory frame of the bug instructions according to its `domid` and `process id`. Then it reads the bug instructions according to the virtual address of the bug using XenAccess, and then it leverages the `jmp` instruction to replace the bug instructions, and the destination address of the `jmp` instruction is the binary patch.

4. EVALUATION

We have created a proof-of-concept prototype of CloudER. To verify the effectiveness and performance 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. [19], as well as the additional real-world exploits which contain the format string attack and buffer overflow attack. The evaluation is performed on the following configuration. Xen-3.0.0 preforms on the Intel Pentium Dual T5600 1.83GHz machine with 1.0G memory. And the Domain0 is equipped with the Linux 2.6.12 kernel. Protected VM is allocated with 128M memory with the Linux 2.6.12 kernel. Tested programs are in the protected VM and compiled by gcc 4.1.1 and linked with glibc 2.1.2. Note that in

protected VM, we close the address randomization function by the command "echo 0 > /proc/sys/kernel/randomize_va_space" to prevent the stack randomization, as address randomization may obfuscate the `error_address` our patch depends on(maybe slight changes to the OS loader help us overcome).

4.1 Effectiveness

4.1.1 Wilander's Benchmark Test-Suite

There exists different buffer overflow attacks in the publicly available Wilander's Benchmark Test-Suite, and we select buffer overflow bugs aim at stack and bss objects. However, Wilander's benchmark writes the shellcode in the program, in order to emulate the network application, we modify it to let the user input the shellcode from the network. Interested readers can reference [19] for more details.

4.1.2 Real-World Attacks

We further evaluate the effectiveness of our system with real-world attacks, including buffer overflow and format string. We select three applications to test CloudER, and the bugs in these software are described in Table 1. In order to trigger the Demand Emulation to detect them, we do the following modification to some applications. For ncompress, we input the long file name from the network to trigger the buffer overflow attack. We note that the modifications to the applications are simply for the sake of following Demand Emulation's network input tainting and tracking. For all these applications, CloudER is able to correctly detect the exploits and patch the vulnerabilities.

In Table 1, we show the bug and patch size in our experiment. Wilander's benchmark is an example of malicious instruction to multiple memory space. ncompress is an instance that bug instructions contain string copy function `strcpy` [9]. ATPhttpd is an example of maliciously written instruction to single memory [4]. wu-ftpd is an example of format string bug(`vsprintf`) [3].

4.2 Performance Overhead

We evaluate the performance overhead of patch generation. The patch generation overhead contains automatically detecting the bug, locating the bug and generating the patch, saving and recovering the protected VM as well as inserting the binary patch. Table 2 shows the performance overhead including the *Detector*, *Locator & Generator*, *Save & Recovery* , *Patch Inserter* and *Total Time*. `Detector` is the time span from CloudER pausing the protected VM to detecting the bug. *Locator & Generator* is the time span CloudER locates the bug and generates patch for the bug. *Save & Recover* is the time span CloudER saves the protected VM and recovers it. *Patch Inserter* is the time CloudER wraps the bug instructions with the jmp instruction and inserts the binary patch into the mmaped memory space. *Total Time* is the time span CloudER pauses the domain during detecting, locating, patching and recovery. Results show that CloudER can generate the patch within seconds.

5. DISCUSSION

The current CloudER system has a number of non-trivial limitations that warrant further research. First, due to the bug location capability, our patch may not remedy the root cause. Second, our patching function can only generate the binary patch for preventing the attack re-launched, and if the attacker leverages the same bug to write other object, our system will generate a new patch, thus there may be several patches for one bug. What's more, our binary patch is not intelligent enough that it can prevent the maliciously writing to the same object. For example, when the attacker writes to a buffer, the first time he writes to the return address, and the second time to the function pointer(suppose the function pointer's address is lower than the return address). In current implementation, we just update the patch and shorten the range between `start_address` and `error_address`. However, we believe, after a long time patching, the binary patch will be better than before. Third, currently our automatical patching generation algorithm can only apply for the buffer overflow which aims at stack and bss object. Our patch mechanism can not prevent attack which aims at heap object, and also not be effective for stack object with ASLR. For other bugs such as heap overflow, integer overflow and double free, we cannot automatically generate binary patch. Fourth, since we use the save & restore mechanism to recover the VM states, it's worth nothing that when we save the VM, its resources on the host machine will be de-allocated, especially the network connections to it will be lost.

6. RELATED WORK

Program patching is important to guarantee the availability of the application. Researchers are searching the efficient automatic tools which can patch the bugs in the programs. These approaches can be divided into source code level patch and binary level patch. For example, AutoPaG [7], Exterminator [10], PASAN [16] and ShieldGen [17] provide the patch for the source code. And other works, such as CleanView[14], Livepatch [8], Pannus [12] and Katana [15], generate the `hot` patch for the binary code. Since the commodity software has no source code available, and the deployed applications in the cloud can not stop for recompiling their source code. Binary level patch meets the requirements of the customers. However, current binary patch methods focus on the function level patch, which is coarse-grained. In practice, we find that the bug often contains two or three instructions at binary level, thus CloudER replaces the bug instruction sequence with fine-grained binary patch (several bytes long), which can be quickly deployed online. Recently, there are two new binary patch methods proposed: First-Aid [5] generates the binary patch by changing the environment based on the feature of the known bugs. First-Aid only exposes the known attack next time, but it does not make the program immune from the attack. Nuwa [20] is a patch tool which provides the patch offline for the VM images. Compared with the two new methods, CloudER applies the temporary patch online to protect the VM from being attacked, and guarantees the software can provide the services without termination. Further, CloudER can also help the cloud providers or the software vendors apply the permanent offline patch for the program by using the bug detection and location information from CloudER.

7. CONCLUSIONS

In this paper, we present the design, implementation, and evaluation of a framework for automatic software vulnerability location and patching in the cloud. Given a working maliciously written exploit (e.g., a buffer overflow attack) which may be previously unknown, our system is able to catch the attack, and automatically analyze the binary code and identify the malicious instructions. Furthermore, within seconds, our system can recovers itself, and automatically generates the binary patch and apply it into the program's execution space. The evaluation using the Wilander's buffer overflow benchmark as well as a number of real-world exploits successfully demonstrates its effectiveness.

Table 1: The Effectiveness of CloudER with Programs and Their Vulnerabilities

Program	Size(Byte)	Attack Type	Detector	Locator	Patch Generator	
				Bug Size	Patch Size	(Prevented?)
Wilander's Benchmark	13,208	Buffer Overflow	✓	5B	41B	✓
ncompress-4.2.4	50,047	Buffer Overflow	✓	5B	82B	✓
ATPhttpd 0.4b	41,085	Buffer Overflow	✓	7B	23B	✓
Wu-ftpd 2.6.0	379,658	Format String	✓	5B	57B	✓

Table 2: Automatic Patch Generation Time

Benchmark/Program	Attack Type	Detector	Locator & Generator	Save & Recovery	Patch Inserter	Total
Wilander's Benchmark	Buffer Overflow	0.419s	1.789s	3.956s	1.127s	7.291s
ncompress 4.2.4	Buffer Overflow	0.967s	3.573s	4.028s	3.458s	12.026s
ATPhttpd 0.4b	Buffer Overflow	1.056s	2.134s	4.834s	2.124s	10.148s
Wuftpd 2.6.0	Format String	3.871s	4.280s	5.376s	2.464s	15.991s

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by grants from the Chinese National Natural Science Foundation (61073027, 60773171, 90818022, and 61021062), the Chinese 973 Major State Basic Program (2009CB320705), and US National Science Foundation (0546173, 0855141). Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/.

[2] Smashing the stack for fun and profit. In *Phrack 7, 49*, 1996.

[3] CVE-2000-0573. Format string in wu-ftpd 2.6.0.

[4] CVE-2002-1816. Buffer overflow in atphttpd 0.4b.

[5] Q. Gao, W. Zhang, Y. Tang, and F. Qin. First-aid: surviving and preventing memory management bugs during production runs. In *Proceedings of the 4th ACM European conference on Computer systems(EuroSys '09)*, pages 159–172, 2009.

[6] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM European conference on Computer systems(EuroSys'06)*, 2006. http://people.cs.ubc.ca/ andy/taint-xen.

[7] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. Autopag: Towards automated software patch generation with source code root cause identification and repair. In *Proceedings of ACM Symposium on InformAtion, Computer and Communications Security(ASIACCS'07)*, Singapore, March 2007.

[8] Livepatch. http://sourcehoge.net/software/livepatch/.

[9] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[10] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, Dec. 2008.

[11] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid(CCGRID '09)*, pages 124–131, 2009.

[12] Pannus. http://pannus.sourceforge.net/.

[13] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference(ACSAC 2007)*, pages 385 –397, 2007.

[14] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles(SOSP '09)*, pages 87–102, 2009.

[15] A. Ramaswamy, S. Bratus, S. Smith, and M. Locasto. Katana: A hot patching framework for elf executables. In *Proceedings of the 4th International Workshop on Secure Software Engineering (SecSE 2010). Co-published in the International Conference on Availability, Reliability and Security*, pages 507 –512, 2010.

[16] A. Smirnov and T. cker Chiueh. Automatic patch generation for buffer overflow attacks. In *Proceedings of the Third International Symposium on Information Assurance and Security(IAS'07)*, 2007.

[17] C. Weidong, P. Marcus, and W. H. J. nad Locato Michael E. Shieldgen:automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 252–266, 2007.

[18] Wikipedia. Format string attack. http://en.wikipedia.org/wiki/Format_string_attack.

[19] 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)*, pages 273–286, 2003.

[20] W. Zhou, P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala. Always up-to-date: scalable offline patching of vm images in a compute cloud. In *Proceedings of the 26th Annual Computer Security Applications Conference(ACSAC '10)*, pages 377–386, 2010.