Volume 3, Issue 6, June 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

A Signature-Free Buffer Overflow Attack Detection Using DST

Kotha Jothsna[*]
DRK Institute of Science and Technology
Department of CSE, Hyderabad , India

Dr.R.V.Krishniah
DRK Institute of Science and Technology
Department of CSE, Hyderabad, India

Abstract— Now days internet threat takes a blended attack form, targeting individual users to gain control over networks and data. Buffer Overflow which is one of the most occurring security vulnerabilities in Internet services such as such as web service, cloud service etc. This motivated by the observation those buffer overflow attacks typically contain executable code whereas legitimate user requests never contain executables in most internet services. Unlike the previous detection algorithms, a new SigFree uses a Distributed Spaning Tree (DST) technique, fast, and hard for exploit code to evade. It blocks attacks by detecting the presence of code, it is a signature free, thus it can block new and unknown buffer overflow attacks; this is also immunized from most attack-side code obfuscation. To do so, we pay particular attention to the formulation of an appropriate fitness function and partnering instruction set. Moreover, by making use of the intron behaviour inherent in the DST paradigm, we are able to explicitly obfuscate the true intent of the code. All the resulting attacks Defeat the widely used in Intrusion Detection System.

Keywords— Linear Genetic Programming, code injection, Intrusion Detection Systems.

I. INTRODUCTION

The history of internet security, buffer over-flow is one of the most serious vulnerabilities in computer systems. Buffer overflow vulnerability is a root cause for most of the cyber attacks such as server breaking-in, worms, zombies, and bonnets. A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and depending on what is stored there, the behaviour of the program itself might be affected. Although taking a broader viewpoint, buffer overflow attacks do not always carry binary code in the attacking requests (or packets); code-injection buffer overflow attacks such as stack smashing probably count for most of the buffers overflow attacks that have happen in the real world. Although lot of research[2-10] has been done to tackle buffer overflow attacks, existing defences are still quite limited in meeting four highly desired requirements: 1) Simplicity in Maintenance; 2) Transparency to existing (legacy) server OS, application software, and hardware; 3) Resiliency to obfuscation; 4) Economical Internet-wide deployment. As a result, although several secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to be successful on a daily basis.

To overcome the above limitations, Recently, X. Wang et al.[11] proposed a SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that “the nature of communication to and from network services is predominantly or exclusively data and not executable code”. Their experimental study shows that the dependency-degree-based SigFree could block all types of code-injection attack packets tested in our experiments with very few false positives. Moreover, SigFree causes very small extra latency to normal client requests when some requests contain exploit code. However, Sig Free cannot fully handle self-modifying code and cannot fully handle the branch-function-based obfuscation. Further, Sig Free does not detect attacks such as return-to-libc attacks that just corrupt control flow or data without injecting code.

In this paper, we propose a novel SigFree, an online buffer overflow attack blocker using *DST*[42] instead of *code abstraction* [11] to protect internet services. The idea of DST is motivated by an important observation by searching the network that is nature of communication to and from network services is predominantly or exclusively data and not executable code”. For detecting and blocking the code-injection buffer overflow attack messages targeting at various internet services such as web service and cloud services, Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate user requests never contain executables in most internet services.

The DS works as follows: SigFree is an application layer detector that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree uses a novel technique called DST. The aim of Distributed spanning tree is used to detect the malicious code in the network like p2p, grid,and cloud. In this context, a popular direct querying algorithm is called expanding ring, or TTL-based flooding. With this algorithm, resources are connected through a graph and the algorithm forwards queries through the graph to find a particular resource. With expanding ring algorithm, it is well known that search performances are affected by the communication graph topology. The structure of the Distributed Spanning Tree, described in the following, provides better performances than usual topologies as trees or random graphs.

II. Related Work

Recently, several researchers proposed different detection and prevention methods to detect and prevent the buffer overflow attacks. We have classified these into three categories: 1) Prevention/Detection techniques of Buffer Overflows;2) Worm detection and signature generation;3) Machine code analysis for security purposes.4)DST.

2.1. Prevention/ Detection of Buffer Overflows

The Existing prevention/detection techniques of buffer over-flows attacks can be roughly broken down into six classes: Class 1A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools [13], [14], [15] have been developed. The bug-finding techniques used in these tools, which is general belong to static analysis, include but are not limited to checking and bugs-as-deviant-behaviour. Class 1A techniques are designed to handle source code only, and they do not ensure completeness in bug finding. In contrast, SigFree handles machine code embedded in a request (message). The Class 1B: Compiler extensions. "If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler". Three such compilers are StackGuard [16], ProPolice 17], and Return Address Defender (RAD) [18]. DIRA [19] is another compiler that can detect control hijacking attacks, identify the malicious input, and repair the compromised program. Class 1B techniques require the availability of source code. In contrast, SigFree does not need to know any source code. Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Pax [20], LibSafe [21], and e-NeXsh [22]. Class 1C: Techniques need to modify the OS. In contrast, SigFree does not need any modification of the OS. Class 1D: Hardware modifications. A main idea of hard-ware modification is to store all return addresses on the processor [29]. In this way, no input can change any return address. Class 1E: Defence-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of Pax [21].Address-space randomization, in its general form [23], can detect exploitation of all memory errors. Instruction set randomization [2], [3] can detect all code-injection attacks, while SigFree cannot guarantee detecting all injected code. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. "Repeated attacks will require repeated and expensive application restarts, effectively rendering the service unavailable" [6]. Class 1F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows area code running symptom. If such unique symptom s can be precisely captured, all buffer overflows can be detected. Class 1B, Class 1C, and Class 1E techniques can capture some but not all—of the running symptoms of buffer overflows. For example, accessing non executable stack segments can be captured by OS modifications; the compiler modifications can detect return address rewriting; and process crash is a symptom capture d by defence-side obfuscation. To achieve 100 percent coverage in capturing buffer overflow symptoms, dynamic data flow/taint analysis/program shepherding techniques were proposed in Vigilante [5], Taint Check [4], and [24]. They can detect buffer overflows during runtime. Covers [6] and [7]. Post crash symptom diagnosis extracts the "signature" after a buffer overflow attack is detected. A more recent system called ARBOR [25] can automatically generate vulnerability-oriented signatures by identifying characteristic features of attacks and using program context. Moreover, A RBOR automatically invokes the recovery actions. Class 1F techniques can block both the attack requests that injection code and the attack requests that do not contain any code, but they need the signatures to be firstly generated. Moreover, they are either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. In contrast, although SigFree could not block the attack requests that do not contain any code, SigFree is signature free and does not need any changes to real-world services.

2.2 Worm Detection and Signature Generation

Because buffer overflow is a key target of worms when they propagate from one host to another, SigFree is related to worm detection. Based on the nature of worm infection symptoms, worm detection techniques can be broken down into three classes: [26]. [Class 2B] techniques use such local traffic symptoms as content invariance, content prevalence, and address dispersion to generate worm signatures and/or block worms. Some examples of Class 2B techniques are Earlybird [8], Autograph [9], Polygraph [10], Hamsa [27], and Packet Vaccine [28]. [Class 2C] techniques use worm code running symptoms to detect worms. It is not surprising that Class 2C techniques are exactly Class 1F techniques. Some examples of Class 2C techniques are Shield [29], Vigilante [5], and COVERS [6]. [Class 2D] techniques use anomaly detection on packet payload to detect worms and generate signature. Wang and Stolfo [30], [31] first proposed Class 2D techniques called PAYL. PAYL is first trained with normal network flow traffic and then uses some byte-level statistical measures to detect exploit code.

Class 2A techniques are not relevant to SigFree. Class 2C techniques have already been discussed. Class 2D techniques could be evaded by statistically simulating normal traffic [32]. Class 2B techniques rely on signatures, while SigFree is signature free. Class 2B techniques focus on identifying the unique bytes that a worm packet must carry, while SigFree focuses on determining if a packet contains code or not. Exploiting the content invariance property, Class 2B techniques are typically not very resilient to obfuscation. In contrast, SigFree is immunized from most attack-side obfuscation methods.

2.3 Machine Code Analysis for Security Purposes

Although source code analysis has been extensively studied (see Class 1A), in many real-world scenarios, source code is not available and the ability to analyse binaries is desired. Machine code analysis has three main security purposes: (P1) malware detection, (P2) to analyse obfuscated binaries, and (P3) to identify and analyse the code contained in buffer overflow attack packets. Along purpose of P1, Chritodorescu and Jha [33] proposed static analysis techniques to detect malicious patterns in executables, and Chritodorescu et al. [34] exploited semantic heuristics to detect obfuscated malware. Along purpose P2, Lakhotia and Eric [35] used static analysis techniques to detect obfuscated calls in binaries, and Kruegel et al. [36] investigated disassembly of obfuscated binaries. SigFree differs from P1 and P2 techniques in design goals. The purpose of SigFree is to see if a message contains code or not, not to determine if a piece of code has malicious intent or not. Hence, SigFree is immunized from most attack-side obfuscation methods. Nevertheless, both the techniques in [37] and SigFree disassemble binary code, although their disassembly procedures are different. As will be seen, disassembly is not the kernel contribution of SigFree. Fnord [38], the pre-processor of Snort IDS, identifies exploit code by detecting NOP sled. Binary disassembly is also used to find the sequence of execution instructions as an evidence of an NOP sled [12]. However, some attacks such as worm CodeRed do not include NOP sled and, as mentioned in [12], mere binary disassembly is not adequate.

Very recently, Wang et al. [11] proposed a SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code". However, their method has following limitations: First, Sig Free cannot fully handle the branch-function-based obfuscation, causes control to be transferred to the corresponding location f(x). By replacing unconditional branches in a program with calls to the branch function, attackers can obscure the flow of control in the program. We note that there are no general solutions for handling branch function at the present state of the art. Second, Sig Free cannot fully handle self-modifying code. Self-modifying code is a piece of code that dynamically modifies itself at runtime and could make Sig Free mistakenly exclude all its instruction sequences. Third, the executable shell codes could be written in alphanumeric form. Such shell codes will be treated as printable ASCII data and thus bypass our analyser. Their Scheme can successfully detect alphanumeric shell codes; however, it will increase computational overhead. Therefore, it requires slight tradeoffs between tight security and system performance.

Fourth, Sig Free does not detect attacks such as return-to-libc attacks that just corrupt control flow or data without injecting code. However, these attacks can be handled by some simple methods.

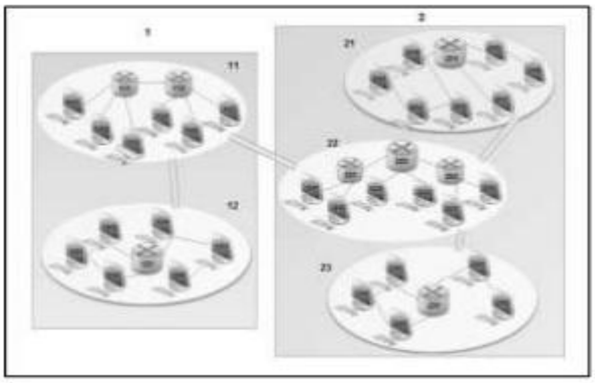
To avoid these problems, Josthna et al. [40] proposed a A Signature-Free Buffer Overflow Attack Blocker Using Genetic Programming. However, it is not efficient while detecting buffer overflow attacks in large networks.

After considering all these characteristics, we proposed a new Signature-Free Buffer Overflow Attack Detector Using Distributed Spanning Tree (DST)[41]

2.4 . The Distributed Spanning Tree Structure

The development and widespread usage of networks is mainly due to their efficient overlay routing and the location function, especially in global storage utilities and applications. We propose to apply an algorithm that defines a deterministic way to efficiently store and share the collected data between the nodes.

These nodes are able to form a distributed and decentralized network with a dynamic adaptation without affecting the whole functioning of the network. In order to realize the flexible and balanced collection of information among the nodes, we required an efficient approach that can scale to a large number of peers exchanging many control messages. The choice of a DST algorithm satisfies these requirements. These solutions present a relevant robustness since the global functioning of the network is totally independent of each application node. A DST can be described at three different levels as shown in Fig.1. The logical level is an abstract vision of the DST. At this level, tree nodes—that are groups of computers—are linked together by abstract links. Then comes to the interconnection level that implements internodes links with TCP/IP links. Finally, there is the topological level which describes how the TCP/IP links map on a real network.



III. BUFFER OVERFLOW ATTACKS

The core behaviour of an overflow attack lies in the simple observation that just because an address space of a variable declared in a program might be allocated of a specific size, this does not stop the same program from attempting to access memory outside of the allocated space. In order to make use of such a weakness, the attacker requires three components: (1) program used by the target system that possesses inherent overflow vulnerability; (2) Knowledge of the

size of memory reference necessary to cause the overflow; and (3) The correct placement of a suitable exploit to make use of the overflow when it occurs. The skill in crafting such an attack lies in how an exploit is hidden and ensuring that the memory referenced outside of the allocated space corresponds to the code defining the desired malicious behaviour.
There are two variants of buffer overflow attacks: Code- Injection (CI) attack, where attackers insert a piece of malicious code into the victim application's address space and then steer the application's control to the injected code; return to libc (RTL) attack, where attackers directly steer the control of the victim application to a function pre-existing in its address space, e.g., a library function. In both cases, attackers hijack the control of the vic-163 Tim application, by modifying a control-sensitive data structure such as a return address and changing it to either a location on the stack (CI attack) or a location in the text or code region (RTL attack). From the above analysis, a buffer overflow attack packet must include a 4-byte of hijack destination word that corresponds to a memory address on the stack or in the text region. Furthermore, to increase the success probability and robustness of a buffer overflow attack. The attackers almost always replicate the hijack destination word in the packet so as to accommodate differences in the address of the target control-sensitive data structure due to different combinations of compiler, loader, operating system, and command-line arguments.

IV. PROPOSED METHOD

The proposed work consists of prevention and detection of buffer overflows. This work proposes SigFree, a real-time, signature-free, out-of the-box, application layer blocker for preventing and detecting buffer overflow attacks, which is one of the most serious cyber security threats. The objective is to propose a global architecture that permits an efficient buffer flow attack Detection System where participants can exchange information following a network model, and thus providing services to a trace back application that strengthens the network security against attacks; or at least permits a fast and effective reaction to this kind of threats.
The solution proposed is designed to elaborate the defences against these large scale attacks by the correlation of the suspicious evidence provided and stored by the architecture entities from different geographical locations. Each participant gains a global view of the intrusion activity through this collaboration. To achieve this objective, we take into consideration some requirements in terms of performance and deployment. In fact, the processing and the bandwidth used, as well as the storage data must be minimized and conceptual security mechanism must be added to permit the access control for each entity in the architecture. Contain executables whereas legitimate user requests never contain executables in most Internet services, SigFree using DST blocks attacks by detecting the presence of code. Our SigFree method uses DST, in that propagation algorithm to send message every computer in the network and trace back algorithm querying the number of computers to detect the attacks.

4.1. Description of the architecture

Our principal contribution is the proposition of the architecture that serves as a model to implement a security system against buffer overflow attacks. This model targets detection systems with the possibility to place some applications at the very top level that can use the collected data of the detection system. The innovation in this case is the addition of a new level between the application and the detection system which is the use of the DST. This new level permits to indexing the information and to distribute it among the participant computers instead of implementing a central collection entity. The Fig. 2 shows that each level of the architecture. Here, we present it with an abstraction degree that permits us to put both independent and specific functions on each level. In fact, a node can contain one or more levels. We will detail this later by giving examples. The first level is the closest to the physical network. We called it the Network Level. In this level, equipment belongs to the underlay network. To be more specific, an entity in this level can for example be an IP router with the basic routing and addressing functionalities. The second level is the Security Level. In this level, we can classify our detection system entities. The implementation of all solutions in terms of detection system can be done here. Its functionalities are the ones of detection. When the analysed traffic in this level is detected as an attack, then, an alert is triggered and a primitive is sent to the upper level. This level will react to the alert accordingly. We can note that a detection system can be integrated to the equipment and the concerned entity will be represented by the above levels of the architecture. The third level of the model is the P2P or network Level which includes the DST proposed in our architecture to index and to distribute the information among the nodes.

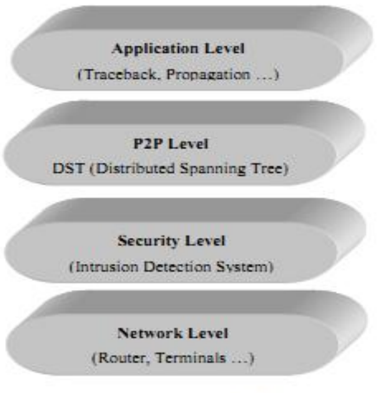

Application Level
(Traceback, Propagation …)
P2P Level
DST (Distributed Spanning Tree)
Security Level
(Intrusion Detection System)
Network Level
(Router, Terminals …)

Fig.2 Architecture Levels

This level receives the information collected and concerning the analysed traffic from the Security Level. When an alert is sent by the lower level, which means that an attack is detected, the internet level treats the received data and indexes the information on the specific DST node (identified by a node Id) depending on the objected calculated. We also see the possibility in this case, to integrate this level module to the equipment that already has the Network and the Security Levels. The last level is the Application Level. This last level is general in the description of our architecture. It can implement all possible management systems that use the indexed information on attacks by the DST level to react. We propose in our case to add a trace back application solution to integrate more complete defence architecture than only a detection system. This trace back module is a part of the future works since the three first levels of the architecture were implemented as we will see in the next sections. By removing any central analysis entity in the architecture, we propose a fully distributed solution. But in choosing this method, we must be sure that the collected data are correlated to ensure a better detection of buffer overflow attacks and also a reaction to them.

4.2. Propagation Algorithm

Based on above architecture, we have proposed two algorithms to detect the buffer overflow attacks. The aim of the propagation algorithm[41] is to efficiently send messages of to every computer that is part of a DST about the buffer overflow attack. This algorithm is the simplest one and is very similar to the classical tree parallel traversal. The root node initiates the traversal by sending a message to all its children. Then, recursively, when a non-leaf node receives a message, it forwards it to its children. Note that, here, we organized network as a tree.

Algorithm 1: Propagation algorithm [41]

```
Procedure $Propagate(msg)$
$Propagate\_Sub(h, msg)$
end procedure
procedure $Propagate\_Sub(s, msg)$
if $s = 0$ then          //The message is sent to a leaf
process $msg$ locally    //End of the Propagation
else
for all $child \in routing\_table[s]$ do
$child \rightarrow Propagate\_Sub(s-1, msg)$
end for
end if
end procedure
```

The DST propagation algorithm is presented as Algorithm.1. This algorithm uses two procedures. Propagate_Sub is a recursive procedure which propagates the message of an attack and Propagate is the procedure which initializes the propagation. The procedure Propagate_aux takes two parameters: msg, the propagated message, and s, the level of the called node in the tree (DST). Because non-leaf nodes are distributed over their descendants, every computer acts as a leaf, as a node of stage 1, …. And as the root node. His parameter s tells the computer which node receives the message. If s = h, the computer must act as the root node. If s = h - 1, it must act as the child of the root node. If s = 0, the computer must act as a leaf and it does not forward the message further. If $s \neq 0$, the computer acts as a stage s forwarding node. So, this node forwards the message to its children. To do it, the computer takes the list of computers that represent the children of its stage s node. For each of them, it asks him to forward the message as a stage s - 1 node, child of the stage s node. The propagation of attack is initialized by the Propagate procedure. This procedure must be called by a computer that is part of the DST. By calling Propagate_sub(h, msg), it asks to himself to act as the root node to forward the message msg. Because a computer always uses itself as the representative of the nodes that contain it, every computer receives only one distant message. We can conclude that the number of distant messages of a broadcast on an n-node DST is n - 1 because the computer that initializes a broadcast does not receive any distant messages. So, the complexity order of the broadcast is O(n) messages. Traces of Algorithm.1 show that the algorithm runs h + 1 recursions. So, the algorithm runs in $h + 1 \leq \log a(|C|) + 2$ steps. We can conclude that the algorithm time complexity is O (log(n)) time units.

4.3. Traceback Algorithm

The second algorithm of our proposed method is Traceback .The main aims of this algorithm is querying a number of computers which grows exponentially like a TTL-based graph flooding algorithm.This algorithm sees a propagate-like algorithm to query a sub tree of stage 1, then a sub tree of stage 2, and so on, until the DST is completely flooded or until the query is positively answered about an buffer overflow Attack.

Algorithm 2: Trace back algorithm [41]

```
Procedure $Traceback(tfs)$
```

```
AL$\leftarrow Traceback\_sub(0,tfs)$
TA$\leftarrow 1$
While $TA \leq h \wedge AL = \emptyset$ do
  $temp \leftarrow \emptyset$
  For all $child \in routing\_table[TA]$ do
    if $TA \neq self$ then
      $temp[child] \leftarrow child \rightarrow Traceback\_sub(TA,tfs)$
    end if
  end for
  for all $temp\,AL \in temp$ do    //joins found resources
    AL$\leftarrow AL \cup tempAL$
  end for
  $TA \leftarrow TA + 1$
end while
return AL
end procedure
procedure $Traceback\_sub(TA,tfs)$
if $TA = 0$ then
  $AL \leftarrow List\ of\ Attackers$
else
  $temp \leftarrow \emptyset$
  for all $child \in routing\_table[TA]$ do
    $temp[child] \leftarrow child \leftarrow Traceback\_sub(TA - 1, tfs)$
  end for
  $AL \leftarrow \emptyset$                         // List of found resources
  For all $tempAL \in temp$ do        // Join found resources
    $AL \leftarrow AL \cup tempAL$
  end for
end if
return AL
end procedure
```

with equal probability and interchanges their respective positions.

The Traceback algorithm uses two procedures. Traceback_Sub is a recursive procedure which propagates requests in subtrees and Traceback is the procedure which controls the search. To look for a resource matching a query, you need to call the Traceback procedure and pass your query as a parameter. The Traceback_Sub procedure takes two parameters to broadcast the request: TA the height of the subtree and tfs, traffic flow signature description. If TA equals 0, we query a leaf. In this case, the computer gathers the list of resources that match the query and returns the list to the caller. If TA is not equal to 0, the computer propagates the query to its subtree of height TA. - 1 To do it, for each child of its node of level TA, the computer sends a message to its representative. This message asks the computer to gather the list of resources that match the query in its subtree of height TA - 1. Then, the computer waits for the answers of the queried subtrees and merges the lists of matching resources. Finally, it returns the merged list to its caller. The Traceback procedure takes one parameter (tfs) which describes the Traffic Flow Signature. The procedure starts by searching locally a resource that matches the tfs and stores the list of found resources in the list AL. Then, starting with a height of 1, the procedure queries subtrees of increasing height until querying the whole tree or until finding a resource. To query a subtree of height TA, the Search procedure sends a message to a computer of each child of its node of level TA but one. We remind that a computer always chooses itself to represent nodes that contain the computer. So, with the test TA $\neq$ Self, we avoid to query the subtree that contains the computer because this subtree is the subtree that has been queried during the previous iteration. Once the queried sub trees return their lists of found resources, the computer merges The lists and prepares a new iteration in the case where no resources is found. At the end, it returns to the client the list of found resources that match the query.

V. SECURITY ANALYSIS

In this section, we analyse the security of our scheme as follows: As indicated in the introduction, two basic approaches are considered in the design of buffer overflow attacks. Our Objective is to develop the attack itself whilst maximizing the probability of executing the malicious code, CA [11] proved very inefficient at manipulating register references (using the standard CA search operators) than Sig-free method using code abstraction. By using DST [41], we expect to avoid this problem. In the following we describe a series of three experiments in which the instruction set is incrementally expanded, thus increasing the search space, but providing for greater freedom in the resulting program content (thus a wider range of behavioural properties). This case results in code that has the capacity to intermix attack and obfuscation. Compared to existing scheme [11], our scheme based DST Detects the all buffer overflow attacks efficiently and securely. Therefore, SigFree using DST is securing than SigFree using code abstraction while detecting the attacks.

VI. PERFORMANCE ANALYSIS

In this section, we implemented a Sig-Free prototype using the C++ programming language in the Win64 environment. The stand-alone prototype was com-piled with Borland C++ version 5.5.1 at optimization level O2. The experiments were performed in a Windows 2007 server with Intel Pentium 4, 3.2-GHz CPU, and 1-Gbyte memory. We measured the processing time of the stand-alone prototype over all (2,910 totally) 0-10 Kbyte images collected from the above real traces. We set the upper limit to 10 Kbytes because the size of a normal web request is rarely over that if it accepts binary inputs. To evaluate the performance impact of SigFree to web servers, we also implemented a proxy-based SigFree prototype. Fig. 2 depicts the implementation architecture. It is comprised of the following modules. URI decoder. The specification for URLs limits the allowed characters in a Request-URI to only a subset of the ASCII character set. This means that the query parameters of a request-URI beyond this subset should be encoded [39]. Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of SigFree is to decode the request-URI. ASCII filters. Malicious executable codes are normally binary strings. In order to guarantee the throughput and response time of the protected web system, if a request is printable ASCII ranging from 20 to 7E in hex, SigFree allows the request to pass. For the requests that SigFree identifies as normal, SigFree forwards them to the web server, Apache HTTP Server 2.0.54 hosted in a Linux server with dual Intel Xeon 1.8-Gbyte CPUs. The individual Clients send requests from a predefined URL list. The documents referred in the URL list are stored in the web server. In addition, the prototype implementation uses a time-to-live-based cache to reduce redundant HTTP connections and data transfers. Rather than testing the absolute performance overhead of SigFree, we consider it more meaningful measuring the impact of SigFree on the normal web services. Hence, we measured the average response latency of the connections by running http load for 1,000 fetches. Whenever there are no buffer overflow attacks, the average response time in the system with SigFree is only slightly higher than the system without SigFree. This indicates that, despite the connection and ASCII checking overheads, the proxy-based implementations does not affect the overall latency significantly. The architecture was implemented and its performance was proved to be good. The aim of these simulations is to compare the behaviours and the performances of tracing algorithms on top of three overlay network topologies: tree, pseudorandom graph, and DST. Our comparison is limited to these two topologies because; they are the most commonly used for self-organized networks whereas static or index-oriented topologies cannot be used in this context. To simulate the execution of tracing algorithms, we use the algorithm for the tree and the graph topologies: the initiator peer contacts its neighbours and waits for a reply; if no resource is found, then the initiator asks its neighbours to contact their neighbours and waits for their replies, and so on, until the end of the tree or the graph is reached. Hundred different types of resources are available, and every computer has a probability of 10 percent to own a resource of each type. Each search request stops either when it finds a node with the requested resources or when the whole structure is traversed.

About the overlay topologies characteristics, trees are bidirectional and their arty is 5. Graphs are also bidirectional, connected, and the degree of each node is 5. Finally, the DST is made in a way that each node has five children. These degrees were chosen because they show the best performances in our simulations. More precisely, we run some tests at various scales to find out these optimal degrees. Then, we use them for all the simulations by considering that these degrees are always optimal in our experiments. However, these values depend on the links throughput and the probability to find a service. Changing one of these parameters implies that the chosen degrees would no longer be optimal. The simulations show that the average time needed to process a request depends on the request arrival rate. This is an ordinary observation. When the number of initiated requests increases, the system becomes more and more loaded and messages spend more time in a waiting queue before being sent. When the number of requests that enter the system becomes higher that the number of requests that leave it, the system becomes saturated.

VII. CONCLUSION

We have proposed A novel Signature-Free Buffer Overflow Attack Detector Using DST that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats. Our method does not require any signatures, thus it can detect new unknown attacks. SigFree is immunized from most attack-side and good for economical Internet-wide deployment with little maintenance cost and low performance overhead. The Results show that code bloat property of the traceback provides suitable means to hide the actual attack by mixing exploit instructions with introns that have no effect toward the success of the attack. Furthermore, evolved attacks discover different ways of attaining sub-goals associated with building buffer overflow attacks, hence mimicking the core attack with different instructions. Finally, we have proved that our method is secured and efficient than existing scheme [11].

References

[1] B.A. Kuperman, C.E. Brodley, H. Ozdoganoglu, T.N. Vijaykumar, and A. Jalote, "Detecting and Prevention of Stack Buffer Overflow Attacks," Comm. ACM, vol. 48, no. 11, 2005.

[2] G. Kc, A. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.

[3] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03), Oct. 2003.

[4] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.

[5] 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), 2005.

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

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

[8] S. Singh, C. Estan, G. Varghese, and S. Savage, "The Early bird System for Real-Time Detection of Unknown Worms," technical report, Univ. of California, San Diego, 2003.

[9] H.-A.Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," Proc. 1 3t h U SE NI X Security Symp. (Security), 2004.

[10] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatic Signature Generation for Polymorphic Worms," Proc. IEEE Symp.Security and Privacy (S&P), 2005.

[11] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Z hu, "SigFree: A Signature-Free Buffer Overflow Attack Blocker", IEEE TRANSAC TIONS ON DEPEN DABLE AND SECURE COMPUTING, VOL. 7, NO. 1, JANUAR Y-MARCH 2010.

[12] Hilmi Güneş Kayacık, Malcolm Heywood, Nur Zincir-Heywood, "On Evolving Buffer Overflow Attacks Using Genetic Programming", In Proc. Of GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

[13] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities,"Proc. Seventh Ann. Network and Distributed System Security Symp. (NDSS '00), Feb. 2000.

[14] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," IEEE Software, vol. 19, no. 1, 2002.

[15] H. Chen, D. Dean, and D. Wagner, "Model Checking One Million Lines of C Code," Proc. 11th Ann. Network and Distributed System Security Symp. (NDSS), 2004.

[16] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard:Automatic Adaptive Detection an d Prevent ion of Buffer-Overflow Attacks, " Proc. Se v en t h US EN I X Security Sym p .(Security '98), Jan. 1998.

[17] GCC Extension for Protecting Applications from Stack - S mashing Attacks, http:/ /www .research.Ibm.com/trl/projects/security/ssp, 2007.

[18] T. cker Chiueh and F.-H. Hsu, "Rad: A Compile-Time Solution to Buffer Overflow Attacks," Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS), 2001.

[19] A. Smirnov and T.cker Chiueh, "Dira: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks," Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.

[20] Pax Documentation, http://pax.grsecurity.net/docs/pa x.txt, Nov. 2003.

[21] A. Baratloo, N. Sing h, and T. Tsai, "Trans parent Run-Time defense against Stack Smashing Attacks," Proc. USENIX Ann. Technical Conf. (USENIX '00), June 2000.

[22] G.S.Kc and A.D.Keromytis, "E-NEXSH: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing," Proc.21st Ann. Computer Security Applications Conf. (ACSAC), 2005.

[23] S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," Proc.14th USENIX Security Symp. (Security), 2005.

[24] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Shepherding," Proc. 11th USENIX Security Symp. (Security), 2002

[25] . Z. Liang and R. Sekar, "Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models," Proc. 21st Ann. Computer Security Applications Conf. (ACSAC), 2005.

[26] R.Pang, V.Yegneswaran, P.Barford, V.Paxson, and L. Peterson, "Characteristics of Internet Background Radiation," Proc. ACM Internet Measurement Conf. (IMC), 2004.

[27] Z. Li, M. Sanghi, Y. Chen, M.Y. Kao, and B. Chavez, "Hamsa: Fast Signature Generation for Zero-Day Polymorphic Worms with Provable Attack Resilience," Proc. IEEE Symp. Security and Privacy (S&P '06), May 2006.

[28] X.F. Wang, Z. Li, J. Xu, M.K. Reiter, C. Kil, and J.Y. Choi, "Packet Vaccine: Black-Box Exploit Detection and Signature Generation,"Proc. 13th ACM Conf. Computer and Comm. Security (CCS), 2006.

H.J.Wang, C.Guo, D.R.Simon, and A.Zugenmaier, "Shield: Vulnerability - Driven Network Filters for Preventing Known Vulnerability Exploits," Proc. ACM SIGCOMM '04, Aug. 2004.

[29] K. Wang and S.J. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," Proc. Seventh Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2004.

[30] K. Wang, G. Cretu, and S.J. Stolfo, "Anomalous Payload-Based Worm Detection and Signature Generation," Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2005.

[31] O. Kolesnikov, D. Dagon, and W. Lee, "Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic," Technical Report GIT-CC-04-13, College of Computing, Georgia Tech, 2004.

[32] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," Proc. 12th USENIX Security Symp. (Security '03), Aug. 2003.

[33] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant, "Semantics-Aware Malware Detection," Proc. IEEE Symp. Security and Privacy (S&P), 2005.

[34] A.Lakhotia and U. Eric, "Abstract Stack Graph to Detect Obfuscated Calls in Binaries," Proc. Fourth IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM '04), Sept. 2004.

[35] C. Kruegel, W. Robertson, F.Valeur, and G.Vigna, "Static Disassembly of Obfuscated Binaries," Proc. 13th USENIX Security Symp. (Security), 2004.

[36] Fnord Snort Preprocessor, http://www.c ansecwest.com/spp_fnord.c, 2007.

[37] B. Schwarz, S.K. Debray, and G.R. Andrews, "Disassembly of Executable Code Revisited," Proc. Ninth IEEE Working Conf.Reverse Eng. (WCRE), 2002.

[38] T. Berners-Lee, L.Masinter, and M. McCahill, Uniform Resource
Locators (URL), RF C 17 38 (Proposed Standard), updated by RFCs 1808, 2368, 2396, 3986, http://www.ietf.org/rfc/rfc1738.txt,2007.

[39] Http Load: Multiprocessing Http Test Client, http://www.acme.com/software/http_load, 2007.

[40] Sylvain Dahan, Laurent Philippe, and Jean-Marc Nicod, "The Distributed Spanning Tree Structure", IEEE Trans. Parallel and Distributed Systems, vol.20, no.12, pp.1738-1751, December 2009.

[41] D. Thamizh Selvam, P.S. Vinayagam, P. Syam Kumar and Dr. R. Subramanian, "A Peer-to-Peer Architecture to collaboratively Propagate and Traceback DDoS Attack Information using DST", International Journal of Computer Applications,Vol.5, No.3, August 2010. pp.8-14.

[42] Kotha Jothsna, Dr.R.V.Krishniah, "SigFree: A Signature-Free Buffer Overflow Attack Blocker using Genetic Programing",
International Journal of Emerging Technologies and Advanced Engineering" Volume 3, Issue 2, February, 2013.