Using Greedy Hamiltonian Call Paths to Detect Stack Smashing Attacks

Mark Foster, Joseph N. Wilson, Shigang Chen

Department of Computer and Information Sciences and Engineering
University of Florida
Gainesville, Florida 32611
mfoster@cise.ufl.edu

Abstract. The ICAT statistics over the past few years have shown at least one out of every five CVE and CVE candidate vulnerabilities have been due to buffer overflows. This constitutes a significant portion of today's computer related security concerns. In this paper we introduce a novel method for detecting stack smashing and buffer overflow attacks. Our runtime method extracts return addresses from the program call stack and uses these return addresses to extract their corresponding invoked addresses from memory. We demonstrate how these return and invoked addresses can be represented as a directed weighted graph and used in the detection of stack smashing attacks. We introduce the concept of a Greedy Hamiltonian Call Path and show how the lack of such a path can be used to detect stack-smashing attacks.

1 Introduction

The term buffer overflow refers to copying more data into a buffer than the buffer was designed to hold. A buffer overflow attack takes places when a malicious individual purposely overflows a buffer to alter a program's intended behavior. The most common form of this attack deals with the attacker intentionally overflowing a buffer on the stack so that the excess data overwrites the return address that resides just below the buffer on the stack. Thus when the current function returns, control flow is transferred to an address chosen by the attacker. Commonly, this address is a location on the stack where the attacker has injected his/her own malicious code inside the same buffer. This type of buffer overflow attack is also referred to as a stack smashing attack since the buffer resides on the stack. Stack smashing attacks are one of the most common forms of buffer overflow attacks due to their simplicity of implementation.

Buffer overflow attacks have been a major security issue for a number of years. Wagner et al. [1] extracted statistics from CERT advisories showing that between 1988 and 1999 buffer overflows accounted for up to 50% of the vulnerabilities reported by CERT. Wagner cites several other statistics showing where buffer overflows were at a minimum of 23% of the vulnerabilities in different databases. A more recent look at the ICAT statistics shows that a significant number of the CVE and CVE candidate vulnerabilities were due to buffer overflows. For the years 2001,

2002 and 2003 buffer overflows accounted for 21%, 22%, and 23% of the vulnerabilities respectively [2]. In addition, SecurityTracker.com released statistics for the time period between April 2001 and March 2002. These statistics show that buffer overflows were the cause behind 20% of the vulnerabilities reported by SecurityTracker [3]. These more recent statistics reinforce the case made by Wagner et al. Buffer overflows are a significant issue for system security.

The purpose of this paper is to introduce a new method for detecting stack smashing and buffer overflow attacks. While much work has been focused on detecting stack-smashing attacks, few approaches use the program call stack to detect such attacks. We propose a new method of detecting stack smashing attacks that relies solely on intercepting system calls and information that can be extracted from the program call stack and process image. Upon intercepting a system call, our method traces the program call stack to extract return addresses. These return addresses are used to extract what we refer to as *invoked addresses*. In the process image, return addresses are preceded by *call* instructions. These *call* instructions are what placed the return addresses on the stack and then transferred control flow to another location. An address that was invoked by a *call* instruction is referred to as an *invoked address*. We use the return and invoked addresses to create a weighted directed graph. We have found that the graph constructed from an uncompromised process always contains a Greedy Hamiltonian Call Path (GHCP). This allows us to use the lack of a GHCP to indicate the presence of a buffer overflow or stack smashing attack.

The rest of this paper is organized in the following manner. In Section 2 we discuss related work. In Section 3 we introduce our new method and prove its correctness using induction. Sections 4 and 5 discuss the limitations and benefits of our proposed method. Our conclusions from this study are stated in Section 6.

2 Related Work

One of the most notable approaches to detecting and preventing buffer overflow attacks is referred to as StackGuard [4]. Cowan et al. created a compiler technique that involves placing a *canary* word on the stack next to the return address. When a function returns, if the canary word has been modified, it implies that the return address has also been modified. The only downfall of Stackguard is that programs are only protected if they have been recompiled with a specially enhanced compiler.

Baratloo et al. proposed two new methods referred to as Libsafe and Libverify [5]. Libsafe uses the saved frame pointers on the stack to act as upper bounds when writing to a buffer. Libverify uses a similar approach to Stackguard in that a return address is verified before a function is allowed to return. Libverify does this by copying each function into the heap and overwriting the original beginning and end of each function with a call to a wrapper function. One downfall of this method is that the amount of space in memory required for each function is double that of what the process would require if not using Libverify.

One approach proposed by Feng et al. is VtPath [6]. VtPath is designed to detect anomalous behavior but would also work well in detecting buffer overflow attacks.

VtPath is unique in that it uses information from a program's call stack to perform anomaly detection. At each system call VtPath takes a snapshot of the return addresses on the stack. The sequence of return addresses found between two system calls creates what is referred to as a virtual path. A training phase is used to learn a program's normal virtual paths. When online, virtual paths not experienced in the training phase are considered an anomaly.

3 Overview of Technique

We propose a new method of detecting stack-smashing attacks that deals with checking the integrity of the program call stack. The proposed method operates at the kernel-level. It intercepts system calls and checks the integrity of the program call stack before allowing such system calls to continue. To check the integrity of a program's call stack we extract the return address and invoked address of each function that has a frame on the stack. Using the list of return addresses and invoked addresses we can create a weighted directed graph. We have found that a properly constructed weighted directed graph of a legitimate process always has the unique characteristic of a Greedy Hamiltonian Call Path (GHCP). We refer to this as a call path since it corresponds to the sequence of function calls that lead us from the entry point of a given program to the current system call. This call path is greedy because when searching for this path within our weighted directed graph, we always choose the minimum weight edge when leaving a vertex. Furthermore, this path is Hamiltonian because every vertex must be included *exactly once*. Most significantly, we have found that the lack of such a path can be used to indicate that there has been a stack smashing or buffer overflow attack.

3.1 Constructing the Graph

The task of constructing a weighted directed graph from the program call stack involves five major steps. We demonstrate these five steps on an example program. The functions, their source code, starting and ending addresses in memory for the example program are shown in Table 1.

The five major steps include:

Step 1 - Collect Return Addresses. Using the existing frame pointer, trace through the program call stack to extract the return address from each stack frame.

Step 2 – Collect Invoked Addresses. For each return address extracted from the stack, find the *call* instruction that precedes it in memory. Extract the invoked address from that *call* instruction. At this point we can create a table of return address/invoked address pairs. For the program shown in Table 1, the return and invoked addresses in Table 2 would be extracted.

In Table 2 it is easy to see how the addresses that start with 0x0804… correlate to the addresses in Table 1. The addresses that start with 0x420… are simply the addresses of C library functions that are used by our program. The last address, 0xc78b1dc8 is the kernel address of the system call function *execve*(). Addresses such as 0x420b4c34 and 0x420b4c6a correspond to the system call wrapper in our C

Table 1. Example program we use to demonstrate graph construction.

Function Name	Starting Address in Memory	Ending Address in Memory	Function's Code
f3()	0x08048400	0x0804842a	execve(…);
f2()	0x0804842c	0x08048439	f3();
f1()	0x0804843c	0x08048449	f2();
main()	0x0804844c	0x0804845f	f1(); return 0;

Table 2. Return Address/Invoked Address Pairs.

Return Address	Invoked Address
0x08048321	0x42017404
0x42017499	0x0804844c
0x08048457	0x0804843c
0x08048447	0x0804842c
0x08048437	0x08048400
0x08048425	0x420b4c34
0x420b4c6a	0xc78b1dc8

library. The additional addresses at the beginning of the table (i.e. 0x08048321, 0x42017404 and 0x42017499) are the addresses corresponding to *_start* and *__libc_start_main*. The purpose of these functions is not pertinent to this paper.

Step 3 - Divide Addresses into Islands. Once the values in Table 2 have been obtained we can begin construction of our weighted directed graph. Our final graph contains a node for each of the addresses in Table 2. However, before we can make each address into a node we must first categorize our addresses into what we refer to as islands. Our addresses are divided into islands based on their locations in memory. For example, addresses that begin with 0x0804… are part of a different island from addresses that begin with 0x420…. Addresses are further divided on whether they are a return address or an invoked address. In this example we have four islands. These islands are show in the Figure 1.

Note that the address 0xc78b1dc8 was not placed into an island. This is because this address represents the first instruction of our system call. It represents a unique node later on. We add this address' node when we begin adding edges. The address 0x08048300, the first instruction in the *_start* procedure, was added even though it is not in Table 2. This address is part of an ELF header and is loaded into memory, thus it can be extracted at runtime. Every program must have an entry point and therefore can be part of our graph.

Step 4 – Adding Edges. Recall the return address/invoked address pairs we have listed in Table 2. Each of these pairs are connected with a zero weight edge leading from the return address to the invoked address. At this time we can add a node for the address 0xc78b1dc8, and subsequently add a zero weight edge to it from it's corresponding return address. All of the edges added thus far are part of our final GHCP.

To complete this step, we attempt to give each invoked address node an edge to every return address node in the same memory region. These edges are weighted with the distance in memory between the two nodes. For example, the node with address 0x0804842c has a directed edge with weight 1b leading to the node with address

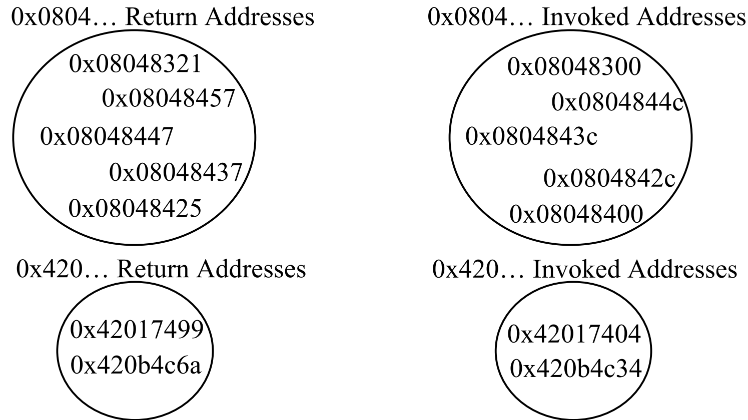

Fig. 1. Example program's addresses divided into islands.

0x08048447. In addition, the node with address 0x0804842c also has a directed edge leading to 0x08048457 with a weight of 2b. The edges leading from invoked address node 0x0804842c to every return address node of the same memory region are shown in Figure 2.

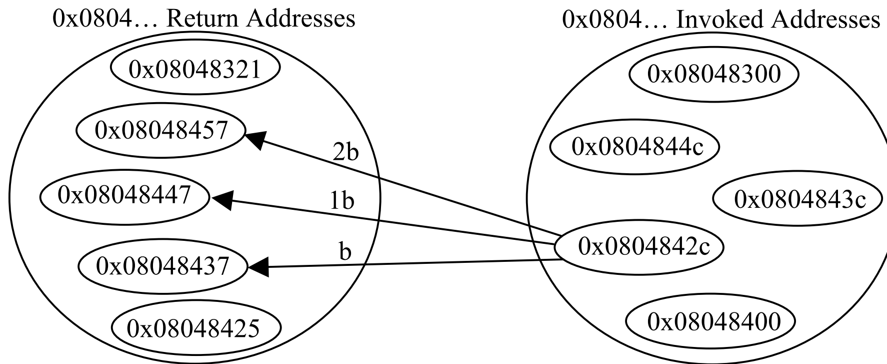

Fig. 2. Edges leading from Invoked Address node 0x08048418.

Note that the node 0x0804842c was not given an edge to nodes 0x08048321 and 0x08048425. This is because these edges would have resulted in negative weights which we do not allow in the graph. This concept is explained more thoroughly in the next subsection.

Once all of the appropriate edges have been added, the graph is complete. We have omitted the drawing of our completed graph due to the crowded nature of a graph of such a simple program.

3.2 Graph Construction Explained

Inspection of Table 2 allows us to see a relationship between a value in the ith row of column one and the $(i-1)^{th}$ row of column two. For example, we know that 0x0804842c is the address of the first instruction in our *f2()* function. Let the row with this address be the $(i-1)^{th}$ row. This means that the return address in the ith row is 0x08048437. Since we also know that the last instruction in our *f2()* function is at address 0x08048439, we know that this return address is inside of *f2()*. Furthermore, we can see in Figure 2 that when a minimum weight edge leaving the address node of 0x0804842c is chosen, it leads to the return address node of 0x08048437. Stated more formally, if we let return addresses be denoted with ω, and invoked addresses be denoted with α, a given invoked address, α_i, should have a minimum weight edge leading to return address, ω_{i+1}. This leads to the idea that every graph's GHCP is no different from the Actual Call Path (ACP) of the progrm.

It turns out, this is exactly what we need. All programs that have not fallen victim to a stack smashing or buffer overflow attack posses this ACP. We can find this ACP by searching for a GHCP. Our method must be greedy to insure that we chose the minimum weight edge when leaving a given node. In addition, since our path must include each vertex exactly once, our path is Hamiltonian. If we are unable to find such a GHCP, then we know that our ACP has been disrupted. This implies the likely occurrence of a stack smashing or buffer overflow attack.

To demonstrate why this works, suppose the function *f2()* were vulnerable to a buffer overflow attack. Suppose the attack overwrites the return address of *f2()* with the address 0x0804844a. This results in the edge from Figure 2 that was labeled with '*b*', now being labeled with a '*1e*'. Thus when a minimum weight edge leaving the address node of 0x0804842c is chosen, it no longer leads to the proper node. It leads to the address node 0x08048447, whose edge is labeled with a '*1b*'. This same address node is also the result of choosing a minimum weight edge when leaving the address node of 0x0804843c. Having two edges that both lead to the same node disrupts our GHCP. There no longer exists a path that is both Greedy and Hamiltonian. When the lack of a GHCP is detected, we know that a stack smashing or buffer overflow attack has occurred.

One assumption we make is that two functions in memory never overlap and that the initial instruction of a function is always the invoked address. We realize that some programs written in assembly may not abide by this assumption. However, all compiled programs and most assembly programs will satisfy this constraint.

To summarize, our ACP represents the expected GHCP. However, we provide multiple paths leaving a given invoked address to give that invoked address a choice when determining our GHCP. By providing a choice, it allows the other return addresses to act as upper bounds. The upper bounds created by other return addresses limit the potential range of addresses that a given return address can been overwritten and modified to by an attacker. There already exists an inherent lower bound since we do not include negative weight edges. Recall that invoked addresses are likely the address of the first instruction for a given function. Thus it makes sense that unaltered execution flow of a given function should never lead to an instruction that resides at a lower memory address than the first instruction of that function.

3.3 Proof by Induction

In order for us to rely on the nonexistence of a GHCP to indicate the presence of a stack smashing attack we must first prove that a GHCP exists for all uncompromised programs. In this section, we consider the case in which there are no recursive function calls. Knowing that our graph has two types of edges, those leaving return addresses and those leaving invoked addresses, we can simplify this proof. Since there is always exactly one edge leaving a given return address, we know this edge is always part of our GHCP. We can exploit this feature of our graph to simplify our proof. With this feature, we now only need to prove that in the ACP each invoked address always has a minimum weight edge leading to its corresponding return address. We prove, using induction that this holds true for all unobjectionable programs. Our formal inductive hypothesis is as follows:

Theorem 3.1 *For all unobjectionable programs in which n different functions have been called, where* $n \geq 1$*, every invoked address* α_i*, for* $i < n$*, has a minimum weight edge leading to the return address* ω_{i+1}*.*

With this stated we must first prove our base case.

Base Case. In this case there is one active function and no other calls have been made. Our assertion that α_i, for $i < n$, has a minimum weight edge leading to return address ω_{i+1}, is vacuously true. Alternatively, we can say that the GHCP corresponds to the ACP, because they are both null.

Inductive Case. For our inductive case we must prove that if the GHCP corresponds to the ACP for n calls, it corresponds to the ACP when the $(n+1)^{st}$ call is made. Stated more formally, we assume the following to be true:

$$GHCP_n = ACP_n = \alpha_1, \omega_2, \alpha_2, \omega_3, \alpha_3, \omega_4 \ldots \omega_n, \alpha_n$$

Thus we must prove the following to be true:

$$GHCP_{n+1} = ACP_{n+1} = \alpha_1, \omega_2, \alpha_2, \omega_3, \alpha_3, \omega_4 \ldots \omega_n, \alpha_n, \omega_{n+1}, \alpha_{n+1}$$

The $(n+1)^{st}$ call results in adding the two additional nodes, ω_{n+1} and α_{n+1}, to our graph. This also results in the additional edges, (α_i, ω_{n+1}) and (α_n, ω_{i+1}), being added to our graph. Since we know that GHCP = ACP, as long as every invoked address α_i, has a minimum weight edge leading to the it's corresponding return address ω_{i+1}, we must prove the following proposition.

Proposition: *For each i, where* $i < n$, $weight(\alpha_i, \omega_{i+1}) < weight(\alpha_i, \omega_{n+1})$

Before proceeding any further we must define some variables.

Table 3. Variables for the induction proof.

L_i	Length in bytes of the ith function.
α_i	Address of the first byte of the ith function.
$r\alpha_i$	The offset to the return address inside the ith function. ($r\alpha_i = \omega_{i+1} - \alpha_i$)

We also assume that two separate functions loaded into memory never overlap. Therefore, we must prove our proposition for two different scenarios namely, $\alpha_i < \alpha_n$ and $\alpha_n < \alpha_i$.

We can construct an abstract version of our graph as it would exist the moment our $(n+1)^{st}$ call is made. This version of our graph, Figure 3, illustrates the relationship

between the function that made the $(n+1)^{st}$ call and any other invoked/return address pairs.

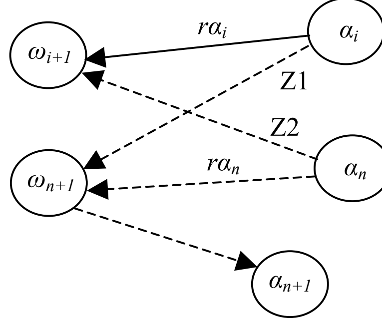

Fig. 3. Abstract graph once $(n+1)^{st}$ call is made. A solid line represents an existing edge. A dotted line represents a new edge.

Now we prove our proposition holds for both scenarios. For the scenario

$$\alpha_i < \alpha_n,$$

we know the following must also be true

$$\omega_{i+1} < \omega_{n+1}.$$

Therefore we can conclude

$$weight(\alpha_i, \omega_{i+1}) = r\alpha_i = \omega_{i+1} - \alpha_i < \omega_{n+1} - \alpha_i = Z1 = weight(\alpha_i, \omega_{n+1}).$$

Thus our proposition holds true for our first scenario. Given the second scenario

$$\alpha_n < \alpha_i,$$

we know the following must also be true

$$\omega_{n+1} < \omega_{i+1}.$$

Therefore we can conclude

$$weight(\alpha_i, \omega_{n+1}) < 0,$$

and since our graph does not contain negative edges, our proposition still holds true.

It might seem logical to conclude that we also need to prove a second proposition. This second proposition is stated below.

Proposition 2: *For each i, where i < n,* $weight(\alpha_n, \omega_{n+1}) < weight(\alpha_n, \omega_{i+1})$

Proving this proposition for the first scenario we find

$$weight(\alpha_n, \omega_{i+1}) < 0.$$

Once again, since our graph does not contain negative edges, our proposition still holds true. With the second scenario we find

$$weight(\alpha_n, \omega_{n+1}) = r\alpha_n = \omega_{n+1} - \alpha_n < \omega_{i+1} - \alpha_n = Z2 = weight(\alpha_n, \omega_{i+1}).,$$

Thus we can prove that our 2nd proposition also holds for both scenarios. However, if Proposition 1 holds, then this second proposition is unneccessary. When we arrive at the point where we must choose an edge leaving α_n, since we are searching for a GHCP, our only feasible choice is $r\alpha_n$ leading to ω_{n+1}. If the first proposition holds, every ω_{i+1} for $i < n$, has already been visited. Thus the only choice that maintains a Hamiltonian path is ω_{n+1}.

In conclusion, we have proven that when the $(n+1)^{st}$ call is made, every invoked address α_i, still has a minimum weight edge leading to it's corresponding return address ω_{i+1}. This in turn proves that if the GHCP corresponds to the ACP for n calls, it corresponds to the ACP when the $(n+1)^{st}$ call is made. Therefore we know that the

lack of a GHCP demonstrates that some form of stack smashing or buffer overflow attack has occurred.

3.4 Recursion

Recursion is the case where $\alpha_i = \alpha_n$ for some $i < n$. When this is the case, we have two different scenarios that may create a problem.

- $\omega_{i+1} = \omega_{n+1}$
- $\omega_{i+1} > \omega_{n+1}$

The first scenario creates a problem because α_i has two equal weight edges leading to ω_{i+1} and ω_{n+1}. Subsequently, these two equal weight edges are also the minimum weight edges leaving α_i. When searching for a GHCP, we won't know which edge to choose. The second scenario creates a problem because α_i has a minimum weight edge leading to ω_{n+1}. To address these scenarios we add a new theorem to our graph construction.

Theorem 3.2: *If $\alpha_i = \alpha_n$ for some i, where i < n, we don't allow the edge (α_i, ω_{n+1}) in our graph.*

With this theorem being stated, we must now prove that our GHCP still coresponds to our ACP even with this condition. We now revisit each case of our induction proof in the previous section.

It is important to note that we are not concerned about the scenario where $\omega_{i+1} < \omega_{n+1}$, for the same reasons we were not concerned about the Proposition 2 in Theorem 4.1.

Base Case (n = 1). Since this case has only one active function, the new condition has no affect on it. Once again, the GHCP corresponds to the ACP, because they are both null.

Inductive Case (n > 1). For our inductive case we must prove that if the GHCP corresponds to the ACP for *n* calls, it corresponds to the ACP when the $(n+1)^{st}$ call is made even when our new condition is applied. We know that $GHCP_n$ still corresponds to ACP_n. We know this because before the $(n+1)^{st}$ call is made, α_n is the last node in our ACP_n. Hence, ω_{n+1} does not exist yet and neither of our scenarios create a problem yet.

Once the $(n+1)^{st}$ call is made, we must still prove that when our additional condition is followed that $GHCP_{n+1}$ corresponds to ACP_{n+1}. Fortunately, we know the following:

$$\text{If } i < n, \text{ then } i \neq n$$

Thus,

$$(\alpha_i, \omega_{n+1}) \neq (\alpha_i, \omega_{i+1})$$

Since the edge (α_i, ω_{i+1}) is never the same edge as (α_i, ω_{n+1}) we can safely remove (α_i, ω_{n+1}) from our graph and our GHCP is not affected. Since (α_i, ω_{i+1}) is always left unmodified, we know that our GHCP still exists. Figure 4 shows our abstract graph when the $(n+1)^{st}$ call is made for when $\alpha_i \neq \alpha_n$ and $\alpha_i = \alpha_n$.

Figure 4 illustrates that when the $(n+1)^{st}$ call is made, regardless of whether $\alpha_i \neq \alpha_n$ or $\alpha_i = \alpha_n$, $GHCP_{n+1}$ still corresponds to the ACP_{n+1}. Our new condition never alters our (α_i, ω_{i+1}) edges.

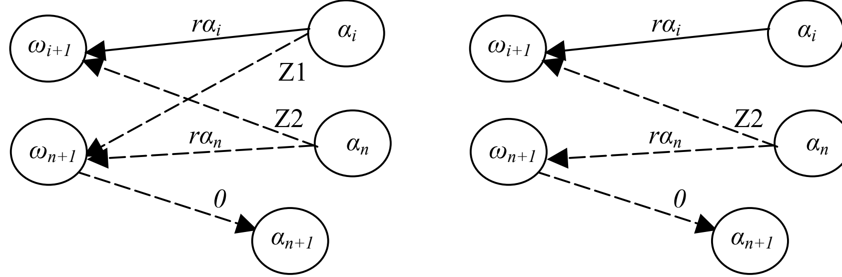

Fig. 4. Abstract graph when recursion is present, once the $(n+1)^{st}$ call is made. Left: $\alpha_i \neq \alpha_n$, Right: $\alpha_i = \alpha_n$.

To summarize, we use other function's return addresses to perform bounds checking on a specific function, say *f()*. We do not allow multiple invocations of *f()* to create bounds criteria for itself.

4 Limitations

One limitation of our GHCP analysis is that it depends on the existence of a valid frame pointer. In most cases when the return address is overwritten, the frame pointer is also overwritten. Without a valid frame pointer, there is no way to trace through the stack to extract return addresses. However, in the case where there is no valid frame pointer, we already know that some form of buffer overflow or stack smashing attack is underway. A system that implements our proposed method would detect, before even generating a graph, that no valid frame point exists. A system that only tests for the ability to trace up a stack is too easily evaded by an attacker to warrant a stand alone buffer overflow detection system. However, due to the prevalence of attacks that could be detected with such a test, we believe it should be incorporated.

There are two methods with which an attacker might be able to evade detection of a system using GHCP analysis. The first method an attacker could use to evade detection is to perform a buffer overflow attack that overwrites a return address to a new return address but leaves the frame pointer unmodified. The second method an attacker could use to evade detection, is to perform a buffer overflow attack that overwrites the return address and frame pointer and also injects code onto the stack. The first few instructions of this injected code must restore the return address and frame pointer to their original values. The injected code would then jump to preexisting code the attacker wants to execute. Both of these methods could work, but they exhibit a major limitation. The new return address or the preexisting code jumped to by the injected code must reside in the same function as the original return address. Recall that each invoked address must have a minimum weight edge to its corresponding return address. If the new return address is inside of another function, the attacker risks destroying the minimum weight edge between the invoked address and the original return address. Likewise for the second method. When a system call is made, a return address is placed on the stack. Thus, if the injected code has jumped to a piece of code in another function, the attacker risks this return address not having

a minimum weight edge from its corresponding invoked address. While both methods are possible, the challenges facing the attacker are much more rigorous than without GHCP analysis.

Another limitation of our method is its ability to deal with function pointers. Currently, we use return addresses to trace through memory and find a corresponding invoked address. These invoked addresses are part of a *call* instruction in memory. The bytes in memory representing a *call* instruction include an address or offset to an address. In either case we can extract the address invoked by this *call* instruction. In the case of function pointers, the *call* instruction is often calling an address that is in a register. We have no way to determine what address was in this register when this *call* instruction executed. However, we propose one could easily modify a compiler to store invoked addresses on the stack. This would give us the ability to always be able to determine a return address' corresponding invoked address. Function pointers would no longer be a limitation.

5 Benefits

A system designed for buffer overflow detection using GHCP analysis has a number of benefits. First, our system does not require access to a program's source code. Secondly, it does not require that a program be compiled with any specially enhanced compiler or even have the executable binary file rewritten unless wants to verify call through pointers to functions. In addition, our system does not require linking with any special libraries or place any additional burden on the application programmer. Many intrusion detection systems also rely on a training phase with a program to learn it's normal behavior. After a training phase, the system can monitor a program to ensure that it doesn't deviate from the behavior observed during the training. Our system does not require any training phase.

Our method is similar to a nonexecutable stack because it makes it extremely difficult for an attacker to execute malicious code on the stack. However, our method provides a number of benefits the executable stack does not. For example, in addition to stack smashing attacks, our method can also detect heap smashing attacks. Furthermore, it is likely to also detect a similar attack that uses the bss or data segment. Our method would also detect most attempts to rewrite a return address to another location in preexisting code. A nonexecutable stack would not detect such an attack. Lastly, our method allows code with a legitimate stack trace to execute code on the stack. In cases where an uncompromised process needs to execute code on the stack, the nonexecutable stack would not allow such a process to proceed.

Our method also provides the framework for even more concise buffer overflow detection system. Currently, one of the limitations of our method is that we rely on other functions in the call path for our bounds checking criteria for a given function. A compiler could easily be modified to inject a dummy function in between every function in a given program. The code for the ith dummy function would consist of only the code required to call the $(i+1)^{st}$ dummy function. By calling the sequence of dummy functions before starting *main*() we would place the necessary bounds checking criteria on the stack that we need for any function in our program. The cost

of this is in the compilation and start up times of the program. In addition, computation of the GHCP would only require time proportional to the number of active functions.

6 Conclusions

In this paper we have introduced a novel method for detecting stack smashing and buffer overflow attacks. We have shown how the return addresses extracted from the program call stack can be used along with their corresponding invoked addresses to create a weight directed graph. We have also introduced the concept of a Greedy Hamiltonian Call Path (GHCP) that exists in such a graph for all unobjectionable programs. Thus, the lack of a GHCP can be used to indicate the existence of a stack smashing or buffer overflow attack.

In addition, our work has laid the framework for an even more concise detection system for stack smashing and buffer overflow attacks. Using our methods in conjunction with an enhanced compiler could remove the limitations experienced by our system involving function pointers and programs with few active functions. An existing compiler could be easily modified to include the necessary modifications.

We have begun implementing a prototype for our method. Early results show a promising outlook for low overhead. Future work includes continued development of our prototype with more exhaustive testing of overhead and compatibility with items such as setjmp/longjmp calls. We expect such items to be compatible with our method but it remains unconfirmed and beyond the scope of this paper.

References

1. D.Wagner, J. Foster, E. Brewer, A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities", In Proceedings 7th Network and Distributed System Security Symposium, Feb, 2000.
2. ICAT Vulnerability Statistics, Downloaded at: http://icat.nist.gov/icat.cfm?function=statistics
3. SecurityTracker.com Statistics, Found at: http://www.securitytracker.com/learn/securitytracker-stats-2002.pdf
4. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Waggle, Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", Proceedings of the 7th USENIX Security Conference, San Antonio, TX, 1998.
5. A. Baratloo, N. Singh, "Transparent Run-Time Defense Against Stack Smashing Attacks", In Proceedings of the 2000 USENIX Technical Conference, San Diego, CA, Jan, 2002.
6. H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, W. Gong, "Anomaly Detection Using Call Stack Information", IEEE Symposium on Security and Privacy, Berkeley, CA, May, 2003.