Automatic Diagnosis and Response to Memory Corruption Vulnerabilities

Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, Chris Bookholt
Cyber Defense Laboratory
Department of Computer Science
North Carolina State University
{jxu3, pning, ckil, yzhai, cgbookho}@ncsu.edu

ABSTRACT

Cyber attacks against networked computers have become relentless in recent years. The most common attack method is to exploit memory corruption vulnerabilities such as buffer overflow and format string bugs. This paper presents a technique to automatically identify both known and unknown memory corruption vulnerabilities. Based on the observation that a randomized program usually crashes upon a memory corruption attack, this technique uses the crash as a trigger to initiate an automatic diagnosis algorithm. The output of the diagnosis includes the instruction that is tricked to corrupt data, the call stack at the time of corruption, and the propagation history of corrupted data. These results provide useful information in fixing the vulnerabilities. Moreover, the diagnosis process also generates a signature of the attack using data/address values embedded in the malicious input message, and is used to block future attacks. Such a signature is further associated with the program execution state to reduce false positives without decreasing the detection rate. The proposed techniques enable the development of a decentralized self-diagnosing and self-protecting defense mechanism for networked computers. We report the implementation experience and experimental evaluation of a prototype system on Linux.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*invasive software*; C.2.0 [**Computer-Communication Networks**]: Security and Protection

General Terms

Security, Experimentation

Keywords

memory corruption attack, randomization, attack diagnosis, message filtering

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'05, November 7–11, 2005, Alexandria, Virginia, USA.
Copyright 2005 ACM 1-59593-226-7/05/0011 ...$5.00.

1. INTRODUCTION

Cyber attacks against Internet connected computer systems, including those in the critical infrastructures, have become relentless in recent years. Malicious attackers break into computer systems using a variety of techniques. The most common method is to exploit memory corruption vulnerabilities such as buffer overflow, format string, and double free. These vulnerabilities are not only exploited by individual intruders, but also facilitate the success of large-scale Internet worms and distributed denial of service (DDoS) attacks. Many defensive methods against such attacks have been investigated in the past several years, including static analysis techniques (e.g., [8, 14]), compiler extensions (e.g., [11, 13]), safe library functions (e.g., [2]), execution monitoring techniques (e.g., [22, 21]), and intrusion detection (e.g., [4, 30, 38]).

Recent fast spreading worms (e.g., Code Red [5], SQL Slammer [6], Blaster [7]) motivated the investigation of more efficient and effective defense mechanisms that can stop such attacks. Shield [39] has been developed to provide temporary protection of vulnerable systems after the vulnerabilities are identified but before patches are properly applied. However, Shield requires manually generated signatures derived from known vulnerabilities. Honeycomb [23], Autograph [18], EarlyBird [34], and Polygraph [27] attempted to automatically generate attack (especially worm) signatures from attack traffic. A limitation of these signature generation techniques is that they all rely on other means to identify attack traffic. Moreover, they all extract signatures in a syntactic manner, while ignoring the semantics of the attacks. Thus, it takes additional effort to pinpoint the vulnerabilities exploited by the attacks, though the derived signatures are helpful in patching the vulnerabilities.

Several recent approaches attempt to monitor the execution of potentially vulnerable programs through code instrumentation or emulation, and recover from attacks when they occur. TaintCheck [28] performs dynamic taint analysis (via binary program emulation) to track the propagation of network input data, and raise alerts when such data is directly or indirectly used illegitimately. TaintCheck also generates semantic-based attack signatures from the network input data that eventually leads to an alert. DIRA [36] instruments the code to maintain a memory update log while a program is executed, and rolls back the memory updates to "clean" state when an attack is detected. STEM [33] takes a reactive approach; after an attack is detected, it replaces the vulnerable program with an automatically patched version, which selectively emulates potentially vulnerable code seg-

ments (through code instrumentation). All these approaches can handle attacks exploiting unknown vulnerabilities to a certain extent, and provide useful information in patching the vulnerabilities. However, the use of program emulation, or code instrumentation introduces substantial overhead in program execution. It is desirable to seek alternative, more efficient mechanisms to achieve the same goals.

In this paper, we develop a novel technique to automatically identify (unknown) memory corruption vulnerabilities. The proposed technique enables us to trace back to the vulnerable instructions that corrupt memory data upon a memory corruption attack. It is observed that a program in a randomized system crashes with an extremely high probability upon such attacks. Based on this, our technique uses a crash of a randomized program as a trigger to start the diagnosis of memory corruption vulnerabilities. The output of the diagnosis process includes the instruction exploited by the attacker to corrupt critical program data, the stack trace at the time of the memory corruption, and the history that the corrupted data is propagated after the initial data corruption. The automated diagnosis process provides useful information in fixing the diagnosed vulnerabilities. Moreover, our technique also automatically generates signatures of the attacks exploiting both known and unknown vulnerabilities. Specifically, such a signature consists of the program state at the time of attack and the memory address values used in the attack, allowing efficient and effective protection of the vulnerable program by filtering out future attacks. These techniques enable the development of a fully decentralized self-diagnosing and self-protecting defense mechanism for networked computer systems.

Our approach shares a similar goal to TaintCheck [28], DIRA [36], and STEM [33]. That is, we would like to automatically recover from attacks and prevent future ones. However, our approach uses different mechanisms to perform recovery and prevention of future attacks. Unlike TaintCheck which catches potential network attacks via dynamic taint analysis of network inputs, our approach uses a backward tracing approach to identify memory corruption vulnerabilities starting from the program crash points. Unlike DIRA which constantly monitors the execution and maintains a memory update log, our approach starts analysis only when there is an attack, and thus does not introduce performance overhead during normal server operations. Moreover, unlike STEM [33], which selectively emulates potentially vulnerable code segments, our approach automatically identifies vulnerable instructions, generates attack signatures, and uses content filters to prevent future attacks, thus introducing less overheads. Finally, our signature generation method is an enhancement to the one used by TaintCheck. By incorporating program state into each signature, we can significantly reduce the false positive rate without decreasing the detection rate. The contributions of this paper are listed below:

1. We develop a model for general memory corruption attacks against programs running in a system with address space randomization. The model can be used for the analysis and defense of such attacks.
2. We develop a series of techniques to automatically diagnose memory corruption vulnerabilities. Our techniques enable a victim system under attack to quickly locate the instructions exploited by a memory corruption attack to overwrite critical program data. Such information allows the victim system to protect itself against similar future attacks, and also facilitates programmers to fix the problems quickly.
3. We enhance the signature generation techniques in [28] to reduce the false positive rate. Our analysis and experiments indicate that associating program state with signature can effectively reduce the false positive rate and performance overhead without increasing false negatives.
4. We implement the proposed system and perform a sequence of experiments to evaluate the proposed techniques on Linux.

The rest of this paper is organized as follows. Section 2 gives an overview of the proposed system and its architecture. Section 3 describes a model for memory corruption attacks, which guides the development of our approach. Section 4 presents the proposed automatic diagnosis technique. Section 5 discusses our automatic response techniques for attacks. Sections 6 and 7 present implementation and evaluation of our system, respectively. Section 8 discusses related work, and Section 9 concludes this paper.

2. SYSTEM OVERVIEW

In this paper, we focus on network service applications (e.g., *httpd, sshd*), because they are critical to the availability of the Internet infrastructure and are frequent targets of attacks.

We harness the results in address space randomization (e.g., [3, 29, 42]) to facilitate the detection of attacks exploiting both known and unknown memory corruption vulnerabilities. It is well-known that a memory corruption attack typically causes a randomized program to crash due to the difficulty in predicting memory addresses. Thus, we use the crashes of such randomized programs as the triggers to start vulnerability analysis.

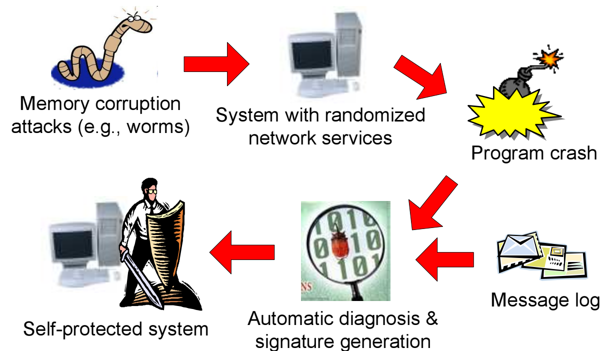

Figure 1: Detecting, diagnosing, and recovering from memory corruption attacks

Figure 1 illustrates the life cycle of the detection, diagnosis, and recovery process in the proposed system. The proposed system runs network service applications using address space randomization. When a memory corruption attack (e.g., a new worm) attempts to exploit an unknown vulnerability in a service program, it typically causes the corresponding process to crash. This crash then triggers the automatic analysis of this attack and the vulnerability, with the input from the process image at the time of crash (e.g.,

the program counter, stack trace) and the relevant messages received by the service process. Our proposed approach then automatically analyzes all the information, identifies the specific vulnerability in the program and the message causing the crash, and generates a signature of the attack, which is used to block future attacks exploiting the same vulnerability.

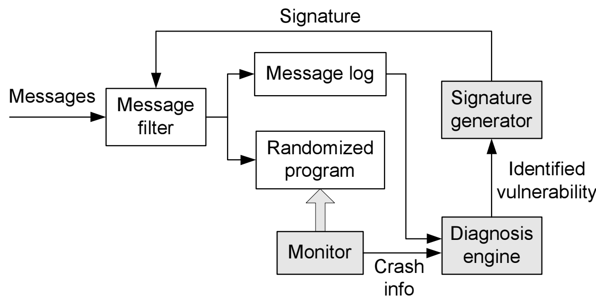

Figure 2: System architecture

Figure 2 shows the architecture of the proposed system. We use a *monitor* to supervise a randomized version of each network service program. All the incoming messages to such a service program are passed through a *message filter*, which initially forwards all the messages to the service program. These messages are also stored in the *message log* for possible future analysis. The monitor has minimum interaction with the randomized program after starting it. The responsibilities of the monitor include (1) identifying the crash of the program (e.g., by catching memory access violation signals), and (2) upon the program crash, collecting information (e.g., program counter and stack trace) necessary for automatic vulnerability analysis. When the automatic diagnosis is triggered by a program crash, the monitor passes the collected crash information to the *diagnosis engine*, which then automatically analyzes the crash information and the messages that potentially cause the crash. The monitor may re-execute the crashed program for a few more times to collect more information. Using the proposed techniques, the diagnosis engine can pinpoint attack message(s), the vulnerable instruction exploited by the attack message, the stack trace at the time of crash, and the history of the corrupted data propagation. The diagnosis engine then generates a signature for the attack, which is used by the message filter to drop future attack messages that exploit the same vulnerability.

3. MODELING MEMORY CORRUPTION ATTACKS

In this section, we describe a model for memory corruption attacks, which will be used to guide the later discussion of our approach.

We first define a few terms used in this model. In a victim program under memory corruption attack, we refer to an instruction that is tricked to overwrite critical program data (e.g., return address on the stack) as a *corrupting instruction*, denoted as c. Note that one memory corruption attack may involve multiple corrupting instructions. Among the corrupting instructions, we refer to those that corrupt program data purely based on the network input as *initial corrupting instructions*. By contrast, the other corrupting instructions may simply propagate corrupted data (e.g., copy to other locations). We observe that to take control of a victim process, an attack usually needs to execute a control flow transfer instruction (e.g., `jmp`, `ret`, and `call`) to further execute the code injected or chosen by the attack. We call such a control flow transfer instruction a *takeover instruction*, denoted as t. Finally, we refer to the instruction that causes a process to crash as a *faulting instruction*, denoted as f.

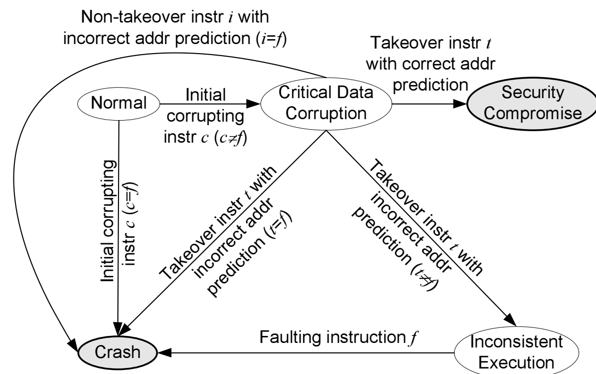

Figure 3: State transition of a randomized program under a memory corruption attack

Figure 3 shows the state transition of a randomized program under a general memory corruption attack. In normal situations, the program stays in the *Normal* state. In order to compromise the target program, a memory corruption attack sends maliciously constructed messages to the program. Due to security vulnerabilities, the corrupting instruction c in the program is tricked to overwrite critical program data (e.g., return address on the stack) while processing such an input. An attacker needs to predict certain memory address (e.g., the above return address) to corrupt critical program data, and then execute a takeover instruction t (e.g., `ret`) to transfer the program control to the code injected or chosen by the attacker. In a successful attack where the attack uses a correctly predicted address, the program moves from the *Normal* state into the *Critical Data Corruption* state by executing a corrupting instruction c, and then transits into the *Security Compromise* state (and takes over the victim program) after executing a takeover instruction t.

Address space randomization makes the attacker's prediction of critical addresses difficult, and forces the attack to use incorrect addresses. As a result, a memory corruption attack leads to one of the following four cases:

- **Case I:** Executing a corrupting instruction directly causes the process to crash. In this case, the corrupting instruction is the faulting instruction (i.e., $c = f$). The victim program directly transits into the *Crash* state from the *Normal* state. For example, in a format string attack, when a *printf*-like function uses malicious network input as an address to corrupt memory, it will crash due to the use of an invalid address value.
- **Case II:** The attack corrupts some critical program data without causing the crash. Thus, the program successfully transits into the *Critical Data Corruption* state from the *Normal* state. However, the process crashes when executing a later non-takeover instruc-

tion i that causes memory access violation. (We distinguish this case from case III, where the victim program executes a takeover instruction.) The program then transits from the *Critical Data Corruption* state to the *Crash* state. In this case, the instruction i is the faulting instruction (i.e., $i = f$). For example, in a stack buffer overflow attack, the overrun may corrupt return address as well as other stack variables. An instruction that dereferences such a variable before the function returns may cause a crash.

- **Case III:** The attack corrupts critical program data without causing the crash. However, the process crashes when executing the takeover instruction t, since the takeover instruction attempts to access an incorrect address. In this case, the takeover instruction is the faulting instruction (i.e., $t = f$). Thus, the program first transits from the *Normal* state into the *Critical Data Corruption* state, and then into the *Crash* state. For example, in a stack overflow attack, the corrupted return address would be invalid and cause the crash when the `ret` instruction is executed.
- **Case IV:** The program successfully transits into the *Critical Data Corruption* state, as in cases II and III. Then the compromised program successfully executes the takeover instruction without an immediate crash, and continues to execute for some time before it executes a faulting instruction. (Such cases have been shown in a previous study [41].) In other words, the corrupted, randomized program transits into the *Inconsistent Execution* state even though the attacker makes wrong address prediction, and then transits into the *Crash* state when it executes a faulting instruction f. In this case, the faulting instruction is executed after c and t.

The key difference between Case I & II is that no datum has been corrupted in Case I, while some data have already been corrupted in case II.

4. DIAGNOSING MEMORY CORRUPTION VULNERABILITIES

Based on the above discussion, we propose a *backward tracing* approach to automatically locating the (unknown) memory corruption vulnerabilities exploited by novel attacks. As discussed in Section 2, the monitor starts each network service program using a randomized version of the program. When this process crashes due to memory corruption attacks, the monitor takes control by intercepting the memory access violation exception. The monitor and the diagnosis engine then start the automatic diagnosis process.

Our approach aims to identify the first one or several corrupting instructions during a memory corruption attack. This instruction can be used to easily identify the counterpart in the program source code with the stack trace and possibly auxiliary data collected during compilation, thus providing useful information in fixing the vulnerability. In the next section, we also use this information to automatically generate a signature of the memory corruption attack, and deploy the signature at the message filter to block future attacks.

Consider the four cases discussed in Section 3. We observe that the takeover instruction t is a dividing factor of these four cases. Before the takeover instruction is executed (cases I and II), the victim process only executes the instructions intended in the original program, though the operands may be corrupted. However, after the takeover instruction is executed (case IV), the victim process may have executed instructions injected or chosen by the attack. Thus, it is more difficult to trace back to the initial corrupting instruction.

We adopt the following strategy to develop our approach. We first eliminate case IV by converting a Case-IV crash into one of the other three cases. For a Case-III crash, we need to identify the takeover instruction, which is also the faulting instruction. Since a takeover instruction is a control flow transfer instruction in case III, the takeover instruction must have accessed an incorrect address pointed by the corrupted data. We then locate the address where this corrupted data is stored, re-execute the crashed program, and at the same time monitor this address to catch the instruction that writes the corrupted data. This instruction is obviously a corrupting instruction, which either initially corrupts the data, or helps propagate the corrupted data. In the latter situation, the corrupted data may have been copied from another location. Thus, we iteratively perform the above process to identify the previous location and corrupting instruction, until we reach the point where the program receives the network input data.

In a Case-I crash or a Case-II crash where the faulting instruction uses a faulty address derived from the malicious input, there is no need to further trace back, since we can already identify the vulnerable instruction affected by the malicious input. However, in a Case-II crash where the faulty address is derived from corrupted data, we need to identify the source of the faulty address, and then trace back to the initial corrupting instruction as in case III.

Our approach consists of the following two steps:

1. **Convert Case-IV crashes.** This step eliminates Case-IV crashes by converting such crashes into one of the other three cases. Intuitively, we re-execute the monitored program with a different memory layout, and force the takeover instruction or an instruction before it to crash the process, if the original crash is a Case-IV crash. At the end of this step, we can distinguish these cases by checking the faulting instruction. If the faulting instruction is a memory access instruction (e.g., `mov`), the (converted) crash is either case I or case II. If the faulting instruction is a control flow transfer instruction (e.g., `jmp`, `ret`), the (converted) crash must be case III.
2. **Trace the corrupting instructions.** The goal of this step is to trace back as close to the initial corrupting instruction as possible. As discussed above, a Case-I crash or a Case-II crash where the faulting instruction uses a faulty address derived from the malicious input, there is no need to further trace back. In the remaining situations, we first identify the location where the corrupted address used by the faulting instruction is stored, and then trace back to the instruction that writes the corrupted data. This process may repeat until we reach the network input data.

In both steps, the monitor supervises the re-execution of the victim program, stops the monitored process occasionally to collect data, and facilitates the diagnosis engine to

trace the attack and to eventually identify the potentially unknown memory corruption vulnerability. The method can be viewed as an automated debugging process.

A critical issue in this diagnosis process is to locate the faulting instruction. Though the faulting instruction f causes the process to crash, the process image at the time of crash does not include the specific address of f. In particular, the CPU's program counter (PC) register does not point to f, rather it contains the address of the next instruction to be executed should f complete. Because f could be any instruction using a faulty address (e.g., function call or return), the address of f may be very far from the address in PC.

We propose to use *monitored re-execution and programmable breakpoints* to locate the faulting instruction. In the following, we first present this technique, and then discuss the two steps of our approach in detail.

4.1 Identifying the Faulting Instructing

To correctly determine the address of the faulting instruction f, we distinguish between two cases: a *simple case* where f is the immediate preceding instruction of the current PC, and a *complex case* where f is an indirect control flow transfer instruction that uses a corrupted target address. These two cases can be distinguished as follows: if the current PC value matches the invalid memory address that causes the access violation, then the faulting instruction f must be an indirect control flow transfer. We then have the complex case. Otherwise, we have the simple case, and f is the immediate predecessor of the current PC. The invalid memory address is stored in the control register `cr2` on Linux/IA-32. We implemented a kernel patch to retrieve this value and made it accessible from the user-space monitor.

In the simple case, the faulting instruction must be the one right before the address held by the PC register at the time of the memory access exception. Now let us discuss how to locate the faulting instruction in the complex case.

In normal program debugging, the call stack trace at the time of exception can be used to identify the last function called, and the faulting instruction f should be a part of this function. Unfortunately, we cannot trust the call stack information, because it might have been maliciously corrupted by the on-going attack. Instead, we use the following algorithm.

We first identify all indirect control flow instructions in the program, and place them in a candidate set C. This is done by scanning the code segment of the monitored process image. This set includes all indirect jumps, indirect function calls, and function returns. The faulting instruction f must be one member in the candidate set.

We then decode and compute the target address a of each member m in C using the process machine state at the time of the memory exception. If for an m, its target address a does not match the value of the current PC register, m is eliminated and removed from C. If there is only one instruction in C at the end of this process, it is the faulting instruction f. If there are still multiple instructions in the resulting C, we use programmable breakpoints to select the real faulting instruction.

Specifically, we set a programmable breakpoint at each of the candidate instructions in C, and re-execute the program (under the supervision of the monitor) with the logged messages replayed to it. The program will have the memory access violation as before due to the attack. During the re-execution, the monitor will receive a breakpoint exception whenever an instruction in the candidate set C is executed. The faulting instruction f is the instruction at the last breakpoint before the memory access violation.

4.2 Converting Case-IV Crashes

We now describe how to convert a Case-IV crash into one of the other three cases. While the purpose of the following algorithm is to eliminate and convert Case-IV crashes, we do not need to distinguish the four cases. In fact, when a crash is detected, we have no way to differentiate the cases. Therefore, our algorithm takes any crash as an input, eliminates the possibility of Case-IV via random re-execution, and converts it to the other cases.

Recall that in a Case-IV crash, a takeover instruction does not directly cause a crash, but rather a subsequent instruction does due to semantically inconsistent execution. Case-IV exists because the takeover instruction uses a corrupted but still valid address. Our method converts Case-IV crashes into other cases by making the corrupted address invalid for the takeover instruction. We use *re-execution with non-overlapping memory layout* for this purpose. Intuitively, when a process crashes, the monitor re-executes the program with message replay. But this time, it creates the address space of the process in such a way that its memory regions do not overlap with those used in the crashed process. While the new process has changed with a completely different memory layout, the address values sent by the attacker in the logged message remain the same. As a result, the re-execution makes a previously valid address invalid. Should the re-execution reach the takeover instruction, it would have an immediate memory access exception, resulting in a Case-III crash. It is also possible that the re-execution would crash before reaching the original takeover instruction. In either case, a Case-IV crash is converted to one of the other three cases.

One possible concern is: Can an attacker predict the relative addresses used in offset-based instructions (e.g., offset-based `jmp`)? Modern compilers generate offset-based instructions directly and place the offset values in the read-only code segment. Thus, the attack will not be able to corrupt these offset values. An attacker may also use offset-based instructions in the attack-injected code. Fortunately, the attack has to rely on a control flow transfer instruction (i.e., the takeover instruction) based on absolute addresses to execute injected code. Thus, the process will crash at the takeover instruction before the attack has a chance to transfer the control flow to the injected code, if the earlier instructions do not cause the crash.

It is certainly possible that during the re-execution with non-overlapping memory layout, an invalid address in the original process becomes a valid one in the re-execution. This implies that a non-Case-IV crash may be converted into a Case-IV crash. To deal with such situations, we re-execute the program for a second time using a memory mapping that does not overlap with the original execution and the first re-execution. Given the non-overlapping memory layouts of the executions, at least two of the three instances will be non-Case-IV crashes. These two instances must crash at the same faulting instruction, because any faulty address can be valid in at most one memory mapping. Note that the

diagnosis engine can easily determine the correspondence of instructions in different crashes by reversing the memory re-mapping (using the information about how the re-mapping is performed). Thus, the diagnosis engine can determine the non-Case-IV crashes through a majority voting, and choose any one of them for the subsequent analysis.

To support this approach, we modified the Linux kernel process execution and memory management code by adding a special system call `execve_rand()`. Besides the normal arguments in the original `execve()`, the system call takes an additional argument – a list of memory regions that should not be used during the lifetime of the process. Before each re-execution, the monitor collects the memory layouts from the previous executions and pass the list to the kernel. All memory regions on the Linux operating system are created via the `mmap()` system call. The modified kernel consults the list from `execve_rand()` whenever `mmap` is called. The modified kernel routine makes sure that such a request does not violate the non-overlapping requirement. This is done by relocating the `mmap` request to random, un-allocated region.

It seems that this approach is only applicable to programs that uses no more than 1/3 of the available address space, which is between 1 and 2 GB on a 32-bit system. Most network service applications, however, have small memory fingerprints (much less than the above limit). Thus, we consider this approach practical.

After choosing a non-Case-IV crash, we can determine partially which case it belongs to by examining the faulting instruction. If the faulting instruction f is a control flow transfer instruction (e.g., `ret`), this must be a Case-III crash. Otherwise, it must be a Case-I or Case-II crash. In a Case-I crash, the faulting instruction must have used an invalid address computed from the malicious input. However, there may be two sub-cases in a Case-II crash: (a) The faulting instruction causes the crash because it accesses an invalid address computed from the *malicious network input*, and (b) the faulting instruction uses *corrupted data* as an address. For the sake of presentation, we call them as Case-II(a) and Case-II(b), respectively.

In a Case-I or Case-II(a) crash, we can already identify how the faulting instruction causes the crash with the malicious input. In a Case-II(b) or Case-III crash, we need to trace back to the corrupting instruction that writes the corrupted data. For convenience, we call an instruction that uses corrupted data as an address the *victim instruction*, denoted as v. The takeover instruction in a Case-III crash and the faulting instruction in a Case-II(b) crash are both victim instructions.

4.3 Tracing the Corrupting Instructions

Once a victim instruction v is identified, we can obtain $TargetAddr(v)$, the address that instruction v is trying to access. Note that the victim instruction cannot be a direct addressing (e.g., `mov 0x12345678, eax`) or a constant based indirect addressing instruction (e.g., `mov eax, [0x4321]`), because the addresses involved in such instructions are hard coded in the read-only code segment and cannot be modified by the attack.

We discuss how to find the address l where $TargetAddr(v)$ is stored in the program's address space using the addressing modes of the IA-32 architecture [15]. In the first situation, the location where $TargetAddr(v)$ is stored is predictable and can be determined. For example, when the victim instruction is `ret`, it is easy to see that $TargetAddr(v)$ is stored at the top of the stack. In the second situation, the victim instruction may use two-level indirect addressing mode, and thus the address l where $TargetAddr(v)$ is stored can also be easily determined. For example, when the victim instruction is `jmp [ebx+esi]` (i.e., jump to the address retrieved from the memory location at $x =$ `ebx+esi`), we can easily infer that the address l must be x. However, in the third situation, the victim instruction may use one-level indirect addressing, and the derivation of l is more complicated. For example, the victim instruction could be `mov [ebx+esi], ebp` (i.e., move the value at address $x =$ `ebx+esi` to register `ebp`), where $TargetAddr(v)$ is computed by adding the values in registers `ebx` and `esi`. We currently rely on binary data dependency analysis to handle this case. In our experiments, such cases can be easily handled. We speculate that in many cases, binary analysis technique can effectively solve this problem. For example, consider the following two instructions `mov [ebx+esi], ebp; mov eax, [ebp]`, where the latter instruction is the victim instruction. Through binary data dependency analysis, we can easily determine that the address l must be the value of `ebx+esi`. However, the general situations where $TargetAddr(v)$ may be computed from the corrupted data require additional research. In this paper, we show we can at least automatically identify the unknown vulnerabilities in the first two situations and many cases in the third one when we can identify where $TargetAddr(v)$ is stored, but leave the general solution to the third situation as future work.

The next issue is to find out which instruction(s) writes $TargetAddr(v)$ to address l. The writing instruction is potentially an initial corrupting instruction. We use *hardware watchpoint*, which is available in most modern processors, to find out the corrupting instruction c. Specifically, the monitor sets the hardware watchpoint register to address l and re-executes the program with the logged messages. The hardware support in the processor guarantees that an exception is raised whenever data is written to address l. To exploit this hardware watchpoint feature, we provide an exception handler (in the monitor) to intercept such exceptions. The exception handler inspects if the value being written is $TargetAddr(v)$. If yes, the exception handler records the address of the writing instruction. This re-execution, as the previous executions, will certainly crash. The last writing instruction that causes the watchpoint exception before the crash is the corrupting instruction we are looking for.

It is possible that after a data item is corrupted, it is copied multiple times before finally being written to address l. We iteratively find all the instructions that perform the value propagation and retrieve their respective stack traces. These provide a history of the propagation of the corrupted data. In practice, the number of times the corrupted data is propagated should be very small, which is confirmed in our evaluation using real world applications.

4.4 Discussions

It is certainly possible that a crash is due to accidental rather than malicious memory fault. While the algorithm in the previous section is discussed in the context of malicious memory attacks, it could be applied to crashes due to both types of memory faults. Regardless the nature of the fault, it is triggered by some external messages. Our algorithm

does not need to distinguish between these two types, and should be able to perform the same type of diagnosis and locate the source of corruption. This, combined with the response method to be discussed in Section 5, can potentially prevent both types of crashes. It would be interesting future research to evaluate the effectiveness of our system for both accidental and malicious memory faults.

Our current approach for tracing memory corruption vulnerabilities has a limitation. As discussed earlier, it cannot guarantee to trace back to the initial corrupting instruction if the data corrupted by this instruction is transformed in arbitrary ways before being used as a faulty address. Despite this limitation, our approach has successfully diagnosed the memory corruption vulnerabilities under attacks in our experimental evaluation. This shows our approach is useful in practice. In our future work, we will investigate other approaches that have the potential to provide a more general solution. One possibility is to trace the attacks through instructional-level dynamic data dependency analysis, which can provide detailed information on how the corrupted data is propagated before the faulting instruction is executed.

5. AUTOMATIC RESPONSE TO ATTACKS

One goal of vulnerability diagnosis is to stop future attacks that exploit the same vulnerability. We use the result from the vulnerability diagnosis to generate attack signatures, and drop messages that match the attack signatures at the message filter. In order to reduce false positives (without increasing false negatives), we use automatically retrieved program state in combination with the signatures.

The key to responding to (unknown) attacks is to distinguish between normal and malicious attack messages. This falls into the general scope of anomaly detection, and has been shown to be a very difficult (if not impossible) problem. The most widely used approach for attack detection is the use of *attack signatures*: byte sequences extracted from existing attacks against a vulnerability. Previously, attack signatures are generated manually after attack messages are collected in network sensors. There have been recent advances in automatic worm signature generation (e.g., EarlyBird [35], Autograph [17], and Polygraph [27]). These methods usually rely on simple thresholds (e.g., number of failed connections) or external ways to classify suspicious flows, and use message contents from the classified flows to extract worm signatures. These systems work best when there is an outburst of worm traffic for accurate signature extraction. However, they are not sufficient for non-worm and stealthy worm attacks. TaintCheck [28] uses an instruction emulator to capture attack signatures at the cost of high performance overhead (between 5-35 times). We observe that program randomization causes an attacked program to crash, which can be used as a definitive sign of intrusion. In this section, we will harness this observation and the results from the vulnerability diagnosis to identify and block attacks. This can be achieved with low overhead due to the non-intrusiveness of randomization to program execution.

5.1 Basic Message Signature

The basic response scheme is similar to the previous signature generation schemes in that we use critical byte sequences from the attack. Specifically, we use the result from the bug diagnosis algorithm from the previous section. Recall that the diagnosis method can identify the corrupting instruction c that is tricked to overwrite critical program data. As a result, we can obtain the (invalid) address y that corrupting instruction c tries to write and the value x that c is writing. The byte sequences x and/or y can potentially serve as a message signature. We then search through the logged messages to confirm the validity of these values. Depending on the attack, both or one of them are embedded in the malicious messages. With a single set of attack messages, we can identify the critical byte sequences in the messages and use them as the signature to block the attacks. Whenever a message containing the byte sequences x and/or y is received, the message filter drops it and also resets the corresponding connection. Thus, this approach allows the application to graciously handle attacks as dropped connections without having its internal state corrupted.

Attack messages can be altered slightly while still being successful. For example, in a stack smashing attack, as long as the corrupted return address falls within the range of the buffer, the attack can still be successful. Therefore, using exact values of x and/or y as a signature can be evaded by such attack variants. To detect such variants, we adopt the method used in [28], that is, matching only the first three bytes in x and/or y in the message.

5.2 Correlating Message Signature with Program Execution State

A message signature generated from the basic scheme consists of short byte sequences. While previous work has shown that such signatures can result in very low false positives (typically under 1%), for servers with high volume of traffic, however, even such a low false positive rate may not be acceptable. To more accurately detect attacks and further reduce false alarm rate, we propose to correlate the message that a program receives with its internal program execution state.

We observe that a network server application can interact with the external clients in many different program states in order to perform the underlying service protocol. An attack that aims to exploit a specific security vulnerability has to send its attack messages at a specific server execution state that exposes the vulnerability. If we can find out to which program execution state the attack message must be sent, we only need to apply the message signature to network messages received in that state. For all network messages received in other program execution states, no message filtering is necessary. This not only can reduce false positive rate without decreasing the detection rate, but also can speed up the message filtering process. For example, if a buffer overflow vulnerability is exposed to a remote attack in *State X*, we only need to inspect messages received in this vulnerable state.

There is a large amount of state information that is available inside non-trivial server applications. Assuming that we understand the semantics of the application, we could use application specific states such as the values of certain critical data variables dynamically retrieved from the process address space as an indication of application state. We could also modify the application to explicitly expose such execution state to the external message filtering mechanisms. These application-dependent approaches can be the most accurate indication of program state. However, such approaches will require manual intervention and are not appropriate to be used as automatic response to attacks.

Our goal is to automatically and transparently retrieve program execution state that can be used in combination with the generated message signatures. Toward this end, we propose to use the application's call stack trace as an indication of the server protocol state. An application's call stack trace should be different at different state of the server protocol execution. Thus, the call stack information can be combined with the message signature to reduce false positive rate (without decreasing the detection rate).

During the bug diagnosis phase, when a message that matches the signature is received, we record its current call stack trace. We abstract out most of the information except the the chain of function return addresses and the current program counter as: $(pc, ret_n, \ldots, ret_1)$, where pc is the program's current program counter, and $ret_n, \ldots, ret_1$ is the chain of return addresses. Note that pc and the function return addresses $ret_n, \ldots, ret_1$ should be transformed into a canonical form so that they are comparable in different randomized versions of the program. During detection phase, we raise an alarm and drop a connection only when both of the following conditions are met: (1) when the signature is matched in the message; and (2) the program's current execution state matches the recorded state from the analysis. Depending on the nature of the application protocols and the size of the call stack, the order of message signature match and program call stack match can be changed in order to speed up processing. For applications that receive a large amount of data, first inspecting call stack is better, since we do not need to inspect each and every byte of the message. For smaller messages, performing message signature matching first should be better.

We illustrate the correlated message filtering scheme using the OpenSSH server as an example. Figure 4 shows the message receiving states and their corresponding program call stack traces for the OpenSSH server while running the SSH Protocol Version 1 using password authentication. Message sending states of the server is not shown here since we are mainly concerned with how to inspect messages received from a client. The execution states are shown in ovals and named according to what type of information the server is expecting from a remote client. For example, the state *session key* is where the server is waiting for the protocol session key from the client. The call stack trace in different receiving states are also shown in the figure. As we can see, each receiving state has a unique call stack trace. OpenSSH server has a vulnerability that can be exploited in the *passwd* receiving state. In this case, the generated signature is the three-byte sequence extracted from the message and the call stack trace *(read, do_authloop, do_authentication, main)*. An incoming message is dropped only when both match conditions are satisfied. The method incorporates semantics because it not only uses byte sequences but also the program execution state. In Section 7, we show that the combined signature eliminates all false positives in our experimental evaluation.

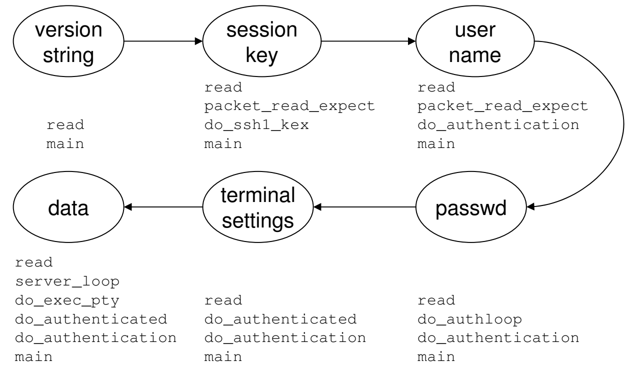

Figure 4: Message receiving states and respective call stacks for OpenSSH server

6. IMPLEMENTATION

We have implemented a prototype system with the components shown in Figure 2. The monitor, the diagnosis engine and the signature generator are implemented as an integrated user-space program. It uses the Linux *ptrace* system facility for controlling application execution. Once our monitor program launches the target application, it suspends itself and waits for any abnormal signals sent to the target application on the `waitpid()` system call. Therefore, our monitor program does not incur any performance overhead during normal application execution. Only when the target application has abnormal behavior such as memory access violation, does the monitor wakes up and runs the diagnosis algorithm. The *ptrace* system interface allows us to inspect the process machine and memory state and perform the analysis required by the algorithm.

The signature matching algorithm is implemented using user space system call tracing and interception. We modified the Unix system utility *strace* to perform message inspection and call stack walk. When a message matches the signature, we insert an error return value to the system calls (e.g., *read* and *recv*). The server process detects such a return code as network communication error and drops the connection before the malicious data reach its address space. Due to the user-space implementation, we are not able to accurately assess the performance of the message filtering algorithms. There are frequent context switches between our filter and the target server process, and also frequent user/kernel mode switches to read messages received in the target process. The overhead of these switches dominates the time for our experiments. We plan to move the message filtering algorithm to the kernel space in the future and evaluate the performance implications of associating message signatures with call stack traces.

In two published results that use kernel level interceptions, the overhead is around 10% even for extreme cases. In Flashback [37], memory checkpointing and system call logging together only incur less than 10% overhead in the worst case for Apache web server. Similarly, in Shield [40], the more complex state-machine based message filter only incurs at most 11% throughput degradation itself. We believe that a kernel space implementation of our algorithms would not introduce more overhead since our logic is much simpler.

Currently, due to the user-space implementation of the message filter, the monitor program and the filter run separately. Ideally, these two components should run simultaneously while the target application is running. Once the filter is implemented in the kernel space, the integration of the two can be done easily.

7. EXPERIMENTAL EVALUATION

We discuss our experience using our prototype system and present experimental evaluation results in this section. We first illustrate the benefits of the automatic bug diagnosis algorithm using an example, and then present the evaluation results for the automatic bug diagnosis algorithm and the response method.

7.1 Automated vs. Manual Diagnosis

The goal of bug diagnosis is to identify the security vulnerability that is exploited by an on-going attack, which causes the application to crash due to address space randomization. The proposed diagnosis method can identify the attack at the time of corruption, which is better than methods that identify the attack at the time of the use of corrupted data. One of the major benefits of our method is that, through automation, it improves the efficiency of problem diagnosis, which is usually performed manually. While it is difficult to quantify, we use an example to illustrate the degree of improvement. We compare our diagnosis methods to manual diagnosis using a debugger.

We use a simple stack buffer overflow attack example in an open source web server *ghttpd-1.4*. Figure 5 shows the source code that has a stack buffer overflow vulnerability. The unchecked stack buffer `temp[]` is in `Log()`, which performs server logging functionalities. Whenever a URL request is sent to the server, the request handler `serveconnection()` calls `Log()` to log the requested URL. Within `Log()`, the message is formatted and written to `temp` using the C library function `vsprintf()` which does not perform bounds checking. An attacker exploits this bug by sending an extremely long URL and uses classic stack smashing technique to take control. Address space randomization will cause a working exploit to crash the server process.

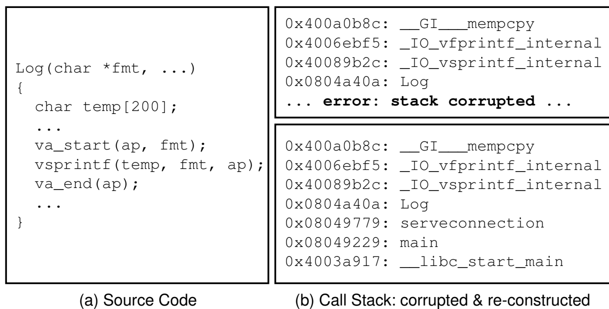

Figure 5: Vulnerability diagnosis for ghttpd

In a manual debug session, a programmer would have few clues regarding the cause of the crash. We tried running the server using the symbolic debugger *gdb*. When the crash occurs, *gdb* was only able to print the current program counter register `$eip`. The value of `$eip` is largely useless because it contains a corrupted value passed on from the malicious attack message. It is very difficult to infer the address and type of the previous instruction that actually causes the crash. Normally, a programmer can rely on the call stack trace at the time of crash for further diagnosis. Unfortunately, in this case, the debugger cannot print the stack trace because it has already been corrupted due to the attack. Therefore, using the debugger alone, the diagnosis process would have taken a substantial amount of time even for experienced programmers with knowledge of the source code, especially for large, complex applications.

Our technique improves the efficiency of diagnosis. In the same example above, at the time of program crash, our analyzer analyzed the memory image of the crashed process, determined that the crash was due to an indirect control flow instruction because the register values `cr2` and `eip` match (i.e., the faulting address and the current PC value match). The analyzer then forms the candidate set C of all indirect control flow instructions in the program, and eliminates a significant portion of the candidates through address decoding using the crash image. The analyzer then re-executed the program to locate the `ret` instruction within `Log()` as the direct cause of the crash. We then used hardware watchpoints to watch the memory locations where the return address were stored. Our analyzer caught any instruction that wrote to that location using invalid address value. The last instruction that wrote to the location was the corrupting instruction. Our tool recorded the call stack trace at the time of writing (the top trace shown in Figure 5(b)). The top stack trace in the figure shows that the corrupting instruction is in the C library function `__GI__mempcpy()` with a chain of C library internal function calls originated from the server function `Log()`. Using the instruction address `0x0804a40a` shown, we mapped it to the source code line where the vulnerable function `vsprintf()` is called (`_IO_vsprintf_internal` is the library symbol for `vsprintf`). This entire process was automated and done without application specific knowledge, and certainly improved the debugging and diagnosis process significantly.

The top stack trace in Figure 5(b), however, is not complete: the callers beyond `Log()` cannot be recovered. This is because the attack has corrupted a substantial portion of the call stack beyond the stack frame of `Log()` (stack grows downward). A critical component among those corrupted is the saved frame pointer which points to the stack frame of the caller of `Log()`. While the partial stack trace is in itself invaluable, a complete call stack trace would be preferred. The complete stack trace is re-constructed by combining the call stack trace at the entrance of the library function `__GI__mempcpy()` and the partial stack trace as shown in bottom on of Figure 5(b) at the time of corruption (and hardware watchpoint exception). The complete stack trace shows the circumstance under which corruption occurs. In addition, the reconstructed stack trace also shows the function call arguments and their values (not shown in the figure), which can facilitate further diagnosis.

7.2 Effectiveness of Diagnosis

We use a number of real world network service applications to evaluate the effectiveness of the method. For a given application P, we first search for the publicly reported memory corruption vulnerabilities. We then find exploits against the identified vulnerabilities. This second step is very time-consuming but can usually be accomplished by using existing exploit programs on the Internet.

The effectiveness evaluation is focused on the following question: How precisely can our method automatically identify the function where the original memory corruption vulnerability is located? In our experiments, we first recover the call stack trace at the time of memory corruption using the diagnosis algorithm, and then compare it to the call stack

trace retrieved by manual code inspection and debugging. A number of real world network applications with different types of memory corruption vulnerabilities are used in our evaluation. For the tested applications, we were able to correctly identify all the vulnerable functions at the time of corruption. The applications tested in our experiments are in Table 1.

Table 1: Tested server applications

Program	Description	Vuln./Attack Type
ghttpd	web server	buffer overflow
rpc.statd	NFS stat server	format string
OpenSSH	secure shell server	integer overflow
icecast	media streaming svr	buffer overflow
Samba	file and print service	buffer overflow

One application examined in our experiments is worth further discussion. The vulnerability in the OpenSSH server is a signed integer overflow. In the server function `detect_attack()`, a 32-bit integer value `l` is assigned to a 16-bit variable `n`. The cast from 32- to 16-bit makes `n=0` under attack. The value `n-1` (which is `0xffff`) is subsequently used to mask variable `i` (retrieved from the attack message) that is used to index an array `h[]`. Because `n-1` does not have any effect on the masking operation, attackers can force `h[i]` to be arbitrary address values. The crash in the randomized system occurs when `h[i]` is accessed. Our analyzer can correctly locate the vulnerability to the function `detect_attack()`. However, currently, we cannot automatically identify this as an *integer overflow* vulnerability due to lack of type information. More accurate diagnosis requires additional investigation.

7.3 Evaluation of Automatic Response

Once signatures are generated, the filtering of attack traffic is pretty straightforward. Thus, our evaluation in this part is focused on the quality of the generated signatures, especially the false positive rate. In particular, we would like to see how much improvement we can get using correlation of message signature with program execution state. We used two server applications with which we were able to run large scale experiments for evaluation: *OpenSSH* server and the *ghttpd* web server.

In the first case, we run a secure shell session using the OpenSSH server to transfer 21 gigabytes of data. The data includes all the files in Fedora Core 3 Linux and the home directory in our lab. The transferred files include text, source code, application and library binaries, as well as many image, video and audio files. We used the signature we generated for the vulnerable OpenSSH server in the message filter and evaluated the alerts reported by the filter. Without using the call stack information, our message filter reported 1245 signature matches for the 21 gigabyte of encrypted data transfer. Since the data transferred are not attack messages, these alerts are all false positives. Using the program call stack information together with the message signature, all false alerts were eliminated. This is because the alerts are all reported in the *data* receiving state as shown in Figure 4, while the recorded attack messages are against the *passwd* receiving state. This shows that the call stack trace is an effective way to discriminating different execution states for OpenSSH sever.

We have also evaluated the idea using the *ghttpd* web server. The traffic is generated using Spec-Web99 benchmark using the static get suite. We were not able to detect any match for the generated signature. A possible reason is that the incoming traffic to the web server are mostly plain text URLs, which do not match the binary attack signature.

The false positive results are listed in Table 2. The *Pure message signature* line uses only pattern sequence as signature. The *Correlated message filtering* line is the proposed method that associates pattern with call stack trace.

Table 2: Number of false positives

Signature Type	OpenSSH	ghttpd
Pure message signatures	1245	0
Correlated message filtering	0	0

The experience running the web server did lead to a caveat in using call stack trace as program semantic state to improve pattern-based signature matching. Applications such as OpenSSH that run complex protocols can benefit greatly from the use of call stack trace. This is because complex protocol implementations are typically structured in a way that models its internal execution state transitions. Therefore, using call stack can clearly discriminate between different server receiving states. The benefit of using call stack is less clear for simple protocols such as HTTP that has few receiving states. Our analysis of the *ghttpd* web server shows that when receiving client messages, the server stays in the main loop. From our monitor's perspective, its call stack does not change. Therefore, the benefit of using call stack trace is application dependent. The use of call stack trace does not affect the correctness of the signature filtering mechanism even when it does not help reduce false positives. Therefore, the additional benefits it offers to applications such as OpenSSH still makes it a useful technique. Although modeling execution state using call stack is application and protocol dependent, the general idea of associating program data/execution state with message signatures can still be potentially beneficial to many applications. Further research is required to identify more useful program semantic states.

8. RELATED WORK

Intrusion detection systems (IDSs) (e.g., Snort [31, 4], Bro [30], and NetSTAT [38]) have been used to detect attacks. Most current IDSs that are being used are based on signatures of known attacks, though some other approaches have been proposed to detect attacks using statistical techniques (e.g., NIDES [16]). These signatures are traditionally generated manually, and the generation of signatures is usually slow and error prone. To deal with the delay in signature generation, and particularly to deal with the threat of fast spreading worms, several approaches [23, 18, 34, 27] have been developed to generate (worm) signatures from attack traffic, including polymorphic worms [27]. However, all these approaches rely on other means to identify the attack traffic. The approach proposed in this paper starts to generate the signature for an unknown attack upon detection by our system. Moreover, the proposed technique offers an opportunity to detect, recover, and defend similar attacks in a purely decentralized manner.

TaintCheck [28] uses binary program emulation to track the propagation and use of *tainted* external data. It prevents the use of tainted data for jump addresses and format strings. We believe that within our system architecture, the technique used by TaintCheck, i.e., dynamic information flow tracking, can be a part of the crash diagnosis engine. We will study how to best leverage such techniques in our future research. Adopting a reactive approach, STEM [33] identifies potentially vulnerable code segments based on program crashes or instrumented applications running on honeypots, and then selectively emulate the vulnerable code segments by instrumenting the source code. The emulation allows a vulnerable program to recover from attacks by rolling back the changes made by an attack. Our approach can use similar attack detectors as in STEM, but differ in that it uses message filter instead of error virtualization for blocking future attacks. Despite the similar purpose of detecting and recovering from unknown memory corruption attacks, our approach also differs from these techniques. In particular, our approach automatically identifies the vulnerability (at the binary level) exploited by novel attacks, and prevent future attacks by signature-based detection and filtering. In addition, our approach introduces insignificant performance overhead during normal program execution because it does not use emulation or code instrumentation. ARBOR [25] is a system that can generate signatures for buffer overflow attacks. ARBOR uses statistical analysis of message sizes to find buffer lengths, while we use program analysis to find signatures for a larger class of attacks. In a follow-up work [24], the authors of ARBOR used a similar approach to this paper to locate corrupting instructions due to buffer overrun, localize attack message, and then generate signature based on message length and contents. Their method requires application level protocol specification for better signature quality.

Our approach is similar to FLIPS [26] which uses anomaly classifier, signature filter, and instruction set randomization (ISR). While ISR only detects code injection attacks, we use address space randomization to detect a broader range of memory corruption attacks. In [1], *shadow honeypots* are used to verify and therefore reduce false alerts from anomaly detectors. The approach requires an additional *shadow process* in addition to the regular server process. The shadowed process runs more slowly than the original server process due to code instrumentation. Our approach only uses one randomized server process with very little performance overhead during normal operations.

Shield [39] is a vulnerability based protection system, which uses manually generated signatures to filtering out attacks exploiting known vulnerabilities. Our approach, which targets at unknown memory corruption vulnerabilities, is complementary to Shield. Indeed, our approach can be combined with Shield to provide automatically generated signatures, and protect against attacks exploiting both known and unknown vulnerabilities.

Address space randomization has been proposed as a generic defense mechanism against memory corruption attacks (e.g., [3, 29, 42]). Our system uses address space randomization to detect memory corruption attacks and trigger the diagnosis and response process. Recent work [32] has shown that it is possible to brute-force attack randomized systems. Clearly, moving to 64-bit platform or using more fine-grain randomization will minimize the chance of such attacks. On the other hand, the goal of our system is precisely to defeat such attacks. Upon the first brute-force attack, we analyze the crash and generate signatures to block future attacks. With better problem diagnosis and signature generation schemes, we will eventually be able to defeat such attacks. We will study these approaches in our future research.

A number of *ad hoc* techniques and tools have been developed to protect known vulnerabilities. Examples include detection and prevention techniques against stack buffer overflow attacks [11, 9, 13], and format string attacks [10]. Since these techniques are targeted at specific types of bugs, and they can certainly be used in our system and serve as specialized detectors. However, we believe that randomization is a more generic and broader detection methods that can cover these aforementioned techniques. Several virtual-machine based tools (e.g., [12, 19, 20]) have been developed to record and analyze OS-level events and their dependencies to facilitate intrusion analysis. The approach presented here is complementary to these techniques.

9. CONCLUSION AND FUTURE WORK

In this paper, we developed a novel approach to automatically diagnose unknown memory corruption vulnerabilities on the basis of program randomization. This approach allows a network service program to quickly identify the vulnerability being exploited. We also generate message signatures that can be used to prevent future attacks. Furthermore, we associate program's execution state, in particular its call stack trace, with the attack signatures learned from the attack messages. The association can be used to significantly reduce the false positive rate without decreasing the detection rate for certain applications. The proposed techniques enable the development of a fully decentralized self-diagnosing and self-protecting defense mechanism for networked computer systems.

A number of issues require future investigations. A more general dynamic data dependency analysis method is required for more efficient diagnosis. The message filtering mechanism should be moved to the kernel space for better performance and better integration with the diagnosis engine. And finally, more experiments are required to evaluate our system.

10. ACKNOWLEDGMENTS

We would like to thank our shepherd, Somesh Jha, for his excellent comments and timely responses. We also thank Dawn Song and the anonymous reviewers for their many suggestions for improving this paper. This work is partially supported by the National Science Foundation (NSF) under grants ITR-0219315 and CCR-0207297.

11. REFERENCES

[1] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[2] A. Baraloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. `http://www.research.avayalabs.com/project/libsafe/`. White Paper.

[3] S. Bhatkar, D.C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium (Security '03)*, August 2003.

[4] Brian Caswell and Marty Roesch. Snort: The open source network intrusion detection system. http://www.snort.org.

[5] CERT. http://www.cert.org/advisories/CA-2001-19.html.

[6] CERT. http://www.cert.org/advisories/CA-2003-04.html.

[7] CERT. http://www.cert.org/advisories/CA-2003-20.html.

[8] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*, November 2002.

[9] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 01)*, April 2001.

[10] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, August 2001.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, June 1998.

[12] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

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

[14] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizingsensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[15] Intel Corporation. *Intel Architecture Software Developer's Manual, volume 2, Instruction Set Reference*, 1999.

[16] H.S. Javits and A. Valdes. The NIDES statistical component: Description and justification. Technical report, SRI International, Computer Science Laboratory, 1993.

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

[18] H.A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[19] S.T. King and P.M. Chen. Backtracking intrusions. In *Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP)*, October 2003.

[20] S.T. King, Z.M. Mao, D.G. Lucchetti, and P.M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[21] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[22] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997.

[23] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[24] Zhenkai Liang and R. Sekar. Automated, sub-second attack signature generation: A basis for building self-protecting servers. In *To appear in 12th ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[25] Zhenkai Liang, R. Sekar, and Daniel C. DuVarney. Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self-healing systems. In *USENIX Annual Technical Conference*, April 2005.

[26] M. Locasto, K. Wang, A. Keromytis, and S. Stolfo. Flips: Hybrid adaptive intrusion prevention. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

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

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

[29] PaX Team. http://pax.grsecurity.net/docs/aslr.txt.

[30] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[31] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA conference*, 1999.

[32] Hovav Shacham, Matthew Page, Ben Pfaff, EuJin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address Space Randomization. In *Proceedings of 11th ACM Conference on Computer and Communications Security (CCS)*, October 2004.

[33] S. Sidiroglou, M.E. Locasto, S.W. Boyd, and A.D. Keromytis. Building a reactive immune system for software services. In *Proceedings of USENIX Annual Technical Conference*, pages 149 – 161, April 2005.

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

[35] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

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

[37] Sudarshan Srinivasan, Srikanth Kandula, Christopher Andrews, and Yuanyuan Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *The proceedings of the annual Usenix technical conference (USENIX'04)*, June 2004.

[38] G. Vigna and R. A. Kermmerer. NetSTAT: A network-based intrusion detection approach. In *Proceedings of the 14th Annual Security Applications Conference*, December 1998.

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

[40] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of ACM SIGCOMM*, August 2004.

[41] Jun Xu, Shuo Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. An Experimental Study of Security Vulnerabilities Caused by Errors. In *Proceedings of IEEE International Conference on Dependable Systems and Networks*, Goteborg, Sweden, July 2001.

[42] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *Proceedings of 22nd Symposium on Reliable and Distributed System (SRDS)*, October 2003.