Anomalous Path Detection with Hardware Support

Tao Zhang Xiaotong Zhuang Santosh Pande Wenke Lee

College of Computing

Georgia Institute of Technology

Atlanta, GA, 30332-0280

{zhangtao, xt2000, santosh, wenke}@cc.gatech.edu

ABSTRACT

Embedded systems are being deployed as a part of critical infrastructures and are vulnerable to malicious attacks due to internet accessibility. Intrusion detection systems have been proposed to protect computer systems from unauthorized penetration. Detecting an attack early on pays off since further damage is avoided and in some cases, resilient recovery could be adopted. This is especially important for embedded systems deployed in critical infrastructures such as Power Grids etc. where a timely intervention could save catastrophes. An intrusion detection system monitors dynamic program behavior against normal program behavior and raises an alert when an anomaly is detected. The normal behavior is learnt by the system through training and profiling.

However, all current intrusion detection systems are purely software based and thus suffer from large performance degradation due to constant monitoring operations inserted in application code. Due to the potential performance overheads, software based solutions cannot monitor program behavior at a very fine level of granularity, thus leaving potential security holes as shown in the literature. Another important drawback of such methods is that they are unable to detect intrusions in near real time and the time lag could prove disastrous in real time embedded systems. In this paper, we propose a hardware-based approach to verify program execution paths of target applications dynamically and to detect anomalous executions. With hardware support, our approach offers multiple advantages over software based solutions including minor performance degradation, much stronger detection capability (a larger variety of attacks get detected) and zero-latency reaction upon an anomaly for near real time detection and thus much better security.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; C.1.0 [**Processor Architectures**]: General

General Terms

Security, Design, Performance

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'05, September 24-27, 2005, San Francisco, California, USA.

Copyright 2005 ACM 1-59593-149-X/05/0009...$5.00.

Keywords

Anomaly Detection, Hardware Support, Anomalous Path, Control Flow Monitoring, Monitoring Granularity

1. INTRODUCTION

Moore's law has brought plenty of computing power and memory within the reach of inexpensive embedded systems. In the mean time, we are moving toward a world of pervasive computing and network connectivity, as intelligent embedded products surround us in our daily lives.

Embedded systems are indispensable components in critical national infrastructures such as power grids, aviation control, biosensors, highway traffic controllers etc. In such environments, embedded systems play a key role of sensing, synthesizing, relaying, and communicating important real time information. In many cases, they are also responsible for real-time control and actuation. However, being used in a networked environment exposes such embedded systems to potential attacks. Embedded software is in nascent stage (mostly written in C and assembly) and like any other software could be plagued with security flaws. Potential holes like software bugs, misconfigurations, misuses etc., render systems vulnerable to malicious attacks in a networked environment. In recent years, CERT reports that the rate of discovery of new vulnerabilities has doubled; accompanying such vulnerabilities is a dramatic increase in intrusion activities [1]. Such an attack on an embedded system could be highly disastrous and its impact very severe. For example, a malicious attack could bring down a critical subsystem and possibly the whole power grid could be tripped across the country. Thus, it is very important to detect and stop attacks on such critical systems in near real-time.

To protect computer systems from unauthorized penetration, intrusion detection is an important component of the overall security solution. Due to following reasons, intrusion detection has become an indispensable means to help secure a networked computer system. (1) A completely secure system is impossible to build. (2) Protecting software through cryptographic mechanisms, such as in the XOM [15] architecture, cannot prevent software's internal flaws (such as buffer overflows), which might be exploited by deliberate intruders. (3) Other operational mistakes, such as misuses, misconfigurations etc., may jeopardize the systems as well. (4) In addition to system inputs (such as input strings, data etc.), embedded systems are typically designed to react to external stimuli – an attacker can also launch an attack by faking such external stimuli in addition to the above.

Traditionally, intrusion detection can be classified into misuse detection and anomaly detection. Misuse detection tries to identify known patterns of intrusions with pre-identified intrusion signatures. On the other hand, anomaly detection assumes the nature of an intrusion is unknown, but will somehow deviate the

program's normal behavior. Misuse detection is more accurate, but suffers from its inability to identify novel attacks. Anomaly detection can be applied to a wide variety of unknown (new) attacks, however the distinction between normal behavior and anomalies must be properly defined to reduce the number of false positives (or false alarms). This paper will focus on anomaly detection.

A number of anomaly detection techniques have been proposed. Most early anomaly detection approaches analyze audit records against profiles of normal user behavior. Forrest et al. [2] discovered that a system call trace is a good way to depict a program's normal behavior, and anomalous program execution tends to produce distinguishable system call traces. A number of papers [7][8] focus on representing system call sequence compactly with finite-state automata (FSA), but these schemes are easily evadable because they use very little information and monitor with coarse granularity. Recent advances [3][4][5] propose including other program information to achieve faster and more accurate anomaly detection.

In general, approaches that check the program with finer granularity should be able to detect more subtle attacks. Both [5] and [6] expose important attacks that current anomaly detection mechanisms are unable to detect. To detect such anomalies, we must follow and check each branch instruction and further each execution path. However, software anomaly-detection systems already suffer from huge performance degradation even when operating at system call level. Thus, refining the granularity further in a software-based solution could lead to severe performance loss and is thus not viable. Due to inability to work at finer granularities, software based intrusion detection cannot offer near real-time detection capabilities which are highly essential for embedded systems that are a part of critical infrastructure as discussed above. Moreover, the anomaly detection software itself can be attacked like any other software.

In this work, we present a hardware-based scheme to detect anomalies by checking program execution paths dynamically. Our anomalous path checking scheme offers multiple advantages over software-based solutions including near zero performance degradation, much stronger detection capability and zero-latency (near real-time) reaction upon an anomaly. These capabilities are ideal for real-time responsiveness and resilience.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 motivates and elaborates our anomalous path checking technique. Section 4 proposes a hardware implementation of the technique. Section 5 discusses several additional implementation issues. Section 6 evaluates precision of the anomalous path checking technique and efficiency of our hardware implementation. Finally, section 7 concludes the paper.

2. RELATED WORK

First, we would like to clarify the difference between anomaly detection and the buffer overflow detection. There has been a lot of work on using specialized software and hardware to detect buffer overflows such as [22][23][24]. These techniques focus on attack mechanism rather than symptoms and plug the hole so that attacker cannot use the particular exploit. However, due to many sources of vulnerabilities, it is possible to start an attack using other exploits and thus, a mechanism that detects attack based on its symptoms rather than a particular exploit is always desirable. As against buffer overflow mechanisms that detect illegal tampering of memory, our scheme can be used to detect any kind of attack, as long as the attack changes normal program control flow somehow (and thus, our technique is a symptom-based technique as against based on a particular exploit). Buffer overflow is merely one exploit that can cause anomaly. Other examples include format string attacks, Trojan horses and other code changes, maliciously crafted input, unexpected/invalid arguments/commands to divert control flow into buggy/rare paths etc. Our scheme can also detect viruses alternating normal behavior of other applications, like a WORD MARCO virus. Actually, many viruses cause other legal applications (WORD, internet explorer etc.) to behave illegally to perform damage since the virus itself is very compact and has limited code. Such altered behavior can be detected as an anomaly from the normal one. In short, our scheme deals with a much broader problem than buffer overflows. Moreover, the ***biggest*** benefit of anomaly detection is its ability to prevent future/new attacks rather than countering existing attacks. It is certain new attacks will be developed (based on new exploits) and it is obviously a bad strategy to research on countermeasures to them only after they have caused a huge damage.

Researchers have shown that a great number of attacks can be detected by analyzing program behavior during its execution. Such behavior could include system calls, function calls, control and data flows etc. We will first discuss system call based anomaly detection, and then other solutions that provide finer granularity in terms of detection capabilities.

Anomaly Detection via System Call Monitoring

System calls are generated as the program interacts with the kernel during its execution, examples of which are *fopen()*, *fgets()*, and *fclose()*. Forrest et al. [2] argue that system call trace appears to be a good starting point for anomaly detection. System call trace can be considered as a distilled execution trace leaving many program structures out. Although system call trace is a great simplification of the whole program activities, storing and checking against all normal system call traces is a tremendous design effort. Exemplary approaches include [7][8].

To motivate our work, we first discuss FSA (Finite State Automata) based techniques. To construct the FSA for system call monitoring, each program statement invoking a system call becomes a state on the state machine diagram. The transitions between states are triggered by system calls. Each transition edge in the FSA is labeled by the triggering system call and the target state is determined by the feasible control flow. The state machine can be easily constructed through static analysis of the program [3] or by dynamically learning the system call trace and the program counter [4]. One significant property of establishing the FSA based on static program analysis is that no false positives will be generated due to the conservative nature of static analysis. Thus, none of the normal executions will trigger alarms and if the state machine reaches the error state, there is a guarantee that something is anomalous.

Towards Finer Granularity

Although FSA-based approaches have a nice property of zero false positives, later work points out that simply checking the FSA for system calls is not sufficient and may cause false negatives in some cases. For instance, Wagner et al. notice that impossible paths may result at call/return sites which the attacker can exploit [3]. This can cause the anomalous control flow along those paths being undetected. They propose a second model called abstract

stack model, which records call stack information to solve the problem. In [5], the authors find out that even the abstract stack model can miss important anomalies. In Figure 1.a, although the condition in line two is false (i.e. the person executing the program is not a super user), the attacker can cause a buffer overflow to return from f() to line seven instead of line four, which actually grants him super user privilege. Since the system call trace is unchanged, no alarm will be triggered. This serious drawback can lead to attacks getting through.

[5] provides a makeup solution called vtPath to detect the case in Figure 1.a by considering more program information. By constructing two hash tables to store not only the possible return addresses but the so-called *virtual paths* (i.e. the sequence of procedure entry/exit points and return addresses traversed between two system calls), the false negative can be avoided. However, [5] also acknowledges that other anomalies may be left undetected. As shown in Figure 1.b, when there is no system call in function f(), a buffer overflow attack can easily grab undue privilege without being detected. This example shows that the granularity at system call level is not fine enough to detect many anomalies in reality, since anomaly can take place *between* system calls.

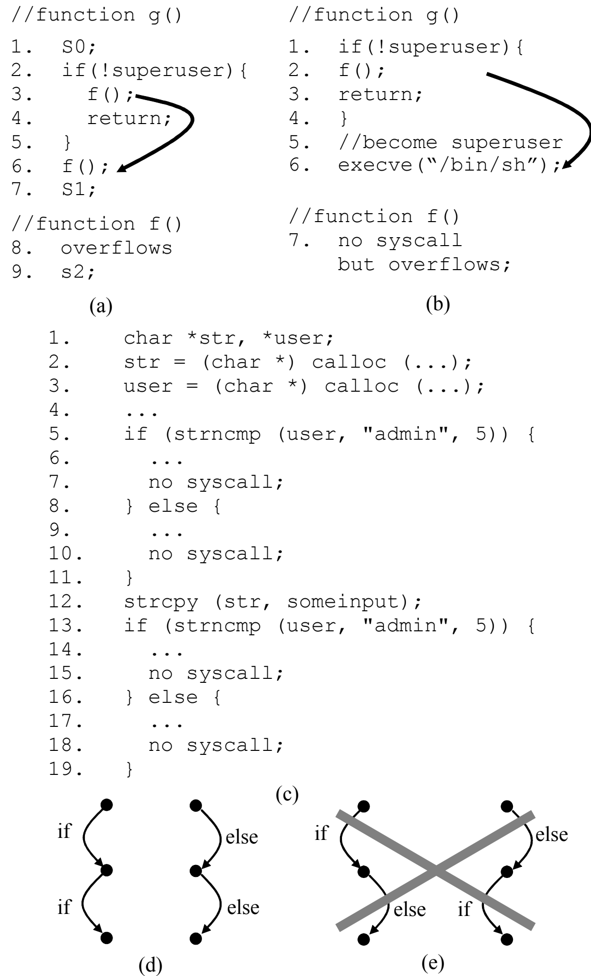

Figure 1. Attacks prompting detection at finer granularity. (Examples cited from [5] and [6]).

[11] proposes incorporating system call arguments into the detection model, which is also an example of incorporating more information. However, the attack shown in Figure 1.b does not depend on system call arguments. Most recently, [9] categorizes system call based anomaly detection systems into "black box", "gray box" and "white box" approaches. Systems relying not only on system call numbers but also on extra information extracted from process memory fall into "gray box" category. [9] further systematically studies the design space of "gray box" approaches and acknowledges the importance of monitoring granularity to the accuracy of the system. Their follow up work [10] gives a new "gray box" anomaly detection technique called *execution graph*, which only accepts system call sequences consistent with program control flow graph. However, similar to [5], due to the limitation of monitoring granularity, execution graph is not able to detect the attack in Figure 1.b either.

One possible solution to the problem illustrated in Figure 1.b is to verify all *jump instructions*, which are instructions that can cause non-sequential execution, including all conditional/ unconditional branches, function calls/returns, indirect jumps, etc. In our previous work [12], compiler records whether each instruction in the original program binary should be a jump instruction and collects all valid target addresses for each jump instruction. At runtime, a hardware anomaly detector monitors program execution based on the information collected by the compiler. Inside the hardware anomaly detector, there is also a special hardware component called return address stack (RAS) to handle function calls/returns and detect stack smashing attacks efficiently. RAS is simply a stack for pushing/popping return addresses. Each return address of a dynamic return instruction is compared against the return address popped up from the top of the RAS. If they do not match, an anomaly is detected. RAS is automatically spilled to memory and restored from memory by hardware when program call nesting level is larger than the size of the RAS. In summary, our previous work [12] checks each jump instruction and achieves very fine-grained monitoring granularity. It is able to detect a part of attacks occurred during two system calls. For example, it easily detects the attack shown in Figure 1.b. since the attack is based on stack smashing and is detected by RAS. The essentially same attack can also be done by tampering a function pointer target by buffer overflows. Most likely, the indirect function call will have an invalid target and will be detected by the indirect-jump checking component in our previous approach.

3. ANOMALOUS PATH CHECKING

3.1 Motivation

Checking jump instructions *individually* still faces difficulties in detecting certain anomalies. Figure 1.c illustrates the problem. In this example, we have two strings defined, namely *str* and *user*. If the string *user* equals *admin*, the application will be granted super-user privilege. The code consists of two if-else statements. Between the two if-else statements, there is a *strcpy* library function call to copy some inputs to string *str*. Notice that, under a normal execution, the user can be either *admin* or *guest*. Thus, if we focus only on the individual branches, since each if-else is regarded as an independent jump and since both branches of the jumps are possible, a scenario where *if* branch is taken in the first conditional branch and *else* branch is taken in the second one will be regarded as a normal execution. However, string *user* is not changed throughout this code segment, thus in fact either the two *if* branches are both taken or the two *else* branches are both taken in any legal execution, as shown in Figure 1.d. In other words, it is not possible to have one if-else statement taking *if* branch and the

other taking *else* branch, as shown in Figure 1.e. Nevertheless, since the *strcpy* function may cause buffer overflow and overwrite the string *user*, the attacker can change the contents in *user* after one if-else statement, guiding the program on the two paths shown in Figure 1.e.

The key aspect of this attack is that the attacker is not tampering return addresses or function pointers. Instead, he tampers critical control decision data. Unfortunately, as pointed out in [6], none of the existing techniques mentioned earlier can detect this type of anomalies. Even our previous work fails, since it checks each jump instruction separately. It is only aware that for the first if-else statement, both *if* and *else* branches might be taken in normal cases; and the same is true for the second if-else statement. Therefore, it will not trigger alarm for the case shown in Figure 1.e. In other words, without correlating multiple jump instructions, anomalous execution paths cannot be discovered when each jump instruction jumps to a target that is considered normal. Therefore, in order to identify this kind of attacks, we should enhance anomaly detection with ***anomalous path checking***, in which the program dynamic execution paths are checked against their normal behavior collected through profiling/training. A program's dynamic execution path is determined by the jump instructions along the path and their target addresses. Our anomalous path checking technique relies on and incorporates techniques to check each individual jump instruction done in our previous work (described briefly in section 2).

As concluded in [5], the ability of anomaly detection systems heavily relies on the granularity level it monitors. An anomaly detector solely relying on system call trace is crippled if an attack takes place between two system calls. It is easy to think about other examples that can foil the anomaly detector even if limited program control flow information is considered. Checking each jump instruction as proposed in our previous work can lead to very fine-grained anomaly detection. With anomalous path checking, we further enhance detection capability, because multiple jump instructions on a path can be correlated to find path anomalies that will otherwise be left undetected. A simple example of such anomaly is shown above (Figure 1.c). In reality, there could be many similar or even more subtle anomalies, requiring the strongest anomaly detection technique we can offer.

3.2 N-jump Path Checking

Checking program execution path is much more complicated than checking each jump instruction separately. Theoretically, the number of possible paths could be exponential to the number of jump instructions in the path. This is so since each jump instruction (if assumed a conditional branch) could have two possible directions, i.e. taken or not taken. In other words, checking the entire execution can be very expensive and infeasible to be implemented for runtime monitoring. In this work, we propose to analyze the whole execution path using sliding windows, where each window is a segment (say, n jumps) of the execution path. The anomaly detector verifies *whether these path segments follow the normal execution path*. In our scheme, the jump target of each direct jump instruction has been checked separately (done in our previous work and discussed in section 2) and an unconditional branch has only one possible target. Thus, we only need to care about conditional branches and indirect jumps, which we call *multi-target* jump instructions. We further define an ***n-jump path*** as an execution path on which exactly n multi-target jump instructions are encountered. An n-jump path can be uniquely decided by the address of starting jump instruction and the directions of the n multi-target jump instructions along the path. In our work, *the direction of an indirect jump is represented by its target addresses*. ***N-jump path checking*** involves collecting n-jump paths seen during training to build a normal n-jump path set and checking n-jump paths during detection against normal n-jump paths. If an n-jump path never seen in the training is detected, we regard it as an anomaly.

We will show later that a larger n can help detect more attacks but also incurs more false positives and more performance overheads. The criterion to choose n is to select as big an n as possible given that both false positive rate and performance overhead are acceptable. Thus, the user can start with a large n then gradually reduces it until the two metrics are acceptable.

3.3 Training Phase

During training, we run the program to be monitored in a clean environment and use an existing technique [14] to record whole program paths (WPPs) of it during its execution. A WPP records the entire program execution path under a specific program input. Each input will generate a WPP, which represents a normal execution of the program (without intrusions). From each dynamic multi-target jump instruction recorded in a WPP, an n-jump path for that jump instruction can be extracted. The n-jump path records directions of all the n jumps along the path starting from the current jump instruction. It is identified by the address of the current jump instruction and the directions of all the n jumps along the path. Essentially, a WPP is segmented into a collection of n-jump paths using a size n sliding window with sliding step 1. N-jump paths obtained from different WPPs are finally aggregated together. This means as long as an n-jump path is normal for any one of the inputs, then it is regarded as normal. The union set is used to detect path anomaly at runtime.

The length or amount of training is controlled by the user. The goal is to have adequate training so that the detector has reasonably low false positive rate. In our experiments, we will show training for our scheme is effective.

3.4 Detection Phase

At runtime, the anomaly detector maintains a record of the latest n multi-target jumps and their directions, and creates a dynamic n-jump path based on these recorded n jumps. The n-jump path is identified using the address of the first jump (the oldest of the n jumps) instruction and the directions of the following n jumps. The anomaly detector then checks whether this n-jump path exists in the set of n-jump paths collected during training. If not, the detector reports an anomaly. We will show later in the paper that the process can be optimized at run-time using hardware.

3.5 Detection Capability

As discussed in [5], the detection capability of an anomaly detection system depends on its monitoring granularity. For anomaly detection systems operating at system call granularity, attacks between two system calls will not be detected. An example is shown in Figure 1.c. Ideally, we want to collect the dynamic instruction sequence of a program and check whether the sequence of instructions is a normal one. Anomalous path checking technique approaches this ideal case by monitoring dynamic program paths. A dynamic program path is determined by the jump instructions along the path and their target addresses. Thus, with anomalous path checking, although the attacker may be able

to modify instructions between two jumps, he cannot create new jump instructions in his attack code. The control flow of his attack code has to be a straight line and the attack code cannot include any function call, which is the most restrictive so far.

Anomalous path checking actually subsumes system call sequence based anomaly detection because each function call is a jump instruction itself and each system call is normally invoked through a library function call in libc [9]. Once the dynamic program path is known, the system call sequence along the path is determined too. The sequence of system calls made along a normal program path has to be a normal system call sequence, which means system call sequence based scheme cannot detect any attack undetected by anomalous path checking. On the other hand, a normal system call sequence does not guarantee a normal program path. The distribution of system calls in a program is sparse and irregular. There may be millions of jump instructions between two system calls and the program path between two system calls can be tampered without being detected under system call sequence based scheme.

However, checking the whole program path imposes too much cost in terms of both storing normal paths and runtime checking. Thus, we choose to check n-jump paths instead. Checking all n-jump paths cannot guarantee that the entire program execution path is normal due to the limited length of n-jump paths. It is necessary to figure out the cases when path anomalies cannot be detected with n-jump path checking. Figure 2 illustrates one such case. Here, two normal paths (1) and (2) share at least n-1 jump instructions in the middle. Path (3) is an anomalous path but it cannot be detected. When path (3) enters the shared part, the detector will think it is on path (1). When path (3) leaves the shared part, it is considered to be on path (2) with n-jump path checking. The reason is that n-jump path checking can only check a path traversing no more than n multi-target jump instructions. Fortunately, our experiments show that the scenario in Figure 2 rarely happens.

We do not try to design a general algorithm to decide the value of n for an arbitrary application, since it is highly application dependent. However, it is very clear that a higher n leads to higher security strength and higher performance overhead. Our suggestion is that user picks the largest n value with which the performance degradation is still acceptable.

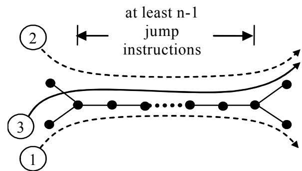

Figure 2. Anomaly path that cannot be detected.

4. HARDWARE IMPLEMENTATION

4.1 The Advantages

Performance

Although it is obvious that finer granularity brings better detection capability, the performance overhead it introduces is formidable under software based approaches, e.g., [3][4][5]. Software based anomaly detection systems suffer from large performance degradation when operating even at the system call granularity [3]. With hardware support, performance degradation can be made very small since the monitor is entirely implemented in hardware and works in parallel with the instruction execution. In comparison, in system call tracing based software approaches, intercepting system call alone can incur 100% to 250% overhead [4].

Anomaly Detector Tamper Resistance

Anomaly detection software, as any software, might itself be attacked. To alleviate this problem, software anomaly detectors are normally implemented as a separate software module. In that way, the detector is not affected by the vulnerabilities of the monitored program. For example, system call based monitoring normally depends on special system tools such as strace to trace system calls rather than instrumenting program binary.

However, such separation is not possible when we monitor the program at a very fine granularity. Normally there will be a dynamic jump in every 10 instructions. Imagine the performance degradation of the monitored program when there is a context switch after every 10 instructions. Thus, we have to transform the monitored binary to insert monitoring code into it, which means the anomaly detector faces potential attacks due to the vulnerabilities not only in the detector but also in the monitor program. It is possible that the monitoring code is tampered or bypassed due to vulnerabilities. On the other hand, our hardware anomaly detector resides in a secure processor, achieving tamper resistance.

4.2 Hardware Architecture Overview

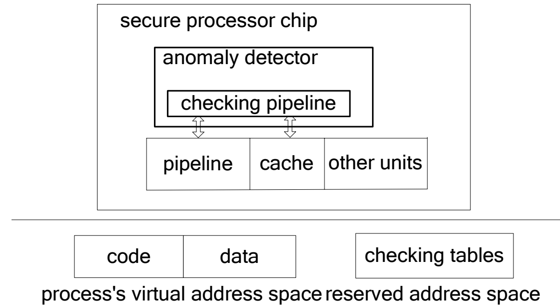

Figure 3. Architectural overview.

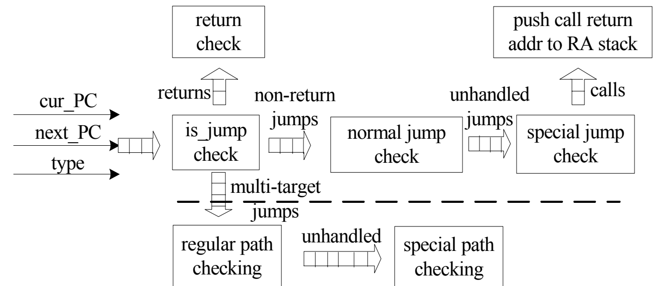

Figure 4. Checking pipeline.

Figure 3 shows an overview of hardware components. Our anomalous path checking mechanism is built into a secure processor. The processor chip (upper part) is physically secure and the external memory (lower part) is open to physical attacks. The secure processor guarantees the confidentiality and integrity of all code and data in external memory. Hardware anomaly detector resides in the secure processor and is tamper resistant. Checking tables record normal behavior of the program (including normal n-jump paths) and are utilized by the checking pipeline. Upon program start, checking tables are loaded into a reserved segment of program virtual address space and are only accessible to the anomaly detector itself. The program itself has no access to

checking tables. Attempts to read/write checking tables from programs will be detected and prevented by the processor. Moreover, physical attacks to checking tables are detected by the integrity-checking scheme of the secure processor. Thus, no software or hardware attack can tamper program normal behavior data.

The checking pipeline (refer to Figure 4) checks committed jump instructions against the checking tables. The checking pipeline is designed to be isolated from the original execution pipeline to minimize implications to the latter. There are data paths from the execution pipeline to the checking pipeline to transmit information required by anomaly detection including current operation code, current PC address and computed next PC address etc.

The internal of the checking pipeline is shown in Figure 4. The pipeline stages at the top of the figure (marked by the dotted line) check individual jump instructions in a similar way as in our previous work. The first stage checks whether each instruction is and should be a jump instruction. If it is a valid jump instruction, the second stage further checks each direct jump instruction (a conditional or unconditional branch) to make sure its jump target is valid. Whether an instruction should be a jump instruction and the valid targets of direct jumps can be easily collected by examining the original program binary. The information obtained is recorded in checking tables. At runtime, dynamic instructions are checked against those tables. Return instructions are checked specially by a hardware managed return address stack (RAS), a brief description of which is in section 2. Our techniques can also handle special cases such as setjmp()/longjmp().

The checking of indirect jumps is incorporated in our path checking stages shown at the bottom of Figure 4. Our path checking stages follow the first stage and consist of two stages, namely regular path checking and special path checking. Among the checking tables, the ones that are used for our anomalous path checking are called *path checking tables*, which record all normal n-jump paths. The details of the path checking stages will be discussed in the next section.

4.3 Implementation Details

The goal of the anomalous path checking hardware is to check dynamic n-jump paths against normal n-jump paths collected during training efficiently. An n-jump path is an execution path with n multi-target jump instructions on it. For example, in Figure 5 the thick line shows a path starting from the end of BB1 going through three basic blocks. Along that path, 3 conditional branch instructions are encountered, i.e. the last instructions of BB1, BB2 and BB3. Obviously, we can uniquely define an n-jump path by the address of its first jump instruction and the directions of the n multi-target jump instructions on the path. To indicate the directions of the n multi-target jump instructions, we define *n-jump path vector* as follows.

N-jump Path Vector: a vector marking the directions of the n multi-target jump instructions on the n-jump path.

Thus, if we only consider conditional branches that have two target addresses, e.g. either fall-through or jump to a particular target address, an n-bit vector is enough to represent an n-jump path vector. For the example in Figure 5, the 3-jump path can be uniquely defined as starting at jump instruction A with 3-jump path vector 011. Here, we assume left branch is marked as 0 and right branch is marked as 1. The most significant bit is the direction of the starting jump instruction. Therefore, an n-jump path can be represented succinctly because only the directions instead of the addresses of jump instructions are recorded except for the header of the path. For this example, the path checking table only records that the jump instruction A as the header and directions of the 3 jump instructions which are left-right-right, i.e. 0-1-1. This simplification is possible because the correct jump targets of each direct jump instruction (e.g. the left branch of instruction A should jump to the beginning of BB2 but not other places) are verified by the other pipeline stages at the top of Figure 4 — refer to section 4.2.

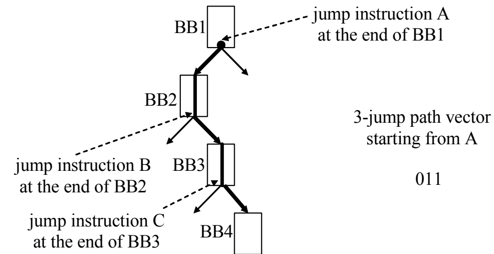

Figure 5. An example of n-jump path.

Therefore, n-jump paths can be grouped according to their headers. For example, in Figure 5, all 3-jump paths starting from instruction A can be grouped and stored together. We group and store all normal n-jump paths in the path checking tables (Figure 3). There are two tables used in anomalous path checking. Regular path checking table is used in regular path checking stage and special path checking table is used in special path checking stage.

One difficulty on path checking table design is that the data in it cannot be entirely regular, since we have to consider indirect jumps that may reach more than two targets and whose directions are represented by their target addresses. Although indirect jumps are not easy to handle, the majority of dynamic jump instructions are direct jumps, either conditional or unconditional. Unconditional branches have only one possible target and are handled by other stages. Conditional branches have two jump targets that are known by examining program binary. For conditional branches, only 1 bit is needed to indicate whether the branch is taken or not taken. Moreover, a large portion of indirect jump instructions are actually return instructions, which are checked separately as mentioned in section 4.2 and are not considered in path checking. Normally, only less than 2% of dynamic jump instructions are non-return indirect jumps. In conclusion, the jump instructions requiring irregular path data (rather than a single bit) are rare, which inspired our two-stage path checking scheme. The regular path checking stage handles most n-jump paths consisting of only conditional branches (those causing collisions after hashing are not handled), while all other cases are left to the special path checking stage.

<u>Regular Path Checking</u>

This stage handles dynamic n-jump paths consisting of only conditional branches. Thus, each jump instruction in the path has two directions. The n-jump path vector becomes an n-bit vector. The data structure for regular path checking is called regular path checking table and is shown in Figure 6. For each jump instruction, all normal n-jump paths starting from that jump instruction are stored together. Here, we use an all_path_vector to record all normal n-jump paths for each jump instruction. In this case, each n-jump path vector is an n-bit number, and therefore corresponds to a unique number within the range of [0, 2n-1]. There are a total

of 2n possible distinct paths. The all_path_vector contains 2n bits, in which each bit corresponds to an n-jump path. With a bit set in the all_path_vector, the corresponding n-jump path is indicated to be normal and vice versa. Essentially, it is a bit vector recording which n-jump path is normal. The least significant bit represents path 0…0, the next bit presents path 0…1, and so on. It is noteworthy that we store the all_path_vector instead of all normal n-jump path vectors. This can be explained from two aspects. First, the all_path_vector is of fixed size, i.e. 2n bits, while the number of normal n-jump paths is not fixed. Fixed size entries are much easier to manage with hardware, leading to smaller hardware cost and performance degradation. Secondly, the number n we are currently dealing with is small. We will show in section 6 that a reasonably small n is good enough to achieve very low false negative rate. For small n, the space taken by the all_path_vector is comparable to that for all the normal n-jump path vectors. For instance, if n=5, the all_path_vector requires 25=32 bits, whereas each 5-jump path vector needs 5 bits. In other words, if there are more than 6 normal 5-jump paths start with the same jump instruction, the all_path_vector will cost less space.

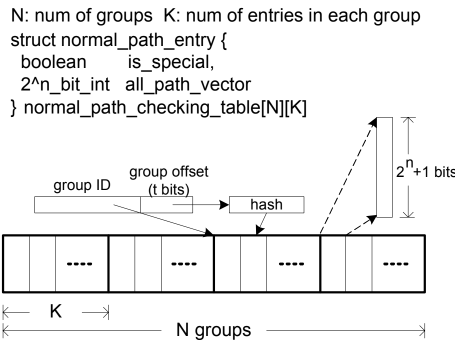

Figure 6. Data structure for regular path checking.

Refer to Figure 6 again. The structure regular_path_entry consists of the all_path_vector and a boolean variable called is_special, which if set indicates that the corresponding jump instruction should be handled by the special path checking stage. Entries in the regular path checking table have to be retrieved using the address of the starting jump instruction. Hash table is the natural way to organize the entries in the regular path checking table. To retrieve a regular_path_entry, we first calculate the hash value of its jump instruction address then use the hash value to index into the hash table. However, there are a couple of problems with a simple hash table design. First, a simple hash table cannot exploit the code locality. The hash table entries for two adjacent jump instructions could be far away. Moreover, the processor always fetches a cache block of data from external memory. Thus, if one uses a simple hash table mechanism, out of multiple regular_path_entry records in one block it is very possible that only one record in the fetched block is touched before its eviction. This is very inefficient. Second, hashing can cause collisions. To avoid fetching the wrong information for a jump instruction, each hash entry has to be tagged which could waste space significantly esp. on an embedded processor. Due to the above reasons, our anomaly detection system abandons a simple hash table design and instead deploys a specially optimized data structure called groupwise hash table.

In Figure 6, the instruction address is divided into two parts: group ID and group offset. Assume there are N groups and each group contains K entries, then the ith group starts at address i*K. Inside each group, the group offset is first hashed then indexed into one of the entries in the group. The advantage of groupwise hashing is: 1) hashing saves space for non-uniformly distributed addresses of jump instructions; 2) hashing is only performed inside each group and each group has a number of sequentially stored entries. In this way, we can exploit spatial locality, because adjacent jumps are most likely located in the same group, their information will be stored close to each other. Some jump addresses may be hashed to the same location causing collisions though. As we observe, with a reasonable hashing function and a proper setup of N and K, collisions rarely happen. In case of a collision, we indicate that the jump should be handled in the special path checking stage, which is done by setting the is_special bit in the corresponding entry of the jump after hashing. Therefore, no tag is necessary to distinguish jumps hashed to the same entry since all the colliding jumps will be handled separately by the special path checking stage. With this customized hash table design, regular path checking table records all normal n-jump paths that are consisted of only conditional branches for non-colliding jumps.

To decide the parameters for the hash table, assume that the group offset is t bit long and the instructions are 4 bytes long, then 2t*¼ instruction addresses are hashed into K slots. The number of jump instructions is about 12% of all instructions, thus the actual number of jump instructions that will be hashed into a group is about 2t*¼*12%. Thus, we can select K to be 1.5*2t*¼*12%, which largely avoids conflicts and the space wastage is below 30%. We can choose a proper t, so that K entries only occupy a few cache blocks, which improves locality among the entries of the same group. After K is decided from t, N should be ⌈M/2t⌉, where M is the code size in bytes.

As discussed above, groupwise hash table not only improves cache performance, but also is an important optimization to reduce the memory requirement of checking tables on resource constraint embedded devices. We also observe an important fact that there is no need to record the whole PC address to identify a branch target. We only need to record the base address of the code section and the offset to the base address to identify a branch target. In our scheme, we assume that the architecture is a 32-bit one and the code size is smaller than 222 bytes (4MB - a large number for most embedded applications). Thus, we only need 22 bits to identify a branch target instead of 32 bits. This is another important technique to reduce sizes of checking tables.

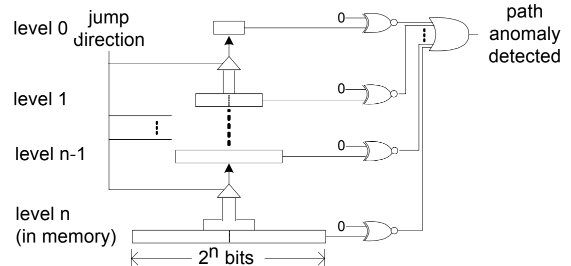

Figure 7. Regular path checking hardware diagram.

The path checking hardware for *regular path checking* is shown in Figure 7. The hardware maintains n bit vectors on chip. The n bit vectors are organized into n levels (level 0 to level n-1). The sizes of the bit vectors vary from 1 bit, 2 bits…to 2^{n-1} bits, where the level k bit vector contains 2^k bits. Level n is assumed to be in the memory and it is the *all_path_vector* of the current jump instruction. Upon reaching a jump instruction, the *all_path_vector*

of the jump instruction is fetched. Then the direction of the jump instruction (only two possibilities in regular path checking, left branch or right branch) is used to select half of the bits (either higher 2^{n-1} bits or lower 2^{n-1} bits) in the *all_path_vector* to be stored in the level n-1 bit vector. Meanwhile, at each level of the bit vectors, half of the bits are selected according to the direction of the current jump and moved one level up, i.e. half of the level k bit vector (either higher 2^{k-1} bits or lower 2^{k-1} bits) is selected and moved to the k-1 level bit vector, where $k \in [1,n-1]$. Finally, if any bit vector becomes 0, a path anomaly is detected. A 0 bit in a bit vector means the corresponding n-jump path was never recorded in the normal runs during training. All bits in a bit vector being 0 means no path from this point is normal thus there is an anomaly. Thus, our hardware implementation of n-jump path checking _detects the anomaly as soon as possible_ rather than waits for another n-1 jump instructions following the current jump and builds the n-jump path then checks whether the n-jump path is seen during training. For example, if the selected half bits of the *all_path_vector* of the current jump are all 0, then an anomaly is detected at once since the dynamic n-jump path starting from the current jump goes to an anomalous direction at the first step.

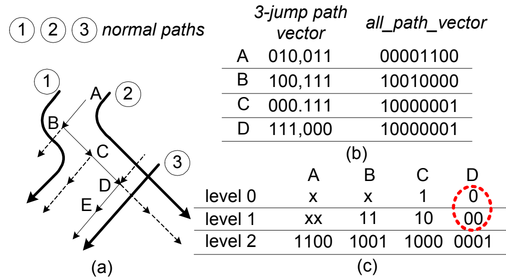

Figure 8. An example for regular path checking: checking anomalous path A→B→C→D→E.

To explain the above scheme further, we give an example in Figure 8. There are three normal paths marked as thick lines. The actual execution follows A→B→C→D→E. Figure 8.b shows 3-jump path vector and all_path_vector for each jump instruction. As in Figure 5, left branch is marked as zero and right branch is marked as one. The most significant bit is the direction of the starting jump instruction. Figure 8.c shows the contents of the on-chip bit vectors after reaching each jump instruction. After instruction A is reached and we know its direction to be zero, i.e. left branch, the lower 4 bits of A's all_path_vector are loaded into the level-2 bit vector. Since A goes to the left branch, no 3-jump path vectors going towards the right branch are needed. At B, the bit vector in level two is moved to level 1 and further pruned based on the direction B takes, i.e. the upper 2 bits are moved to the level-1 bit vector. Upon reaching D, its left branch is taken, however both level 0 and level-1 bit vectors are 0 indicating path anomaly. This is so since no normal 3-jump path starting from B follows the actual execution path B→C→D→E (causing level 0 to be 0), and no normal 2-jump path starting from C follows the actual execution path C→D→E (causing the level 1 bit vector to be 0).

Special Path Checking

When the *regular_path_entry* loaded during regular path checking has _is_special_ bit set (which indicates that the corresponding jump instruction leads to a collision during hashing), the n-jump path is sent to the *special path checking* stage. Also, an indirect jump instruction can cause all the current pending n-jump paths in regular path checking stage be sent to special path checking stage including the n-jump path starting from the indirect jump instruction. The data structure in the special path checking stage is a hash table called *special path checking table*. It hashes the address of the starting jump instruction to index into the table and retrieves all the normal n-jump paths for the jump. The hash table is tagged and collisions are handled by a linked list. Special path checking table records all normal n-jump paths for colliding jumps and normal n-jump paths containing indirect jumps for non-colliding jumps. To support special path checking, the starting jump instruction corresponding to each level in regular path checking hardware is recorded and updated when the vector in the level is updated. In addition, when special path checking stage is not idling, the direction of conditional branches and the target address of indirect jumps have to be sent to it. The former is used to check the conditional branches in an n-jump path and the latter is used to check indirect jumps.

Since an indirect jump could have multiple targets, we use the target address to identify the direction of an indirect jump (as against 0/1 to specify the direction of a conditional branch with true or false outcome). The target of an indirect jump along a program path is collected during training. With the presence of indirect jumps, an n-jump path vector cannot be an n-bit vector any more. The actually length of n-jump vector depends on the number of indirect jumps in the path. In addition, we have to record the positions of indirect jump target addresses in an n-jump path vector for proper handling.

The irregularity of data structure complicates special jump checking stage and there is no interesting optimization opportunity. Thus, the latency in special path checking stage is longer. However, in most cases, only a small percentage of n-jump paths reach this stage, thus the performance impact is small. During our experimentation, we find that in most cases less than 20% of n-jump paths reach this stage.

Our staged anomaly detection hardware is carefully designed to satisfy the needs of embedded systems. It achieves minor performance degradation by pipelining the anomaly detection. Most requests are handled in the first and fast stage. Only a small portion will enter the slower stage. Since our design can achieve very good performance without further architectural optimizations, the requirement of expensive hardware resource such as large on-chip buffers could be avoided. In addition, the hash table in normal jump check stage is also particularly designed to eliminate the tags for the hash table, reducing the pressure on physical memory in embedded systems.

5. OTHER CONSIDERATIONS

Function call is just one type of jump instruction; therefore, paths across function boundary are checked in the same manner. DLLs or dynamic libraries are shared by many processes and are loaded on demand. They can be checked in the same way as normal programs as long as checking tables are loaded together with the DLLs. The jump instruction addresses in the checking tables for a DLL should be relative to the beginning of the DLL, so that they are independent to the actual location the DLL is loaded into the program's virtual address space. System calls like *fork* and *exec* family can create copies of the running process or overwrite it completely. A copy of the checking tables can be created if the process has been forked. In addition, new checking tables are

loaded to the memory space if an *exec* family system call is executed.

The *setjmp()/longjmp()* functions are used in exception and error handling. *setjmp()* saves the stack context and other machine state for later recovery by invoking *longjmp()*. Thus, after *longjmp()*, the program resumes as if the *setjmp()* just returned. The correct handling of *longjmp()* is achieved by adopting the same mechanism in our previous work[12]. The basic idea is that the anomaly detector is aware of the execution of *setjmp()* and saves the current state of return address stack upon a *setjmp()*. The RAS is recovered to that particular state when *longjmp*() is called. With path checking, the context saved upon a *setjmp()* should also include the state of path checking hardware, such as the n-1 on-chip bit vectors (Figure 7). After *longjmp()*, the state of path checking hardware is restored accordingly.

6. EVALUATION

The evaluation of an anomaly detection system consists of two aspects: precision (detection capabilities) and performance. In this section, we show that our anomaly detection system achieves both good precision and low performance overhead.

Table 1. Daemon programs and vulnerabilities.

Server Program	*total # of vulnerabilities*	*buffer overflow*	*format string*	*other bugs*
telnetd	2	1	0	1
wu-ftpd	5	2	2	1
xinetd	2	2	0	0
crond	1	1	0	0
syslogd	1	0	1	0
atftpd	1	1	0	0
httpd (CERN)	2	1	0	1
sendmail	4	4	0	0
sshd	2	1	0	1
portmap	1	0	0	1

In our experiments, we choose 10 daemon programs as benchmarks, as listed in Table 1. Those daemon programs are very common in UNIX/LINUX based systems. They have well-known vulnerabilities so that we can evaluate our scheme in a standard way. These programs are important for networked embedded systems since they offer light-weighted implementations of essential system services for networked systems that have small memory footprints. Some of them provide critical system services, such as crond, syslogd etc. We expect their wide presence in future embedded systems due to rapidly increasing popularity of UNIX based embedded operating systems, such as eCos, embedded Linux etc. Others provide essential Internet services, such as sshd, httpd, wu-ftpd etc. and thus they are likely to become integral components of networked embedded systems. We intentionally choose old versions of these programs with well-known vulnerabilities. Those vulnerabilities will be used in false negative measurement. Table 1 also lists type and number of vulnerabilities in each of these daemon programs used. To make our results more convincing, we also provide results for SPEC 2000 integer benchmark programs in a couple of important experiments.

To evaluate precision of our anomaly detection system, we implement it in an open-source IA-32 system emulator Bochs [26] with Linux installed. Good precision means both low false positive rate and low false negative rate. False positive is acknowledged as a very difficult problem in anomaly detection. False positives are generally inevitable for training-based anomaly detection systems. However, in our experiments, we show that training for our scheme is effective and as long as user performs reasonable amount of training, false positive rate is near zero.

To train our daemon programs, we used training scripts that generated commands to exercise the systems as in [4][5]. These scripts generate a random sequence of mostly valid commands, interspersed with some invalid commands. The distribution of these commands along with their parameters is set to mimic the distributions observed under normal operation. We keep exercising the daemon programs using training scripts until no new normal n-jump paths are discovered for a long time. Then we think training is done and n-jump paths learnt during training are all normal n-jump paths of the program. Figure 9 shows convergence in terms of percentage of normal 9-jump path learnt for three relatively big daemon programs (sshd, httpd and sendmail). The figures are plotted against the number of jump instructions executed in the program being learnt. The graph uses a linear scale on Y-axis (percentage learnt) and a logarithmic scale on X-axis (number of jump instructions processed). The curves are smoothed.

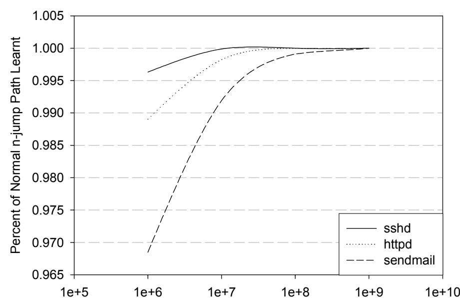

Figure 9. Convergence on selected benchmarks.

Convergence speed is an important measurement since it governs the amount of training time required to achieve a certain level of false positive rate. Slower convergence speed means user has to spend more time to train the system. Figure 9 shows for all of three daemon programs, our detection algorithm converge around after a few hundred million jump instructions are encountered and processed at runtime. No new n-jump paths can be learnt even we train the system for a much longer time. Convergence on smaller daemon programs are not shown since they converge even faster.

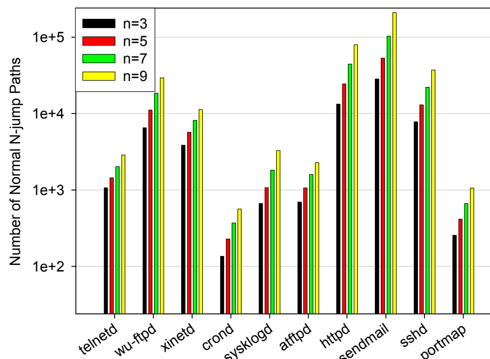

Figure 10. Number of n-jump path.

The above results show that training for our anomaly detection system is very effective. One fundamental reason of the high convergence rate is that although the number of potential n-jump

paths is huge (the number of static branches multiplies 2^n), the real n-jump paths seen during program execution is very limited. Figure 10 shows the absolute numbers of n-jumps paths learnt after convergence. It is clear that most benchmarks have a very small number of n-jump paths even they are run by billions of instructions. The result is not surprising. A lot of work, for example [18], has shown that a significant part of branches are highly biased and many of them are always taken or not-taken during the program execution. Another important reason is a well-known fact that most program execution time is spent on a small number of program hot paths, which can be easily learnt through training.

In [4] and [5], the authors also estimated false positive rate of their systems. They modified the training scripts slightly in terms of commands distribution and other parameters, and then used the modified script to test against their anomaly detection system running in detection mode after training. We tried the same method in our experiments. We found even with modified training scripts, no false positives are observed for all daemon programs after sufficient amount of training (after convergence). We are aware that training scripts cannot capture normal user behavior completely. When applied in real world, it is very likely that our scheme incurs false positives. However, we are confident that the false positive rate would be very low (near zero) and our system does not require excessive human interference.

A good anomaly detection system has to achieve very low false negative rate too, i.e., very high anomaly detection rate. However, it is expected to be futile to measure detection rate of our anomaly detection system against traditional attacks (stack smashing etc.) as in previous work, because those attacks should be easily prevented under our scheme. To prove that, we tested our scheme against our benchmarks with known vulnerabilities listed in Table 1. Most vulnerabilities examined are due to buffer overflows. Other vulnerabilities include format string attacks and unexpected program options etc. We found that our system successfully identified all the attacks because they cause anomalous paths. In addition, we tried the testbed of 20 different buffer overflow attacks developed by John Wilander [20]. The authors claim that the combination covers all practically possible buffer overflow attacks in which attack targets are the return address, the old base pointer, function pointers or long jump buffers. In all cases, the attacks were detected.

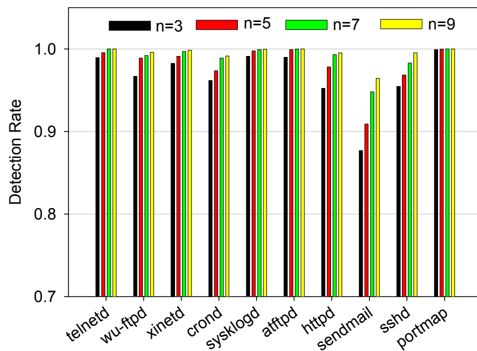

Figure 11. Detection rate of randomly inserted anomalous paths (daemon programs).

To further measure detection capability of our scheme, we choose to perform random fault injection experiments. Fault injection is a classical method for security analysis. A detailed discussion of it can be found in [25]. During the execution of benchmark programs, we randomly pick up branches to inject faults to divert their directions. The total number of faults injected during the execution is huge. For each randomly chosen branch, we divert the branch from its correct direction/target. Such diversion may or may not create an anomalous path. If it does, we check whether our anomaly detection system is able to detect that anomalous path. If our anomaly detection system fails, that is a false negative. Note that each fault is isolated to others. For each fault (each branch direction diversion), we check whether our scheme can detect it. Thus, we achieve a similar effect of running the benchmark numerous times and in each execution injecting only one fault and detecting it using our scheme. Figure 11 shows the results of our random fault injection experiments. The detection capability of our system depends on the value *n* in n-jump path. As illustrated in section 3.5, a larger n leads to higher detection rate. Our results show that when n=3, the average detection rate of randomly inserted anomalous paths is 94.1%; when n=5, the average detection rate is 97.4%; when n=7, the average detection rate is 98.9%; when n=9, the average detection rate is 99.5%. From the above results, our anomaly detection system achieves very high detection rate and thus exhibits very low false negative rate. The fundamental reason is again that the number of normal n-jump paths is very limited, as shown in Figure 10. Thus, the cases illustrated in Figure 2 are rare. The cases are even rarer with larger n. False negatives are due to the reason explained in section 3.5.

To enforce our results further, we also performed the same fault injection experiment on SPEC2000 integer programs. Normal profile for each program is collected using a single standard training input data set. We found that our scheme got very good detection rate again for such complicated programs and with such limited training. For example, with n=9, the average detection rate is 99.3%.

Next, we evaluate the performance aspect of our anomalous path detection system. To measure performance accurately, we collect instruction trace during program execution and feed the instruction trace into a cycle accurate processor simulator Simplescalar [16] targeted to X86 [17]. All hardware modeling is done in Simplescalar. Each benchmark is simulated in a cycle accurate way by 2 billion instructions. The default parameters of our processor model are shown in Table 2. We selected X86 based processor model since x86 compatible embedded processors are gaining a lot of popularity now, such as AMD Elan SC series, Cyrix MediaGX, and National Semiconductor Geode etc. Second important reason for using X86 is that it is the only one for which we could find full system emulator.

Table 2. Default Parameters of the Processor Simulated

Clock frequency	600MHZ	L1 I-cache	32K, 32 way, 1cycle 32B block
Fetch queue	8 entries	L1 D-cache	32K, 32 way, 1cycle 32B block
Decode width	1	L2 Cache	none
Issue width	2	Memory latency	first chunk: 30 cycles, inter chunk: 1 cycles
Commit width	2	Branch predictor	bimod
RUU size	8		
LSQ size	8	TLB miss	30 cycles

There is no performance comparison with software-based approaches since no current software based anomaly detection

system is able to achieve same security strength as our scheme. Moreover, even with weaker detection capability, software based solutions already suffer from huge performance degradation, please refer to [3] for detailed numbers. On the other hand, our design only degrades performance slightly.

Performance degradation under our scheme is mainly due to additional memory accesses to checking tables. Checking tables record program normal behavior and path checking tables are a part of them. Intuitively, larger checking tables lead to higher memory system pressure and larger performance degradation. Figure 12 shows optimized sizes (in bytes) of checking tables for each benchmark with different values of n. Checking table becomes large with a larger n, since there are more n-jump paths to record and information for each n-jump path takes more space. Generally, the size of checking tables is in terms of tens of KB and is small, considering modern embedded systems can have plenty of RAM for user applications. For example, iPAQ hx2755 PocketPC has 128MB RAM installed.

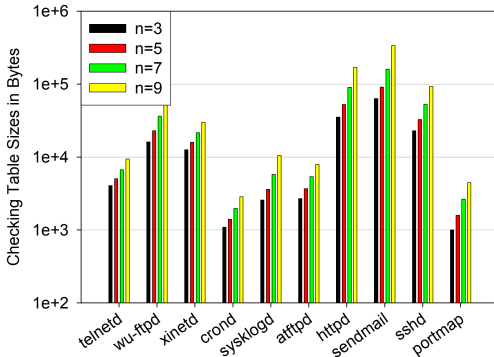

Figure 12. Optimized Checking table sizes.

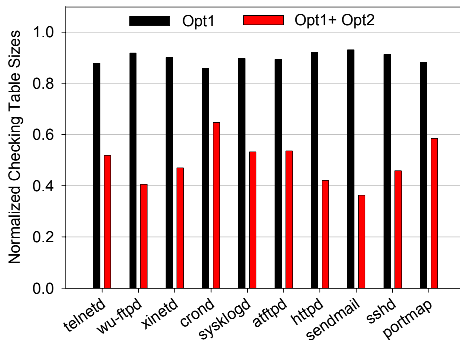

Figure 13. Effects of optimizations to checking table sizes.

Figure 13 shows the effects of our optimizations to reduce checking table sizes, which is particularly important for embedded systems. Opt1 denotes the optimization that uses offsets to represent target addresses. Opt2 denotes the groupwise hash table optimization. Checking table sizes are normalized to the naïve case without any optimization. From the results, by using the offset to the base address of the code section to represent a branch target address, we reduce checking table sizes by 10.1% on average. If both optimizations are enabled, we reduce checking table sizes by 50.7% on average. The results show that our optimizations, especially groupwise hash table, are highly effective to reduce memory requirement of checking tables on embedded systems.

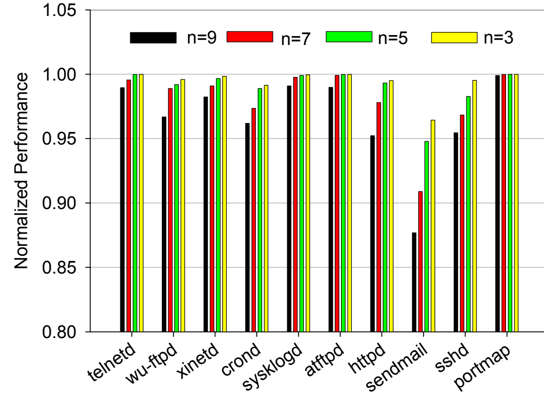

Figure 14. Performance results.

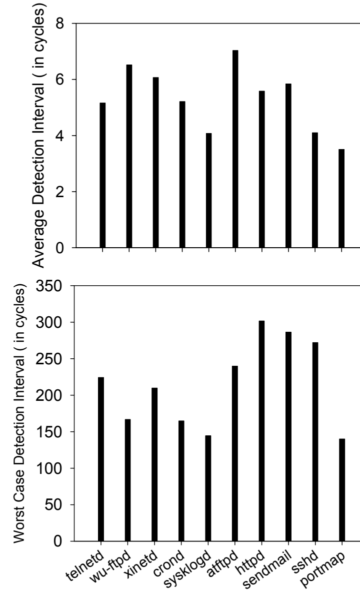

Figure 15. Intervals between fault injection and fault detection when n=9.

Figure 14 shows performance results. Performance degradation under different values of n (the length of paths checked) is shown. The IPCs (instruction per cycle) are normalized to the baseline processor without anomalous path checking. With a larger n, detection strength is stronger, but the size of path checking tables will be larger, leading to degraded cache performance and overall performance. When n=9, the average performance degradation is 3.4%; when n=7, the average performance degradation is 2.0%; when n=5, the average performance degradation is 1.0%; when n=3, the average performance degradation is 0.6%. It is clear that our hardware implementation keeps performance degradation minor even when n is large. Benchmark *sendmail* is a lot larger than other benchmarks used, thus it always has significantly larger performance degradation than the others. We also measured performance degradation for SPEC2000 integer programs. As in fault injection experiments, training is done using standard training input data sets. We found that the performance degradation is also minor for SPEC2000 integer programs. In particular, when n=9, the average performance degradation is only 2.7%.

Finally, real-time attack detection and response is another important advantage of our scheme over software-based schemes. In our fault injection experiments, we also measured the time interval between fault injection and fault detection when n=9. Figure 15 shows the intervals in cycles in the average and worst cases. Cycle numbers are collected in the same way as in performance evaluation. We found that in most cases the fault is detected immediately so the average interval is only several cycles. In worst cases in which we can still detect the fault, the fault is detected only after additional n-1 jumps are executed. The interval in worst cases can reach a couple of hundreds of cycles. Overall, in such short time, no serious damage can be done to the system.

7. CONCLUSION

In this paper, we propose a hardware-based approach to verify the program execution paths of target applications dynamically and to detect anomalous executions on critical embedded systems where near real-time detection is extremely important. With hardware support, our approach offers multiple advantages over software-based solutions including very small performance degradation; much stronger detection capability and zero-latency reaction upon anomaly thus much better security.

The performance degradation is very low mainly due to the efficient design of our staged checking pipeline. A second reason for low degradation is checking for intrusion is done in parallel with the instruction execution causing almost no extra overheads. The detection capabilities of the system offer detection of the attacks at fine granularity and within a few cycles of incidence. Moreover, the detection comes almost free with very low degradation. The hardware structures introduced by our approach are lightweight and through optimizations, we reduce the size of the checking tables by almost 50% and reduce their memory footprint to fit the design philosophy for embedded processors. Overall, we believe that near real time detection capability at fine granularity levels, with low performance and resource overheads should make our solution a practical one.

8. REFERENCES

[1] Allen Householder, Kevin Houle, and Chad Dougherty, “Computer Attack Trends Challenge Internet Security”, *IEEE security and Privacy*, Apr. 2002.

[2] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, “A Sense of Self for Unix Processes,” In Proceedings of the 1996 IEEE Symposium on Security and Privacy, 1996.

[3] D. Wagner, D. Dean, “Intrusion Detection via Static Analysis,” In Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001.

[4] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors,” In Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001.

[5] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, Weibo Gong, “Anomaly Detection Using Call Stack Information,” IEEE Symposium on Security and Privacy, May, 2003.

[6] Henry H. Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, Barton P. Miller, “Formalizing Sensitivity in Static Analysis for Intrusion Detection,” In *Proceedings of* the 2004 IEEE Symposium on Security and Privacy, 2004.

[7] A.Kosoresow, S.Hofmeyr, “Intrusion Detection via System Call Traces,” IEEE Software, vol. 14, pp. 24-42, 1997.

[8] C. Michael, A. Ghosh, “Using Finite Automate to Mine Execution Data for Intrusion Detection: A preliminary Report”, RAID 2000.

[9] Debin Gao, Michael K. Reiter, Dawn Song, “On Gray-Box Program Tracking for Anomaly Detection”, 13th USENIX Security Symposium, pages 103-118, August 2004

[10] Debin Gao, Michael K. Reiter and Dawn Song, “Gray-Box Extraction of Execution Graphs for Anomaly Detection”, the 11th ACM CCS conf., pages 318-329, October 2004.

[11] C. Krügel, D. Mutz, F. Valeur, G. Vigna, “On the Detection of Anomalous System Call Arguments”, In Proceedings of ESORICS 2003, pages 326-343, Norway, 2003.

[12] Tao Zhang, Xiaotong Zhuang, Santosh Pande, Wenke Lee, “Hardware Supported Anomaly Detection: down to the Control Flow Level,” Technical Report GIT-CERCS-04-11.

[13] James R. Larus, “Whole Program Paths,” PLDI 1999.

[14] Y. Zhang and R. Gupta, ”Timestamped Whole Program Path Representation and its Applications,” PLDI 2001.

[15] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” *ASPLOSIX,* Nov. 2000.

[16] Doug Burger and Todd M. Austin. “The SimpleScalar Tool Set Version 2.0”.

[17] Vlaovic, E and S. Davidson, “TAXI: Trace Analysis for X86 Interpretation”, In Proc. Of 2002 IEEE International Conference on Computer Design.

[18] D. Grunwald, D. Lindsay, and B. Zorn. "Static methods in hybrid branch prediction". PACT 1998. Pages: 222 – 229.

[19] R. Jasper, M. Brennan, K. Williamson, B. Currier, D. Zimmerman, “Test Data Generation and Feasible Path Analysis”, ISSTA 1994, pp. 95-107.

[20] J. Wilander and M. Kamkar. “A comparison of publicly available tools for dynamic buffer overflow prevention”. In 10th NDSSS, 2003.

[21] Scut. “Exploiting format string vulnerabilities”. TESO Security Group.

[22] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Point-Guard: Protecting Pointers From Buffer Overflow Vulnerabilities,” Proceedings of 12th USENIX Security Symposium, Washington DC, Aug., 2003.

[23] 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,” 7th USENIX Security Conf., pages 63-78.

[24] G.E. Suh, W. Lee, and S. Devadas, “Secure Program Execution via Dynamic Information Flow Tracking”, ASPLOS 2004.

[25] A.K. Ghosh, T. O'Connor, G. McGraw, “An automated approach for identifying potential vulnerabilities in software”, 1998 IEEE Symposium on Security and Privacy, pp. 104-114.

[26] Bochs: the Open Source IA-32 Emulation Project, http://bochs.sourceforge.net.