SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities

Gang Chen, Hai Jin, *Senior Member*, *IEEE*, Deqing Zou,
Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi

Abstract—Buffer overflow attacks still pose a significant threat to the security and availability of today's computer systems. Although there are a number of solutions proposed to provide adequate protection against buffer overflow attacks, most of existing solutions terminate the vulnerable program when the buffer overflow occurs, effectively rendering the program unavailable. The impact on availability is a serious problem on service-oriented platforms. This paper presents SafeStack, a system that can automatically diagnose and patch stack-based buffer overflow vulnerabilities. The key technique of our solution is to virtualize memory accesses and move the vulnerable buffer into protected memory regions, which provides a fundamental and effective protection against recurrence of the same attack without stopping normal system execution. We developed a prototype on a Linux system, and conducted extensive experiments to evaluate the effectiveness and performance of the system using a range of applications. Our experimental results showed that SafeStack can quickly generate runtime patches to successfully handle the attack's recurrence. Furthermore, SafeStack only incurs acceptable overhead for the patched applications.

Index Terms—Software reliability, buffer overflow vulnerability diagnosis, attack prevention

1 Introduction

With the fast development of service-oriented computing paradigm, the demand for high availability of services grows significantly. Unfortunately, system security and availability are still severely hindered by memory vulnerabilities. Programs written in unsafe languages like C and C++ are particularly vulnerable, where attackers can exploit memory vulnerabilities to control vulnerable programs. Among memory vulnerabilities, buffer overflow vulnerabilities have posed a major threat to the security of computer systems. According to the US-CERT vulnerability notes database, there are 11 buffer overflow attacks in 20 vulnerabilities with the highest severity metric [1]. However, the response to such attacks is slow: previous study shows that it takes 28 days on average to diagnose vulnerabilities and generate patches [2]. During this long vulnerable time window, users have to either tolerate the costly downtime, or experience problems such as intermittent crashes and potential attack's recurrence if they continue to use the software—neither of the temporary mitigation is desirable. Therefore, it is very important to provide a practical and efficient solution to survive and prevent the buffer overflow attacks.

- *G. Chen, H. Jin, D. Zou, W. Zheng, and X. Shi are with the Cluster and Grid Computing Lab, Services Computing Technology and System Lab, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: deqingzou@hust.edu.cn.*
- *B.B. Zhou is with the Centre for Distributed and High Performance Computing, School of Information Technologies, University of Sydney, Sydney, NSW 2006, Australia. E-mail: bbz@it.usyd.edu.au.*
- *Z. Liang is with the Department of Computer Science, School of Computing, National University of Singapore, Singapore 117417. E-mail: liangzk@comp.nus.edu.sg.*

Manuscript received 18 Oct. 2012; revised 17 Mar. 2013; accepted 28 May 2013; published online 10 June 2013.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2012-10-0253.
Digital Object Identifier no. 10.1109/TDSC.2013.25.

In a buffer overflow attack, the attacker aims to change the control flow of the vulnerable program to gain access to a system. There are various ways to exploit buffer overflow vulnerabilities. The most widely used one is to overflow buffers on the stack to modify the return address or the saved frame pointer. Although this type of attack is no longer directly exploitable with the advancement of buffer overflow defense technology, it is still the basis for denial-of-service attacks or advanced memory exploits, such as attacks exploiting the windows structured exception handler mechanism [3]. There are other complex exploit forms of stack-based buffer overflow attacks, for instance, overrunning a local buffer to corrupt some other code pointers, such as function pointers, `longjmp` buffers and global offset table entries.

A number of solutions have been proposed to protect applications against the buffer overflow vulnerabilities. They are based on techniques including return address defense on the stack [4], [5], [6], [7], array bounds checking [8], pointers protection via encryption [9], and address space layout randomization [10], [11], [12]. Although these solutions prevent buffer overflow exploits, they drop the availability of the applications because they usually terminate the applications when buffer overflow attacks are detected. Terminating an application is often unacceptable due to high availability requirements from service-oriented platforms.

Since memory error exploits are caused by vulnerabilities, patching the vulnerabilities is the ultimate solution. Keromytis [18] proposes the "patch on demand" concept, integrating the vulnerability discovery, patch generation, and patch application cycles into a system. Solutions such as PASAN [20] have been developed to automatically generate patches of memory errors, but they mainly focus on patch generation, which still needs service restart to apply the patches. To help programs survive vulnerabilities and thus improve their availability, a great number of solutions have been proposed to tolerate attacks, such as attack signature diagnosis [17], [19], and self-healing approaches [34], [15], [16]. However, they do not solve the problem satisfactorily due to the lack of compatibility or accuracy. Therefore, there is a strong need of an end-to-end system that *both generates and applies patches with production-grade performance and safety maintenance*. First-Aid [21] is such a system. However, this approach is designed only to protect heap buffers. It is not effective for other type of buffers, especially for stack-based buffers. Stack-based buffers are tightly coupled with binaries, and thus are harder to relocate while the program is running.

In this paper, we propose a novel system called *SafeStack*, which can quickly and automatically generate patches to stack-based buffer overflow vulnerabilities and apply the patches without stopping the vulnerable service. The key technique of SafeStack is *memory access virtualization*, which relocates memory objects to selected locations. Once an attack is detected, SafeStack identifies the stack objects that trigger the stack buffer overflow, generates runtime patches which consist of vulnerability signatures, bug-triggering buffers and vulnerability treatments, and applies them to move these stack objects into protected memory areas. SafeStack is designed to learn from attacks. Although the application still suffers the first attack, SafeStack can diagnose the vulnerability origin from this attack and generate patches to “fix” the stack-based buffer overflow vulnerability. As a result, the automatically generated SafeStack patches enable the application to survive exploits and continue to process subsequent requests.

Compared with previous approaches, SafeStack has the following advantages:

- *Efficiency.* SafeStack consists of an online production system and an offline triage system to leverage failure diagnosis and availability. Specifically, the online system is used to detect faults and apply patches while the offline system is used to identify the bug-triggering buffers and then to generate patches. Therefore, the failure diagnosis does not influence the work of applications. In our experiments, for each of nine stack-based buffer overflow bugs in eight applications, SafeStack can automatically generate patches just in a few seconds and enable the applications to survive subsequent attacks.
- *Safety.* Most buffer overflow defense tools try to improve the security of the applications without considering the availability, while most software fault tolerance tools tend to ignore buffer overflow bugs or reduce the security to improve the availability. These approaches are not desirable for stack-based buffer overflow bugs in applications with high availability requirements. SafeStack moves the bug triggering stack objects out of the stack, which prevents attackers to corrupt the return address, the saved frame pointers and some other code pointers. Furthermore, as the “faulty” stack objects are moved into and randomly located in the protected memory areas, the system does not introduce uncertainty or misbehavior into the application’s execution.
- *Scalability.* SafeStack is not tied to any specific memory protection tools. In the current implementation, SafeStack adopts a method similar to that introduced in DieHard [22] which can provide high probabilistic memory safety. SafeStack can easily be combined with other useful tools to collaboratively protect the application against memory bugs.

We have implemented a Linux prototype and evaluated it using eight applications which have stack-based buffer overflow vulnerabilities, including four web servers, an FTP server, a streaming audio server, and two desktop applications. Additionally, we evaluated SafeStack’s performance with the SPEC INT2000 benchmarks. Our experimental results showed that SafeStack can quickly generate runtime patches and successfully handle the attack’s recurrence after applying the patches. Furthermore, the patches generated by SafeStack only incur reasonable overhead.

In summary, we made the following contributions:

- We propose SafeStack, a novel end-to-end system to diagnose and patch stack-based buffer overflow vulnerabilities. SafeStack automatically generates and applies patches to vulnerabilities to mitigate buffer overflow exploits while preserving system availability.
- We develop a technique, *memory access virtualization*, which allows runtime relocation of memory objects through binary instrumentation.
- We implemented SafeStack on a Linux system, and evaluated its effectiveness and performance using real-world attacks.

The rest of the paper is organized as follows. The SafeStack overview, including the high-level ideas, architecture, the work flow, and important steps of SafeStack are introduced in Section 2. Section 3 describes the important techniques and a few challenges we faced in implementation. The experimental and analytical results are presented in Section 4. Section 5 gives an overview of related work. Finally, we summarize our contributions in Section 6.

2 SafeStack

In this section, we first introduce new high-level ideas for the design and development of SafeStack. Then, we discuss how system components function in dealing with stack buffer overflow bugs.

2.1 Memory Access Virtualization

To understand a stack-based buffer overflow attack, we show a typical stack layout for a running function in Fig. 1a. It consists of the function’s parameters, the return address, the previous frame pointer, and the local variables. There are two stack-based buffers in the local variables. If the program fails to check the buffer bounds when it accesses the two buffers, out-of-bound buffer access can change stack variables adjacent to the buffers. Attackers can use accesses to these buffers to corrupt the stack. For example, attackers can use a buffer overflow exploit to overwrite the return address on the stack with another address. When the victim function returns, the program control is transferred to malicious code injected by attackers.

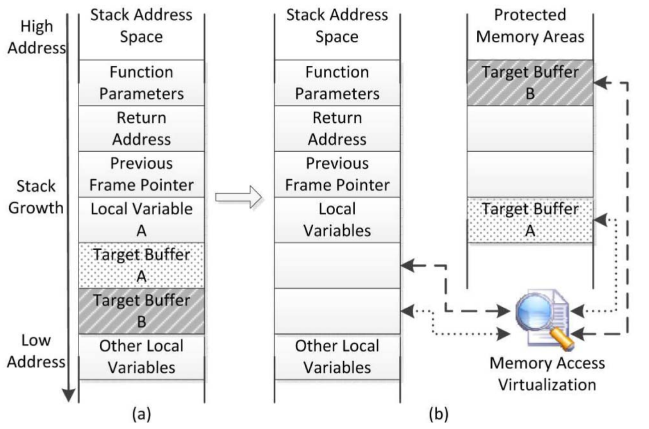


Fig. 1. Memory access virtualization.

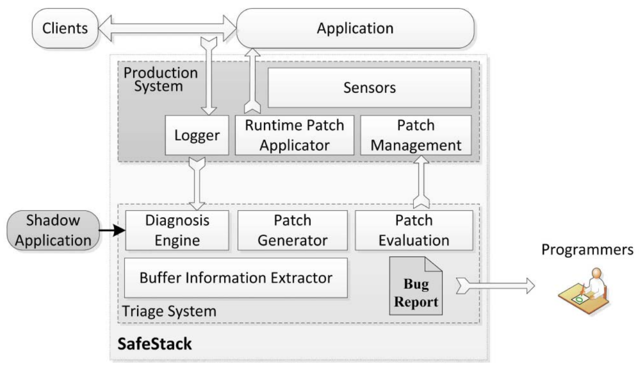


Fig. 2. Dual-system architecture of SafeStack.

There are two major challenges in automated buffer overflow attack diagnosis and patching, that is, *how to automatically identify the buffers that are overflowed in an attack* and *how to mitigate attacks to preserve system availability without rebuilding the vulnerable program's binary*. To better deal with both challenges, we develop a new technique called *memory access virtualization*. Using this technique, we can relocate memory objects to other locations to maintain a program's functionality at runtime. With the ability of memory object relocation, for attack diagnosis we can move stack buffers to a monitored memory region to detect whether some of them have out-of-bound access, while for attack prevention we can move vulnerable buffers into protected memory areas, and then write values into (or take values out of) the corresponding protected memory areas instead of the original stack address space. Therefore, we can safely mask buffer overflow attacks such as an out-of-bound write or an out-of-bound read and make the program continue to execute normally, instead of throwing up an exception and terminating the program—an undesirable situation for an important business server program.

Based on this technique, we build SafeStack to automatically diagnose and patch stack-based buffer overflow attacks. Specifically, we use *memory access virtualization* mechanism to relocate the bug-triggering buffer into protected memory areas, as shown in Fig. 1b. In this way, the two vulnerable buffers are protected from being used to overrun the control data.

2.2 A Dual System Architecture

To preserve availability while reducing the impact on the running application, we introduce a dual system architecture that consists of an online production system and an offline triage system, as illustrated in Fig. 2. The online production system is the deployment of the application in the production environment. The offline triage system is a shadow deployment of the application, which is used to identify the bug-triggering buffers and to generate patches.

The online production system includes a set of *sensors* for detecting exploits and identifying their types at runtime, a *logger* component to log network inputs for exploit diagnosis in the triage system, a *patch management* component for receiving and managing the patches that are generated and evaluated from the triage system, and a *runtime patch applicator* component for applying the patches with a dynamic instrumentation tool.

The offline triage system includes a *buffer information extractor* component for extracting information about stack buffers from program binaries, a *diagnosis engine* component for identifying the bug-triggering buffers in the shadow application, a *patch generator* component for generating runtime patches according to the results of diagnosis engine, and a *patch evaluation* component for evaluating whether the patches can survive the exploits. The patch evaluation component sends the results to the patch management component and generates an exploit report according to the results of the patch evaluation to help programmers perform postmortem exploit analysis.

2.3 Workflow

The Workflow of SafeStack is illustrated in Fig. 3. The logger component of the online production system records network inputs to a program into a log file and maintains a shadow call-stack. It logs network inputs by sessions, and only stores recent sessions to keep the log size under a user-specified threshold. When an attack is detected, the online system sends the log file and the shadow call-stack with the core dump file to the offline triage system for the analysis.

In the offline triage system, the buffer information extractor extracts the stack-buffer-related information from the program binaries. This step is indicated as Step 1 in Fig. 3. Once the diagnosis engine receives an attack analysis request from the production system, the triage system enters the diagnosis phase to analyze the attack information, gets the candidate vulnerable functions, and replays the inputs from the log file, indicated as Step 2 in Fig. 3. According to the buffer information, the triage

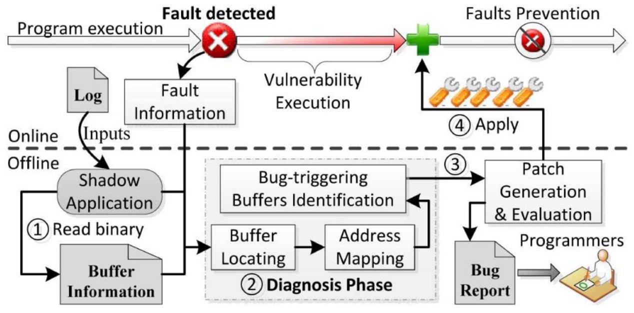


Fig. 3. SafeStack workflow.

system dynamically instruments the program to move the stack buffer objects inside the candidate vulnerable functions to the monitored memory areas, and identifies bug-triggering buffers through testing.

Once a bug-triggering buffer is detected, the patch generator generates runtime patches, which are evaluated by the patch evaluation component to determine whether they are able to make the application tolerate attack recurrences (Step 3 in Fig. 3). If so, the patch evaluation component sends the generated runtime patches to the production system. At the same time, the system also generates a report according to the patch evaluation for postmortem vulnerability analysis. If no patch is generated, the system will raise an exception to indicate that the problem may not be a stack buffer overflow.

Finally, the runtime patch applicator in the online production system applies patches generated from the triage system using dynamic binary instrumentation (Step 4). In this way, SafeStack temporarily fixes the vulnerability and prevents subsequent attacks of the same type, which helps the production system withstand the buffer overflow attacks until the vendor's official patch becomes available.

The design of SafeStack is especially suitable for cloud systems or data centers, which are rich in resources. Take a cloud system as an example. We can deploy the production system in several virtual machines for different users and deploy just one triage system in a separate virtual machine. The generated patches can be easily distributed to any VM running the vulnerable application.

3 Implementation

It is not straightforward to develop a system based on the high-level ideas discussed in the previous section. Techniques need to be developed for memory access virtualization, stack buffer relocation, bug-triggering buffer identification, patch generation and evaluation, and runtime patch application. To investigate these important techniques and to test the feasibility of our high-level ideas, we developed a prototype SafeStack on a Linux platform. In our implementation, a dynamic instrumentation tool Pin [24] is used to monitor all the program instructions to check their memory accesses. The exploit detection sensor in current implementation detects program crashes by catching the signal SIGSEGV. We also use the protected heap space as the protected memory areas because significant achievements have been made recently in dealing with heap-related vulnerabilities [22], [23].

3.1 Memory Access Virtualization

The key technique of SafeStack is *memory access virtualization*, which allows memory objects to be relocated at runtime. It is the basis of both attack diagnosis and vulnerability patching.

Memory access virtualization adopts an *object-relocation* table to map the original memory address of an object to a new address. The mapping is relatively simple if the access to a buffer is within that buffer. Otherwise, the mapping has to be done carefully as the system needs to consider whether it is an out-of-bound access or a legitimate access to a different variable.

We focus our discussion on the memory access virtualization mechanism for simple arrays. The mechanism works in a similar way for other types of local buffers, such as structures, structure arrays, unions, and union arrays.

There are two main types of array element access: *direct access* and *indirect access*. For example, defining a character array of size 20 as `a[20]`, `a[5]` is a direct array element access using the constant index 5, and `a[i]` is an indirect array element access using the index variable `i`.

For the direct access, the array index is a constant. The offset from the frame pointer or the stack pointer is determined at the compilation time. In this case, the effective memory address is inside the object-relocation table, and the addressing mode is *"Base plus Offset."* The base is the base register (EBP or ESP) and the offset is the value between the variable (e.g., `a[20]`) and the base register. SafeStack simply replaces this original address with its corresponding new address before executing the instruction. There is a case that the stack buffer subscript is a constant and out of range, such as `a[24]`. Fortunately, this explicit out of bound access can be detected in the testing phase before the application is released, and it is reasonable that we assume the program does not have this explicit out of bound access.

For the indirect access, the array index is an expression. The effective memory address is the sum of the starting address of the array and the size of the array element multiplied by the array index, and the addressing mode is *"Base plus Index plus Offset."* The base is the base register, the offset is the value between the starting address of the array and the base register, and the index is the index register (storing value `i`, which is scaled by the size of the array element). SafeStack first calculates the starting address of the array, finds the corresponding new address according to the object-relocation table, calculates the final new address, and replaces the original address with it before executing the instruction.

In addition to these two types of array element access, arrays can also be accessed by pointers to arrays. For example, a program can define a pointer variable pointing to an array, and access array elements using the pointer variable. To handle this type of access, SafeStack first replaces the original address with the corresponding new address, and subsequently all the offset calculation based on this pointer can be directed onto the corresponding new address without the needs to map new address for each instruction subsequently. Similarly, most of the time the address of an array is passed as an argument to a function, for example, the address of an array is passed to the function `strcpy` to copy data into the array. In this case, the value of the argument has been replaced with the corresponding new address. This solution avoids searching the stack frame and checking every instruction for address mapping, and thus has better performance. For SIMD instructions, they also specify the addresses of arrays to process. Therefore, SafeStack can replace them with corresponding new address before processing.

We summarize the memory access virtualization procedure with pseudocode in Procedure 1. The memory access virtualization procedure inserts different functions for direct and indirect array element accesses and the pointer referenced access, respectively. The memory access virtualization procedure rewrites the memory address due to the limitation of the instrumentation tool PIN, which cannot get the actual memory address at the instrumentation time. Note that the LEA instruction is a special case. It is often used for efficient computation rather than memory access. So SafeStack moves the corresponding new address into the destination register directly, and there is no need to execute the original LEA instruction.

Procedure 1. Memory Access Virtualization Procedure
Input: *ins*
Output: none

```
 1: if instruction ins is a read/write memory
    operation then
 2:   get the base register (regBase);
 3:   if regBase is valid then
 4:     get the index register (regIndex);
 5:     if regIndex is valid then
 6:       insert a call (OriginalBufferSIB) before ins to
          get addr;
 7:       rewrite addr in ins;
 8:     else
 9:       insert a call (OriginalBufferDirectAccess)
          before ins to get addr;
10:       rewrite addr in ins;
11:     end if
12:   end if
13: else if instruction ins is $LEA$ then
14:   get the destination register (dest);
15:   insert a call (LeaOperation) before ins to rewrite dest;
16:   delete ins;
17: end if
```

In Procedure 2, we show a representative procedure *OriginalBufferSIB* which is used to deal with the buffer itself access for a variable array subscript. The input *originalEa* is the address of an array element; *regBaseValue* is the content of base register; *regIndexValue* is the content of index register; *scale* is the scale factor, and *disp* is the displacement. The procedure calculates the starting address of the array and calls a procedure *LookupTable* to get the corresponding new address retrieved from the object-relocation table. SafeStack returns a calculated new address if found, otherwise returns the original memory address.

Procedure 2. Buffer SIB Procedure.
Input: *originalEa, regBaseValue, regIndexValue, scale, disp*
Output: *addrOriginal, addrNew*

```
1: $addrOriginal = regBaseValue + disp$;
2: $addrNew =$ LookupTable($addrOriginal$);
3: if $addrNew > 0$ then
4:   return $addrNew + regIndexValue * scale$;
5: else
6:   return $originalEa$;
7: end if
```

Using actual stack address for the stack buffer access can cause high overhead. However, in most cases, the offset from the frame pointer is sufficient to determine a stack buffer inside a function. In such cases, we can greatly simplify the above procedure for memory access virtualization by using the process id, thread id, the function starting code address, and an offset to uniquely determine a stack buffer. In this way, we not only eliminate the operation of locating the memory address of stack buffers, but also decrease the overhead of the operation for finding the corresponding heap address.

3.2 Stack Buffer Relocation

To relocate a stack buffer to the protected heap space, SafeStack allocates the same size of the corresponding buffer on the heap space, adds padding around it, and places canaries around the corresponding heap objects for bug-triggering buffer identification. Similar to the heap management for the moved buffers in DieHard [22], the allocated buffers are placed in the *miniheaps*. However, the buffers are not randomly placed on the heap as the padding is added around the allocated buffers to avoid using more memory space for identification of bug-triggering buffers.

SafeStack maps the stack buffers to new heap addresses when the value of the stack pointer decreases, i.e., when the buffers are allocated. It only causes a reasonable overhead when stack buffers are allocated on the stack, but not initialized yet. Otherwise when the stack buffer has been initialized, SafeStack must copy the data from the stack buffer to the new heap address.

When a function is completed, the associated memory space on the heap is freed and the mapping relationship is removed from the object-relocation table.

3.3 Bug-Triggering Buffer Identification

When a CALL instruction is detected, SafeStack must instrument this function if the function is inside the candidate vulnerable functions and contains stack buffers.

To determine whether buffers in a function are overflowed in the attack, SafeStack adds padding around the corresponding heap objects and identifies the bug-triggering objects with canary checking [4]. Specifically, canaries (that is, guard values) are placed around each object when it is moved to the heap region. When the triage system replays an attack, it checks whether the canaries have been tampered before the function returns. If so, the corresponding stack buffer will be identified as a bug-triggering buffer. If the triage system cannot find the bug-triggering buffers, it will enter into a heavy instrumentation, which moves all stack buffer objects into the heap space for the detection. In the heavy instrumentation, all the functions containing stack buffers will be instrumented.

The bug-triggering stack buffer is sent to the next step for patch generation. However, the diagnosis phase is not terminated immediately. This is because some data in the vulnerable source space could corrupt other buffer objects, which may cause the system to repeatedly enter and exit the vulnerability diagnosis phase till all the vulnerabilities are found. If the monitoring time is kept too long, however, the patch generation will be delayed and thus the online system kept in the vulnerability phase for a long time. In our current implementation, the monitoring time is chosen for returns of three instrumented functions after one bug-triggering buffer is identified, or a specified constant time is reached, or the program exits.

3.4 Patch Generation and Evaluation

Once the bug-triggering stack buffers are identified, patches are generated to "isolate" them on the stack. A runtime patch consists of a vulnerability signature, a bug-triggering buffer, and a vulnerability treatment. Based on the technique developed in First-Aid [21], SafeStack uses the return addresses of three most recent functions on the stack as the vulnerability signature. The bug signature can be determined during the time when the bug-triggering buffer is identified by checking the call-stack to obtain three most recent functions. The bug-triggering buffer contains the buffer size and the offset of the frame pointer. All vulnerability treatments are of the same type in the current implementation, i.e., moving the stack buffer to the heap and then mapping the memory addresses of the buffer on the stack to the corresponding addresses on the heap. However, SafeStack provides a common patch format, which can be easily integrated into the First-Aid system to provide more comprehensive protection for programs.

The generated patches are evaluated before they are applied to the online production system. The triage system first restarts the shadow deployment of the application and applies patches to the program, and then replays the inputs to check if the application can survive the exploit. If not, the system continues to monitor the patched application to find other vulnerabilities. If all the patches that the system can generate have been applied, but the exploit still exists, this exploit may not be caused only by stack buffer overflow and the triage system will throw an exception to the production system. A bug report is also generated during this process.

3.5 Runtime Patch Application

When receiving patches from the triage system, the patch management component in the production system notifies the runtime patch applicator to apply patches. For the patched application, once the vulnerability signature is matched by three most recent functions in the call-stack, the memory access virtualization mechanism will move the bug-triggering buffers to the heap address space. The moved objects are randomly placed on the heap to avoid them being overrun from each other [22].

The memory access virtualization mechanism ensures that all the read/write memory operations to the bug-triggering buffers are directed into its corresponding heap objects. This can be guaranteed by the memory access virtualization mechanism. Therefore, the patched application can stay clear of the stack buffer overflow vulnerability and prevent its recurrence.

3.6 Other Issues and Limitations

To diagnose buffer overflow vulnerabilities, we need to obtain stack buffer information, which includes function information and variable information. Function information contains function name (for nonstripped binaries), starting address, and the number of stack buffer variables. Variable information contains the variable size, offset from the frame pointer for each local buffer variables, and buffer parameters.

With stack buffer information, we can identify all those instructions which do the memory reading/writing or change the ESP/EBP register, and then infer local variable information from the instructions to determine the sizes of buffers and variables. Most of the times a buffer is referenced with a buffer pointer and an index and thus we can identify the buffer pointer from the base register. However, a direct memory access to the stack buffer can cause a buffer to be split into multiple small ones, but it is not common, as observed in our experiments. For inlined functions, the buffers originally allocated in its stack frame, but now they are allocated in the parent functions. Fortunately, inline functions work directly on application binaries, where they have already been inlined and compiled, and there is no need for any special treatments.

The stack buffer information can be extracted from program binaries. Currently, a disassembly library Libdasm [25] is used for the task. More accurate buffer information can also be obtained by integrating more sophisticated binary analysis tools such as IDAPro [26].

For most buffer overflow attacks, the call stack will be corrupted because the return address is corrupted. SafeStack uses a shadow call stack to maintain the call stack information and thus to prevent the system from entering into the heavy instrumentation. Our shadow call stack not only contains the current call stack information, but also the candidate vulnerable functions most-recently returned. In our current implementation, we maintain three most recently returned functions as the candidate vulnerable functions. There is a problem when using `setjmp/longjmp` in the program, which can disturb our work of shadow call stack in our current implementation. Although this is not common, improving it remains our future work.

As our main focus in the approach is to use memory access virtualization to generate accurate patches and apply patches, we assume the system states are in the exact same state after reset before each replay and therefore the attack can be reproduced in the triage system. SafeStack records recent network sessions using a first-in-first-out queue, as most attacks are caused by network inputs in the same short network session. For some applications such as banking applications, queries replay method may result in damaging data. This can be done with virtual machines or checkpoints of existing solutions. This remains our future work.

4 Experimental Evaluation

Our evaluation platform consists of three machines connected with 100-Mbps Ethernet. The first two machines are each configured with the Intel E5300 dual core 2.6-GHz processors, 2-MB L2 cache, and 2-GB memory. One is used for the production system and the other is for the triage system. The third machine is configured with Intel E5200 dual core 2.5-GHz processors, 2-MB L2 cache, and 2-GB memory. It is used to run clients. The operating system kernel is Linux 2.6, and Pin 2.8-37300 is used for dynamical instrumentation.

We evaluated the effectiveness of SafeStack with respect to survivability and performance. To test the survivability, we used a range of multiprocess and multithreaded applications as our testsuite. To test the performance, we first measured the overhead at a function call level, and then the overall performance at the application level.

TABLE 1
Applications Used in Evaluation

Application	Version	Vulnerability ID	Application Description	No. of Patches	Attack Recurrence Prevention
Apache	1.3.31	CVE-2004-0940	Web Server	1	Yes
T-HTTPd	2.21	CVE-2003-0899		1	Yes
Light-HTTPd	0.1	CVE-2002-1549		2	Yes
ATP-HTTPd	0.4b	Bugtraq ID 8709		1	Yes
ATP-HTTPd-i		Manually Injected		3	Yes
ProFTPD	1.3.3	CVE-2010-4221	FTP Server	1	Yes
Icecast	1.3.11	CVE-2002-0177	Streaming Audio Server	1	Yes
Newspost	2.1.1	CVE-2005-0101	Usenet Auto-poster	1	Yes
Prozilla	1.3.6	CVE-2005-2961	Download Accelerator	1	Yes

4.1 Overall Effectiveness Analysis

As shown in Table 1, there are eight applications used in our test, which are four web servers, i.e., Apache (enable the module Mod_include), T-HTTPd [27], Light-HTTPd [28], and ATP-HTTPd [29], an FTP server ProFTPD [30], a streaming audio server Icecase [31], an automatic news posting tool Newspost [32], and a download accelerator Prozilla [33].

We evaluated SafeStack using nine stack-based buffer overflow vulnerabilities in the applications: eight of the vulnerabilities are real-world vulnerabilities caused by buffers of characters. We also synthesized an additional vulnerability caused by buffers of integer, which was manually injected into the ATP-HTTPd (named ATP-HTTPd-i in our experiment results). To simulate attack recurrences in real attack scenarios, we mixed normal inputs with the vulnerability triggering inputs.

There were two types of stack overflow vulnerabilities in our test. One was caused by unbounded access to the stack. For example, the injected vulnerability in ATP-HTTPd-i was caused by an out-of-bound access in a `for` loop. The other was memory corruption via pointer operations. For example, there was an unbounded memory-copy vulnerability in Prozilla. Attackers can construct a malicious file which caused a negative number to be used as a copy parameter passed to a function `strncpy` to induce the memory corruption.

The overall effectiveness of SafeStack is illustrated in the last two columns in Table 1. The values in column *No. of Patches* referred to the number of runtime patches generated by SafeStack, i.e., the number of stack buffers moved out of the stack. All the patched applications avoided recurrence of the same attack, as is indicated by the last column.

We illustrate more details about SafeStack using the web server ATP-HTTPd as an example. SafeStack first analyzed the shadow call stack after detecting an attack. The shadow call stack contained six functions and three candidate vulnerable functions. The buffer overflow was caused by unbounded check memory copy `sprintf` which corrupted a stack buffer in function `http_send_error`. Then, SafeStack moved all the buffers in these vulnerable functions to the heap and adopted the canary checking technique once the vulnerable functions were called in the triage system. The stack buffer with a corrupted canary was identified as the bug-triggering buffer. In this experiment, SafeStack extracted three stack buffers in the function `http_send_error` from the program binary code and identified one buffer as the bug-triggering buffer. After that, SafeStack generated a patch for this buffer which contained a vulnerability signature (i.e., function `http_send_error`, `http_send_file` and `http_handler`), a bug-triggering buffer (i.e., the offset from the frame pointer was -844 and the size was 816 bytes) and a vulnerability treatment (i.e., stack objects relocation). SafeStack then evaluated the patch by applying it to the program after restarted the program and checking whether the patched program can survive the exploit after replaying the inputs. In our experiment, the generated patch can survive the exploit and the patch was applied in the production system. Therefore, the patched application in the production system can stay protected from the stack buffer overflow vulnerability and prevent its recurrence, which will be elaborated in the next section.

We also examined the accuracy of the buffer information extractor. There are five stack buffers in the function `Log` of the web server Light-HTTPd, but SafeStack identified them as two large buffers. The two actual bug-triggering buffers were identified as one large bug-triggering buffer. That is, SafeStack generated one patch, instead of two, in this function—the other patch was generated in another function. Although SafeStack cannot stop the two actual buffers in the large bug-triggering buffer from overrunning each other, it can stop them from corrupting other objects in the stack, making the application survive the exploit.

4.1.1 Prevention of Attack Recurrence

We evaluated the capability of attack prevention by comparing SafeStack with the restart method. We used a lightweight web server ATP-HTTPd in this experiment, and the result was shown in Fig. 4. From the figure, we can see that SafeStack can effectively prevent subsequent attacks caused by the same vulnerabilities after patches are applied into the applications. SafeStack spent about 1 s to generate a runtime patch in the triage system. After applying the patch, the performance of ATP-HTTPd remained stable irrespective of attacks occurrence.

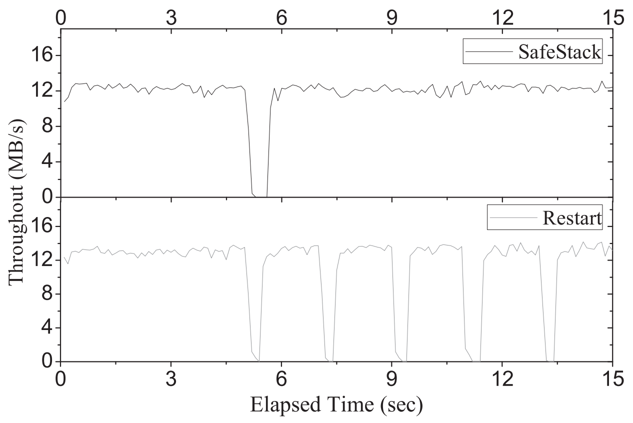


Fig. 4. Comparison between SafeStack and restart.

4.2 Performance Analysis

To measure the performance of SafeStack, we evaluated the elapsed time in the patch generation phase and the patch evaluation phase for each application in the triage system. The starting point for the measured time was when the triage system received an attack analysis request. Fig. 5 showed the average time for the patch generation and the patch evaluation, respectively.

As shown in Fig. 5, the average patch generation time varied between 0.462 s for ATP-HTTPd-i and 11.582 s for Newspost, and the average patch evaluation time ranged from 0.432 s for ATP-HTTPd-i to 11.172 s for Newspost. For desktop applications Newspost and Prozilla, the triage time was relative high. This is because it was mainly caused by the application itself, which needed a long time from the application start to the detection of an attack. For example, before posting a file to a malicious server, Newspost needed to wait for 10 s to post by default. After that, Newspost crashed when processing a malicious crafted package sent by the server. Therefore, it needed about 10.009 s to exploit the vulnerability. In the figure we can also see that the average patch generation time was almost the same with the average patch evaluation time except for Prozilla. The reason was that SafeStack terminated the application after three instrumented functions returned after detecting an attack. At that time, the file download had not been started. However, in the patch evaluation phase, SafeStack successfully made the patched application download the file, which needed more time without a doubt.

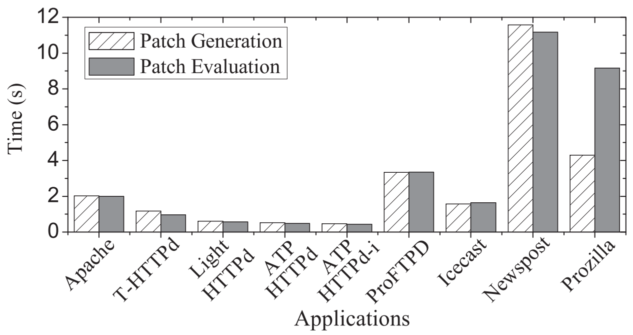


Fig. 5. Recovery performance.

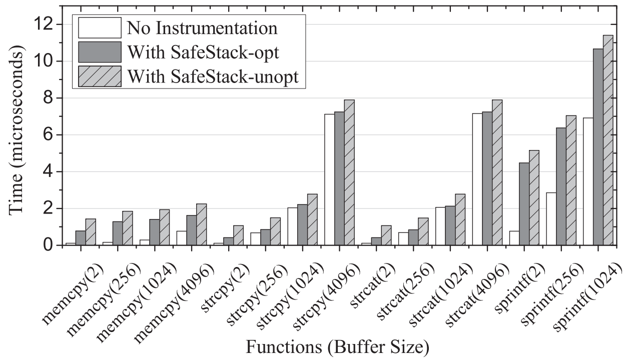


Fig. 6. Performance on function granularity.

4.3 Microbenchmarks

We did microbenchmark testing to evaluate the worst-case performance of our memory address virtualization mechanism. The stack buffers we used were character arrays. We evaluated SafeStack under both the optimized memory address virtualization mechanism and the unoptimized memory address virtualization mechanism.

To test the performance of our memory address virtualization mechanism for the former, we used four commonly used string handling functions: `memcpy`, `strcpy`, `strcat`, and `sprintf`. We measured three execution times for each of them: time without instrumentation, time for optimized SafeStack, and time for unoptimized SafeStack. In the test, we marked a stack buffer as a bug-triggering buffer and loaded a patch in SafeStack. The source buffer we used was allocated on the heap space. We changed the buffer size from 2 bytes to 4,096 bytes and got the execution time over the average time of 10,000 times of string operations. The result was shown in Fig. 6, except for the case `sprintf`(4,096) because the time was 15.662, 25.247 and $25.914\,\mu s$, respectively, which is too large to show in the figure.

As shown in Fig. 6, the overhead decreased while the buffer size increased. The minimum overhead for optimized SafeStack and unoptimized SafeStack was 1.11 times and 1.935 times for function `memcpy`, 1.756 and 10.902 percent for function `strcpy`, 1.245 and 10.39 percent for function `strcat`, and 61.2 and 65.458 percent for function `sprintf`, respectively.

To test the performance of our memory address virtualization mechanism on stack buffer element access, we used a loop (loop `for` in our experiment) to assign values to the stack buffer and got the execution time of the loop. We used three cases for the stack buffer and changed the buffer size from 2 bytes to 1,024 bytes with incrementing 20 bytes each time. The execution time was the average time of 10,000 times the loop and the result was shown in Fig. 7.

As shown in Fig. 7, the overhead of optimized SafeStack ranged from 1.071 times for 2 bytes and 29.031 percent for 1,024 bytes, and the overhead of unoptimized SafeStack ranged from 12.741 times for 2 bytes and 4.199 times for 1,024 bytes. Besides with this test, we tested the case when the buffer size was 4,096 bytes. The result was $45.853\,\mu s$ for original run, $58.602\,\mu s$ for instrumented run with optimized SafeStack, and $236.857\,\mu s$ for instrumented run with unoptimized SafeStack. The overhead was 27.804 percent and 4.166 time, respectively.

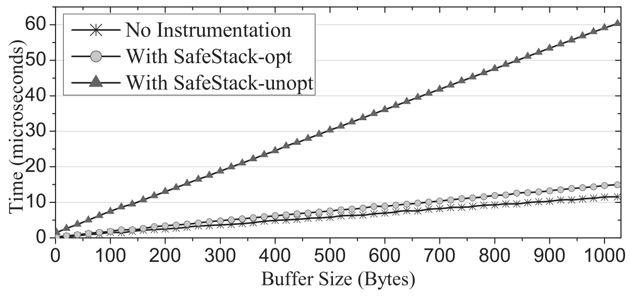


Fig. 7. Performance on loop granularity.

From the two experiments, we can see that our optimization for the memory address virtualization mechanism was effective. The reason was that optimized SafeStack only looked up the object-relocation table to get the corresponding heap address in instrumentation routines while unoptimized SafeStack did this in analysis routines. SafeStack had a modest overhead caused by our memory address virtualization mechanism when the buffer size was relatively big, but the overhead was relatively high when the buffer size was relatively small. Fortunately, for actual used applications, SafeStack only moved the bug-triggering buffers out of stack when the patches took effect. Therefore, the overhead for the analysis was amortized to the entire program execution, and we presented it in the next section.

4.4 Macro Benchmarks

We evaluated the normal execution overhead caused by SafeStack with these eight applications and the SPEC INT2000 benchmarks. In this experiment, we configured SafeStack in two cases. One was only enabling the memory address virtualization mechanism, and the other was enabling the memory address virtualization mechanism and shadow call stack. The workloads we used contained the distributed testing tools and the constructed workloads. For web servers, we adopted the Apache benchmark tool ab to test web servers. For the FTP server ProFTPD, we adopted an FTP benchmark tool dkftpbench. For the streaming audio server Icecast, we adopted a source client Ices [31] which encodes 10 audio files to a stream for broadcasting, and each size of audio files is 3.59 MB. For the Usenet autoposter Newspost, we posted 10 image files whose total size is 12.1 MB to another newsgroup. For Prozilla, we constructed a group of various sizes of files for downloading. The size ranges from 0 byte (empty file) to 256 MB with the average is 49.38 MB. For the SPEC benchmarks, we used the reference data sets as the workload. We selected a stack buffer to patch for each benchmark to get results for two cases. We compared the average response time for server applications and the execution time for desktop applications.

We show the overhead of patched applications in normal execution (i.e., when there were no attacks) in Fig. 8. SafeStack incurred acceptable overhead. It ranged from 0.1 to 20.628 percent with an average of 6.087 percent for the patched run. If the shadow call-stack mechanism is turned on, it incurred additional overhead (8.071 percent on average). For a better performance in a deployment, we can initially leave the shadow call stack off, and activate it when an attack is observed. The following properties of a system make the overhead of SafeStack reasonable. On one hand, SafeStack used the optimized memory address virtualization mechanism and successfully prevented subsequent attacks for all applications. Although there were some recursive functions calls in some applications, we found that such functions were not in the crashed call stack or they have no stack buffers. Therefore, it will not cause the unoptimized memory address virtualization mechanism to be used. On the other hand, our memory access virtualization mechanism worked when the patches took effect, which resulted in the overhead of the address mapping operation amortized to the entire application. In the figure, we can see that the patched run incurred no overhead for some applications because the patches did not take effect for the application's normal execution.

5 Related Work

In this section, we compare SafeStack with other solutions in preventing and responding to buffer overflow attacks.

5.1 Fail-Stop Approaches

There are many tools specially used to defend against stack buffer overflow attacks. For example, StackGuard [4] inserts a canary word before the return address on the stack, and

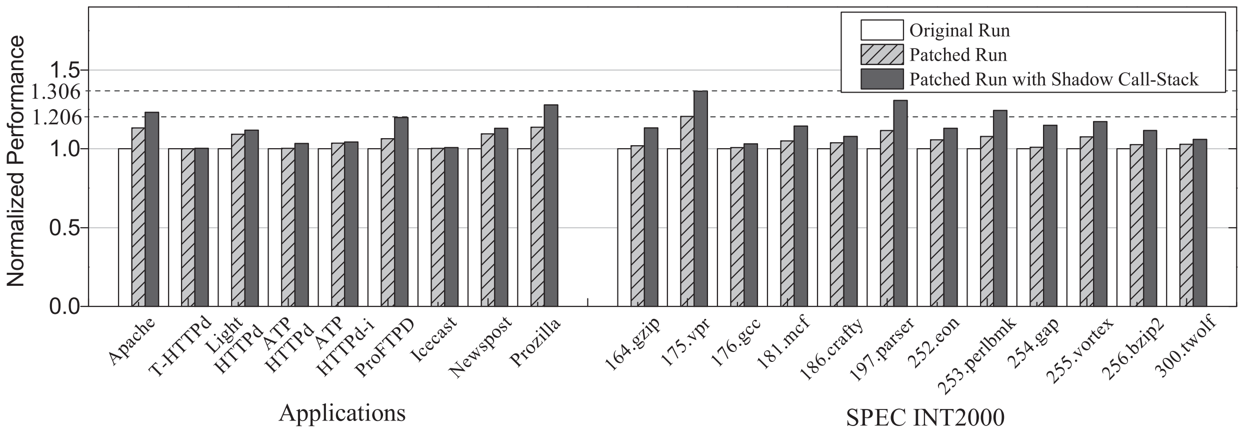


Fig. 8. Overhead for normal execution.

verifies the canary before the function returns. StackShield [5] is similar to StackGuard, but it stores a copy of the return address when entering a function and restores it when the function returns. PointGuard [9] protects pointers by encrypting the pointers when they are stored in memory and decrypting the pointers when they are loaded into CPU registers. CRED [8] is a bounds checker for C and supports for some common uses of out-of-bounds pointers in existing C programs. WIT [35] prevents memory error exploits by the points-to analysis at compile time and enforces write integrity and control-flow integrity at runtime. All these approaches are extensions to the GNU C compiler, but our SafeStack does not rely on the compiler extensions as it extracts the buffer information from the binary code.

Sidiroglou and Keromytis propose the first end-point architecture for automatically repairing software flaws [14]. It adopts source code transformations including memory-safety transformations which target the specific failing function. Specially, it transforms the offending buffer to the heap. Bhatkar et al. propose a comprehensive randomization technique to protect the program against memory error exploits by applying a source-to-source transformation [12]. Specially, it transforms all the stack buffers into heap buffers. The requirement of program source code limits the application of these approaches.

Baratloo et al. propose two methods, Libsafe and Libverify, to defend against stack smashing attacks [36]. Libsafe replaces the vulnerable C library functions with a safe implementation. Libverify is similar to StackShield, but it injects verification codes at the beginning of program execution. A combination of two tools still cannot defend against buffer overflow that is caused by programmer's copy functions or the third library functions, which can be used to corrupt the code pointers.

A binary rewriting defense technology is proposed to protect the return address on the stack [6]. It just applies a combination of well-known disassembly techniques to insert the return address defense code to protect the integrity of the return address with a redundant copy. However, it just only protects the return address, and it is ineffective against corrupting the code pointers.

5.2 Availability-Preserving Approaches

Keromytis characterizes self-healing software systems [37] and proposes a general architecture of a self-healing system, including self-monitor to monitor anomalous behavior, self-diagnosis to identify faults, self-adaptation to generate candidate fixes and self-testing to find the best fix to deploy. Keromytis points out the research direction in this field and we follow this architecture to design our system.

Reboot technology attempts to restart the program when a fault is encountered. It includes the whole program restart, rebooting faulty parts of software components (Micro-reboot [23]) and software rejuvenation [38], which is an active approach to periodically reboot the program to deal with software aging. However, they are unable to tolerate exploits, whose deterministic vulnerabilities can be triggered.

Failure-oblivious computing [34] tolerates memory-related vulnerabilities by manufacturing values for "out-of-bound read" and discarding "out-of-bound write." However, it needs the program to be recompiled with its safe compiler to generate failure-oblivious codes. Moreover, it may result in unpredicted behavior which imposes a new threat to the program.

Rx [13] applies a method to replay the program in a changed execution environment. However, to survive stack buffer overflow attacks, it just drops user's requests which contain the malicious request.

There are several methods developed to deal with heap-related vulnerabilities such as DieHard [22] and First-Aid [21]. First-Aid is an extension to Rx. It tolerates vulnerabilities by identifying the vulnerability types and bug-triggering memory objects, and generating runtime patches to temporarily fix the vulnerability.

Another direction is to learn from attacks and automatically "heal" the vulnerable application through an attack signature, for example, COVERS [17] and Vigilante [40].

Error virtualization [15], [16], [39], [41], [42], [43], [45] is an efficient technique that force a heuristic-based error value from a function where a fault occurs, to bypass "faulty" region of codes. STEM [41] speculatively emulates the faulty region of codes in a virtual core and adopts error virtualization to recover from faults. STEM can also be used in application communities to collaboratively detect and tolerate software faults and vulnerabilities [39]. SEAD [45] adopts a dynamic instrumentation tool to dynamic load STEM into the application. ASSURE [15] uses a dynamic instrumentation tool to implement error virtualization and a rescue point technique to record the information of functions. ASSURE may cause the application not to function well, SHelp [16] alleviates the problem by applying "weighted" rescue points and extends it to a virtualization computing environment. However, all of these techniques change the execution flow of an application and cannot guarantee that new threats are not introduced.

Band-aid patching technique [44] is proposed to guarantee correctness after applying patches. It runs the old and new versions of patched code in sequence. If the old or new version crashes, it is discarded. This approach should be handled with the resource duplication and the coexistence of old and new versions.

6 Conclusions

In this paper, we presented a new system SafeStack which can automatically patch stack-based buffer overflow vulnerabilities. The system focuses on the vulnerability's origin, that is, the vulnerability triggering stack buffers. The system first tries to identify the bug-triggering stack buffers and move them out of the stack to a protected space using a novel memory access virtualization technique, then diagnoses vulnerabilities and finally generates runtime patches to "fix" the vulnerabilities temporarily to protect the applications from the subsequent recurrence of the same attacks. We have developed a prototype system and evaluated the system's effectiveness using a range of applications with different bugs. The experimental results demonstrate that our system can quickly generate runtime patches to successfully tolerate recurrence of the same attack. In addition, patches generated by SafeStack incur low runtime overhead during the applications' normal execution.

Memory access virtualization is a key mechanism we used to deal with stack buffer overflow attacks. As mentioned before, this mechanism can also provide a fine-grained memory randomization for memory objects in runtime. Therefore, it could be used in several other scenarios. For example, it can be used to randomize stack objects of a function once the function is called, which can make the attacks more difficult. Combined with ASLR [10] they provide memory randomization in the load time.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their insightful comments as well as Mingwei Zhang for his help in the discussion of techniques for identifying stack variables. This paper was supported by National 973 Plan of China under grant no.2014CB340600, National High-Tech R&D Program (863 Program) of China under grant no. 2012AA012600, and the National Natural Science Foundation of China under grant no. 61272072.

REFERENCES

[1] "US-CERT Vulnerability Notes Database," http://www.kb.cert.org/vuls/bymetric?open\&start=1\&count=20, 2013.

[2] "Internet Security Threat Report," http://www.symantec.com/enterprise/threatreport/index.jsp, 2013.

[3] M. Nicholls, "Tutorial: SEH Based Exploits and the Development Process," http://www.ethicalhacker.net/content/view/309/2/, 2013.

[4] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. Seventh Conf. USENIX Security Symp.*, pp. 63-78, 1998.

[5] "A Stack Smashing Technique Protection Tool for Linux," http://www.anglefire.com/sk/stackshield, 2012.

[6] M. Prasad and T. Chiueh, "A Binary Rewriting Defense against Stack Based Buffer Overflow Attacks," *Proc. USENIX Annu. Technical Conf.*, pp. 211-224, 2003.

[7] T. Chiueh and F. Hsu, "RAD: A Compile-time Solution to Buffer Overflow Attacks," *Proc. 21st Int'l Conf. Distributed Computing Systems*, pp. 409-417, 2001.

[8] O. Ruwase and M. Lam, "A Practical Dynamic Buffer Overflow Detector," *Proc. 11th Annu. Network Distributed System Security Symp.*, pp. 159-169, 2004.

[9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," *Proc. 12th Conf. USENIX Security Symp.*, pp. 91-104, 2003.

[10] PaX Team, "PaX," http://pax.grsecurity.net, 2013.

[11] S. Bhatkar, D.C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proc. 12th USENIX Security Symp.*, 2003.

[12] S. Bhatkar, R. Sekar, and D. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th Conf. USENIX Security Symp.*, pp. 271-286, 2005.

[13] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures," *Proc. 20th ACM Symp. Operating System Principles*, pp. 235-248, 2005.

[14] S. Sidiroglou and A. Keromytis, "Countering Network Worms through Automatic Patch Generation," *IEEE Security Privacy*, vol. 3, no. 6, pp. 41-49, Nov./Dec. 2005.

[15] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. Keromytis, "ASSURE: Automatic Software Self-Healing Using REscue Points," *Proc. 14th Int'l Conf. Architectural Support Programming Languages Operating Systems*, pp. 37-48, 2009.

[16] G. Chen, H. Jin, D. Zou, B. Zhou, W. Qiang, and G. Hu, "SHelp: Automatic Self-Healing for Multiple Application Instances in a Virtual Machine Environment," *Proc. IEEE Int'l Conf. Cluster Computing*, pp. 97-106, 2010.

[17] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers," *Proc. 12th ACM Conf. Computer Comm. Security*, 2005.

[18] A. Keromytis, "'Patch on Demand' Saves Even More Time?" *Computer*, vol. 37, no. 8, pp. 94-96, 2004.

[19] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," *Proc. 12th ACM Conf. Computer Comm. Security*, 2005.

[20] A. Smirnov and T. Chiueh, "Automatic Patch Generation for Buffer Overflow Attacks," *Proc. Third Int'l Symp. Information Assurance Security*, pp. 165-170, 2007.

[21] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-Aid: Surviving and Preventing Memory Management Bugs during Production Runs," *Proc. Fourth ACM European Conf. Computer Systems*, pp. 159-172, 2009.

[22] E. Berger and B. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," *Proc. ACM SIGPLAN Conf. Programming Language Design Implementation*, pp. 158-168, 2006.

[23] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—A Technique for Cheap Recovery," *Proc. Sixth Conf. Symp. Operating Systems Design Implementation*, pp. 31-44, 2004.

[24] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design Implementation*, pp. 190-200, 2005.

[25] "Libdasm: A Disassembly Library," http://code.google.com/p/libdasm/, 2013.

[26] Hex-Rays, "IDAPro Multi-Processor Disassembler and Debugger," http://www.hex-rays.com/products/ida/index.shtml, 2013.

[27] Symantec, "Thttpd Defang Remote Buffer Overflow Vulnerability," http://www.securityfocus.com/bid/8906/, 2013.

[28] LHTTPd Development Team, "A Light HTTP Server and Content Management System," http://lhttpd.sourceforge.net/, 2013.

[29] Symantec, "Atphttpd Remote GET Request Buffer Overrun Vulnerability," http://www.securityfocus.com/bid/8709/discuss/, 2013.

[30] ProFTPD, "A Highly Configurable GPL-Licensed FTP Server Software," http://www.proftpd.org/, 2013.

[31] Icecast, "A GPL Streaming Media Server," http://www.icecast.org/, 2012.

[32] Symantec, "Newspost Remote Buffer Overflow Vulnerability," http://www.securityfocus.com/bid/12418/, 2013.

[33] Symantec, "Prozilla Buffer Overflow Vulnerability," http://www.securityfocus.com/bid/14993, 2013.

[34] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee Jr., "Enhancing Server Availability and Security through Failure-Oblivious Computing," *Proc. Sixth Conf. Symp. Operating Systems Design Implementation*, pp. 303-316, 2004.

[35] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," *Proc. IEEE Symp. Security Privacy*, pp. 263-277, 2008.

[36] A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-Time Defense against Stack Smashing Attacks," *Proc. USENIX Annu. Technical Conf.*, pp. 251-262, 2000.

[37] A.D. Keromytis, "Characterizing Self-Healing Software Systems," *Proc. Fourth Int'l Conf. Math. Methods, Models Architectures Computer Networks Security*, 2007.

[38] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proc. 25th Annu. Int'l Symp. Fault-Tolerant Computing*, pp. 381-391, 1995.

[39] M.E. Locasto, S. Sidiroglou, and A.D. Keromytis, "Software Self-Healing Using Collaborative Application Communities," *Proc. Internet Soc. Symp. Network Distributed Systems Security*, pp. 95-106, 2006.

[40] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," *Proc. 20th ACM Symp. Operating Systems Principles (SOSP '05)*, 2005.

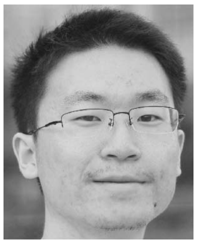
[41] S. Sidiroglou, M.E. Locasto, S.W. Boyd, and A.D. Keromytis, "Building a Reactive Immune System for Software Services," *Proc. USENIX Annu. Technical Conf.*, pp. 149-161, 2005.

[42] S. Sidiroglou, M.E. Locasto, and A.D. Keromytis, "Hardware Support for Self-Healing Software Services," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 1, pp. 42-47, 2005.

[43] S. Sidiroglou, G. Giovanidis, and A.D. Keromytis, "A Dynamic Mechanism for Recovering from Buffer Overflow Attacks," *Proc. Eighth Int'l Conf. Information Security (ISC '05)*, pp. 1-15, 2005.

[44] S. Sidiroglou, S. Ioannidis, and A. Keromytis, "Band-Aid Patching," *Proc. Third Workshop Hot Topics System Dependability*, pp. 102-106, 2007.

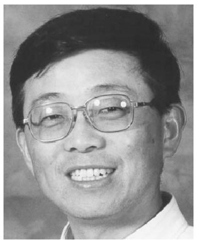
[45] M.E. Locasto, A. Stavrou, G.F. Cretu, and A.D. Keromytis, "From STEM to SEAD: Speculative Execution for Automated Defense," *Proc. USENIX Annu. Technical Conf.*, pp. 219-232, 2007.



Gang Chen received the BS degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2008, where he is currently working toward the PhD degree in computer science and technology. His research interests include software security and reliability in cloud computing.

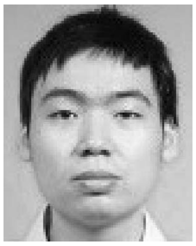
Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 1994. He is currently a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST. He is also the dean of the School of Computer Science and Technology at HUST. He worked at the University of Hong Kong between 1998 and 2000 and as a visiting scholar at the University of Southern California between 1999 and 2000. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of the National 973 Basic Research Program Project of Virtualization Technology of Computing System. He is the member of Grid Forum Steering Group. He has coauthored 15 books and published more than 400 research papers. His research interests include computer architecture, virtualization technology, cluster computing and grid computing, peer-to-peer computing, network storage, and network security. He is the steering committee chair of the International Conference on Grid and Pervasive Computing, the Asia-Pacific Services Computing Conference, the International Conference on Frontier of Computer Science and Technology, and the Annual ChinaGrid Conference. He is a member of the steering committee of the IEEE/ACM International Symposium on Cluster Computing and the Grid, the IFIP International Conference Network and Parallel Computing, and the International Conference Grid and Cooperative Computing, the International Conference Autonomic and Trusted Computing, and the International Conference Ubiquitous Intelligence and Computing. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001. In 1996, he was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany. He is a senior member of the IEEE and a member of the ACM.

Deqing Zou received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST) in 2004. He is currently a professor of computer science at HUST. His main research interests include system security, trusted computing, virtualization, and cloud security.

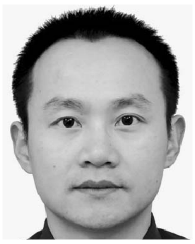


Bing Bing Zhou received the BS degree from the Nanjing Institute of Technology, China, and the PhD degree in computer science from the Australian National University. He is currently an associate professor at the University of Sydney, Australia. His research interests include parallel/distributed computing, grid and cloud computing, peer-to-peer systems, parallel algorithms, and bioinformatics. He has a number of publications in leading international journals and conference proceedings. His research has been funded by the Australian Research Council through several Discovery Project grants.

Zhenkai Liang received the BS degree from Peking University in 1999 and the PhD degree from Stony Brook University in 2006. He is currently an assistant professor at the School of Computing, National University of Singapore. His main research interests include system security, software security, and software debugging. His research focuses on signature generation for remote attacks, malicious program analysis and confinement, web security, and debugging techniques. As a coauthor, he received the ACM SIGSOFT Distinguished Paper Award at the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering in 2009, the Best Paper Award at the USENIX Security Symposium in 2007, and the Outstanding Paper Award at the Annual Computer Security Applications Conference in 2003. He also received the Young Investigator Award from the National University of Singapore in 2008.



Weide Zheng received the BS and MS degrees in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2009 and 2012, respectively. His main research interests include fault tolerance and security issues in cloud computing, virtualization, and networks. He is currently a software engineer at Baidu Inc.



Xuanhua Shi received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. In 2006, he worked as an INRIA postdoctoral researcher with PARIS team at Rennes. He is currently an associate professor in both the Service Computing Technology and System Lab and the Cluster and Grid Computing Lab at HUST. His research interests include cloud computing, data intensive computing, fault tolerance, virtualization technology.