EFFICIENT TAINT ANALYSIS USING MULTICORE MACHINES

by

Mr. Milind Mohan Chabbi

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

2007

STATEMENT BY AUTHOR

SIGNED: ________________________________

APPROVED BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

________________________ ________________________

Dr. Gregory R. Andrews Date

Professor

ACKNOWLEDGEMENTS

I would like to convey my gratitude to my thesis advisors Dr. Gregory Andrews and Dr. Saumya Debray for their continuous support and invaluable advice during the past one and half years. They have always appreciated my work and set goals for me that are of very high standards but at the same time realistic and reachable with constant effort. I would also like to thank them for excellent compiler and concurrent programming theory they have taught me in their classes.

I appreciate willingness of Dr. John Hartman and Dr. Chris Gniady to serve on my thesis committee, and to make time in their busy schedules. I am also very thankful to Dr. John Hartman for the excellent operating system background that he setup in his class.

I thank all members of SOLAR research group for their encouragement and support. My special thanks to Somasundaram Perianayagam and Haifeng He for helping me with implementation details.

Last but not the least I would like to thank my father Mr. Mohan Chabbi, mother Mrs. Shaila Chabbi and brother Mr. Mayur Chabbi for their constant support in every endeavors of my life. They have always stood besides me in my good and bad times, provided moral and financial support and helped me grow in every respect.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Time and again data privacy and computer security are severely threatened by software vulnerabilities. With more and more computing devices getting network connectivity, the exploitation of vulnerable programs by remote users has become a ubiquitous issue. While tremendous effort is carried out to counter software exploitation, no single approach developed so far has been able to satisfactorily provide solid security along with efficient performance. Taint analysis, which is one of the approaches to track information flow to counter program exploits, has shown a promise. In this thesis work, we propose a novel implementation of fine-grained dynamic vulnerability detection by parallelizing the actual computation and taint computation and tapping the power of idle cores on multicore machines to mitigate performance overheads. We propose a paradigm of secure and efficient computing using many cores and use binary rewriting to empower a program with parallelized taint monitoring capability. The challenge lies in minimizing the thread synchronization. We demonstrate the effectiveness of our approach in protecting against various attacks while offering an order of magnitude performance improvement compared to state of the art approaches.

CHAPTER 1

1 INTRODUCTION

1.1 Background

For a layman software user or a system administrator, the security advisory news from their trusted software vendor is often painful. In recent times it has become the norm to get such security advisories on a monthly basis. Most commonly used software, including a seemly innocuous text reader like Acrobat, is known to have been compromised [1]. According to statistics from Carnegie Mellon University's CERT (Computer Emergency Response Team), the number of reported vulnerabilities in software has increased nearly 500% in two years (1999- 2001) [2] as shown in Figure 1. Hence it is imperative to find a trustable and efficient solution to this problem.

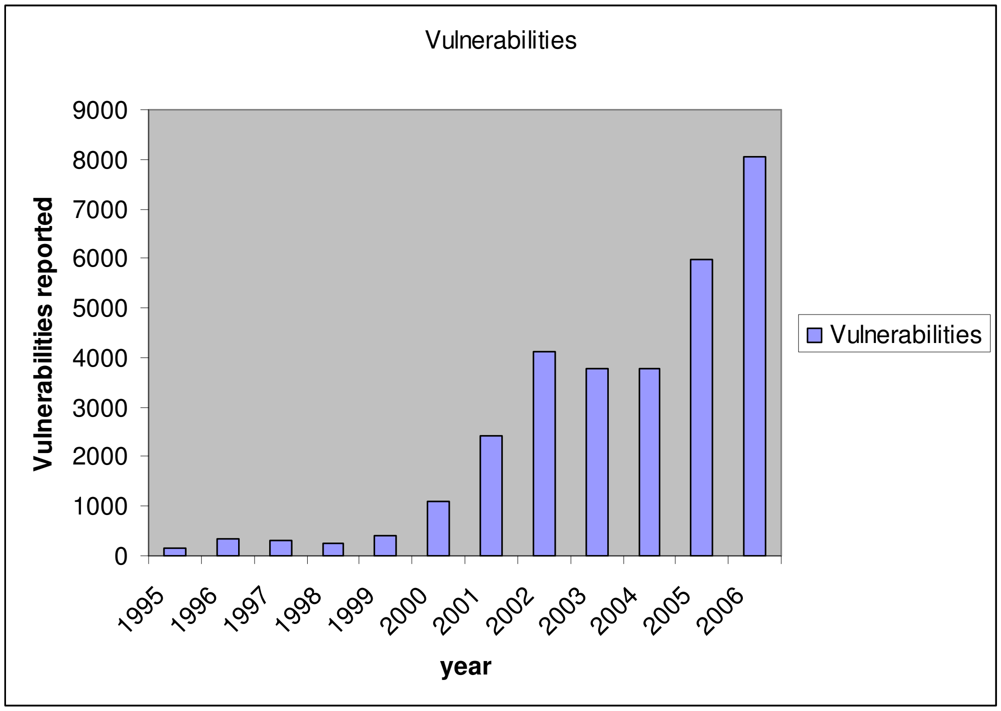


Figure 1: Increase of vulnerabilities in the last decade

1.1.1 What is a vulnerable program?

A vulnerable program is the one that has programming defect in it that can be exploited by an external entity or a malicious user. A vulnerable program is different from a virus or a malicious program in that the program by itself is not compromised. Program vulnerabilities arise at various stages of software development from requirement analysis, design, and coding to maintenance and upgrades. Because of the complexity of software, complete elimination of program vulnerabilities is nearly impossible.

1.1.2 Classification of vulnerable programs

While violation of security policies that would constitute an intrusion can happen in various ways [15], over 50% of the vulnerabilities arise from buffer overflows and another 40% arise from input validation error [15]. A common characteristic of all successful attacks is the ability to change the flow of control, which lets the attacker execute arbitrary code. Buffer overflow vulnerability arises because of the failure to properly check the length of data against the size of a data storage object. Buffer overflow vulnerability can be classified as stack-based or heap-based depending on the location of the vulnerability. Input validation error happens because of the failure to verify the contents of user-given data. Due to input validation error a malicious user-input can be passed to sensitive functions or system calls thereby opening a back door for an attacker. Below we discuss each one of them in detail.

1.1.2.1 Stack-based vulnerability

In a stack-based attack, user-given data is read into a local buffer allocated on the stack. If too much data is written into the buffer, it can overwrite the return address. Well-crafted data can overwrite the function return address with the address of injected code and divert execution to malicious code on function return.

A common buffer overflow attack is shown in Figure 2. A local buffer allocated on the stack is overwritten with ‘A’s and eventually the return address is overwritten, in this case with the return address 0xbdfec0e0. Similar techniques can be used to divert program execution by overwriting the old base pointer, a function pointer present as a local variable, a function pointer present as a parameter, a long jump buffer present as a local variable, and a long jump buffer present as a function parameter.

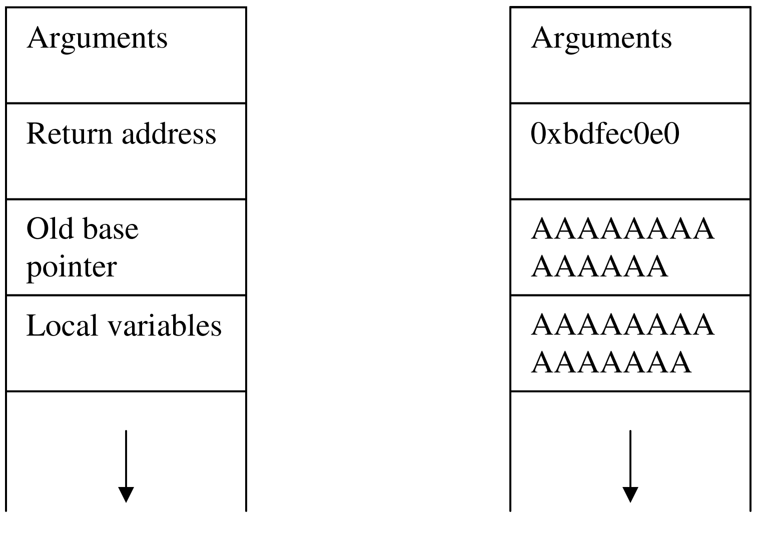


Figure 2: Stack-based vulnerability

1.1.2.2 Heap-based vulnerability

In a heap-based attack, user given data is read into a local buffer allocated on the heap/BSS (Block Started Symbol)/data section. If too much data is written into the buffer without proper bounds check, it can possibly overwrite some function pointer present in the heap/BSS/data section. A well-crafted attack can overwrite a function pointer with the address of injected code and divert execution to malicious code.

A common heap buffer overflow attack is shown in Figure 3. A buffer allocated on the heap is overwritten with 'A's and eventually a function pointer is overwritten, in this case with the address 0xbfef00e0. Similar techniques can be used to divert program execution by overwriting a long jump buffer present on the heap.

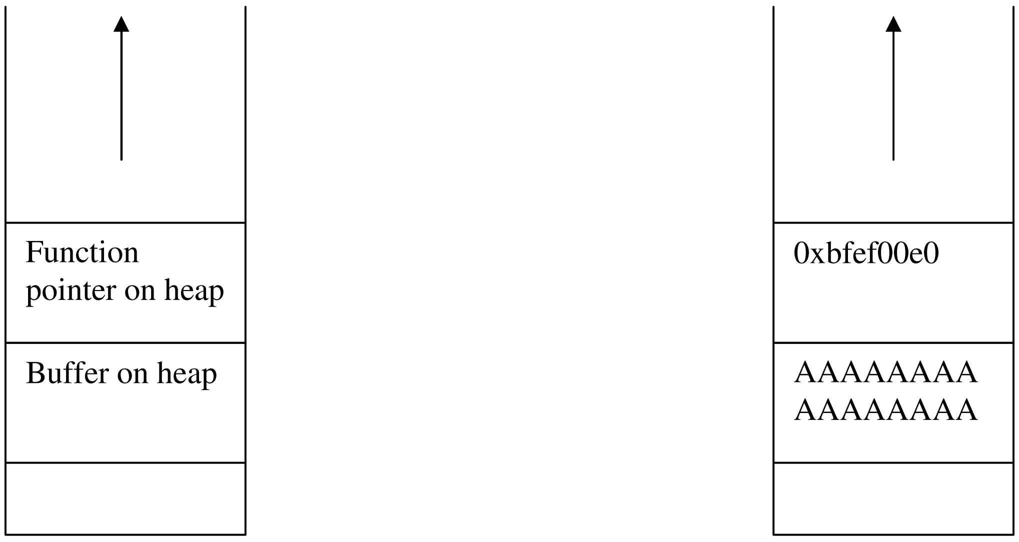


Figure 3: Heap-based vulnerability

1.1.2.3 Input validation error

A classic example of an input validation error is a Format String attack. The use of user input as a format string parameter to the `printf` class of functions can incur security problems [9]. Sensitive data from memory regions can be read by `%s` and `%x` format specifiers and `%n` can be used to write data to memory locations. Similarly, passing user input data to system calls like `popen()` and `execve()` can execute any arbitrary command.

Wrong usage:

```
int func (char *user)
{
     printf (user);
}
```

Correct usage:

```
int func (char *user)
{
     printf ("%s", user);
}
```

1.2 Related work

Vulnerability analysis is the process of determining if a system contains defects that could be exploited by an attacker to compromise the security of the system or that of the platform the system runs on. Numerous approaches have been tried and tested for taint analysis which we briefly discuss below.

1.2.1 Vulnerability detection methodologies

Based on the accuracy with which vulnerability analysis is carried out, we can classify them as coarse-gained analyses and fine-grained analyses. Alerting a non-exploit as an exploit is regarded as a false positive. Failure to alert an exploit is regarded as a false negative.

1.2.1.1 Coarse-grained taint analysis

One of the common approaches taken to prevent intrusion is to block network ports or filter network packets [6]. These approaches do not cause any overhead on the program, but they suffer from high numbers of false positives as well as false negatives.

Another approach in the coarse-grained class is to add function prologue and epilogue (see Chiueh et al. [8]). In this approach, the boundary of every function in the input program is identified, and a sequence of protection instructions is inserted. Similarly, StackGuard [25] places a “canary” word next to (prior to) the return address on the stack. Once the function is done, the new code from compiler first checks to make sure that the canary word is unmodified and intact before jumping to the return address. However, these approaches address only a small class of vulnerabilities and also they require program recompilation. IBM’s GCC (GNU Compiler Collection) extension [10] reorders local variables to place buffers after pointers to protect function pointers from stack-smashing attacks. This approach however cannot catch any heap-based attacks.

Another approach is to carry out static analysis on the source code to detect vulnerabilities. This, however, necessitates availability of source code, which is not commonly available for commodity software. Also assembly code generated from source can be significantly different compared to the actual source due to compiler optimizations like instruction reordering. This opens up a space where vulnerabilities not present in the original source code may appear in the optimized binary and vice versa, leading to higher false positives and false negatives. Thus source code analysis may not provide as much security guarantees as binary analysis.

1.2.1.2 Fine-grained taint analysis

The discussion above clearly indicates the need for more accurate and reliable techniques. In fine-grained taint analysis approach, essentially every operation performed by the program is tracked and is inspected for malicious behavior. In this approach, data originating from user, file, and network ports is marked as untrusted (tainted). The untrusted data is tracked from its point of origin to the point of possible exploit. Any anomalous usage of untrusted data, for example as an argument to a sensitive system call, the address of an indirect call, etc, is flagged as an error. Because the fine-grained approach makes no assumptions about the runtime behavior of program, it has fewer false negatives and false positives. However, it is expensive because potentially every instruction has to be tracked, because doing so would require being able to solve the halting problem.

Fine-grained analysis can be carried out in two ways:

Static analysis: In this case the binary is disassembled and symbolic execution is performed to detect the class of vulnerabilities consisting of the use of tainted data in sensitive operations [11]. However this approach faces difficulties involved in getting correct disassembly, obtaining correct flow graphs, pointer analysis, resolving indirect jump targets, and symbolic execution of loops. Also, static analysis cannot possibly handle all programs.

Dynamic or Runtime analysis: Dynamic taint analysis involves tracking the use of untrusted data during program execution. Currently there are three ways to track taint information [24].

i) **Interpreter-based approach:** Some of the interpreted languages like Perl have a built-in facility to track untrusted data. While in this mode, Perl takes special precautions called *taint checks* to prevent both obvious and subtle traps. Some of these checks are reasonably simple, such as verifying that path directories aren't writable by others. Complex ones involve not allowing a program to use data derived from outside the program to affect something else outside the program. All command line arguments, environment variables, locale information, results of certain system calls (`readdir()`, `readlink()`, the variable of `shmread()`, the messages returned by `msgrcv()`, the password, and shell fields returned by the `getpwxxx()` calls), and all file input are marked as "tainted". Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, or in any command that modify files, directories, or processes.

TaintCheck [17] runs a compiled program in interpreted fashion to track tainted data. In their paper [17], Newsome and Song perform binary rewriting at runtime and run a program in an emulated environment. This allows them to monitor and control program's execution at fine-gained level. On reaching each basic block, TaintCheck translates the block of x86 code into its own RISC (Reduced Instruction Set Computers) like instruction set called UCode. TaintCheck, then instruments Ucode block to incorporate taint analysis code. It then converts Ucode back to x86 and executes the block. This approach shows a slowdown of about 30 times.

ii) **Architecture-based approach:** With custom hardware support, the processor can carry out taint computation for every instruction it executes. In their paper [23] Edward Suh et al. let the operating system identify a set of input channels as spurious, and make processor track all information flows from those inputs. If spurious values are used for an operation, a checker generates a security trap. When the processor generates a security trap, a handler checks if the trapped operation is allowed or not. This approach shows a slowdown of 5.5 times.

iii) **Instrumentation-based approach:** In this approach, the program is instrumented to dynamically trace the propagation of taint data. In their paper [13], Cheng et al. propose a frame work consisting of a configuration file to specify a security policy, a shadow memory to maintain taint information, and a program monitor to perform instrumentation and to intercept system calls. The program monitor inserts additional code for maintaining, propagating, and checking taint status before executing the code.

In their paper [19], Qin et al. propose a Low-Overhead Practical Information Flow Tracking System (LIFT). LIFT minimizes run-time overhead by exploiting dynamic binary instrumentation. LIFT aggressively eliminates unnecessary dynamic information flow tracking, coalesces information checks, and efficiently switches between target programs and instrumented information flow tracking code. LIFT exploits the fact that for most server applications, the majority of tag propagations are from safe data sources to safe destinations.

These approaches have various drawbacks. The interpreter-based approach works only on some languages. TaintCheck [17] takes a little different approach of converting a program written for x86 to RISC, but, it incurs significant overhead due to interpretive execution and runtime instrumentation. Also, runtime analysis leaves less scope for optimizations. The 30-40 times slowdown reported by TaintCheck [17] means it cannot be deployed in production.

The architecture-based approach needs processor support, which is currently not available from any vendor, and further it needs significant operating system support and has less flexibility in security configurations. This means we cannot expect architecture-based approaches any time soon, and also users are unlikely to adopt this approach if they cannot have application-specific security configurations.

The instrumentation-based approach incurs one-time instrumentation overhead and a much higher runtime tracing overhead on each run. However, instrumentation-based approach shows a promising future since there is a wide scope for static analysis. Also, contemporary research on instrumentation [16] proposes low overhead instrumentation techniques. Our technique falls in to this category but has a different approach to the implementation.

1.3 Our contribution

In this thesis, we propose a novel approach to detect program exploitation without prohibitive performance overheads. The key idea is to delegate to a separate concurrently executing thread (which we call a shadow thread), the work of tracking untrusted data from its point of origin to the point of exploit in a program. Since we don't run the original program in interpretive fashion or make it keep track of taint information, the original program does not have the runtime overheads seen in other approaches. Further, since more and more processors are equipped with multicores and most of the software is not written to fully utilize the computing power of multicore machines, we are likely to have some idle cores. Also, in view of the hardware trends showing addition of just not tens, or hundreds but thousands of cores, our idea to utilize some idle cores for vulnerability detection goes a long way to shaping the future of computing. In addition to this, the shadow thread performs fine-grained taint tracking and hence provides high security guarantees. Because of runtime taint tracking, our approach has few false positives.

Our approach uses already established facts like shadow memory, memory tagging, and taint propagation [16, 23, and 17] for vulnerability detection. However, we do so in a completely new way by means of a shadow execution running on an idle core. We use static binary rewriting, which enables us do optimizations (see Section 3.7) that are hard if not impossible with dynamic instrumentation. Because of taint checking by a concurrent thread and the static optimizations, we incur low runtime overhead.

We demonstrate the effective usage of our approach for buffer overflow and format string attack detection on real applications. The preliminary performance results show that there is an order of magnitude performance improvement compared to TaintCheck [17] and we better LIFT [19] in some cases. The rest of this document is organized in the following manner: Chapter 2 provides an overview of the entire system; Chapter 3 describes the design and implementation details; Chapter 4 presents the security assurances that we can make; Chapter 5 evaluates performance and finally Chapter 6 presents conclusions and discusses future work.

CHAPTER 2

2 SYSTEM OVERVIEW

In this chapter, we discuss the key characteristics of our system and provide an overview of the components that make our system. We have developed a proof of concept tool to establish the practical usability and reliability of our approach. Our approach of taint analysis has the characteristics described below, some of which are also seen in other state of the art taint analyzers like TaintCheck [17], and TaintTrace [24].

2.1 Characteristics of our approach

2.1.1 Practical usability due to better performance

Unlike other instrumentation based approaches, our tool does not require the original program itself to compute the taint values for each operation. Instead we spawn a new thread, which runs on an idle core, to perform taint computation. Hence our approach offers better performance.

2.1.2 Reliable security

Like some of the others, our approach computes and propagates the trustworthiness of data through every instruction executed by the program at run time. This fine-grained analysis offers high security guarantees in detecting all kinds of buffer overflow attacks that lead to change of control, format string attacks, input validation errors, etc. Because of the flexible and configurable security, extending the current system to handle other kinds of attacks is simple.

2.1.3 Source code independence

Our approach works by disassembling a binary program. We do not need source code, and we do not need any special compilation for our analysis. This offers a great practical benefit on current and legacy software.

2.1.4 Language independence

Our approach currently works by disassembling the ELF (Executable Linkable Format) x86 binaries. Hence the approach is independent of the source programming language. Further we can extend the current tool to support other architectures.

2.1.5 Hardware independence

Our project is motivated by the fact that many, if not all, applications are finding it difficult to leverage the full advantage of multicore processors because most applications are interactive and they inherently have some sequential components. Currently most of the processors are dual core and the hardware trend shows processors getting 8 or 16 cores in near future and 1000s of cores not in the too distant future. Hence we should be able to schedule our shadow thread on an idle core. Other than multicore, we do not require any hardware support (unlike [10, 11], which heavily depend an architectural support for taint analysis).

2.1.6 Operating system independence

We do not require any support from or modifications to the operating system. We mark the origin of taint data by identifying system calls like `read(), recvfrom(),` etc.

2.2 Schematic diagram

Figure 4 presents the architecture of our taint analyzer. Given an input binary P_{input}, we do the following:

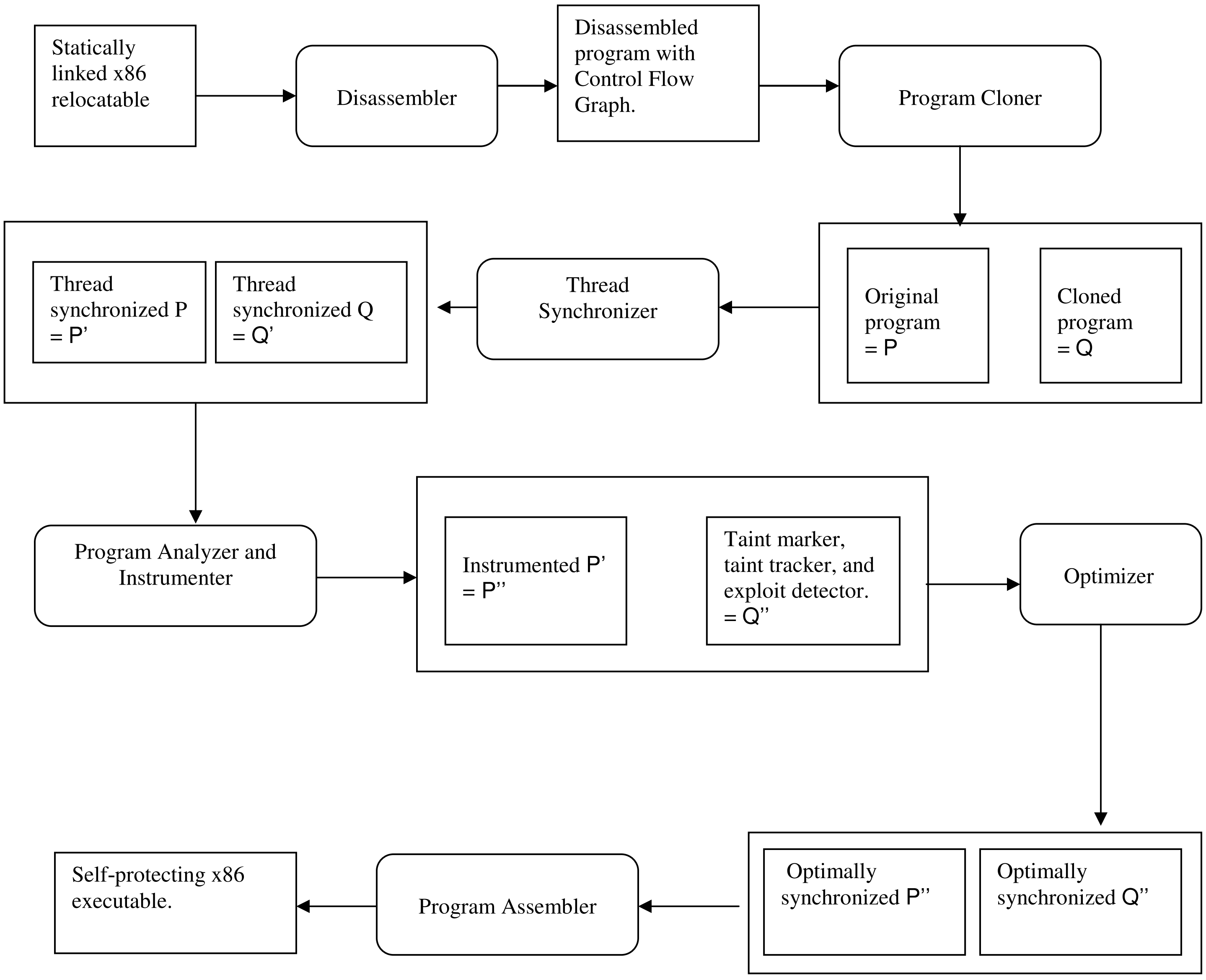
1. The memory area *M* of P_{input} is augmented with an additional memory region *M'* such that for each original memory byte $w \in M$, there is a corresponding tag word $\tau(w) \in M'$. This is straightforward to do by rewriting the binary to incorporate a new data section. The tag word $\tau(w)$ contains the taint value associated with *w*. More specifically if *w* is either copied or arithmetically computed from an untrusted input, then $\tau(w) = 1$, otherwise $\tau(w) = 0$.

2. The program P_{input} is rewritten to consist of two interacting threads: $P_{original}$, which carries out the actual computations of the original program, and P_{shadow} which shadows the computation of $P_{original}$ but computes the trust values instead of actual data values. Both threads share the same address space; this allows them to communicate via shared memory. The shadow thread should be always behind the original thread so that shadow thread can perform the taint computations on behalf of original program. However, shadow program cannot remain too far behind the original program since before it computes the taint and asserts an exploit, the original program might have already been exploited. In the current configuration we decided to keep the shadow thread one basic block behind the original program (see Chapter 3 for exceptions). However this is a configurable option and we can trade off security vs. performance.

3. We create P_{shadow} from P_{input} by replacing the actual computation with equivalent taint-computing instructions. We regard the program points that originate untrusted data as “Taint Sources” and the program points beyond which the attacker has complete control over the program as “Taint Sinks”. The shadow has code to mark the tainted memory regions at taint sources and it has guards that check whether the system is about to be compromised around the potential taint sinks like return address, function pointers, sensitive system call arguments, format string handling functions, etc. We add code to $P_{original}$ and P_{shadow} to synchronize at conditional branches and indirect control transfers so that $P_{original}$ can communicate the transfer targets to P_{shadow}. This is required since P_{shadow} does not compute values and hence it does not have the

information it needs to follow $P_{original}$'s control behavior. If we want to audit return instruction or indirect jumps etc, we also add code to delay $P_{original}$ at these points until P_{shadow} completes the security checks.

4. Finally, we optimize $P_{original}$ and P_{shadow} to remove unnecessary computation and synchronization overheads and the two threads are packaged as a single executable that is invoked in the same way as the original program.



Statically linked x86 relocatable
Disassembler
Disassembled program with Control Flow Graph.
Program Cloner
Original program = P
Cloned program = Q
Thread Synchronizer
Thread synchronized P = P'
Thread synchronized Q = Q'
Program Analyzer and Instrumenter
Instrumented P' = P''
Taint marker, taint tracker, and exploit detector. = Q''
Optimizer
Optimally synchronized P''
Optimally synchronized Q''
Program Assembler
Self-protecting x86 executable.

Figure 4: Schematic diagram

CHAPTER 3

3 DESIGN AND IMPLEMENTATION

We use the PLTO[1] binary rewriting system to implement our prototype taint checker. Input to PLTO is a statically linked x86 relocatable in ELF format. We assume that the relocatable has information about procedures, symbols, and relocations as per ELF format. We consider this to be a reasonable assumption since we are not trying to deal with virus code, but instead trying to catch vulnerabilities that exist in commodity software.

The components of our taint analyzer are shown in Figure 4. Below we describe the steps in creating a self-protecting binary from the original program's binary.

3.1 Disassembling the program

An ELF format statically linked relocatable is fed to PLTO (Pentium Link Time Optimizer) [21] to carry out accurate program disassembly. PLTO is a binary rewriting system that modifies an object program to improve some aspect of its behavior, such as execution time, code size, or security. PLTO is developed for the Intel IA-32 architecture to handle complexities like large number of op-codes, addressing modes, and variable-length instructions. PLTO first disassembles all segments containing code, creates a single instruction stream, and constructs an interprocedural control flow graph for the entire program. It uses relocation information and knowledge about instruction semantics to guide these steps. The output of the disassembly stage is a control flow graph with information about functions, basic blocks, and control transfer edges.

3.2 Creating shadow memory

The next step is to augment the program with a shadow memory as described in Chapter 2. To do this we create a new section called "`.shadow`" in ELF with size |*M*|/2, where *M* is the total addressable virtual memory. Also during program assembly we allocate a new segment for the shadow section. With this configuration, for every memory location $w \in M$, we can compute the corresponding shadow memory location $\tau(w) \in M'$ in the shadow region by subtracting a constant offset from *w*. In other words, $\tau(w) = w - K$, where *K* is a constant during execution of a program. Typically *K* = |*M*|/2. Figure 5(a) below shows the memory layout of a standard ELF format executable in Linux and Figure 5(b) shows the program after augmenting it with a shadow memory region.

[1] PLTO (Pentium Link Time Optimizer) requires statically linked relocatable binaries in order to have all relocation information.

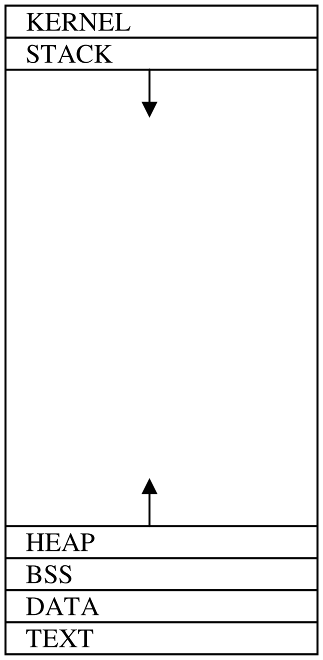


Figure 5(a): Original program

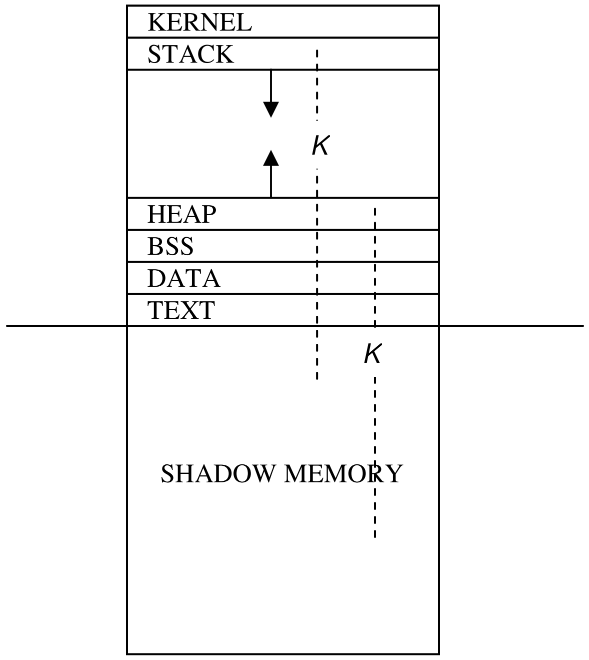


Figure 5(b): Program with shadow memory

3.3 Building the shadow thread

Following the creation of shadow section, we need to create a shadow thread that carries out the task of taint analysis.

3.3.1 Cloning the program to produce the shadow thread

The first step toward building a shadow program is whole program cloning. Given the original program $P_{original,}$, we build $P_{shadow.}$ The program cloner is built using the facilities provided by PLTO to clone functions. A linear sweep over all functions in $P_{original}$ creates clone functions. With this scheme, inter-procedural edges point back to the functions in $P_{original}$ (as shown in Figure 6). However, by maintaining a one–to–one mapping between each function to its clone and each block to its clone, we can redirect the control edges to the corresponding shadow blocks (as shown in Figure 7).

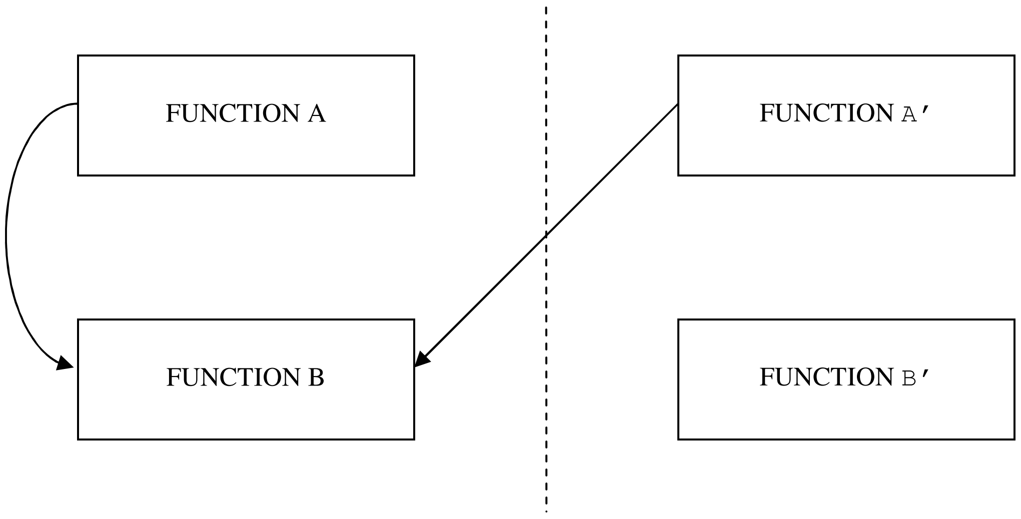


Figure 6: Cloned functions

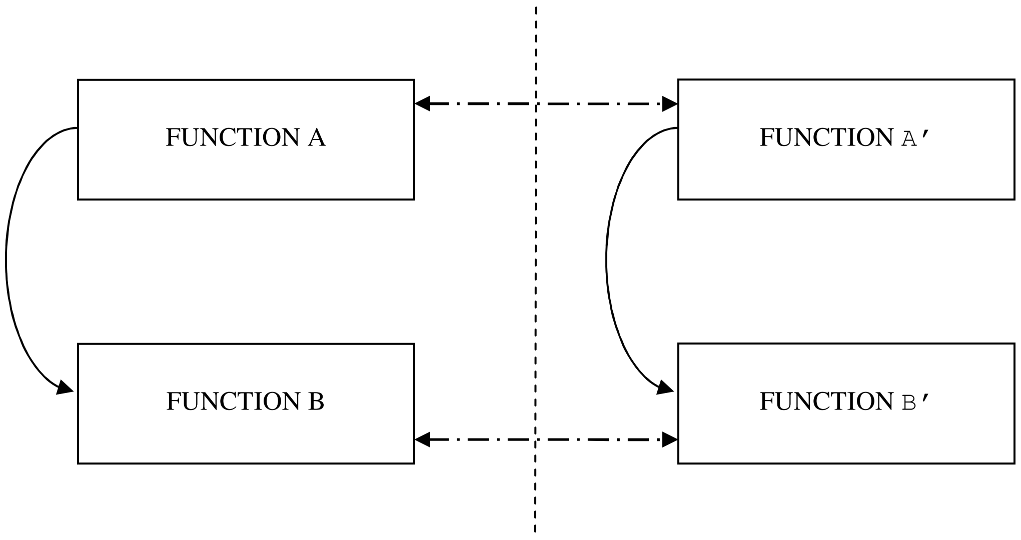


Figure 7: Cloned functions with one-to-one mapping

3.3.2 Control flow imitation in shadow using synchronization

As discussed previously, we need to keep P_{shadow} behind $P_{original.}$ Remember that the shadow program has no information to imitate the control flow of the original program since it does not do actual computation. The thread synchronizer component of our tool handles this task. We use a global flag `TARGET` to achieve signaling between two threads. We do not use any system calls for synchronization since system calls are inherently expensive.

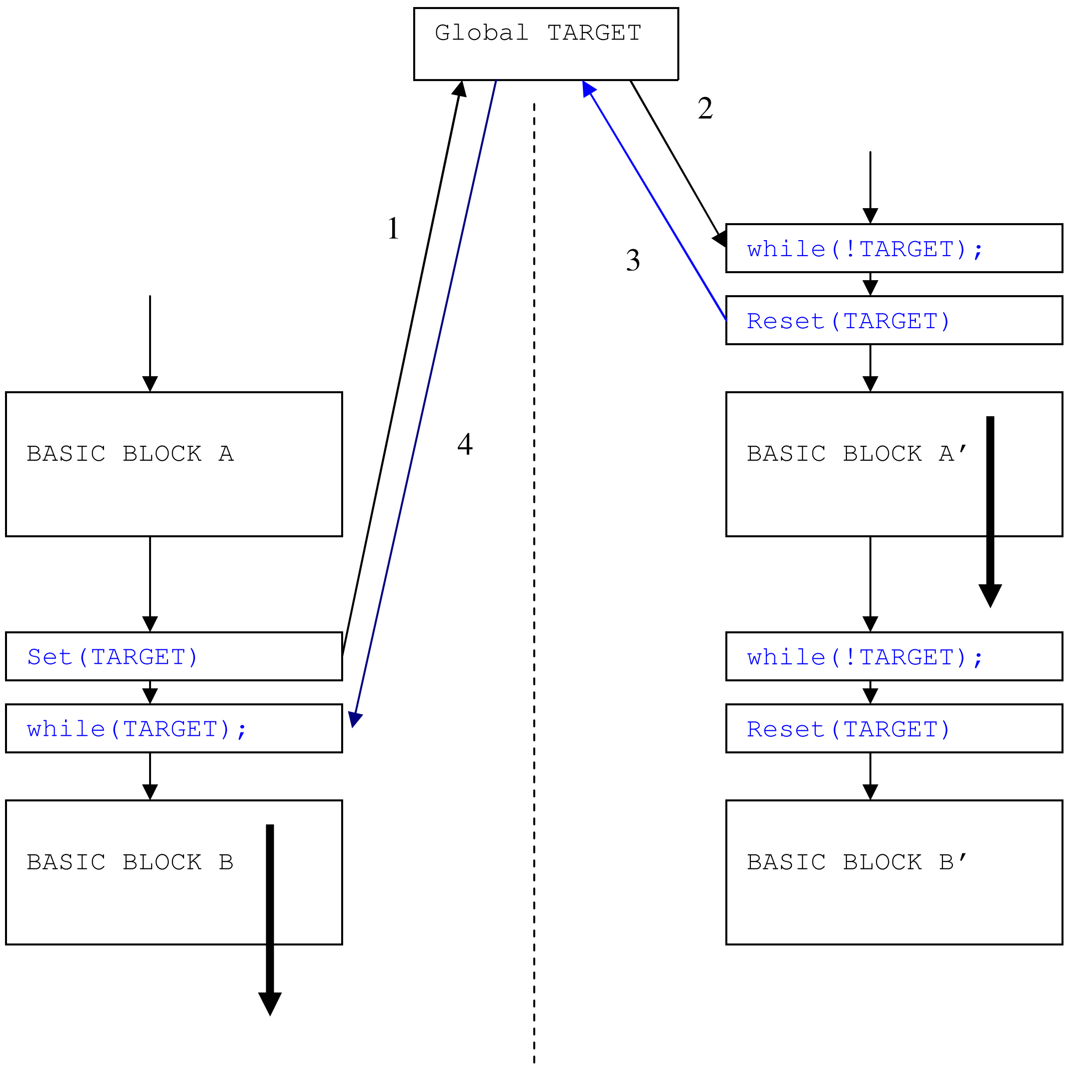


Figure 8: Thread synchronization for simple blocks

Let us assume that basic block A is the program entry point. The shadow program will have A' as its entry point. We augment P_{shadow} and $P_{original}$ as shown in Figure 8. Originally `TARGET` is set to zero. Following is the sequence of steps that happen at runtime:

1. $P_{original}$ finishes executing basic block `A` during which P_{shadow} is spinning waiting for the `TARGET` to be set.
2. $P_{original}$ sets `TARGET` and waits for it to be reset. (Arrow 1 in Figure 8)
3. P_{shadow} observes `TARGET` being set (Arrow 2 in Figure 8), stops waiting and resets `TARGET` (Arrow 3 in Figure 8)
4. P_{shadow} starts executing basic block `A′`.
5. $P_{oroginal}$ observes `TARGET` being reset (Arrow 4 in Figure 8), stops waiting and starts executing basic block `B`.

During step 4 and 5, the two threads are executing concurrently. Though the above example shows the basic case of synchronization, we need to handle more complex scenarios like conditional branches, jump tables, indirect jumps and indirect function calls.

Handling branches:

To handle conditional branches we augment P_{shadow} and $P_{original}$ as shown in Figure 9. Assume that each basic block has a unique identifier. For example, the address of the first instruction of each block can serve as its unique identifier. The following sequence of steps happens at run time:

1. $P_{original}$ executes basic block `A` and branches to basic block `C` or `B`.
2. If $P_{original}$ branches to `C`, it sets `TARGET = C′`. If $P_{original}$ branches to `B`, it sets `TARGET = B′`. In either case $P_{original}$ waits for `TARGET` be reset.
3. On entering `A′`, P_{shadow} copies `TARGET` to `LocalTarget` and resets `TARGET` so that $P_{original}$ can proceed.
4. After executing `A′`, P_{shadow} determines its next basic block by comparing the `LocalTarget` with `C′`. If they are same, it executes block `C′`; otherwise it executes block `B′`, thus imitating $P_{original}$'s execution.

Because of the one-to-one mapping between the basic blocks in the original and shadow programs, we can statically determine the `TARGET` values to be set during program instrumentation.

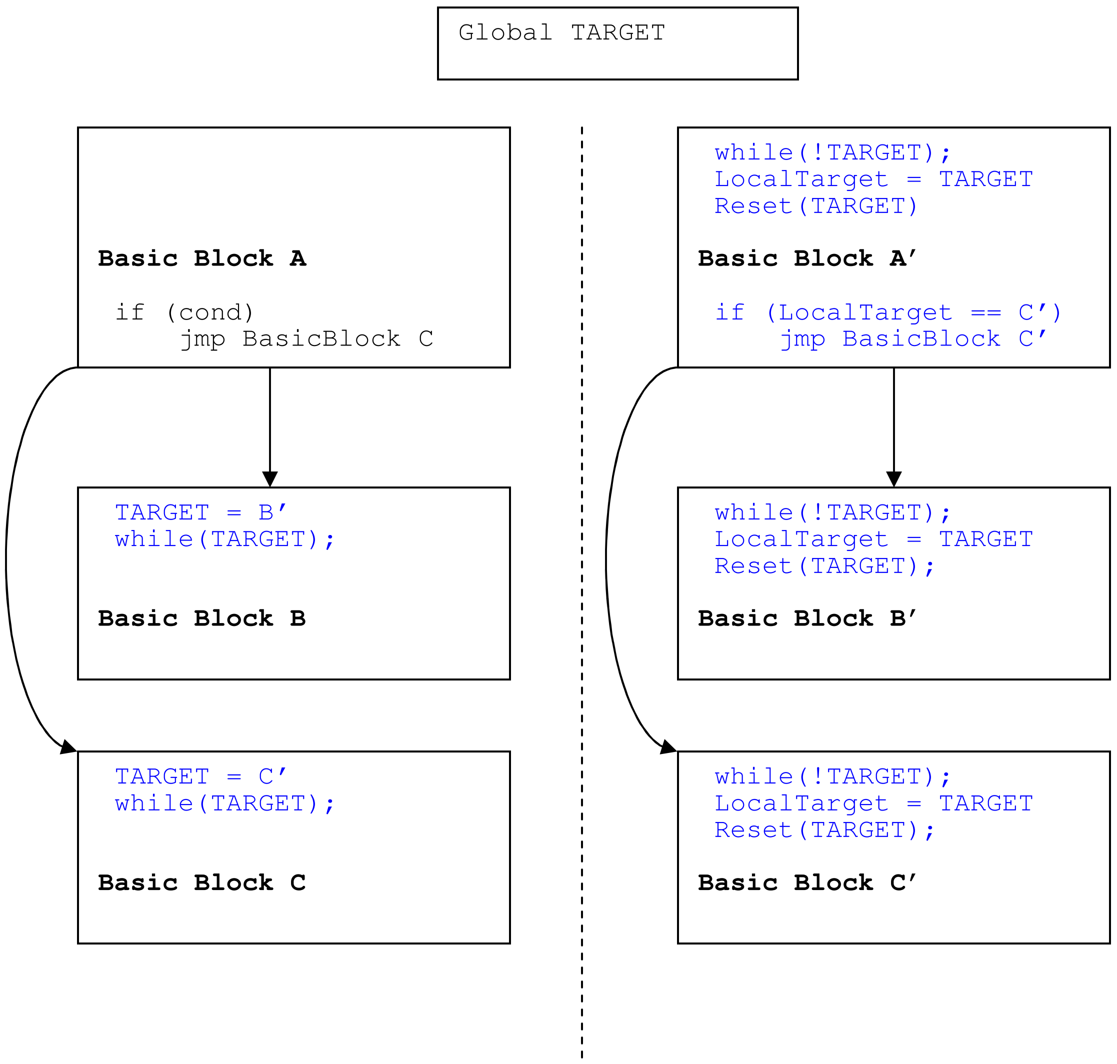


Figure 9: Thread synchronization for conditional branches

Handling indirect jumps:

To handle conditional branches we augment P_{shadow} and $P_{original}$ as shown in Figure 10. Assume $P_{original}$ has an indirect jump after basic block A. The following sequence of steps happens at run time:

1. $P_{original}$ executes basic block `A` and jumps to basic block `B` via an indirect jump.
2. On entering basic block `B`, $P_{original}$ sets `TARGET` to the first instruction of basic block `B'`.
3. On entering basic block `A'`, P_{shadow} makes a local copy of the `TARGET` into `LocalTarget` and resets `TARGET` so that $P_{original}$ can proceed.
4. At the end of basic block `A`. P_{shadow} simply jumps to the address present in `LocalTarget`, which lands it in `B'`.

Because of the one-to-one mapping between instructions, we can statically determine the `TARGET` values to be set during program instrumentation.

We use the same technique for indirect function calls and jumps via jump tables.

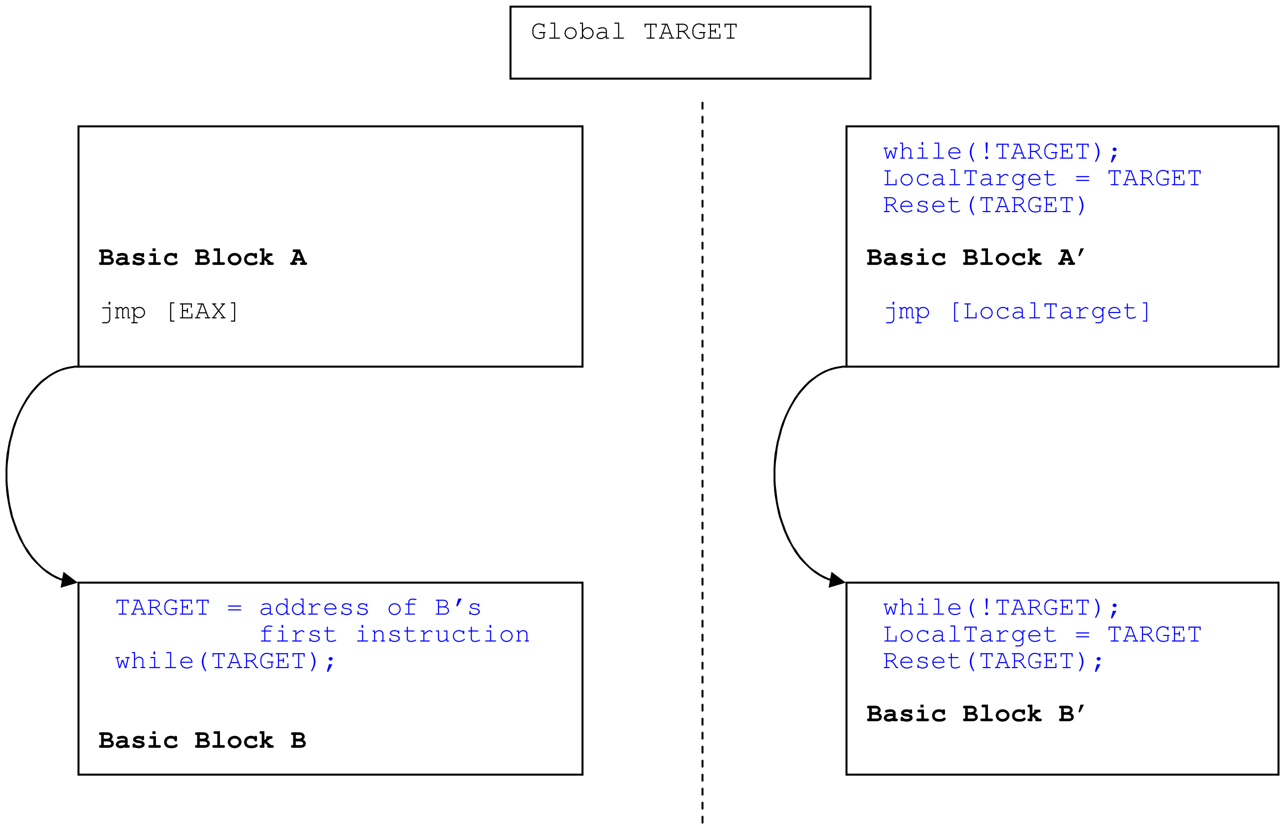


Figure 10: Thread synchronization for indirect jumps

3.3.3 Spawning the Shadow thread

Once the two synchronized threads are created, we need to add code to spawn two threads. We use the `Pthreads` (Portable Operating System Interface (POSIX) threads) library. However, before threads can be spawned, we need the program to set up some initial data structures. Hence we should not have $P_{original}$ expect P_{shadow} to be behind it during this phase. Further, because these initialization routines may call some common functions like `malloc()`, we need these common routines to be available in nonsynchronized mode during program initialization and in synchronized mode after initialization. We break this chicken and egg problem by maintaining one more copy of the program that does not have any synchronization code. Initially we run the nonsynchronized copy from the point of program entry until the program's data structures are set (`_start, _init` etc in Linux context). Once the initial data structures are set up, $P_{original}$ spawns P_{shadow}. $P_{original}$ then calls the main routine in the original program, and P_{shadow} calls the main routine in the shadow program. This is illustrated in Figure 11.

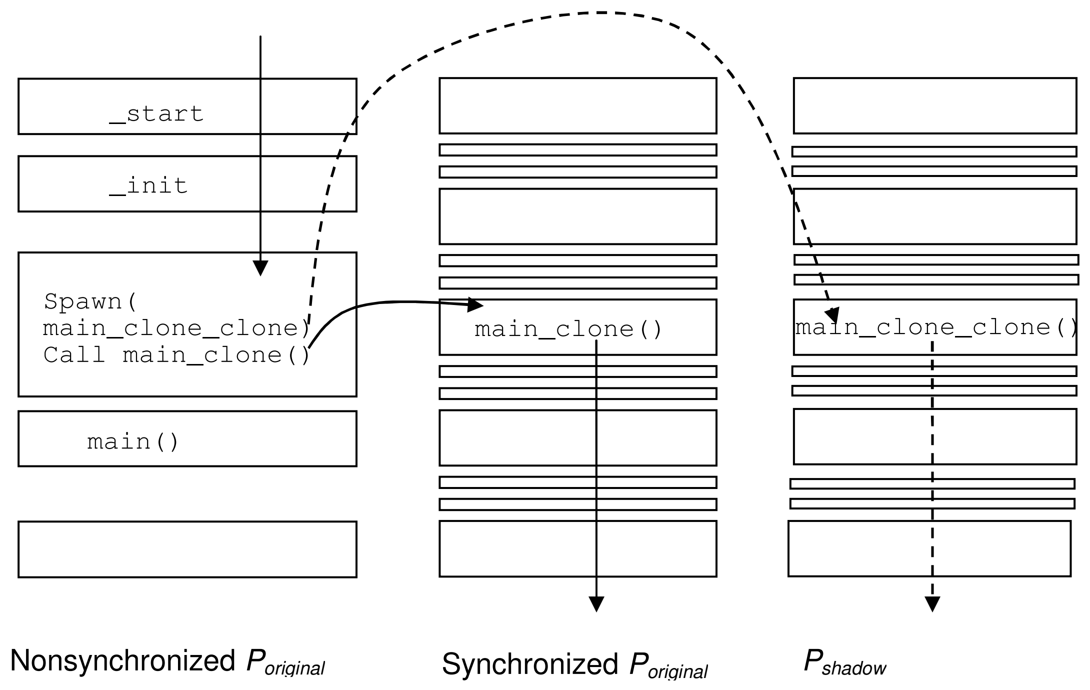


Figure 11: Shadow-thread spawning

A key aspect of shadow thread creation is setting up its stack space. We spawn P_{shadow} particularly taking care that its stack is set up in the shadow memory section as shown in Figure 12:

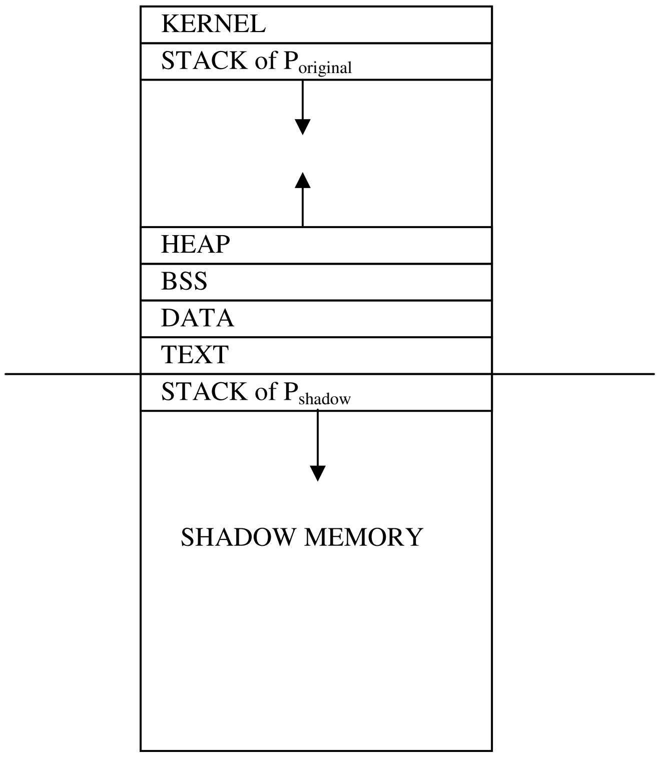


Figure 12: Stack-balanced $P_{original}$ and P_{shadow}

Once we have created two synchronized threads, we use the program analyzer and instrumenter components of our tool to build taint marking, tracking, and security checking into the shadow thread as discussed below.

3.4 Marking the taint sources

In our taint analyzer, we regard any data coming into program address space from an external source as untrusted. The default policy is to regard data read from system calls like `read(), and recvfrom()` as untrusted. The policy can be easily extended to other system calls. In Linux the system calls work using a software interrupt like `int 0x80.` The accumulator holds the interrupt number and other registers contain the arguments to system call. Consider the case of `read()` system call which appears as shown below in x86 disassembly.

```
push    %ebx
mov     0x10(%esp),%edx
mov     0xc(%esp),%ecx
mov     0x8(%esp),%ebx
mov     $0x3,%eax
int     $0x80
```

Immediately after the system call, the following can be inferred:

1. Register EAX contains the number of bytes read (say *N*)
2. Register ECX contains the pointer to the buffer in user space (say *P*).

With this information, it is fairly simple to augment the shadow thread to set *N* bytes of shadow memory starting at location $\tau(P) = P - K$, as tainted. Note however that P_{shadow} does not make any system call by itself. The approach is extendable to fast system calls that use instructions like `SYSENTER/SYSEXIT`.

Another source of taint input to a program is command line arguments and environment variables, which are set up during program loading. The environment variables and arguments to the `main()` function are stored below the kernel stack (as shown in Figure 13). Since we treat command line arguments and environment variables as untrusted, we mark the corresponding shadow region as tainted. This helps us catch exploits that are possible via malicious command line parameters or environment variables (see Section 4.3.2.4).

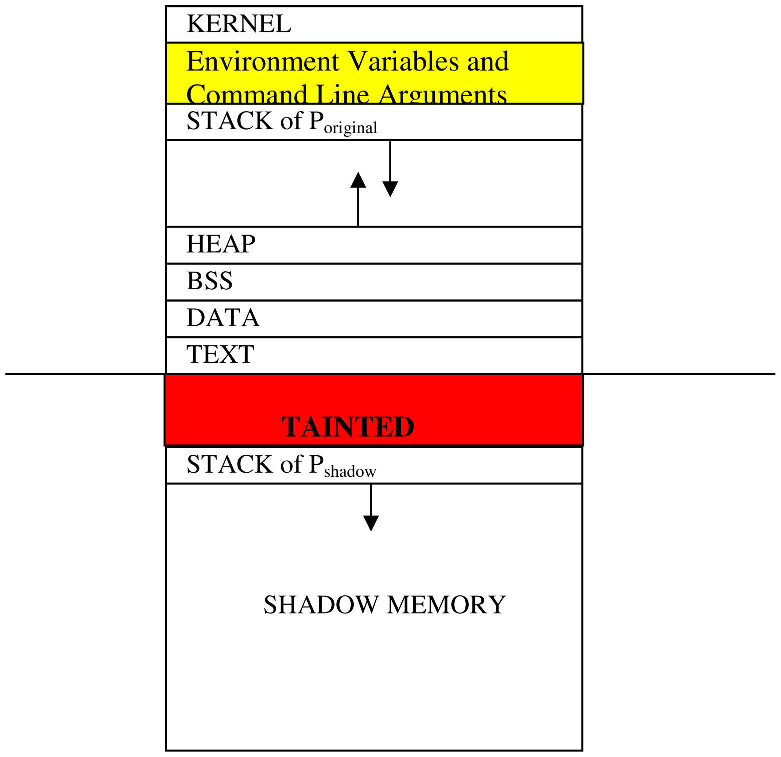


Figure 13: Tainted command-line arguments and environment variables

3.5 Tracking taint propagation

Once injected, untrusted data can flow to other parts of memory in various ways. Newsome and Song have categorized them in to 4 possible ways [17]:

Copy dependency: Copying of a tainted value to a different location taints the new location.

Arithmetic dependency: If tainted data is used as a source operand of a computation, the results of the computation are tainted.

Address dependency: If a tainted value is used to specify the address for a load/store operation, the loaded/stored value depends on the tainted value.

Control dependency: A structure of the form

```
if ( x == 0 )
     y = 0;
else if ( x == 1)
     y = 1;
     ….
```

is the same as `x = y` and hence a method of taint propagation.

We classify each instruction as either *taint causing* (e.g. `mov, add` etc) or as *innocuous* (e.g. `inc, dec`). Each *taint causing* instruction of the form *c = a Φ b* (where *Φ* is an operation that takes *a, b* as source operands and *c* as destination operand) in P_{shadow} is replaced by an operation $\tau(c) = \tau(a)\ LOGICAL\_OR\ \tau(b)$. Each innocuous instruction is simply destroyed. The Table 1 below gives a mapping of some instructions in $P_{original}$ to P_{shadow}.

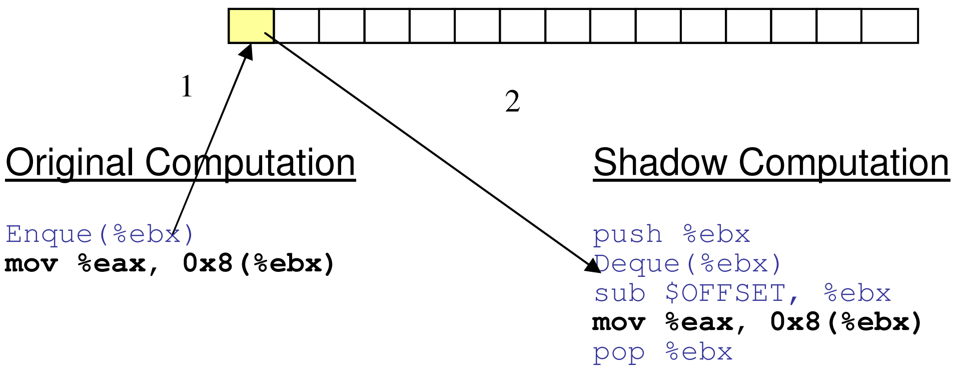
Original Computation	Shadow Computation	Instruction Class
`mov %eax, %ebx`	`mov %eax, %ebx`	Copy Propagation
`mov %eax, 0x8(%ebp)`	`mov %eax, 0x8(%ebp)`	Copy Propagation
`add  %eax, %ebx`	`or  %eax, %ebx`	Arithmetic Propagation
`shl  %eax, 0x8(%ebp)`	`or  %eax, 0x8(%ebp)`	Arithmetic Propagation
`push %eax`	`push %eax`	Copy Propagation
`lea (%eax), %edx`	`mov %eax, %edx`	Address Propagation
`inc %eax`	None	Innocuous

Table 1: Mapping of original instructions to shadow instructions

Any instruction that affects the stack frame (e.g. `push, pop, call, leave, ret` etc) in $P_{original}$ also appears in P_{shadow} so that the stacks of $P_{original}$ and P_{shadow} are always balanced. Hence any memory operand in $P_{original}$ that is relative to the stack pointer or frame pointer can be replicated in P_{shadow}. For example, `0x8(%ebp)` is the same in both $P_{original}$ and P_{shadow}. However, memory operands that are not relative to the current stack frame cannot be replicated in P_{shadow}. To address this problem, we use a shared buffer between $P_{original}$ and P_{shadow} to pass the required information. As an example consider an instruction of the form:

```
mov %eax, 0x8(%ebx)
```

The augmented $P_{original}$ and P_{shadow} to handle this instruction are as shown in Figure 14.



```
Enque(%ebx)
mov %eax, 0x8(%ebx)
```

```
push %ebx
Deque(%ebx)
sub $OFFSET, %ebx
mov %eax, 0x8(%ebx)
pop %ebx
```

Figure 14: Shared buffer between original and shadow threads

At runtime the following two steps happen:

1. The original computation enqueues the value %ebx and proceeds.
2. The shadow computation dequeues the value, subtracts a constant OFFSET to compute the shadow memory address, and then stores the taint flag.

We use a large shared circular buffer and do not use any locking because we always know that the shadow is behind the original by one basic block. $P_{original}$ only needs to update the tail of the shared buffer, and P_{shadow} only needs to update the head of the shared buffer. Because of the large buffer size, the wrap around does not cause the original program to overwrite an unread value. Thus we ensure fast and efficient use of the shared buffer. (The size of the shared buffer can be set to twice the maximum number of enqueues caused by basic blocks in the program. In our current prototype we use a shared buffer of size 1024).

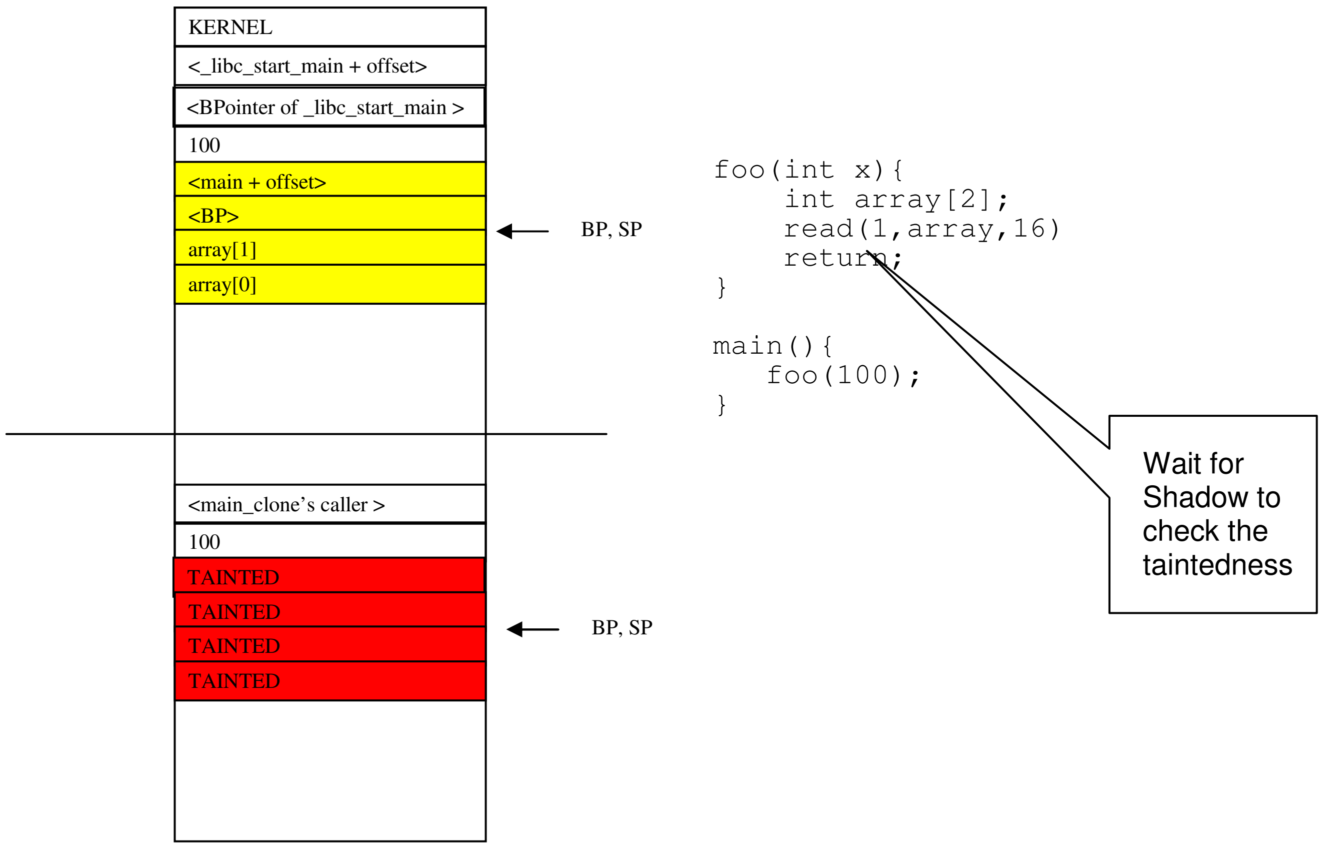
With this scheme, we can track taint propagation due to copy dependency, arithmetic dependency, and address dependency. However, like [17] and most of other taint analysis techniques, we cannot detect control based taint propagation. (See Chapter 6 for ideas).

3.6 Exploit detection

In order to obtain full control of the victim process, every attack has to change the program's control flow in order to execute malicious code. There are only a few ways to change a program's control flow. Attacks may change a code pointer for indirect jumps (e.g. return address overwrite), or inject malicious code at a place that will be executed without malevolent control transfer (e.g. arguments to sensitive system calls). We use the following defense mechanisms to protect against these exploits.

3.6.1 Guarding function returns

Before returning from callee to the caller (x86 instructions `leave` and `ret`), $P_{original}$ waits for P_{shadow} to make sure that the return address or old base pointer is not tainted. The Figure 15 shows a scenario where P_{shadow} detects a return address overwrite.



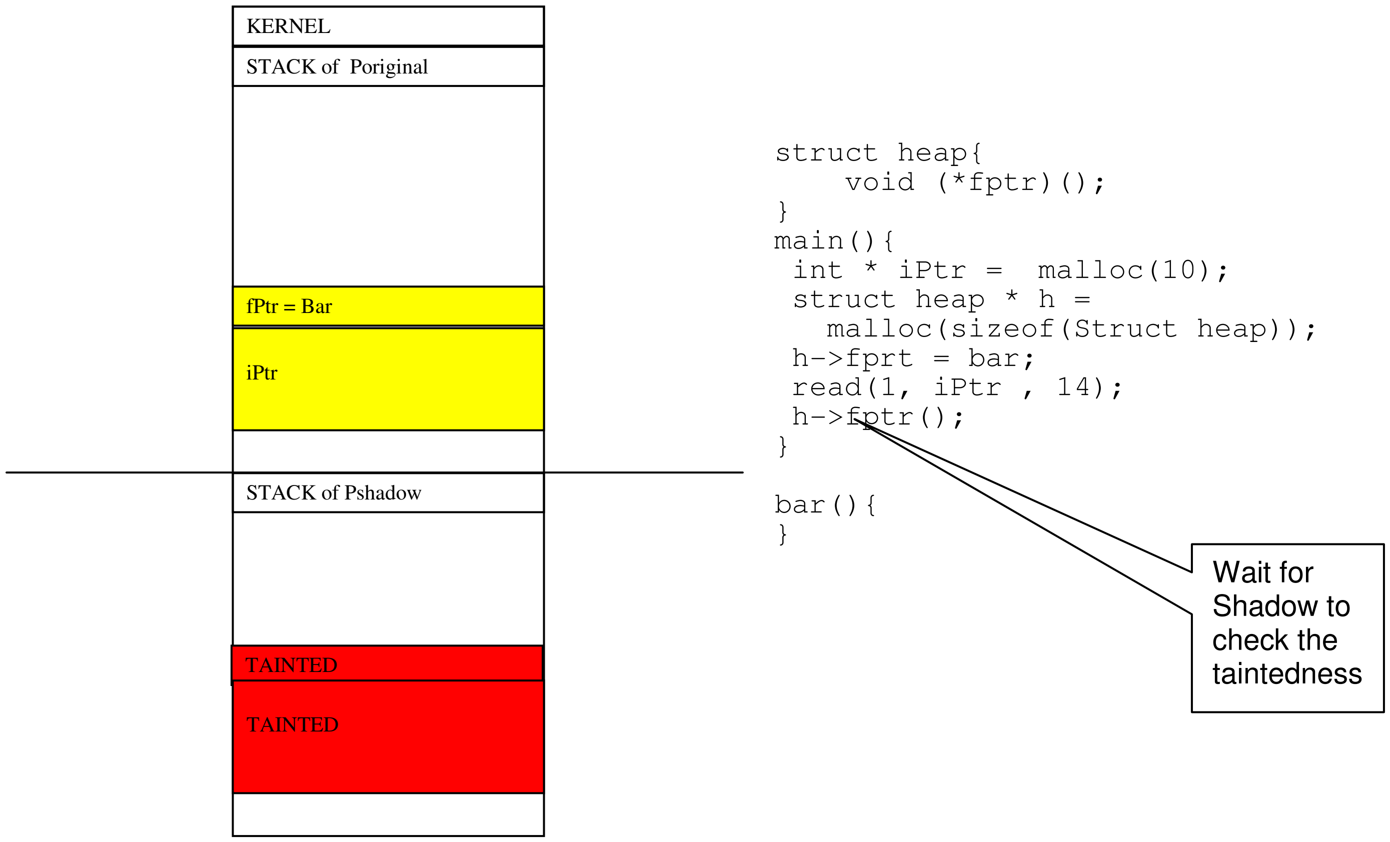
```
foo(int x){
    int array[2];
    read(1,array,16)
    return;
}

main(){
   foo(100);
}
```

Figure 15: Catching return address overwrite

3.6.2 Guarding indirect jumps

Before making an indirect jump, $P_{original}$ waits for P_{shadow} to make sure that the jump target address as well as the contents of jump target location are untainted. Figure 16 shows a scenario where P_{shadow} detects a function pointer overwrite attack.



```
struct heap{
    void (*fptr)();
}
main(){
 int * iPtr =  malloc(10);
 struct heap * h =
   malloc(sizeof(Struct heap));
 h->fprt = bar;
 read(1, iPtr , 14);
 h->fptr();
}

bar(){
}
```

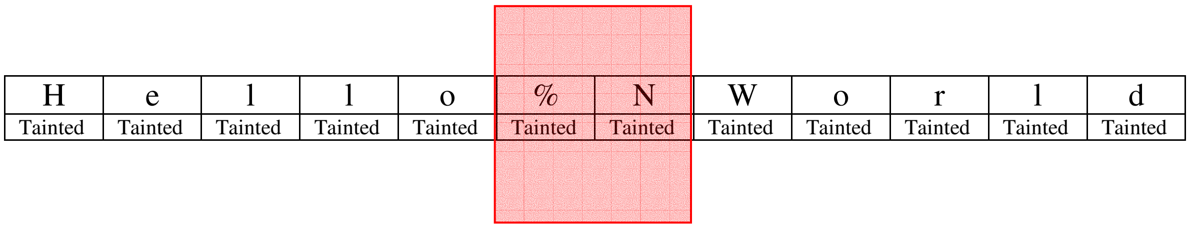
Figure 16: Catching function pointer overwrite

3.6.3 Guarding format string handlers

The format string handling functions are `sprintf()`, `snprintf()`, `fprintf()`, `vprintf()`, `vsprintf()`, `vsnprintf()`, `vfprintf()`, `syslog()` and `vsyslog()`. Among them `vfprintf()` is the basic function on which the other functions are based. To detect format string attacks, we guard `vfprintf()`. Below are the steps to catch a format string attack:

1. On entering `vfprintf()`, $P_{original}$ waits for P_{shadow} to enter its counterpart shadow function.
2. The second argument to `vfprintf()` is the format string. P_{shadow} examines the entire format string looking for occurrence of format specifiers like `%n, %x` etc.
3. If step 2 finds any format specifier in memory location *w*, we examine memory location $\tau(w)$. If $\tau(w)$ is marked as tainted, then we flag a format string attack alert.

Figure 17 shows the scenario where we detect a format string attack.



H	e	l	l	o	%	N	W	o	r	l	d
Tainted	Tainted	Tainted	Tainted	Tainted	Tainted	Tainted	Tainted	Tainted	Tainted	Tainted	Tainted

```
vfprintf(File * fp, char * fmt, …){
      if(fmt contains a “%n”  that is tainted) then
            Flag Warning
}

Foo(){
      char * buffer[256];
      read(1,buffer,256);
      printf(buffer);
}
```

Figure 17: Catching format string attack

3.6.4 Guarding sensitive system calls

We can detect whether particular arguments to sensitive system calls like `execve()` are tainted using a combination of the taint-marking technique and the format string exploit detection technique discussed previously. This in combination with authenticated system calls [19] gives a strong security guarantee for making system calls. We track the x86 software interrupts that cause system calls and check for the taintedness of the strings passed as arguments to the system calls. However by default we do not enable this option in our prototype implementation since some applications may use user given data as argument to sensitive system calls like `execve()`. Users can decide whether to permit a tainted string as system call argument or not.

3.7 Reducing synchronization via static analysis and optimization

Since block by block synchronization between two threads has high overhead, we perform the optimizations below to reduce synchronization. For the following discussion we define a set of functions *descendents(F)* for a function F as follows:

Let $F \rightarrow G$ denote function *F* (directly or indirectly) calls function *G*.
Let $\rightarrow^*$ denote the reflexive transitive closure of the relation $\rightarrow$.

Then, set of *descendents* of function F is defined as:

$$descendents(F) = \left\{ \quad G \mid F \rightarrow^* G \quad \right\}$$

3.7.1 Not monitoring all function return addresses

By default, we synchronized the two threads at each function call return. However if a function *F* has following two properties, we do not need to monitor the function return address:

1. The function *F* by itself has no local variables, and
2. No member of the set *descendents(F)* has local variables.

Functions that exhibit above two properties are assured not to overwrite the return address on their stack frame. They may overwrite return addresses deeper in the stack frame, but that would be caught by the owner function of that stack frame.

3.7.2 Not tracing some functions

The shadow thread need not trace instructions in some of the function calls. To run a function *A* without synchronizing when called from function *B*, we need to have the following properties:

1. Function *A* by itself should not have any store operations.
2. Function *A* by itself is not a taint originating function.
3. No member of the set *descendents(A)* has store operations.
4. No member of the set *descendents(A)* is a taint originating function.
5. Function *B* should not use the return value from function *A* for any store operation.

Functions with these properties are assured to neither originate taint nor propagate taint. They are definitely innocuous functions (e.g. `strcmp()`). Such functions can be run at full speed without synchronization in the original program and need not be called by the shadow program at all. As mentioned earlier (see Section 3.3.3) we have a copy of the program without synchronization, hence we can easily do this.

3.7.3 Optimizing high frequency loops

One of the most commonly occurring loop patterns is sweeping through an array, assigning to array elements with either a constant value or with some user input. In the former case all that shadow thread should do is to set every element of the corresponding shadow memory region as untainted; in the latter case it should mark every element as tainted. To achieve this, the shadow program obtains the array base address, stride, and the number of iterations from the original program, and runs in parallel with the original program, but keeps itself just behind the original program. At the end of the loop both threads synchronize and then proceed as normal.

Figure 18 illustrates a program that sweeps an array to initialize values. Figure 19 shows the modified original program along with its shadow counter part.

```
    for( int i = 0; i < n ; i += k ){
          array[i] = 0;
    }
```

Figure 18: Array initialization loop

```
Global * arrayBase;
Global stride;
Global iterations = 0;
Global done = false;
```

Original Program

```
arrayBase = array;
stride = k;
for( int i = 0; i < n ; i +=k ){
    array[i] = 0;
    iterations++;
}
done = true;
while(done);
```

Shadow Program

```
shadowIterations = 0;
for(;;){
      shadowIterations++;
      while(shadowIterations
            >= iterations ){
                  if(done)
                        break;
      }
      if(shadowIterations >
         iterations){
            shadowIterations = 0;
            iterations = 0;
            done = false;
            break;
      }
      index = (shadowIterations - 1 )
               * stride
      arrayBase[index + OFFSET]
              = UNTAINTED;
}
```

Figure 19: Optimized original and shadow array initializes

The figure 20 illustrates a program reading user input into a buffer. Figure 21 shows the modified original program along with its shadow counter part.

```
for(int I = 0; read(fd,&ch,1); i++) {
      buffer [i] = ch;
}
```

Figure 20: File reader loop

```
Global * arrayBase;
Global stride;
Global iterations = 0;
Global done = false;
```

Original Program

```
arrayBase = buffer;
stride = 1;
for( int i = 0; read(fd,&ch,1); i++ ){
    buffer[i] = ch;
    iterations++;
}
done = true;
while(done);
```

Shadow Program

```
shadowIterations = 0;
for(;;){
      shadowIterations++;
      while(shadowIterations
            >= iterations ){
                  if(done)
                        break;
      }
      if(shadowIterations >
         iterations){
            shadowIterations = 0;
            iterations = 0;
            done = false;
            break;
      }
      index = (shadowIterations - 1 )
               * stride

      arrayBase[OFFSET + index]
         = TAINTED;
}
```

Figure 21: Optimized original and shadow file readers

3.8 Packaging the self-protecting binary

After being modified as described above, a program is packaged into a single program, and assembled back into x86-executable using PLTO's binary-rewriter. The modified program can be loaded and executed in just the same way as the original program.

3.9 Prototype limitations

In our current implementation, if the main process forks a child (using a system call like Unix fork()), we cannot track taint in the child process. This is because the child process will not have the shadow thread spawned. This is more a limitation of the `pthread` library than our implementation. With a different thread library we can solve this problem. Alternatively, with the `pthread` library, we can spawn a shadow thread by carefully setting up its stack in the child process.

We don't support multi-threaded applications in our prototype implementation; we can extend it to support multi-threaded applications by having a shadow thread for each thread in the original program and a global `TARGET` variable for each pair of original and shadow thread. We would also need more synchronization to ensure that taint tags are accurate when potential vulnerabilities are checked.

CHAPTER 4

4 SECURITY EVALUATION

In this chapter we discuss implications of our approach with respect to vulnerability detection. There are two aspects to the security guaranties. First, our approach should catch vulnerabilities; second, it should not trigger false alarms.

4.1 Analysis of false negatives

An exploit that goes undetected is regarded as a false negative. While, a false negative rate of 0% is desirable, in practice a false negative rate should be as small as possible. Under our scheme, we mark all data external to the program as tainted. We track copy propagation, arithmetic propagation, and address propagation of tainted data from source to destination. We guard every indirect control transfer, i.e, control transfer through return address, function pointers, and indirect jumps. With these policies, we catch all attacks that alter jump targets. Most attacks are control attacks, namely hijacking the control flow of victim programs. This is the final step that attackers follow to break into system and this is also the final line of defense before a system is compromised. Because we guard the control flow, we are assured to defend all control hijacks that happen either due to stack smash or heap smash. Further, we check the function call arguments of format string handling functions and never let a user-given string contain format specifiers. Thus we assure catching format string attacks. We can detect tainted arguments to sensitive system calls in the same way.

A less frequent type of attack is a non-control-data attack [7]. Non-control-data attacks corrupt a variety of application data including user identity data, configuration data, user input data, and decision-making data. Examples include random memory bit-flips in applications that can lead to serious security compromises in network servers, and hardware faults that can subvert an RSA (Ron Rivest, Adi Shamir and Len Adleman) implementation. We cannot detect such an attack in our current scheme. Also we do not track control-based taint propagation, but we are unaware of any program exploiting this kind of vulnerability. An attacker can exploit a non control vulnerability to generate a segmentation fault and crash the system. We cannot detect such denial of service attacks.

One more critical issue is the protection of shadow memory. Since both the original thread and the shadow thread are in the same virtual address space, the original program can possibly read from or write to shadow memory by generating a random address in the shadow memory region. In the unlikely event of the original program writing to shadow memory, we could lose the taint values. Again, see the future work section for suggested solutions.

While in our default policy we guard the `printf` class of functions to catch format string vulnerabilities, we cannot catch a format string vulnerability if the user writes a custom format string handler. In such cases we have provided a provision for the user to register all format string handling functions, so that our taint analyzer can guard all user written format string handlers too.

4.2 Analysis of false positives

While our scheme has not shown any false positives in the applications we have tested, we claim the possibility of false positives in the following scenarios:

i) **Intentionally passing user format specifiers to format string handlers:** If a program is written to pass a user-specified formats to a format string handling function, the best option is to modify the program since it is definitely vulnerable. However if that is the design of the program, we have flexibility to configure our tool to not track the format handling functions.

ii) **Intentionally passing user strings to sensitive system calls:** A user given input may become an argument to sensitive system calls like `execve()`, which our taint analyzer flags as error. If it is intended by the programmer to pass user argument to such functions we have flexibility in our tool to configure it. Default option is to not taint check any system call arguments.

iii) **Intentionally executing user injected code:** We do not allow a program to execute user injected code even if it is by design. We do not expect any commodity software to give its user a privilege to execute arbitrary code.

4.3 Security evaluation

To corroborate our claims, we show a wide variety of exploits that our taint analyzer can detect both in synthetic and commodity software.

4.3.1 Security evaluation on synthetic micro-benchmarks

We carried out various buffer overwrite and format string attack experiments by writing small programs with vulnerabilities. The important ones are listed below

4.3.1.1 Detecting old base pointer overwrite

We wrote a small program that reads data from a file into a buffer on the stack. The data overwrites the old stack base pointer. Just before function return, the original program waits for the shadow thread, and the shadow thread confirms overwrite and flags an error.

4.3.1.2 Detecting return address overwrite

Similarly, we wrote a small program that reads data from a file into a buffer on the stack. The data overwrites the return address. Just before function return, the original program waits for the shadow thread, and the shadow thread confirms overwrite and flags an error.

4.3.1.3 Detecting function pointer overwrites on stack

To confirm detection of function pointer overwrites on the stack, we wrote a program that reads data from a file and overwrites a stack buffer resulting in overwriting a function pointer located on the stack. When the program called a function using the function pointer present on the stack, it waits for the shadow thread to verify the sanity of the target address, at which time the shadow thread flagged an error due to a tainted function pointer.

4.3.1.4 Detecting function pointer overwrites on heap

To confirm the detection of function pointer overwrites on the heap, we wrote a program that reads data from a file into a buffer on the heap. However, the data overwrote the allocated memory on heap, and overran a function pointer present on the heap. When the program called a function using the function pointer present on heap, it waited for the shadow thread to verify the sanity of the target address, at which time the shadow thread flagged an error due to a tainted function pointer.

4.3.1.5 Detecting format string attacks

To check format string vulnerability, we wrote a program with vulnerable use of `printf()`. The program read user input and passed it to `printf`(). Our taint analyzer promptly detected the exploit.

Further, we generalized all the above examples by propagating the tainted data by copy and by arithmetic and logic operations. In all cases, we were able to detect exploits of the above mentioned types.

4.3.2 Security evaluation on commodity software

To confirm the efficacy of our approach, we ran a number of programs with known vulnerabilities using our taint analyzer, and we were able to catch the vulnerabilities without incurring any false positives. The following six sub-sections discuss each of them in detail.

4.3.2.1 ATPhttpd server

`ATPhttpd` is a small web server designed for high-performance. The `ATPhttpd` web server version 0.4b has well known buffer overflow vulnerabilities. The problem occurs due to insufficient bounds checking when handling GET requests. As a result, an attacker can issue a GET request with an extremely long file name and can overrun the bounds of an internal memory buffer and overwrite the return address of the function `http_send_error()`. When the function `http_send_error()` returns, the attacker effectively controls the flow of execution. With our taint analyzer we detected the return address overwrite and prevented the attack. For other normal requests, we did not generate any false alarms.

4.3.2.2 Passlogd daemon

`Passlogd` is a sniffer tool for capturing `syslog` messages in transit. Version `0.1c` of `passlogd` has a buffer overflow vulnerability in function `sl_parse()`. The vulnerability is caused by multiple boundary errors in the parser. This can be exploited to execute arbitrary code with root privileges on a vulnerable system by constructing a specially crafted network packet. The exploit program sends a packet that causes the `passlogd` parser to overwrite the return address of function `sl_parse()`. On return from this function, control jumps to an injected shell code. When this program was run with our taint analyzer, we correctly detected the return address overwrite attack. For other innocuous packets we did not generate any false positives.

4.3.2.3 BSD Talkd daemon

`Talkd` is a client-server application shipped with many Unix variants that is used for user communication between hosts on a network. The version of `talkd` that ships with OpenBSD 2.7 and older has a format string vulnerability. When a talk client connects to a talk server and requests communication with a user, `talkd` (the server program) will check to see whether the user is accepting messages. If so, it will print a message to the user’s terminal telling them that `"username@hostname"` wants to chat with them. This is done via a `fprintf()` function, which happens to have passed to it client-supplied data as part of the format string. The `fprintf()` call, in `announce.c`, uses as its format string argument the caller's username and the remote host. The caller's username is provided in the datagram sent by the client. It is thus possible for an attacker to modify a talk client so that a username value containing a malicious format specifier(s) is sent and overwrites memory on the remote server process' stack. It is thus possible to execute arbitrary code remotely, leading to a root compromise. We wrote a talk client program that exploits this vulnerability by passing a username containing the `%n` format specifier. When we ran `talkd` with our taint analyzer and communicated with it using the exploit program, `talkd` detected the tainted format string and stopped the exploit. For all other valid inputs, `talkd` daemon did not cause any false positives.

4.3.2.4 Chpass application using Libutil

The BSD `libutil` that ships with OpenBSD 2.7 or earlier contains a format string vulnerability in the `pw_error()` function, which is used in the `setuid chpass` utility. In `pw_error()`, user input is passed as the only argument to a `printf()` function, making it possible for an attacker to corrupt the stack. If format specifiers are deliberately inserted into the environment variable `EDITOR`, it causes part of string that is passed to the `printf`() function to be tainted. The tainted string will cause the `printf()` function to reference locations deeper into the stack than it should, expecting to see variables that would normally be there. This, combined with the fact that you can write to memory with format specifiers, allows an attacker to, for example, reconstruct the return address of the function so that it points to user-supplied (in the format string) shell code. When the function returns, it will begin executing the shell code on the stack.

A unique feature of this vulnerability is that the tainted string does not originate from external data, but instead from the environmental variables that are set up during program startup.

We set the environmental variable `EDITOR` to contain format specifier `%n` to exploit this vulnerability. When run under our taint analyzer we were able to detect the format string vulnerability and stop the exploit.

4.3.2.5 Libtiff library

`LibTIFF` is a library used to encode and decode images in Tag Image File Format (TIFF). Multiple LibTIFF routines in version `3.5.4` contain heap buffer overflow vulnerabilities in the following functions:

- `NeXTDecode` (in libtiff/tif_next.c)
- `ThunderDecode` (in libtiff/tif_thunder.c)
- `LogL16Decode` (in libtiff/tif_luv.c)

These issues are the result of insufficient validation of user-supplied data. Consequently, a remote attacker may be able to exploit these vulnerabilities by supplying an application using `LibTIFF` with a specially crafted TIFF image.

To detect the vulnerability in `libtiff`, we wrote an application to use the `NeXTDecode()` function to decode a strip of a TIFF image. The application passed a heap allocated buffer to `NeXTDecode()`. We also allocated a function pointer on the heap. To this application, we passed a buffer read from a specially crafted TIFF image that wrongly specified the size of the strip, thereby making `NeXTDecode()` overflow the heap buffer and overwrite the heap allocated function pointer. Later, when the program tried to call the function using the tainted function pointer, it was detected as an exploit by our taint analyzer. For other non-malicious inputs, `NeXTDecode()` was able to decode without any false positives.

4.3.2.6 Cfinger Daemon

`Cfingerd` is a configurable Finger daemon. `Cfingerd` version `1.4.3` (and earlier) has format string vulnerabilities that allow remote users to attain root privileges and execute arbitrary code. `Cfingerd` queries and logs the remote username of users of the service. If an attacker sets up a remote machine that returns specific format strings instead of a valid username, and connects to `cfingerd` from that machine, he can exploit the format string vulnerability. Because `Cfingerd` runs as root, attacker can gain full control of the `Cfingerd` host. We were able to successfully detect an exploit that tried to send a tainted username to `syslog` function of `Cfingerd`.

Table 2, shows different vulnerabilities detected by our taint analyzer.

Application	**Version**	**Vulnerability**	**Detected**	**False positives**
ATPHttpd	0.4b	Stack-based buffer overflow	YES	None
Passlogd	0.1c	Stack-based buffer overflow	YES	None
BSD Talkd	BSD 2.7	Format String	YES	None
BSD Chpass	BSD 2.7	Format String	YES	None
LibTiff library	3.5.4	Heap-based buffer overflow	YES	None
Cfingerd	1.4.3	Format String	YES	None

Table 2: Various kinds of vulnerabilities detected by our taint analyzer

CHAPTER 5

5 PERFORMACE EVALUATION

5.1 *Experimental setup*

We conducted a number of performance tests on real applications to evaluate the slowdown caused by taint checking. We conducted all our experiments on a 32-bit Intel x86 T2050 dual core machine with a 1.60 GHz clock and, 1GB of memory running Linux Fedora Core 5. We built the applications by linking them statically. All the comparisons given in the next section are on statically linked binaries. We evaluated the performance of the programs discussed in Section 4.3.2. However, we do not report the performance of `talkd` and `chpass`, because they are interactive applications involving multiple processes and user interactions. We have also evaluated `Gzip`, which is one of the CPU SPEC (Standard Performance Evaluation Corporation)'s integer benchmarks to evaluate performance in the case of CPU-bound applications.

5.2 *Performance results*

5.2.1 Atphttpd server

We evaluate the performance of the `ATPhttpd` server to measure the time to serve requests for varying file files. Our results are shown in Table 3 and Figure 22. The results show that the server runs about only 50% slower than when the server is run without instrumentation. We observe performance enhancement with the increased file size. This is attributed to the fact that for larger files, the program spends more time in a file reading loop that gets optimized by our optimization techniques discussed in Section 3.7.3. One more reason for better performance for large files is due to the fact that the server spends more time in a system call, writing data to the client socket.

HTML PAGE SIZE	**1KB**	**10KB**	**20KB**	**30KB**
Time in original execution (in sec)	0.01	0.033333	0.04275	0.0678
Time in instrumented program (in sec)	0.040333	0.0594	0.0765	0.101
Slowdown (number of times)	4.033333	1.782	1.789474	1.489676

Table 3: Performance comparison of original and instrumented ATPHttpd server

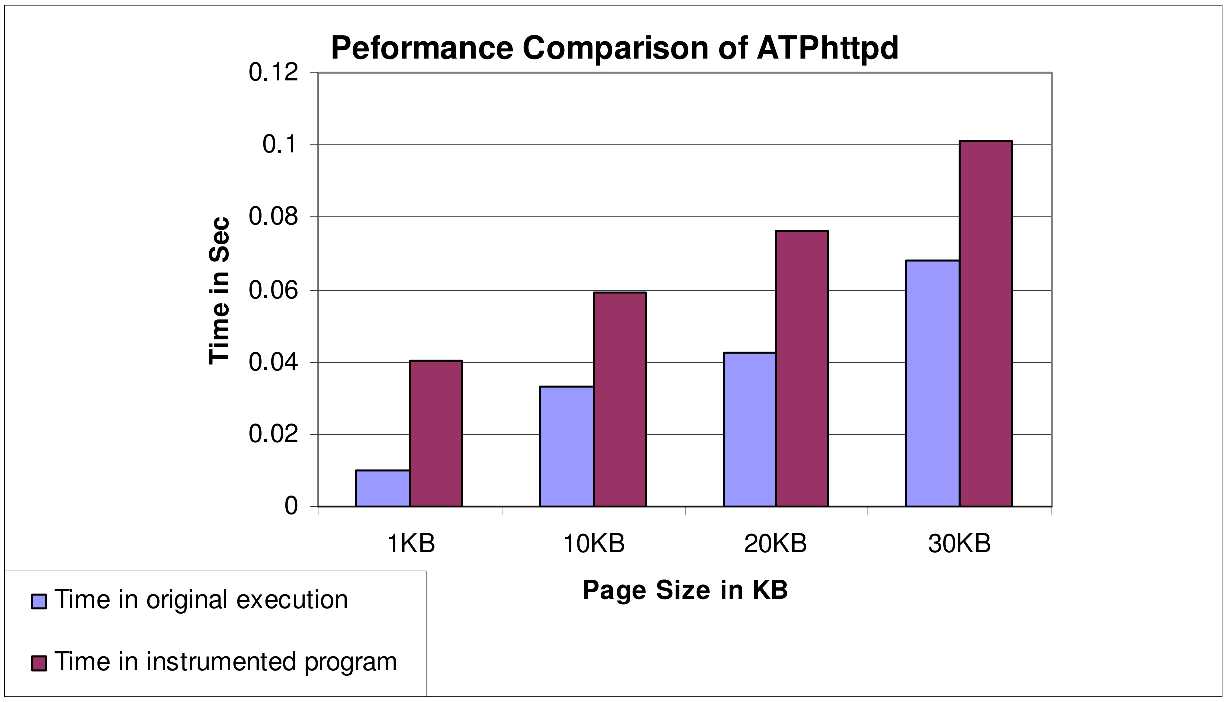


Figure 22: Performance of original Vs. instrumented ATPHttpd server

We also tested the effectiveness of our before-mentioned optimizations (see Section 3.7) on ATPHttpd server for a 30K page fetch. Table 4 and Figure 23 show the effectiveness of each optimization. Not tracing functions that exhibit the properties discussed in Section 3.7.2 reduces the slowdown from 6.3 times to 5.2 times. Relaxing return address check as discussed in Section 3.7.1 brings down the slowdown from 6.3 times to 4.7 times. Optimizing a high frequency file reading loop reduces the slowdown from 6.7 to 2.7 times. When we apply all these optimizations we see a slowdown of just 48%.

Type of computation	Time in sec	Slowdown
Original computation	0.0678	1
Taint analyzer with no optimization	0.4273333	6.302852
Taint analyzer with some functions not traced	0.3536667	5.216323
Taint analyzer with relaxed return address checks	0.3196667	4.714848
Taint analyzer with optimized high frequency loop	0.1866667	2.753196
Taint analyzer with all optimizations	0.101	1.489676

Table 4: Effect of individual optimization on ATPHttpd for 30K page

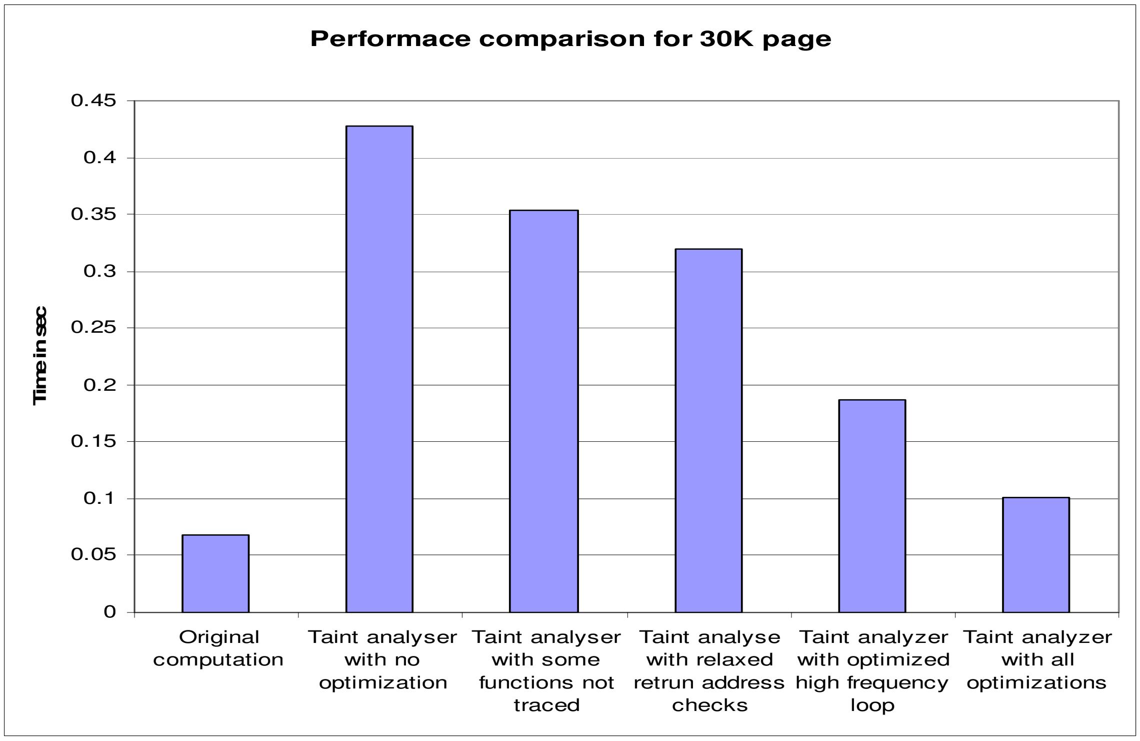


Figure 23: Effect of individual optimization on ATPHttpd for 30K page

5.2.2 Passlogd daemon

We ran the `Passlogd` sniffer tool during an ftp download of size 1.25K that generated 14 packets on the monitored network interface. For each of them we measured the processing time taken by `Passlogd` when run under our taint analyzer. We obtained a slow down of about 1.75 times compared to the non-instrumented original application.

5.2.3 Gzip application

We instrumented the widely used compression tool Gzip and evaluated the performance overhead on several file sizes. Table 5 shows the slowdown due to our taint analyzer. Gzip, which is also one of the CPU SPEC2000 benchmarks, shows better performance on large files. While the worst performance is 2.6 times slowdown for 400KB file, the best is less than 5% slowdown for a 65MB file. The average slowdown is less than 40%. Figure 24 shows the graphical comparison of performance in original Gzip versus the taint analyzer version.

As in the case of ATPHttpd, with the increased file size we see better performance, which we attribute to the fact that for larger files, longer time is spent in file reading and writing system calls. Also, code of $P_{original}$ and P_{shadow} are not clustered in the program memory, i.e, a function $F_{original} \in P_{original}$ is not located close to $F_{shadow} \in P_{shadow.}$ But, every time when the original program is executing $F_{original}$, the shadow thread is executing $F_{shadow.}$ This lack of spatial locality leads to higher amount of page faults for the short-lived programs. However, for programs, which run for longer time, the temporal locality overshadows slowdown due to lack of spatial locality and the performance is improved. See Chapter 6 for optimizations.

Input file size	**400KB**	**1.2MB**	**2.4MB**	**13.6MB**	**65.4MB**
Time in original Gzip program (in sec)	0.042	0.08425	0.153333	2.166	8.012333
Time in instrumented Gzip program (in sec)	0.111429	0.138833	0.216	2.289167	8.3155
Slowdown (number of times)	2.653061	1.647873	1.408696	1.056864	1.037838

Table 5: Performance comparison of original and instrumented Gzip

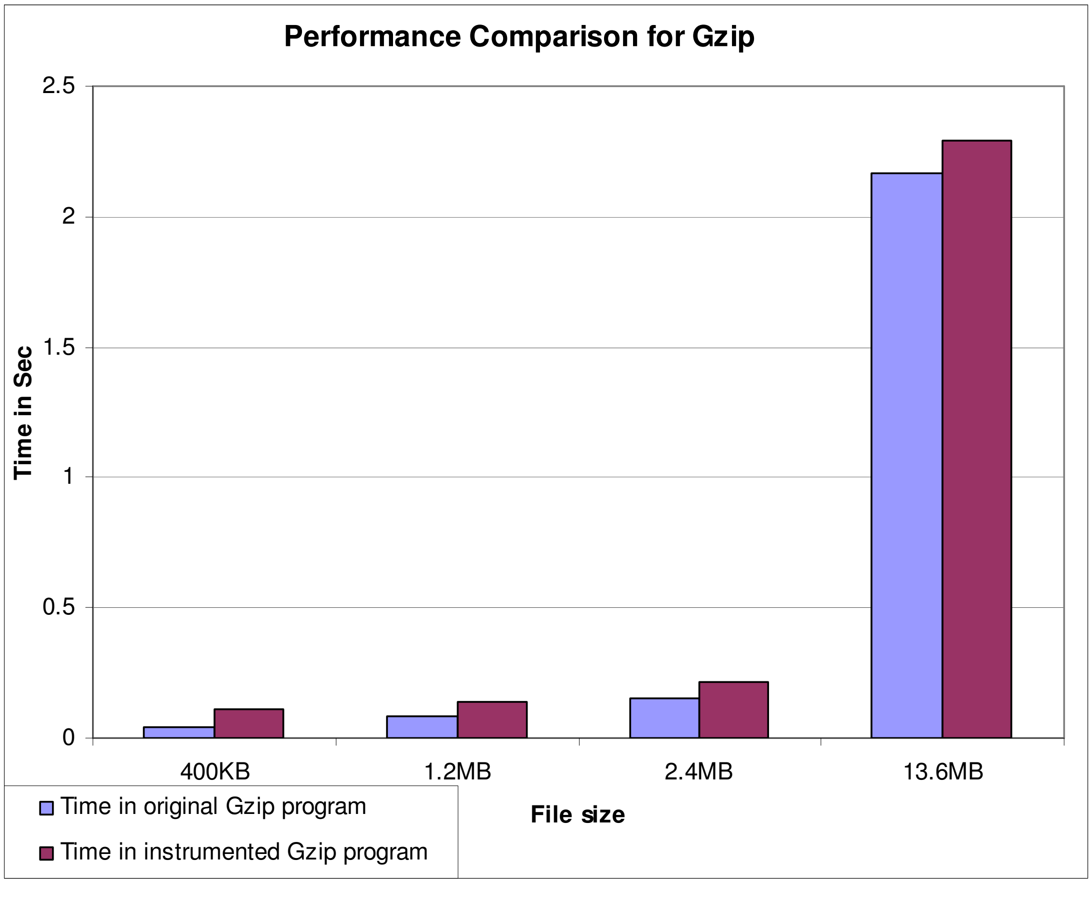


Figure 24: Performance of original Vs. instrumented Gzip

5.2.4 Libtiff library

We evaluated the performance on `Libtiff` library by measuring the time taken for `DecodeNext()` function discussed in Section 4.3.2.5. For an input containing 10 strips of 10KB size each (100KB total), we incurred a performance slowdown of 15 times. This is a higher overhead compared to other benchmarks, however it is explainable by the fact that in the entire program that we wrote, we just call `DecodeNext()` which spends most of its time copying data from one buffer to the other. This leaves us with little scope for optimizations. However, with the increase in the file size we started to see performance improvements. It is attributed to the fact that the function `DecodeNext()` has an innocuous loop as shown below that gets optimized.

```
for (op = buf, cc = occ; cc-- > 0;)
     *op++ = 0xff;
```

For a 100K file, the above loop gets executed 102400 times and the data-copy loop gets executed 10 times (100K file / 10K strip = 10 times). Hence the overhead is in the initialization loop.

Once we optimize this loop, the next bottleneck is the data-copying loop. For a 2MB file, the copy-loop gets executed 205 times. For a 10MB file , the copy loop gets executed 1024 times. However we cannot optimize the data-copying loop since it is a taint propagating part of the program. Table 6 and Figure 25 show the performance of `Libtiff`.

Image file size	**100KB**	**2MB**	**10MB**
Original program	0.002	0.0218	0.07
Instrumented Program	0.0306	0.2016	0.7378
Slowdown (number of times)	15.3	9.247706	10.54

Table 6: Performance comparison of original and instrumented Libtiff

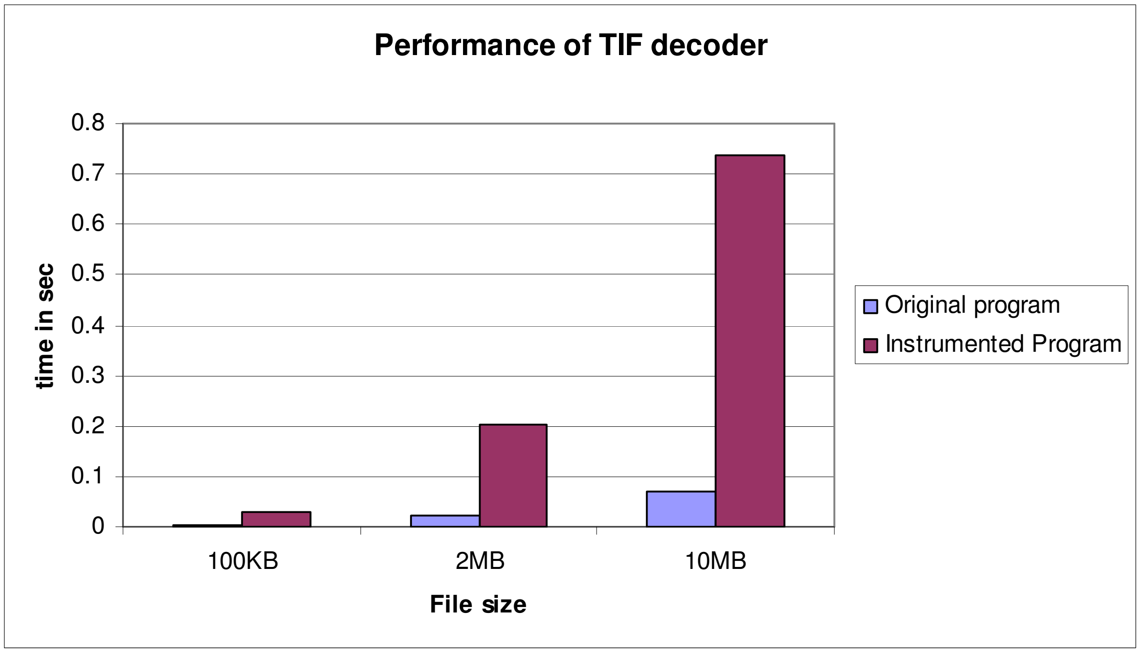


Figure 25: Performance of original Vs. instrumented Libtiff

5.2.5 Cfinger Daemon

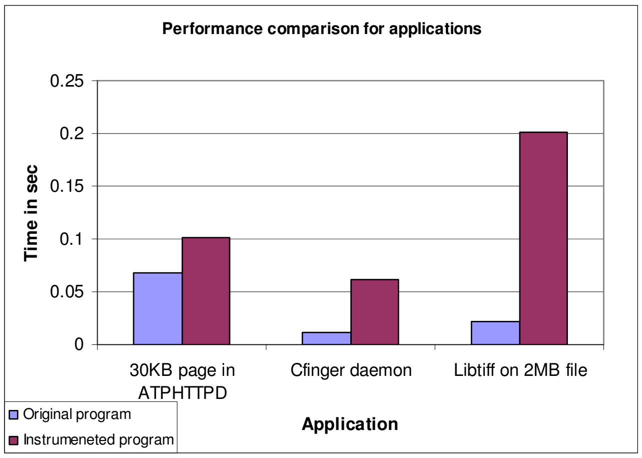
Finally, we evaluated the performance on `Cfingerd`. `Cfingerd` runs under `inetd` and hence it is invoked on each `finger` call. When run without any instrumentation, a finger request was completed in 0.011 seconds. However, when we ran it with our taint analyzer, it took 0.103 seconds to complete. Thus we observed a slowdown of 9.26 times. On applying loop optimization to a string copying loop, the execution time reduced to 0.061 seconds, which is a slowdown of 5.5 times.

Table 7 shows performance of 30K page on `ATPHttpd` server, `Cfinger` daemon, 1.25K page sniffer on `Passlogd`, 13.6MB file compression with `Gzip` and 2M file with `Libtiff` on instrumented applications in comparison with the original applications.

Application	**30KB page in ATPHTTPD**	**Cfinger daemon**	**Libtiff on 2MB file**	**Passlogd with 1.25KB download**	**Gzip with 13.6MB file**
Original program	0.0678	0.011	0.0218	2.45	2.166
Instrumented program	0.101	0.061	0.2016	4.301133333	2.289167
Slowdown (number of times)	1.489675516	5.54	9.247706422	1.755564626	1.056864

Table 7: Effect of our taint analyzer on the performance of different applications

Figure 26(a) shows performance comparisons for the short-lived applications: `ATPHttpd`, `Cfingerd` and `Libtiff`. Figure 26(b) shows performance comparisons for the long-running applications: `Gzip` and `Passlogd`.



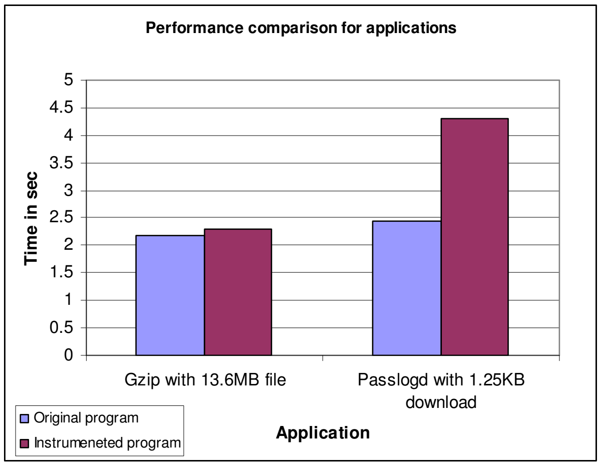


Figure 26(a): Short-lived applications

Figure 26(b): Long-running applications

Effect of our taint analyzer on the performance of different applications

CHAPTER 6

6 CONCLUSIONS AND FUTURE WORK

In this thesis we have shown that vulnerability detection using multicore machines has a promising future. We have made a successful effort to gather static program analysis to get better performance in dynamic taint analysis on the future hardware. With many cores being the future of hardware, we can always afford to dedicate some cores for taint analysis. This brings a new paradigm of secure and efficient computing. In our work we have shown that taint analysis with multithreading using multicore machines overshadows many state-of-art dynamic taint analysis techniques in terms of performance and at the same time provides as much if not more concrete security guarantees as any other. On some benchmarks (like Gzip) our performance is better than LIFT [19], the best performing taint analyzer at the time of writing this thesis.

Our experiments on real world applications including a web server, a chat server, an image decoder, and a password changing application show effectiveness in detecting exploits. Also our performance analyses on an I/O-bound web server and CPU-bound compression tool show that we incur significantly less overhead in comparison with the current state-of-art taint analyzers.

We plan to extend and improve our work in two major directions: improving runtime performance and making more security guarantees.

6.1 Performance enhancement

In the current work we do loop optimization using manual techniques. Also we can optimize only simple loops, i.e loops without nested control flow structure. In our future work we intend to do automatic loop optimization using data and control flow analysis. Also we will optimize many more loops including the ones that involve nested control structures. With most programs spending about 80% of their time in loops, loop optimization has high performance impact.

Currently we do not guard function returns in special cases (see Section 3.7.1). We can improve this by extending to functions that do not have any arrays on the stack. Also, currently we do not trace some functions (see Section 3.7.2). We can safely extend this to functions that only do store operations on local variables (non-pointer) on the current stack frame.

Currently we synchronize the two threads at each basic block. We plan to reduce this synchronization by using a circular buffer of targets, so that threads synchronize only at program exploit points.

In our current implementation, we layout all code belonging to $P_{original}$ in one contiguous memory and all code belonging to P_{shadow} in the other. However, when the original program is executing a function $F_{original} \in P_{original,}$ the shadow thread is executing $F_{shadow} \in P_{shadow.}$ We can utilize this spatial locality and layout code such that $F_{original}$ is always adjacent to

F_{shadow}. This optimization reduces page faults and improves performance for short-lived programs.

6.2 Security enhancement

In the current work, we have a problem if the original program generates an address that accesses the shadow memory. With kernel level thread support, we could solve this problem by guarding the address range accessed by each thread.

We plan to handle control based taint propagation by means of a tainted program counter [17]. Currently we do not handle denial of service attacks; we intend to tackle this issue as well in our future work.

APPENDIX A: Sample code

1. Sample x86 code for fall-through block:

Input code:

```
8804a116 <__pthread_setschedparam$$z0515>:
8804a116:   83 c4 10                add    $0x10,%esp

8804a119 <__pthread_setschedparam$$z26747>:
8804a119:   31 c0                   xor    %eax,%eax
8804a11b:   eb a9                   jmp    8804a0c6
<__pthread_setschedparam$$z26748>
```

Original + Instrumented:

```
880a78c2 <__pthread_setschedparam_clone$$z32031>:
880a78c2:   9c                      pushf
880a78c3:   c7 05 c4 80 04 08 fa    movl   $0x881ef7fa,0x80480c4
880a78ca:   f7 1e 88

880a78cd <__pthread_setschedparam_clone$$z153978>:
880a78cd:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
880a78d4:   00 00 00
880a78d7:   75 f4                   jne    880a78cd
<__pthread_setschedparam_clone$$z153978>

880a78d9 <__pthread_setschedparam_clone$$z153977>:
880a78d9:   9d                      popf
880a78da:   83 c4 10                add    $0x10,%esp

880a78dd <__pthread_setschedparam_clone$$z32032>:
880a78dd:   9c                      pushf
880a78de:   c7 05 c4 80 04 08 23    movl   $0x881ef823,0x80480c4
880a78e5:   f8 1e 88

880a78e8 <__pthread_setschedparam_clone$$z153980>:
880a78e8:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
880a78ef:   00 00 00
880a78f2:   75 f4                   jne    880a78e8
<__pthread_setschedparam_clone$$z153980>

880a78f4 <__pthread_setschedparam_clone$$z153979>:
880a78f4:   9d                      popf
880a78f5:   31 c0                   xor    %eax,%eax
880a78f7:   e9 7e fe ff ff          jmp    880a777a
<__pthread_setschedparam_clone$$z32023>
```

Shadow code:

```
881ef7fa <__pthread_setschedparam_clone_clone$$z63387>:
881ef7fa:   9c                      pushf

881ef7fb <__pthread_setschedparam_clone_clone$$z95168>:
881ef7fb:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
881ef802:   00 00 00
881ef805:   74 f4                   je     881ef7fb
<__pthread_setschedparam_clone_clone$$z95168>

881ef807 <__pthread_setschedparam_clone_clone$$z95167>:
```

```
881ef807:   50                      push   %eax
881ef808:   8b 05 c4 80 04 08       mov    0x80480c4,%eax
881ef80e:   89 05 c8 80 04 08       mov    %eax,0x80480c8
881ef814:   58                      pop    %eax
881ef815:   c7 05 c4 80 04 08 00    movl   $0x0,0x80480c4
881ef81c:   00 00 00
881ef81f:   9d                      popf
881ef820:   83 c4 10                add    $0x10,%esp

881ef823 <__pthread_setschedparam_clone_clone$$z63388>:
881ef823:   9c                      pushf

881ef824 <__pthread_setschedparam_clone_clone$$z95170>:
881ef824:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
881ef82b:   00 00 00
881ef82e:   74 f4                   je     881ef824
<__pthread_setschedparam_clone_clone$$z95170>

881ef830 <__pthread_setschedparam_clone_clone$$z95169>:
881ef830:   50                      push   %eax
881ef831:   8b 05 c4 80 04 08       mov    0x80480c4,%eax
881ef837:   89 05 c8 80 04 08       mov    %eax,0x80480c8
881ef83d:   58                      pop    %eax
881ef83e:   c7 05 c4 80 04 08 00    movl   $0x0,0x80480c4
881ef845:   00 00 00
881ef848:   9d                      popf
881ef849:   09 c0                   or     %eax,%eax
881ef84b:   e9 8c fd ff ff          jmp    881ef5dc
<__pthread_setschedparam_clone_clone$$z63379>
```

2. Sample x86 code for conditional branch block:

Input code:

```
88048598 <http_send_file$$z0071>:
88048598:   83 c4 10                add    $0x10,%esp
8804859b:   89 45 ec                mov    %eax,0xffffffec(%ebp)
8804859e:   85 c0                   test   %eax,%eax
880485a0:   0f 88 59 01 00 00       js     880486ff <http_send_file$$z0088>

880485a6 <http_send_file$$z0072>:
880485a6:   50                      push   %eax
880485a7:   ff 75 08                pushl  0x8(%ebp)
880485aa:   ff 75 0c                pushl  0xc(%ebp)
880485ad:   6a ff                   push   $0xffffffff
 ....................

880486ff <http_send_file$$z0088>:
880486ff:   83 ec 0c                sub    $0xc,%esp
88048702:   ff 75 08                pushl  0x8(%ebp)
88048705:   68 48 c7 42 88          push   $0x8842c748
8804870a:   6a 00                   push   $0x0
8804870c:   68 5b c7 42 88          push   $0x8842c75b
88048711:   68 93 01 00 00          push   $0x193
88048716:   eb da                   jmp    880486f2 <http_send_file$$z26704>
```

Original + Instrumented:

```
880a358f <http_send_file_clone$$z31556>:
880a358f:   9c                      pushf
880a3590:   c7 05 c4 80 04 08 51    movl   $0x881e9351,0x80480c4
880a3597:   93 1e 88
```

```
880a359a <http_send_file_clone$$z153150>:
880a359a:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
880a35a1:   00 00 00
880a35a4:   75 f4                   jne    880a359a
<http_send_file_clone$$z153150>

880a35a6 <http_send_file_clone$$z153149>:
880a35a6:   9d                      popf
880a35a7:   83 c4 10                add    $0x10,%esp
880a35aa:   89 45 ec                mov    %eax,0xffffffec(%ebp)
880a35ad:   85 c0                   test   %eax,%eax
880a35af:   0f 88 d6 03 00 00       js     880a398b
<http_send_file_clone$$z31576>

880a35b5 <http_send_file_clone$$z31557>:
880a35b5:   9c                      pushf
880a35b6:   c7 05 c4 80 04 08 8d    movl   $0x881e938d,0x80480c4
880a35bd:   93 1e 88

880a35c0 <http_send_file_clone$$z153152>:
880a35c0:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
880a35c7:   00 00 00
880a35ca:   75 f4                   jne    880a35c0
<http_send_file_clone$$z153152>

880a35cc <http_send_file_clone$$z153151>:
880a35cc:   9d                      popf
880a35cd:   50                      push   %eax
880a35ce:   ff 75 08                pushl  0x8(%ebp)
880a35d1:   ff 75 0c                pushl  0xc(%ebp)
880a35d4:   6a ff                   push   $0xffffffff

………………

880a398b <http_send_file_clone$$z31576>:
880a398b:   9c                      pushf
880a398c:   c7 05 c4 80 04 08 4f    movl   $0x881e994f,0x80480c4
880a3993:   99 1e 88

880a3996 <http_send_file_clone$$z153190>:
880a3996:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
880a399d:   00 00 00
880a39a0:   75 f4                   jne    880a3996
<http_send_file_clone$$z153190>

880a39a2 <http_send_file_clone$$z153189>:
880a39a2:   9d                      popf
880a39a3:   83 ec 0c                sub    $0xc,%esp
880a39a6:   ff 75 08                pushl  0x8(%ebp)
880a39a9:   68 48 c7 42 88          push   $0x8842c748
880a39ae:   6a 00                   push   $0x0
880a39b0:   68 5b c7 42 88          push   $0x8842c75b
880a39b5:   68 93 01 00 00          push   $0x193
880a39ba:   eb 92                   jmp    880a394e
<http_send_file_clone$$z31574>
```

Shadow code:

```
881e9351 <http_send_file_clone_clone$$z62912>:
881e9351:   9c                      pushf

881e9352 <http_send_file_clone_clone$$z94338>:
```

```
881e9352:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
881e9359:   00 00 00
881e935c:   74 f4                   je     881e9352
<http_send_file_clone_clone$$z94338>

881e935e <http_send_file_clone_clone$$z94337>:
881e935e:   50                      push   %eax
881e935f:   8b 05 c4 80 04 08       mov    0x80480c4,%eax
881e9365:   89 05 c8 80 04 08       mov    %eax,0x80480c8
881e936b:   58                      pop    %eax
881e936c:   c7 05 c4 80 04 08 00    movl   $0x0,0x80480c4
881e9373:   00 00 00
881e9376:   9d                      popf
881e9377:   83 c4 10                add    $0x10,%esp
881e937a:   89 45 ec                mov    %eax,0xffffffec(%ebp)
881e937d:   81 3d c8 80 04 08 4f    cmpl   $0x881e994f,0x80480c8
881e9384:   99 1e 88
881e9387:   0f 84 c2 05 00 00       je     881e994f
<http_send_file_clone_clone$$z62932>

881e938d <http_send_file_clone_clone$$z62913>:
881e938d:   9c                      pushf

881e938e <http_send_file_clone_clone$$z94340>:
881e938e:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
881e9395:   00 00 00
881e9398:   74 f4                   je     881e938e
<http_send_file_clone_clone$$z94340>

881e939a <http_send_file_clone_clone$$z94339>:
881e939a:   50                      push   %eax
881e939b:   8b 05 c4 80 04 08       mov    0x80480c4,%eax
881e93a1:   89 05 c8 80 04 08       mov    %eax,0x80480c8
881e93a7:   58                      pop    %eax
881e93a8:   c7 05 c4 80 04 08 00    movl   $0x0,0x80480c4
881e93af:   00 00 00
881e93b2:   9d                      popf
881e93b3:   50                      push   %eax
881e93b4:   ff 75 08                pushl  0x8(%ebp)
881e93b7:   ff 75 0c                pushl  0xc(%ebp)
881e93ba:   6a 00                   push   $0x0

………………………

881e994f <http_send_file_clone_clone$$z62932>:
881e994f:   9c                      pushf

881e9950 <http_send_file_clone_clone$$z94378>:
881e9950:   81 3d c4 80 04 08 00    cmpl   $0x0,0x80480c4
881e9957:   00 00 00
881e995a:   74 f4                   je     881e9950
<http_send_file_clone_clone$$z94378>

881e995c <http_send_file_clone_clone$$z94377>:
881e995c:   50                      push   %eax
881e995d:   8b 05 c4 80 04 08       mov    0x80480c4,%eax
881e9963:   89 05 c8 80 04 08       mov    %eax,0x80480c8
881e9969:   58                      pop    %eax
881e996a:   c7 05 c4 80 04 08 00    movl   $0x0,0x80480c4
881e9971:   00 00 00
881e9974:   9d                      popf
881e9975:   83 ec 0c                sub    $0xc,%esp
881e9978:   ff 75 08                pushl  0x8(%ebp)
881e997b:   68 00 00 00 00          push   $0x0
```

```
881e9980:   6a 00                   push   $0x0
881e9982:   68 00 00 00 00          push   $0x0
881e9987:   68 00 00 00 00          push   $0x0
881e998c:   e9 68 ff ff ff          jmp    881e98f9
<http_send_file_clone_clone$$z62930>
```

REFERENCES

[1] Adobe Security Advisory: Acrobat and Adobe Reader plug-in buffer overflow, August 16th, 2005. http://www.adobe.com/cfusion/knowledgebase/index.cfm?id=321644

[2] CERT/CC Statistics 1988-2006, http://www.cert.org/stats/

[3] http://www.pcworld.com/article/id,131049-c,gpsreceivers/article.html

[4] Arash Baratloo, Navjot Singh, and Timothy Tsai. "Transparent Run-Time Defense Against Stack Smashing Attacks", in *Proceedings of 2000 USENIX Annual Technical Conference,* San Diego, California, 2000.

[5] Walter Chang and Calvin Lin. "Guarding Programs against Attacks with Dynamic Data Flow Analysis", in *7th Annual Austin CAS International Conference,* February, 2005.

[6] D. Brent Chapman. "Network (In)Security Through IP Packet Filtering", in *Proceedings of the Third USENIX UNIX Security Symposium,* pages 14-16, Baltimore, MD, September 1992.

[7] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. "Non-Control-Data Attacks Are realistic Threats", in *USENIX Security Symposium*, Baltimore, MD, August 2005.

[8] Tzi-Cker Chiueh and Fu-Hau Hsu. "RAD: A Compile-Time Solution to Buffer Overflow Attacks", in *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS),* Phoenix, Arizona, April 2001.

[9] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. "FormatGuard: Automatic Protection From printf Format String", in *Proceedings of the10th USENIX Security Symposium,* Washington, D.C., August 2001.

[10] Hiroaki Etoh and Kunikazu Yoda. "Protecting from stack-smashing attacks", IBM Research Division, Tokyo Research Laboratory, June 2000.[http://www.trl.ibm.com/projects/security/ssp/]

[11] Cova M Felmetsger, V Banks, and G Vigna. "Static Detection of Vulnerabilities in x86 Executables", in *22nd Annual Computer Security Applications Conference (ACSAC'06),* Shanghai, September 2006.

[12] Alex Ho, Michael Fetterman , Christopher Clark , Andrew Warfield , Steven Hand. "Practical taint-based protection using demand emulation", in *Proceedings of the 2006 EuroSys conference*, Leuven, Belgium, April 2006.

[13] Jingfei Kong, Cliff C Zou, and Huiyang Zhou. "Improving software security via runtime instruction-level taint checking", in *proceedings of the 1st workshop on Architectural and system support for improving software dependability,* San Jose, California, pages 18-24, 2006 .

[14] Zhenkai Liang , R. Sekar. "Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models", In *Proceedings of the 21st Annual Computer Security Applications Conference*, p.215-224, December 2005.

[15] Stuart Moore and SecurityGlobal.net. "Security Vulnerabilities", in *NCC-AIIM Seminar 2004 Presentations.* http://www.nccaiim.org/Education/Proceedings/2004/7-Moore-vulnerabilities.ppt

[16] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, and Dirk Grunwald. "Shadow Profiling: Hiding Instrumentation Costs with Parallelism", in *Proceedings of the International Symposium on Code Generation and Optimization (CGO),* March 2007.

[17] James Newsome and Dawn Song. "Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software", in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, San Diego, California, February 2005.

[18] Manish Prasad, and Tzi-cker Chiueh. "A Binary Rewriting Defense against Stack Based Overflow attacks", in *Proceedings of the USENIX. Annual Technical Conference,* San Antonio, TX, June 2003.

[19] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou and Youfeng Wu. "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks", in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture,* pages: 135-148, 2006.

[20] Mohan Rajagopalan, Matti A Hiltunen, Trevor Jim and Richard D Schlichting. "Authenticated System Calls", in *IEEE International Symposium on Dependable Systems and Networks,* June 2005.

[21] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. "PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture", in *Proceedings of 2001 Workshop on Binary Rewriting (WBT-2001),* September 2001.

[22] Weidong Shi, Hsien-Hsin S. Lee, Laura Falk and Mrinmoy Ghosh. “An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors”, in *Proceedings of the 33rd International Symposium on Computer Architecture,* pages 102-113, Boston, MA, June 2006.

[23] G. Edward Suh, Jae W. Lee , David Zhang , Srinivas Devadas. “Secure program execution via dynamic information flow tracking”. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, Boston, MA, October 2004.

[24] Cheng W, Qin Zhao, Bei Yu, and Hiroshige.S. “TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting”, in *11th IEEE Symposium on Computers and Communications (ISCC'06),* Pula-Cagliari, Sardinia, Italy, June 2006.

[25] P Wagle, and C Cowan. “Stackguard: Simple stack smash protection for GCC”, in *Proceedings of the GCC Developers Summit,* pages 243–256, 2003.