# SAM: Security Adaptation Manager

Heather Hinton† Crispin Cowan Lois Delcambre Shawn Bowers

† Ryerson Polytechnic University, Canada Oregon Graduate Institute

E-mail: heather@eecg.utoronto.ca {crispin, lmd, shawn}@cse.ogi.edu

## Abstract

In the trade-offs between security and performance, it seems that security is always the loser. If we allow for adaptive security, we can at least ensure that security and performance are treated somewhat equally. Using adaptive security, we can allow a system to exist in a less secure, more performant state until it comes under attack. We the adapt the system to a more secure, less performant implementation. In this paper, we introduce the Security Adaptation Manager, or SAM. We describe SAM and how we have implemented SAM to take advantage of the different protection strengths offered by the StackGuard compiler. Using SAM to provide StackGuard-based adaptive security provides a form of misuse-based intrusion detection, capable of detecting known and novel attacks.

## 1. Introduction

It seems that the criteria on which software is evaluated is given in order of "importance" as functionality, performance, and finally security. Given that we are not likely to fundamentally change this ordering we turn to to the practice of adaptive security. Adaptive security allows us to implement a system in a high performance, highly functional state for normal use, and then adapt the system to a less performant/less functional/more secure state in the presence of attacks. That is, we adapt the amount of security offered based on the severity of the attack environment in which a system exists.

In this paper, we describe the Security Adaptation Manager (SAM) tool and the adaptive security it provides. We describe the implementation of SAM and its use with the StackGuard compiler [CPM+98, CBD+99] to adaptively protect against buffer-overflow based stack-smashing attacks. The Security Adaptation Manager (SAM) is a front-end adaptation coordinator that monitors unsuccessful stack-smashing attacks. Together with a General Adaptation Space Navigator, SAM implements adaptive security: more or less protection, providing less or more performance in the presence of a more or less hostile environment. The SAM tool allows us to adapt the level of StackGuard protection in a system to correspond to the (perceived) attack environment.

It is not our position that we implement systems with minimal or no protection. However, recognizing that many such systems are already in place, we offer this approach as a means of mitigating the risk of using such a system.

This paper is structured as follows: the remainder of this Section provides a brief introduction and overview of Adaptation Spaces. Section 2 describes buffer-overflow attacks and defenses. In Section 3 we describe the buffer-overflow adaptation space. The implementation of the Security Adaptation Manager (SAM) is described in Section 4. We briefly discuss the (expected) performance results of SAM in Section 5. Section 6 contains a brief discussion of other software-centered pproaches to intrusion detection.

### 1.1. Adaptation Spaces

An adaptation space consists of a condition space and a transition graph: these define when and how we implement adaptations. The condition space is a complete lattice of all possible conditions of interest and their settings. Conditions are "of interest" if they are used to define when to adapt an application. As a simple example, a condition might indicate that a resource has experienced a non-maskable failure. If the system is to continue to offer some (reduced level of) service then it must adapt to this non-maskable failure. In a security context, the conditions of interest correspond to different "attack environments" experienced by the system. Each attack environment represents a class of attacks against the environment and the system.

Given a condition space, there is a corresponding transition graph. The transition graph has one node for

each system configuration (as defined by the condition space) and defines how we can transition between configurations. For the application defined in this paper, it is possible to transition from any one configuration to any other configuration (all possible transitions are allowable). The transition graph therefore represents an implementation space. This defines the system configurations that can survive and thrive under a given set of conditions. Each individual configuration is referred to as an implementation alternative. It is up to the adaptive application designer to define the available implementation alternatives for an adaptation space.

An implementation alternative is specified by a set of conditions. An implementation alternative is considered a feasible choice when its conditions are true. Conditions are defined over event variables that monitor some aspect of the external environment. Note that for any given condition, there may be more than one implementation alternative that is feasible. For example, it is an artifact of the implementation space that when the system is in its optimal state, we may legitimately implement any of the defined alternatives, including that defined for the worst possible state.

In general, if the most desirable implementation alternative is feasible then all lesser implementation alternatives are also feasible. We specify a default preference order, indicating the "desirability" of the implementation alternatives. This is used to indicate the preferred implementation from the set of feasible implementation alternatives.

An adaptation space formalism is used to navigate among combinations of adaptations [CDM+99]. The adaptation space is represented as a partially-ordered graph (a lattice) with a upper and a lower bound. The upper bound corresponds to the ideal situation, where all required resources are available. In a security context, this is the case where no attacks are experienced. The lower bound corresponds to the worst possible situation, where no resources are available. In a security context, this implies that the system is experiencing indefensible attacks. The lower bound represents the least desirable state; conditions that are higher in the lattice are more desirable.

The adaptation and implementation spaces are rich, in that they specify a complete set of resources and their possible configurations. The adaptation space approach allows us to prune the implementation space so that we define an "optimal" set of implementation alternatives for the system. Similar implementations can be coalesced into a single implementation, triggered by multiple conditions.

## 2. Buffer Overflow Defenses

Buffer overflows are one of, if not the, most commonly exploited vulnerability in security-sensitive code. These vulnerabilities are commonly exploited in stack-smashing attacks to gain privileged access to a system. There are two "types" of protection against these attacks. We can protect a system against these attacks, using techniques such as non-executable stacks and stack integrity checks. Alternatively, we can attempt to eliminate or reduce the vulnerabilities that lead to these attacks using techniques such as array bounds checking, memory access checking, type-safe languages and debugging tools.

### 2.1. Protecting the Stack

Non-executable stacks have been implemented with patches to Solaris and Linux [Dik, Des]. These patches make the stack portion of a user process's virtual address space non-executable. Injected attack code cannot therefore be executed. This approach offers a zero performance penalty but requires a specially-patched kernel. These patches are not trivial to implement.

Integrity checks against the stack are also used to protect against buffer-overflow attacks. StackGuard is an example of an integrity check approach. StackGuard is a compiler enhancement to protect programs against stack-smashing attacks [CPM+98, CBD+99] using integrity checks against the stack prior to returning from a function call. StackGuard detects that the return address has been altered before returning from a function (and executing the attacker's injected code). Stack protection is independent of the software executed on a system; protection is provided, regardless of the quality of the code.

### 2.2. Reducing Vulnerabilities

Reducing a system's vulnerability to buffer-overflow attack occurs at the code level. Richard Jones and Paul Kelly have developed a gcc patch [JK95] that does full array bounds checking for C programs. This method prevents all buffer overflow attacks, not just those attempting to alter function activation records. Unfortunately, this method also imposes a substantial performance overhead: a pointer intensive function (ijk matrix multiply) experienced a 30× slowdown. This compiler is also not mature: complex programs such as elm fail to execute when compiled with this compiler.

Purify [HJ92] is a memory debugging tool for C programs. Purify does integrity checking of memory on ev-

ery memory access. The performance penalty imposed by Purify is 2 to 5 times the execution time of optimized code. Purify is more suitable for debugging code than for use with production code. More advanced debugging tools have also been developed, such as fault injection tools [GOM98].

Debugging techniques can only minimize the number of buffer overflow vulnerabilities. They provide no assurance that all buffer overflow vulnerabilities have been eliminated. Thus for high assurance, protective measures such as StackGuard and non-executable stacks should be employed.

### 2.3. StackGuard Protection

The StackGuard compiler provides protection through integrity checks against the function activation record on the stack. StackGuard places a "canary" word next to the return address on the stack (between the return address and the function's local variables). Every function, including main, gets a canary word. When a function returns, it first checks the canary word. If the canary word is still intact (i.e., unchanged), the function will jump to the address pointed to by the return address word.

If the canary word is not intact then the function will invoke the canary_death_handler. This will send a canary-death message to the syslog files and cause the program to (gracefully) terminate. The format of this message is

```
Immunix type ? Canary[?] = ? died with
  cadaver ???????? procedure ???
```

This message reports the number of the canary (the ith canary corresponds to the ith function protected), the value of the canary (reported in hexadecimal format) and the procedure name that contains the corrupted canary [1]

StackGuard protection is not infallible. An attack that proceeds without altering the canary value, either by carefully stepping over the canary word or by including the canary word in the attack string, would fail to be detected.

The initial release of StackGuard protected the canary value by choosing a 32-bit random number as a canary value at program exec() time (a random canary). This makes it intractable for the attacker to guess the canary value [CPM+98]. The StackGuard protection has subsequently been extended to use a "terminator" canary [CBD+99]. It is reasoned that an attacker cannot simultaneously deposit a terminator value (for example, a null character) in the canary's location and move on to alter the return address above the canary. The terminator canary is a 32-bit word comprised of a null byte, a carriage return (0x0D), a line feed (0x0A), and an "EOF" (0xFF in the libc representation). Most string copying functions will halt when they encounter one of these bytes.

The random canary is impervious to all string operations, not just those that terminate on the "usual" termination symbols. The random canary is therefore more secure than the terminator canary. Conversely, the terminator canary is faster than the random canary check because it does not have to look up the current canary value.

A final flavor of StackGuard protection is the "terminator with diversity" canary word(s). With this approach, we add a random number of words after the canary word (and before the function's local variables) on the stack, so that the length of the canary word is not predictable. This StackGuard flavor protects against an attack against the terminator canary implemented using do-while gets to overwrite and rebuild the terminator canary word. While not as secure as the random canary, the terminator with diversity is more secure than the terminator canary implementation.

## 3. StackGuard Adaptation Space

The StackGuard Adaptation Space is defined by the attack environment, that is, the types of buffer-overflow stack-smashing attacks that can be implemented against a system. In the ideal situation, there are no attacks implemented and no special protection is required. In the worst possible situation, it is not possible to defend against the attacks that can be implemented and the only recourse is to take the vulnerable code off-line. The ideal and worst situations bound the adaptation space, which is defined by the (total) ordering shown in Figure 1(a).

Figure 1(b) gives the corresponding implementation space. Note that in Figure 1(b) we do not have an implementation that corresponds to the bounds checking compiler. This is because we have not been able to find a good, working bounds checking compiler for a Linux-based system [2].

The least upper bound of the lattice refers to Unprotected code. We do not recommend, as a rule, that a system contain Unprotected code. However, there may be situations where we can tolerate Unprotected

[1] When placed into the syslog files, Date, Time, Host and Program Name information is prepended to the canary death message. For example, one possible prepended message is:
Apr 22 19:52:06 localhost badnull1:

[2] Purify is Solaris-based and the Jones-Kelly bcc is not mature enough for use.

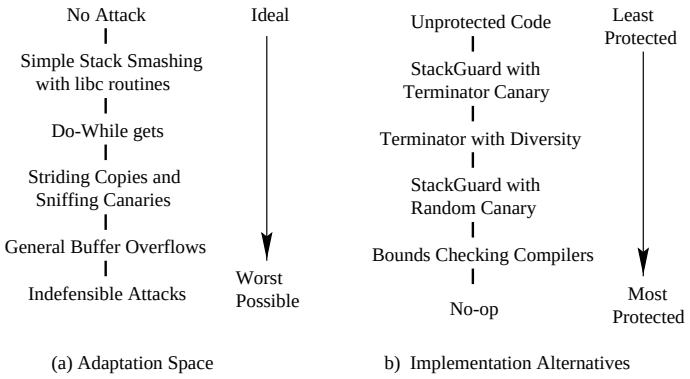


Figure 1. Buffer-Overflow-Based Stack-Smashing Attack Adaptation Space

code, including: we cannot protect the code because we do not have access to the source code and cannot recompile it with the StackGuard compiler; we have a sufficiently good guarantee that the code cannot be accessed (and therefore attacked); the code has been formally verified and proven to be immune to any and all attacks.

The Terminator, Terminator with Diversity, and Random implementations are all flavors of StackGuard-protected code and refer to the type of canary that is used to protect the function pointer. Each flavor protects against the types of attack defined in the corresponding adaptation configuration (Figure 1(a)).

The greatest lower bound of the lattice refers to No-op code. In this situation we are in the presence of an omnipotent and omnipresent attacker and we cannot, by definition, protect the code from attack. This means that our only recourse is to take the affected code off-line and "No-op" it. Because No-op'ing code is a serious action, we do not recommend that entire classes of programs be No-op'ed. Once a vulnerable program is discovered, that program and only that program should be taken off-line [3].

As with a properly defined implementation space, protecting our system at any of the implementation levels defined in Figure 1(b) will provide protection against attacks at that level and all attacks that are higher in the lattice. This means that, for example, the protection provided against general buffer-overflow attacks also implies protection against stack-smashing attacks using libc.

[3]This is consistent with real-world practice. For example, CERT will often issue a warning with the advice that a certain piece of code be taken off-line until a work-around can be discovered and put in place.

### 3.1. StackGuard Monitoring

The SAM tool can be configured to monitor any type of event that is deemed relevant to the system. We provide two types of (StackGuard-relevant) events. The first type of event is the set of death messages reported to the syslog by StackGuard-protected code. We refer to these as monitored events. The second type of event is the set of manual triggers, toggled by the user.

Monitored events are used by SAM to automatically adapt the level of protection within the system. Manual events allow a security administrator to force the adaptation to a more secure implementation than is indicated by the monitored events. This would allow the SA to move to a more secure state as a precautionary measure, for example, after reading a CERT advisory. Note that we cannot use manual events to force an implementation that is less secure than the one specified, and required, by the monitored events.

### 3.2. StackGuard Monitored Event Variables

SAM monitors the canary death messages reported in the syslog files. Each syslog message contains syslog-generated information (describing the date, time, and "originator" of the message) and StackGuard-generated information (the canary death message). The data contained in this message is parsed and separated into Month, Date, Time, HostName, ProgramName, CanaryType, CanaryValue, and AttackedProcedure fields.

The Month, Date, and Time information is used to determine the frequency of unsuccessful attacks. The CanaryType and CanaryValue fields are used to verify which StackGuard implementation is currently in place. Although not currently used by SAM, the AttackedProcedure field can be used as part of an off-line analysis of the vulnerabilities that are being exploited by the unsuccessful attacks.

The HostName and ProgramName are used to identify the sensitivity of the host and program under attack. We consider three classes of host sensitivity: security server, other server and workstation. A security server is one that is critical to the security function of the network. An example of a security server is a Kerberos authentication or a RADIUS server. An example of a non-security, other, server is a file server or a compute server. We respond to attacks against these classes of servers differently: we take a much more conservative view when considering attacks against a security server.

We classify programs as having nobody, user, or root sensitivity. Attacks against root-level programs are the most serious (have greatest associated risks). Attacks against nobody-level programs are judged to be the least serious. We tolerate fewer unsuccessful attacks against root-level programs than against user-level or nobody-level programs.

### 3.3. SAM Conditions

The SAM adaptation conditions are given by the following tuple:

(pgm_host_attack_grid, manual_selector)

The manual_selector event variable represents the user-specified implementation alternative.

The pgm_host_attack grid defines the (monitored) conditions for moving to a more or less secure implementation based on the class of program and host that is being (unsuccessfully) attacked. A typical grid is shown below in Table 1. The value in each corresponds to the number of (unsuccessful) attacks that are observed before we move to a more secure implementation. Once the conditions specified by any cell in this grid are satisfied, we must adapt the system. In Table 1, for example, if there is more than one unsuccessful attack on (any) root-level program on a security server, then we will upgrade the system to a more secure implementation.

Originally, we had defined adaptability based on the percentage of possible programs and hosts within a class being under attack. This approach offered flexibility because we cannot know the numbers of different types of servers within an unknown system. For a given system, defining a trigger based on 1% of these servers has the same effect as specifying one server. Defining a trigger based on 50% of the workstations in a system, however, is easier to understand (and justify) than requiring 13 workstations be under attack.

| | Security | Other | Workstation |
|---|---|---|---|
| Root | 1 | 5 | 10 |
| User | 5 | 10 | 15 |
| Nobody | 10 | 20 | 25 |

Table 1: Pgm_Host_Attack Grid

We have found that it is just as easy to base our decisions on a "raw" number of perceived attacks against a given class. While this might result in a more sensitive set of adaptability conditions, this is not seen as a detriment. Indeed, this approach is more flexible in that it does not allow an attacker to hide their actions by moving from host to host.

### 3.4. Protection Postures

The possible implementation space that SAM needs to consider is quite large. This follows given that we monitor three classes of programs (root, user, nobody), on three classes of hosts (security server, other server, workstation), with five possible security levels (Unprotected, Terminator, Terminator with Diversity, Random, No-op). This leaves us with $9^5$ possible implementation alternatives to consider.

We can immediately prune this space by noting that two of the five possible security states are not recommended for entire classes of programs or hosts (Unprotected and No-op). Taking an entire class of programs or an entire class of hosts off-line, or leaving them unprotected, is a drastic action. We postulate that the number of Unprotected and No-op'ed programs will be small and can be represented by N. Based on these assumptions, we can prune our implementation space to $9^3 + N$. Even this space, however, is too large to be practical.

We therefore consider pre-configured states, or Protection Postures. A protection posture is a predefined system implementation, where each program-host class may have different implementation alternatives. Protection postures allow us to prune the implementation space by pre-configuring a small number of postures that are the most effective system implementations. We define three postures, called Calm, Nervous, Panic in order of preference.

When a system has adopted a Calm protection posture, we trade-off protection for performance as shown in Table 2. The most critical code (root-level permission on a security server) is always implemented at the highest (most-protected) protection level, even in the Calm state. The remainder of the code is protected at the least StackGuard-protected level (note that this differs from the least protected level, which is Unprotected).

| | Security | Other | Workstation |
|---|---|---|---|
| Root | Random | Terminator | Terminator |
| User | Terminator | Terminator | Terminator |
| Nobody | Terminator | Terminator | Terminator |

Table 2: Calm Protection Posture

Once we believe that a system is under attack, we move up to the Nervous protection posture. This posture offers protection against a broader range of attacks than the Calm posture. We protect at the highest StackGuard level (with the Random canary) the root-permission code on the security and other servers and the user-permission code on the security server. The

remainder of the code is upgraded to the Terminator with Diversity level of StackGuard protection.

| | Security | Other | Workstation |
|---|---|---|---|
| Root | Random | Random | TermDiv |
| User | Random | TermDiv | TermDiv |
| Nobody | TermDiv | TermDiv | TermDiv |

Table 3: Nervous Protection Posture

If a system is heavily attacked, we move into the Panic state. In this state we implement the strongest security we have by taking all code to the Random canary level of StackGuard protection. We also take any root-level programs that are under attack into the No-op state.

| | Security | Other | Workstation |
|---|---|---|---|
| Root | Rand/NoOp | Rand/NoOp | Rand/NoOp |
| User | Rand | Rand | Rand |
| Nobody | Rand | Rand | Rand |

Table 4: Panicked Protection Posture

A protection posture is implemented when a corresponding set of conditions, defined by a pgm_host_attack grid, is true. Table 5, below, defines the pgm_host_attack grids corresponding to the protection postures defined above [4]. If any of the conditions in a grid are true, then we must adapt the system to a more secure implementation, where all conditions are true. If all of the conditions of the grid are false, then we may restore the system to the less secure implementation (for which the conditions are all true).

| | Security | Other | Workstation |
|---|---|---|---|
| Root | 1 | 5 | 15 |
| User | 5 | 10 | 20 |
| Nobody | 5 | 15 | 25 |

Table 5(a): Monitored Event Conditions for Calm Posture

| | Security | Other | Workstation |
|---|---|---|---|
| Root | 2 | 10 | 20 |
| User | 10 | 15 | 30 |
| Nobody | 10 | 20 | 40 |

Table 5(b): Monitored Event Conditions for Nervous Posture

According to Table 5(a), once we have seen one attack against a root-level program on a security server, we can no longer remain calm: we must move into the nervous posture.

[4] The values reported in this table correspond to the test values used with SAM.

| | Security | Other | Workstation |
|---|---|---|---|
| Root | 3 | 15 | 25 |
| User | 15 | 20 | 40 |
| Nobody | 15 | 25 | 50 |

Table 5(c): Monitored Event Conditions for Panicked Posture

The protection postures and pgm_host_attack grids are used together to determine the implementation alternatives for a given system given its (attack) environment. For example, let the monitored events indicate that there have been 13 attack attempts against user programs on other servers. This indicates that there is at least one condition required by the Calm posture that is false: we cannot continue with the Calm implementation. From Table 5(b) and (c), we see that this monitored event does not violate the conditions of the Nervous or Panicked postures. Either posture is allowable. However, we prefer to be in the Nervous posture over the Panicked posture for performance reasons, and so the preference ordering will indicate that the Nervous posture is the preferred implementation.

## 4. Implementation

The Security Adaptation Monitoring tool is implemented in Java and Perl (using JDK 1.1 on a StackGuard-protected Linux distribution). SAM has been tested on a stand-alone system using simulated test data (based on real attack patterns observed at both Ryerson Polytechnic University and the Oregon Graduate Institute). SAM successfully adapted the system: simple attacks that were possible (but difficult to implement and therefore gave rise to auditable failure notices) became impossible as the system adapted to a more secure state. SAM also successfully transitioned the system to a more performant state in the absence of any failed attack attempts.

SAM takes as an input a file generated by the Swatch tool [Atk]. Swatch is a Perl utility used to filter all non-SAM relevant messages from syslog. Currently, only canary death messages are considered SAM-relevant. SAM also receives user inputs from a user interface. SAM parses the relevant event variables from these inputs and passes the information to the Generic Adaptation Space Navigator.

The Generic Adaptation Space Navigator is a (Java-based) tool that enables software applications to adapt to changing environments [Bow99]. The Navigator takes as an input an XML [XML] file specifying an adaptation space. The Navigator also takes as input the values of the monitored event variables, and pro-

duces as an output the path to the preferred implementation for the given environment.

Using the preferred implementation returned by the Navigator, SAM must "replace" the existing implementation with the preferred implementation. This is accomplished using a Perl script that is triggered by the Java component of SAM. We maintain three complete "systems", one for each protection posture. That is, we prepend to all directory paths the protection posture that is currently implemented. This means that below the root directory, we have three sub-directories, /calm, /nerv, /panic, as shown in Figure 2.

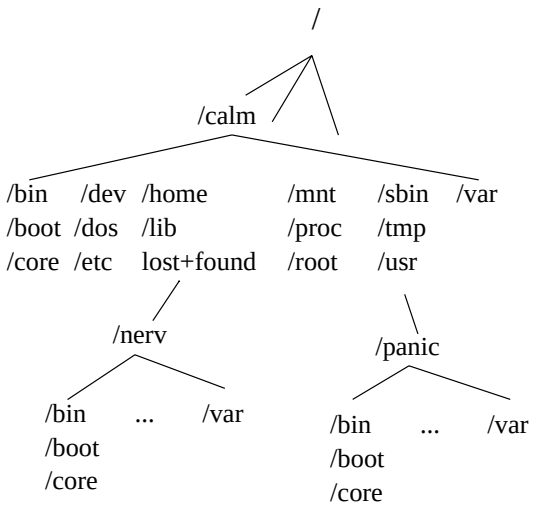


Figure 2. Implementing the Protection Postures

For example, we will have three versions of /etc: /calm/etc, /nerv/etc, /panic/etc. In the /calm directory, all root-level code on a security server will protected with a Random canary, while the remaining code on the remaining servers is protected at the Terminator canary level.

This implementation implies that the details of the protection postures are locally configurable. While we cannot change the fact that there are three postures, we are free to implement these postures as most effective and efficient for a given system.

### 4.1. When to Adapt

When to adapt is based on the number of programs at a host that are under attack. We have stated, for example, that if there has been an unsuccessful attack against a root-level program at a non-security (other) server, then we cannot stay in the calm posture and must adapt to a more secure implementation alternative. Do we consider attacks over all time or over the last x seconds?

If we were to consider attacks over all time, it is trivial to argue that once the system adapts into the most secure implementation, it can never return to a less secure/more performant implementation. If we consider attacks over some period of time, how do we define this period? This is not a trivial problem.

If we pick a period that is too short, we may not "detect" attacks that should cause us to adapt to a more secure state (a technique known in the hacker literature as a "slow scan"). If we pick a period that is too long, it will be very difficult to "recover" from past attacks. Simply by picking a value, we open ourselves up to a denial-of-service attack: all the attacker must do is ensure that the minimum number of attacks are recognized within this time period.

Associated with this problem is the problem of recovering to a more performant state after the number of detected attacks drops off. Do we use the same waiting period? Do we place stricter requirements on this "direction" of adaptation and use a longer waiting period? If we pick a waiting period that is too short, we can easily set ourselves up for a concerted attack against the system in its weaker state. We also run the risk of thrashing the system, that is, wasting cycles adapting between two alternatives, to the point that there are no cycles available for compute jobs. If we pick a period that is too long, then we are not taking full advantage of the adaptive approach to maximize performance.

We recommend that the downgrade window be set at least 10 times as long as the downgrade window. This value is based solely on our observations of the timing of attacks; it will more than likely need to be fine tuned for each individual system.

Because of the difficulty in picking these adapt and wait times, we do not hard-wire this into SAM. Instead, we allow security administrators to pick these values when configuring SAM for their local systems. We have tested SAM (and set up as default values, easily changed within SAM) using an upgrade window of 60 seconds (measured in milliseconds), and a downgrade window of 3600 seconds. This means that an attacker can attempt at most one attack an hour against the least-protected system.

### 4.2. Security of SAM

One, very good, means of attacking a system is to attack the system's defense mechanisms. We must consider the security of StackGuard and the Security Adaptation Manager. In order to subvert the StackGuard mechanism, we must recompile code with a non-StackGuard compiler. This implies that even if SAM

is subverted, the protection offered by StackGuard will continue to be in place.

SAM relies on the outputs of the syslog files, filtered by the PERL utility swatch. If we can subvert either swatch, the swatch configuration files, or the output produced by swatch, we can fool SAM into thinking that the system is not under attack. SAM will therefore not adapt the system to a more secure or more performant implementation. This implies that we must protect all aspects of swatch.

An easier way to subvert SAM is to replace the various StackGuard-compiled programs with non-StackGuard compiled programs. When SAM adapts the system, it will then (unknowingly) adapt less secure code instead of more secure code. This means that we must tightly control write access to the locations where StackGuard compiled code resides.

Another way to attack SAM is to inject false canary death records to the syslog files. This will give the appearance of a system heavily under attack and will cause SAM to upgrade the system to a low performance/high security implementation. This is a form of denial-of-service attack against the system.

## 5. Performance Results

We have run a preliminary performance analysis on a 133 MHz Pentium with 16M of RAM, 16K cache memory and a 256K level 2 cache. The SAM components are implemented in Java and Perl scripts. Because SAM is a continuous process (always running) it will affect the performance (elapsed time, or throughput) for all processes. Using a better Java compiler would help this aspect of performance. Nevertheless, we do not believe that this is a serious drawback as SAM should not be running on a "general user" machine. SAM is a priviliged, security-relevant program. As such, it should be run on a bastion host.

We do not yet have performance results for the tool as used in a production-type environment (i.e., with a more powerful processor) or a "true" attack environment, when we must implement attack-based transitions. All testing of SAM has been done using simulated attack data. A preliminary performance justification follows.

Performance analyses of the StackGuard flavors tell us that the Terminator canary is the most performant, followed by the Terminator with Diversity, and the Random canaries. We also know that the performance overhead imposed by StackGuard'ed code is not significant [CBD+99]. This leads us to postulate that the major contributor to performance slowdown with SAM will be the adaptation time. The adaptation time is the time it takes to re-configure the system to reflect the new SAM-specified protection posture.

As the majority of the performance penalty is that incurred by the creating of symbolic links, the performance hit incurred is negligible. Implementing SAM-specified re-configurations on a heavily loaded laptop had no noticeable effect on other jobs.

One ramification of using symbolic links for implementation transitions is that any processes that are currently running will not be re-configured, mid-execution, to a {more, less} secure state. Instead, they will finished executing in their current state. The next time the process is invoked, it will be invoked in the "new" configuration, corresponding to the current implementation.

We have implemented the transition mechanism with the assumption that processes are short-lived and start-on-demand. There are however, several persistent daemons, including nfsd, sendmail, and inetd (which handles the start on demand for all other daemons). Future versions of SAM must develop a solution that allows these persistent daemons to be killed and re-started automatically. Until then, these daemons will need to be re-started manually when the system is transitioned to a more secure state.

## 6. Discussion

Ghosh et al proposed a software level approach to intrusion detection [GWC98]. They concentrate on the software level because "attacks against computer systems are in fact attacks against specific software programs". Although their approach works best with software that has been modified to provide internal state information, they do allow for intrusion detection based solely on the observable external states of the program. Thus they are not limited to applying this approach to programs for which the source code is available.

Forrest et al. have also proposed a system process level approach [HFS98] to misuse detection. Misuses are those sequences that deviate from those occurring empirically in traces of known normal behavior. This approach requires a modified kernel to trap system calls.

The StackGuard based approach does not require the modification of programs or the kernel. It does, however, require that we have access to the source code, so that we can re-compile it with the StackGuard compiler. We hope that the benefits of a tool such as the Security Adaptation Manager as used with StackGuard will convince more people to release open source.

Together, StackGuard and SAM provide a form of misuse detection, where the misuse pattern is the ca-

nary death message reported by a failed StackGuard-compiled program. StackGuard (and therefore SAM) are different from misuse detection approaches in that it is able to "recognize" novel attacks. For example, StackGuard was developed before the recent attack against a particular edition of lsof was known/published [Zbo99]. Where other misuse detection systems may not be able to detect this type of misuse, StackGuard had no trouble in preventing lsof-based stack-smashing attacks. This implies that StackGuard and SAM will be able to detect new, previously unknown buffer-overflow based stack-smashing attacks, and adapt the system correspondingly.

## 7. Conclusions and Future Work

Although not a "classic" intrusion detection mechanism, the Security Adaptation Manager (SAM) does provide detection of and response to possible attacks. SAM monitors audit logs to detect unsuccessful attacks and to provide an adaptive response. Based on the attack characteristics, SAM will adapt the system to more or less secure states, optimizing the performance of the system in the presence of attacks. SAM also allows the user to force the adaptation to a more secure state than is required by the attack characteristics. This allows security administrators to take pre-emptive steps in the face of (e.g.) CERT advisories.

In future work, SAM can be generalized to arbitrary intrusion detection sources and arbitrary intrusion response adaptations. Because of the generality of the use conditions in an adaptation space, any form of intrusion detection is suitable for driving SAM. Similarly, any portion of the system's security posture that is dynamically configurable can be driven by an intrusion detection-response policy encoded in SAM. For example, SAM can be easily configured to provide adaptability to firewalls and firewall rules. The sample policies presented here only scratch the surface of possibilities.

## References


[Atk] Todd Atkins. The simple watcher and filer. available from ftp://ftp.stanford.edu/general/security-tools/swatch or from ftp://ftp.redhat.com/pub/contrib/SRPMS/swatch-2.2-2.src.rpm.

[Bow99] Shawn Bowers. The general adaptation space navigator, April 1999. Heterodyne Project.

[CBD+99] Crispin Cowan, S. Beattie, R. Day, C. Pu, P. Wagle, and E. Walthinsen. Protecting systems from stack smashing attacks with stackguard, 1999. to appear at Linux Expo 1999, http://www.bitmover.com/ linux-expo/ papers.html.

[CDM+99] Crispin Cowan, L. Delcambre, A. Le Meur, L. Liu, D. Maier, D. McNamee, M. Miller, C. Pu, P. Wagle, and J. Walpole. Adaptation space: Surviving non-maskable failures, 1999. Heterodyne Project.

[CPM+98] Crispin Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wangle, and Q. Zhang. Stackguard: Automatics adaptive detection and prevention of buffer-overflow attacks. In Proceedings 1998 Usenix Security Conference, January 1998. San Antonio, Texas.

[Des] Solar Designer. Non-executable user stack. http://www.false.com/security/linux-stack/.

[Dik] Casper Dik. Non-executable stack for solaris. Posting to comp.security.unix January 2 1997.

[GOM98] Anup K Ghosh, Tom O'Conner, and Gary McGraw. An automated approach for identifying potential vulnerabilities in software. In Proceedings of the IEEE Symposium on Security and Privacy, pages 104–114, May 1998.

[GWC98] Anup K. Ghosh, James Wanken, and Frank Charron. Detecting anomalous and unkown intrusions against programs. In Proceedings of Fourteenth ACSAC, 1998.

[HFS98] S.A. Hofmeyr, S. Forrest, and A. Somayajii. Intrusion detection using sequences of system calls. Journal of Computer Security, 6:151–180, 1998.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In Proceedings of the Winter USENIX Conference, 1992. Also available at http://www.rational.com/support/techpapers/fast_detection/.

[JK95] Richard Jones and Paul Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html, July 1995.

[XML] XML. World wide web consortium extensible markup language (xml). http://www.w3.org/XML/.

[Zbo99] Anthony C. Zboralski. [HERT] Advisory #002 Buffer overflow in lsof. Bugtraq mailing list, http://geek-girl.com/bugtraq/, February 18 1999.