IMPROVING OPERATING SYSTEMS SECURITY: TWO CASE STUDIES

A Dissertation
Presented to
The Academic Faculty

by

Jinpeng Wei

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2009

IMPROVING OPERATNG SYSTEMS SECURITY: TWO CASE STUDIES

Approved by:

Dr. Calton Pu, Advisor
College of Computing
Georgia Institute of Technology

Dr. Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Kang Li
Department of Computer Science
University of Georgia

Dr. Jonathon Giffin
College of Computing
Georgia Institute of Technology

Date Approved: July 30, 2009

To my wife, Qiong, and my daughter, Ellen.

ACKNOWLEDGEMENTS

I wish to thank my advisor, Calton Pu, for his guidance and support. He led me into the realm of Computer Science research and taught me along the way the important skills for a mature researcher. And most importantly, he showed me how to enjoy research as a way of living and eventually I understood.

I wish to thank my wife, Qiong, who has given me continuous love, encouragement, and support over the years, especially the most difficult times during my Ph.D. study. Without her I may not finish my thesis.

I wish to thank my parents, who raised me and gave me the best education that they could think of, and kept watching my progress with their love and considerations.

I appreciate many professors that have directly or indirectly enlightened me, including Dr. Ling Liu, Dr. Mustaque Ahamad, Dr. Jonathon Giffin, Dr. Karsten Schwan, Dr. Alessandro Orso, and Dr. Douglas Blough at Georgia Tech and Dr. Kang Li at University of Georgia.

I also thank Jeffrey R. Jackson and John A. Wiegert at Intel Corporation who exposed me to virtualization technology (especially Xen), Carlos V. Rozas and Anand Rajan at Intel Corporation who inspired my work on kernel control flow integrity, and Dr. Xiaolan Zhang, Dr. Vasanth Bala, and Dr. Arun Iyengar at IBM T. J. Watson Research Center.

Finally, I thank my fellow Ph.D. students in the Distributed Data Intensive Systems Lab, especially Dr. Lenin Singaravelu, Dr. Lakshmish Ramaswamy, Dr. Li Xiong, Dr. Jianjun Zhang, Dr. Mudhakar Srivatsa, Dr. James Caverlee, Dr. Steve Webb,

Qinyi Wu, Younggyun Koh, and Danesh Irani. I feel grateful to have worked with and shared frustration and happiness with all of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

TOCTTOU	Time-Of-Check-To-Time-Of-Use
STIR	Soft Timer Interrupt Request

SUMMARY

Malicious attacks on computer systems attempt to *obtain* and *maintain* illicit control over the victim system. To obtain unauthorized access, they often exploit *vulnerabilities* in the victim system, and to maintain illicit control, they apply various *hiding* techniques to remain stealthy. In this dissertation, we discuss and present solutions for two classes of security problems: TOCTTOU (time-of-check-to-time-of-use) and K-Queue. TOCTTOU is a vulnerability that can be exploited to obtain unauthorized root access, and K-Queue is a hiding technique that can be used to maintain stealthy control of the victim kernel.

The first security problem is TOCTTOU, a race condition in Unix-style file systems in which an attacker exploits a small timing gap between a file system call that checks a condition and a use kernel call that depends on the condition. TOCTTOU vulnerabilities are widespread and cause serious consequences. For example, according to US-CERT (United States Computer Emergency Readiness Team), such vulnerabilities exist in a wide range of applications, affect many operating systems, and often give the attacker unauthorized root access. Our research contributions on TOCTTOU include: (1) A model that enumerates the complete set of potential TOCTTOU vulnerabilities (e.g., 224 TOCTTOU pairs in Linux); (2) A set of tools that detect TOCTTOU vulnerabilities in Linux applications such as *vi*, *gedit*, and *rpm*; (3) A theoretical as well as an experimental evaluation of security risks that shows that TOCTTOU vulnerabilities can no longer be considered "low risk" given the wide-scale deployment of multiprocessors; (4) An event-driven protection mechanism and its implementation in the Linux kernel that defend Linux applications against TOCTTOU attacks at low performance overhead.

The second security problem addressed in this dissertation is kernel queue or K-Queue, which represents a new hiding technique that can be used by the attacker to maintain stealthy control of the victim system after a successful break-in. K-Queue-driven attacks can achieve continual malicious function execution without persistently changing either kernel code or data (from the "gold" distribution), which prevents state-of-the-art kernel integrity monitors such as CFI and SBCFI from detecting them. We have studied a concrete instance of K-Queue-driven attacks that use the soft timer mechanism found in nearly all full-featured operating systems. We demonstrate that an attacker can use soft timer interrupt requests (STIRs) to perform powerful attacks, including key logging, denial of service, and hidden process scheduling. To defend against soft-timer-driven kernel control flow attacks, we propose and implement an approach based on an automated static analysis of the entire kernel that identifies and catalogs all legitimate STIRs in a database. At runtime, a reference monitor in a trusted virtual machine compares each pending STIR with STIRs in the database, allowing the execution of only known good STIRs. Our defensive technique effectively mitigates soft-timer-driven attacks at a low cost (less than 7% for each of our benchmarks).

As the finishing touch of this dissertation, we design and implement a solution to the general class of K-Queue-driven attacks which can exploit IRQ action queues, tasklet queues, soft timer queues, and work queues. Our first contribution is a unified static analysis framework and a set of tools that can generate specifications of K-Queue summary signatures and the corresponding checking code in an automated way. We also design and implement a unified runtime reference monitor based on virtualization that validates K-Queue invariants and guards such invariants against tampering. Finally, we

perform a comprehensive experimental evaluation of the scalability of our static analysis framework and tool set, which shows that different K-Queue analyzers have significant overlapping that can be exploited for better efficiency; and we carry out an evaluation of the complexity and runtime overhead of our K-Queue Checker which suggests ways for further optimization.

CHAPTER 1

INTRODUCTION

Operating systems are privileged programs that hold ultimate control over the computing assets (e.g., CPU, memory, network bandwidth, and files) of any computing system.

However, today's operating systems are not secure, because they contain numerous vulnerabilities. For example, Secunia[1] has reported 2,135 vulnerabilities for Microsoft Windows since 2003. Such vulnerabilities are bad for security because attackers (often called hackers) can exploit them to *obtain* illicit control over the victim operating system and thus access to, or control over, the computing assets managed by the operating system. The large number of vulnerabilities and the ease with which many of them can be exploited often increase the attacker's chance of success. For example, it has been reported that an unpatched Windows XP with SP1 [23] can be compromised within six minutes after installation.

The damage due to malicious compromise to an operating system has increased significantly in recent years. Traditional hackers exploit vulnerable systems mainly for showing off their technical skills. The game is over once the target system is penetrated, and they would like to be noticed. In some sense this style of attacking is good for security, because the damage is one time and remediation can be taken once the comproise is announced. However, most hackers today exploit vulnerabilities for monetary gains, so the real game begins *after* the target system is conquered, and the hackers would like this game to last forever. For example, once getting into a system, today's attackers often collect sensitive information (e.g., credit card numbers and trading

---

[1] http://secunia.com

secrets), install key loggers to steal passwords, or install other kinds of malicious software or malware, which includes spyware, rootkits, virus, worms, Trojans, and stealthy backdoors. Even worse, they can enlist the victim machine into botnets, large collections of compromised machines under the control of a bot master. The largest botnet to date contains more than eight million nodes [24]. These botnets are valuable for sending spam emails or mounting distributed DoS (Denial of Service) attacks, so they are often traded in underground black markets.

In other words, the attackers today are interested not in showing off but in the actual benefit of using the compromised computing system. To maximize their gains, they strive to *maintain* a stealthy control over the victim system. Unfortunately, remaining stealthy is not a tough job for the attacker, because today's mainstream operating systems have a monolithic privilege system – once the attacker becomes the *root* user, he/she can freely modify any state of the system, including the operating system itself, to hide his/her activities. This coarse-grained access control has significantly lowered our confidence on long-running systems, to the extent that most administrators are forced to completely re-install a computer system to regain trust if they suspect that a root compromise has happened. Unfortunately, it is a difficult decision to re-install a computing system in active use. So we are often forced to live with potentially compromised operating systems.

The thesis of this dissertation is to increase our trust on long-running computing systems by improving the security of operating systems. Obviously this is a big topic; therefore we approach it by two concrete case studies. Specifically, we discuss and present solutions for two classes of security problems: TOCTTOU (time-of-check-to-time-of-use) and K-Queue (Kernel Queue), in which TOCTTOU is a race condition vulnerability that can be exploited for privilege escalation and K-Queue is a kernel mechanism that can be misused to maintain stealthy control of the victim kernel. By these

cases studies, we hope to gain insights in how to systematically improve the runtime security of modern operating systems.

We choose TOCTTOU and K-Queue because both of them enable non-obtrusive, stealthy attacks that are of interest to today's hackers. Both of them touch fundamental system software design philosophies that are unfornately bad for security. TOCTTOU, an inherent design flaw (i.e., the lack of transactional support) in modern file systems, has been around for more than 30 years, yet such vulnerabilities continue to be discovered in widely-used applications and the adoption of multi-cores aggravates the security threat represented by such vulnerabilities. K-Queue reflects an inherent *lack* of fine-grained access control of the CPU inside a modern operating system kernel, which allows a malicious extension to easily inject illicit control flows into the kernel.

Our solutions to both problems are inspired by the concept of para-transactional invariants (PTIs). PTIs, runtime properties that remain true *throughout* the execution of a block of program statements, are a general and unifying concept for understanding and preserving runtime properties. We extract and express runtime properties whose violations are the root cause for TOCTTOU and K-Queue. By adding the missing logic into the vulnerable system to eliminate the root causes, we provide effective and efficient solutions to these security problems. Our work helps improve the security of modern operating systems.

1.1. Contributions

This dissertation makes the following contributions in the TOCTTOU problem.

- An abstract model that is capable of enumerating the complete set of potential TOCTTOU vulnerabilities (e.g., 224 TOCTTOU pairs in Linux).
- A systematic search for potential TOCTTOU vulnerabilites in Linux system utility programs, which reveals unknown TOCTTOU vulnerabilities in widely-used applications such as *vi*, *gedit*, and *rpm*.

- A detailed experimental and theoretical study of successfully exploiting TOCTTOU vulnerabilities in real-world applications, on both uniprocessors and multiprocessors, which significantly advances our understanding of TOCTTOU attacks.
- A modular and event-driven defense mechanism (EDGI) and its Linux implementation that defend applications against TOCTTOU attacks at low performance overhead and do not require existing applications to change.

In terms of the K-Queue problem, this dissertation makes the following contributions.

- A definition of K-Queue-driven transient control flow attacks as a new attack class that maintains stealthy control of the victim kernel, and an empirical study of attacks that leverages the soft timer queue.
- An authentication model that uses summary signatures to differentiate K-Queue requests from legitimate and malicious software.
- A unified static analysis framework and a set of tools that can generate specifications of K-Queue summary signatures and the corresponding checking code in an automated way.
- A runtime reference monitor based on virtualization that validates K-Queue invariants and guards such invariants against tampering, which effectively defends potential K-Queue-driven attacks.

1.2. Outline

The rest of this dissertation is organized as follows. Chapter 2 discusses our solution to the TOCTTOU (time-of-check-to-time-of-use) problem. Chapter 3 presents our solution to K-Queue driven transient control flow attacks. Chapter 4 discusses future work and draws the conclusion.

1.3. Mapping From This Dissertation to Existing Publications

This dissertation is based on several published papers by the author. Specifically, the result in Section 2.3 was published in paper [61], the result in Section 2.4 was published in paper [62], the solution in Section 2.5 was published in [44], and Sections 3.2 and 3.3 are based on paper [63].

CHAPTER 2

TOCTTOU

The first contribution of this dissertation research is a complete solution to the well-known and long-standing TOCTTOU problem. We propose the CUU model that is capable of enumerating the complete set of potential TOCTTOU vulnerabilities, and our modular and event-driven defense mechanism (EDGI) and its Linux implementation are also complete.

2.1. Problem Statement

TOCTTOU (time-of-check-to-time-of-use) is a well-known security vulnerability [2] in file systems lacking strong synchronization support (e.g., the Unix file system). A TOCTTOU vulnerability is characterized by two distinct operations [6]. First, a vulnerable program *checks* for a file condition. Second, the program *uses* (operates on) the file, assuming that the established file condition remains invariant during execution. An illustrative vulnerable program is *sendmail*, which used to check for a specific attribute of a mailbox file (e.g., it is not a symbolic link) in step one and append new messages (as root) in step two. However, the checking and appending operations do not form an atomic unit. Therefore, a local attacker (the mailbox owner) can exploit the window of vulnerability between the two operations by deleting his/her mailbox and replacing it with a symbolic link to /etc/passwd. If the replacement is completed within the window and the new messages happen to be syntactically correct /etc/passwd entries with root access, then *sendmail* may unintentionally give unauthorized root access to a normal user (the attacker).

The *sendmail* example shows the structural complexity of a TOCTTOU attack, which requires (unintended) shared access to a file by the attacker and the victim (the

sendmail program), plus the two distinct steps (check and use) in the victim. This complexity plus the non-deterministic nature of TOCTTOU attacks make the detection difficult. For example, TOCTTOU attacks usually result in escalation of privileges, but no immediately recognizable damage. Furthermore, TOCTTOU attacks are inherently non-deterministic and not easily reproducible, making post mortem analysis also difficult. These difficulties are illustrated by the TOCTTOU vulnerabilities recently found in *vi* and *emacs* (Section 2.3.2), which appear to have been in place since the time those venerable programs were created.

TOCTTOU vulnerabilities are a very significant problem. In fact, CERT [58] released 20 advisories on TOCTTOU vulnerabilities between 2000 and 2004. These advisories covered a wide range of applications from system management tools (e.g., /bin/sh, *shar*, *tripwire*) to user level applications (e.g., *gpm*, Netscape browser), and they affected many operating systems, including Caldera, Conectiva, Debian, FreeBSD, HP-UX, Immunix, MandrakeSoft, RedHat, Sun Solaris, and SuSE. In 11 of these CERT advisories, the attacker was able to gain unauthorized root access. A similar list compiled from the BUGTRAQ [11] mailing list is shown in Table 1. TOCTTOU vulnerabilities are widespread and cause serious consequences.

Table 1: Reported TOCTTOU vulnerabilities

Domain	Application Name
Enterprise applications	Apache, bzip2, gzip, getmail, Imp-webmail, procmail, openldap, openSSL, Kerberos, OpenOffice, StarOffice, CUPS, SAP, samba
Administrative tools	at, diskcheck, GNU fileutils, logwatch, patchadd
Device managers	Esound, glint, pppd, Xinetd
Development tools	make, perl, Rational ClearCase, KDE, BitKeeper, Cscope

2.2. The CUU Model of TOCTTOU

Although in general TOCTTOU problems are not limited to file access [14], in this dissertation we focus on file-related TOCTTOU problems. We first propose an abstract model of such TOCTTOU problems (called CUU – "C" stands for "Check" and "U" stands for "Use") that captures all potential vulnerabilities. The model is based on

two mutually exclusive *invariants:* a file object either does not exist, or it exists and is mapped to a logical disk block. For each file object, one of these invariants must remain true between the check and use steps of every program. Otherwise, potential TOCTTOU vulnerabilities arise. This model allows us to enumerate all the file system call pairs of check and use (called exploitable TOCTTOU pairs), between which the invariants may be violated. Guided by this model, we are able to detect concrete TOCTTOU vulnerabilities in real-world applications. From this model we also derive a protection mechanism, which maintains the invariants across all the exploitable TOCTTOU pairs by preventing access from other concurrent processes/users. The practical value of CUU is demonstrated by the mapping of concrete Unix-style file systems to it. We have exhaustively analyzed the file system calls of POSIX and Linux and classified them according to the CUU model. From this classification we enumerated all the exploitable TOCTTOU pairs for POSIX (485 pairs) and Linux (224 pairs).

2.2.1. The Abstract File System

Due to the complexity of the TOCTTOU problem in real file systems, in this section we define a simplified Abstract File System (AbsFS), on which we define the TOCTTOU problem (see Section 2.2.2) and design a defense mechanism (see Section 2.5). In Section 2.2.3 we map concrete file systems (POSIX and Linux) to AbsFS and translate the results from the AbsFS to the concrete file systems.

2.2.1.1. Definition of Abstract File System

The Abstract File System (AbsFS) manages a set of file system (FS) objects. Each file system object consists of a pathname, an ordered set of logical disk blocks, and a mapping of the pathname to the corresponding set of logical disk blocks. For simplicity we assume the AbsFS to contain only contiguous files, i.e., the set of logical disk blocks is sequential for every file, and the AbsFS only needs to map the pathname to the address

(block number) of the initial logical disk block. Let F denote the set of all pathnames and B denote the set of all logical disk blocks, the pathname mapping function **resolve** is defined by:

$resolve: F \rightarrow B \cup \{\varnothing\}, \varnothing \not\subset \mathrm{B}$.

Given a pathname $f \in F$, if the AbsFS object corresponding to *f* exists, with the initial logical disk block number $b \in B$, then we define $resolve(f) = b$. If the AbsFS object corresponding to *f* does not exist, we define $resolve(f) = \varnothing$.

The AbsFS defines an Application Programming Interface consisting of 4 operations on file objects.

Definition 1: *creation(pathname)* is the operation that creates new FS objects in the AbsFS by changing the mapping for pathname f from $resolve(f) = \varnothing$ to $resolve(f) = b$, for some $b \in B$.

Definition 2: *removal(pathname)* is the operation that changes the mapping for pathname *f* from $resolve(f) = b$ to $resolve(f) = \varnothing$.

Definition 3: *normal use(pathname)* is the operation that works on an existing file system object and does not remove it.

Definition 4: *check(pathname)* is the operation that returns a predicate about the named FS object. The predicate may be $resolve(f) = b$ or $resolve(f) = \varnothing$. The file *f* has to be in one of these two states.

An application uses the *creation* operation to create a new FS object, the *check* operation to determine the invariant $resolve(f) = b$ or $resolve(f) = \varnothing$, the *normal use* operation to read or write the FS object, and the *removal* operation to delete an FS object. These four kinds of operations (*creation*, *normal use*, *removal*, and *check*) are all the currently defined AbsFS operations. The *creation* and *removal* operations change the **resolve** mapping, while the *check* and *normal use* operations do not change the **resolve**

mapping. The AbsFS operations and FS object states can be represented in a state transition diagram shown in Figure 1.

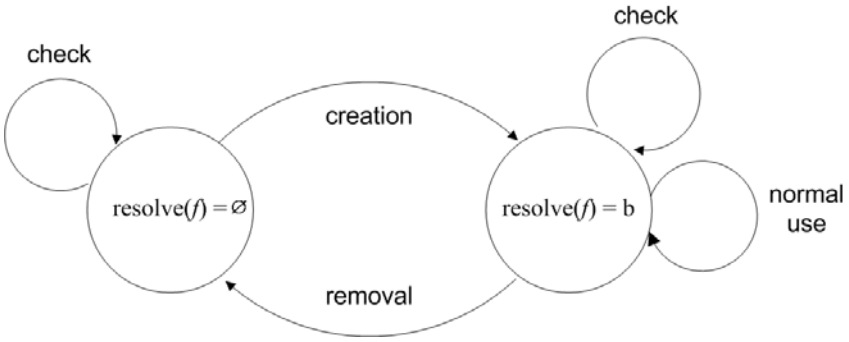

Figure 1: State Transition Diagram for FS Object *f*

2.2.1.2.Concurrent Access to AbsFS

Since the TOCTTOU vulnerability happens with concurrent access by a victim process and an attacker process, we extend the notation above to include explicit modeling of concurrent file system object access.

Definition 5: Safe sequence of AbsFS operations. Given a sequence *O* of AbsFS operations invoked by a process/user on FS object *f*, $O(f) = o_1(f), o_2(f), ..., o_n(f)$, $n > 1$, if $\forall i, 1 \le i \le n-1$, $resolve(f)$ remains an invariant between $o_i(f)$ and $o_{i+1}(f)$, we say the sequence $O(f)$ is a *safe* sequence of AbsFS operations (from the concurrency point of view). Since in most cases all the operations in the sequence belong to the same process/user, for notational simplicity, we omit the process/user id from the sequence. In case of interleaved operations, we will add a superscript to denote the different processes/users.

It is straightforward to see that the exclusive access by a single process to files is safe, i.e., the state of each FS object persists from the end of each AbsFS operation to the beginning of the next AbsFS operation under exclusive access.

Definition 6: Unsafe sequence of AbsFS operations: Given a sequence of operations $O(f) = o_1(f), o_2(f), ..., o_n(f)$, $n > 1$, if $\exists i, 1 \le i \le n-1$, $resolve(f)$ is not invariant

between $o_i(f)$ and $o_{i+1}(f)$, i.e., $resolve_{o_i}(f) \neq resolve_{o_{i+1}}(f)$, $O(f)$ is an *unsafe* sequence of AbsFS operations.

2.2.2. The CUU Model

2.2.2.1. Exclusion of Careless Programming

Before we start the discussion of the TOCTTOU problem, we point out that the TOCTTOU vulnerability is not due to a naively careless programming style. Consider the *sendmail* example. Hypothetically, the *sendmail* could simply open the file name that is the user's mailbox by naming convention (e.g., /usr/mail/username) and then append emails to that file. This simplistic approach fails immediately because the naming convention may or may not hold for all names (e.g., a user may have created a symbolic link from /usr/mail/username to /etc/passwd). To avoid this kind of problems, many system programmers have adopted a more careful programming style. In case of files, this careful programming style establishes a predicate on the file before using it. For example, *sendmail* establishes the predicate $resolve(f) = b$, where b belongs to a regular file, not a symbolic link, before appending messages to f. The predicate $resolve(f) = b$ is an invariant that should remain true as long as the *sendmail* keeps appending messages. We call the predicate an *invariant* instead of pre-condition, because the normal connotation of pre-condition is that it must be true before entering a function, but it may become false after the function has started. In contrast, our invariant must remain true through the duration of file usage.

In the rest of this dissertation we exclude the careless programming style and assume that all system utilities of interest will establish an invariant on a pathname before using it. This is represented in our notation by dividing a sequence of AbsFS operations $O(f) = o_1(f),...,o_i(f),o_{i+1}(f),...,o_n(f)$ into two subsequences. The first subsequence $o_1(f),...,o_i(f)$ is called the "Check" part, and the second subsequence

$o_{i+1}(f),...,o_n(f)$ is called the "Use" part. The "Check" part establishes the invariant $resolve_{o_i}(f)$ and the "Use" part of the sequence relies on the invariant remaining true, i.e., $O(f)$ is a safe sequence of AbsFS operations.

2.2.2.2.TOCTTOU Attacks in AbsFS

Definition 7: A TOCTTOU (Time-Of-Check-To-Time-Of-Use) attack on file object *f* consists of two concurrent processes, victim *v* and attacker *a*, with interleaved AbsFS operations that make *v*'s sequence unsafe. Consider the victim *v* executing the sequence $O^v(f) = o_1^v(f),...,o_i^v(f),o_{i+1}^v(f)...,o_n^v(f)$, divided into the "Check" and "Use" parts. Concurrent with *v*, attacker *a* is able to change the mapping $resolve_{o_i}(f)$ established by *v* during the execution of the sequence $O^v(f)$, transforming it into an unsafe sequence. This is achieved by inserting the sequence $O^a(f) = o_1^a(f),o_2^a(f),...,o_k^a(f)$ between the "Check" and "Use" parts of $O^v(f)$. The result becomes: $o_1^v(f),...,o_i^v(f),o_1^a(f),o_2^a(f),...,o_k^a(f),o_{i+1}^v(f)...,o_n^v(f)$.

To illustrate the definition with concrete scenario, we temporarily move from AbsFS to a Unix-style file system environment. Suppose the invariant established by *v* is $resolve_{o_i}(f) = b$, the attack sequence $O^a(f)$ of *a* can be: first remove *f* and then create a symbolic link named *f* which points to another file object *t* ($resolve(t) = b', b \neq b'$), resulting in $resolve_{o_k}^a(f) = b'$. If the invariant established by *v* is $resolve_{o_i}(f) = \varnothing$, a possible attack sequence $O^a(f)$ is to create the file object *f*, making $resolve_{o_k}^a(f) \neq \varnothing$.

The TOCTTOU attack is successful if $resolve_{o_i}^v(f) \neq resolve_{o_k}^a(f)$ and victim *v* continues execution without realizing the invariant created by the subsequence $o_1^v(f),...,o_i^v(f)$ (the "Check" part) has been violated. Consequently, the subsequence $o_{i+1}^v(f)...,o_n^v(f)$ (the "Use" part) will execute under the assumption of the original invariant, which is no longer true.

The side effect of *v* executing the "Use" subsequence $o_{i+1}^{v}(f)...,o_{n}^{v}(f)$ after a successful TOCTTOU attack is that *v* is actually working on some other unintended file object. For example, if *t* = /etc/passwd in the *sendmail* example, emails may be appended to /etc/passwd.

Proposition 1: Violation of an invariant is a necessary condition for a successful TOCTTOU attack.

The proposition 1 follows from Definition 7. If there is no violation of invariants, the sequence $O^{v}(f)$ is a safe sequence, so there would be no TOCTTOU attack. Consequently, through the entire duration of $O^{v}(f)$, we can prevent TOCTTOU attacks by preserving the invariant established by $O^{v}(f)$ and making the sequence a safe sequence.

2.2.2.3. An Enumeration of TOCTTOU pairs

Definition 8: Consider an unsafe sequence of AbsFS operations $O(f)=o_1(f),o_2(f),...,o_n(f)$, where $resolve_{o_i}(f)\neq resolve_{o_{i+1}}(f)$. The two operations surrounding the violation of the original invariant (the last operation of the "Check" part and the first operation of the "Use" part), $o_i(f)$ and $o_{i+1}(f)$, are called a *TOCTTOU pair*.

It is useful to identify the TOCTTOU pairs explicitly, since the combinations that yield such pairs are non-trivial but manageable. The diagram in Figure 1 shows all the AbsFS operations and the two states in which a file may be. On the left side of diagram is the *non-existent* state, denoted by $resolve(f)=\varnothing$ and on the right side of the diagram is the *existent* state, denoted by $resolve(f)=b$.

Let us consider first the *non-existent* state and the invariant $resolve(f)=\varnothing$. The first term of a TOCTTOU pair is an operation that results in the *non-existent* state of *f*. From the state transition diagram in Figure 1, we see that two operations lead to the *non-existent* state: {*check, removal*}. The *removal* operation explicitly makes *f* non-existent,

while the *check* operation also ends in the *non-existent* state if it does not find the pathname mapping. The second term of the TOCTTOU pair is an operation that starts from the invariant $resolve(f) = \varnothing$ (the *non-existent* state). The two operations that start from the *non-existent* state are: {*check, creation*}. Therefore, the TOCTTOU pairs associated with the *non-existent* state are contained in the set produced by the Cartesian product of {*check, removal*}×{*check, creation*}.

While the Cartesian product contains all the TOCTTOU pairs, we will refine the second term, which corresponds to the "Use" part of the TOCTTOU pair. For an attacker to exploit a TOCTTOU vulnerability for some gain (e.g., escalation of privileges), the victim must be tricked into doing something useful for the attacker in the "Use" part. Examples of useful actions are: (1) set or modify the status information of an existing file object (e.g. make /etc/passwd world-writable); (2) modify the runtime environment of the victim application (e.g. change the current directory); and (3) release the content of a sensitive file object (e.g. read the content of /etc/shadow into memory). Since the *check* operation does not produce any useful results for the attacker, we define *exploitable* TOCTTOU pairs by eliminating the *check* operation from the second term of TOCTTOU pairs.

Table 2: Exploitable TOCTTOU pairs (AbsFS)

Invariant	TOCTTOU Pairs
$resolve(f) = \varnothing$	<check, creation> <removal, creation>
$resolve(f) = b$	<creation, normal use> <check, normal use> <normal use, normal use> <creation, removal> <check, removal> <normal use, removal>

Now we consider the *existent* state of *f*, characterized by the invariant $resolve(f) = b$. The state transition diagram in Figure 1 shows that the set of operations that lead into the *existent* state is {*creation, check, normal use*}, and the set of operations that start from the *existent* state is {*check, normal use, removal*}. So the TOCTTOU

pairs associated with this invariant are in the set {*creation, check, normal use*}×{*check, normal use, removal*}. As a second term of the TOCTTOU pairs, *check* will not produce useful results for the attacker. Consequently, we also eliminate *check* from the list of exploitable TOCTTOU pairs.

By deleting *check* from the second terms, the exploitable TOCTTOU pairs are {*check, removal*}×{*creation*} for the first invariant and {*creation, check, normal use*}×{*normal use, removal*} for the second invariant. Since there are only two invariants in AbsFS, we have enumerated all the exploitable TOCTTOU pairs in Table 2.

Proposition 2: The enumeration of TOCTTOU pairs in Table 2 is complete, i.e., it contains all the exploitable TOCTTOU pairs in AbsFS.

Proof: by construction we have enumerated all the exploitable TOCTTOU pairs in Table 2. There are only two invariants in the state diagram in Figure 1, and we have analyzed all the state transitions in Figure 1.

2.2.2.4. Prevention of TOCTTOU Attacks

In the rest of this section, we will focus on the preservation of invariants across the exploitable TOCTTOU pairs. This protection will be done in two steps. First, we will maintain explicitly the invariant *holder* for each file object. Second, for every file system operation that may change the invariant, we check whether the invoker of the operation is the holder. The operation is allowed if it's invoked by the holder. It is disallowed if it belongs to another process/user.

In Figure 1, we described the state transitions of a file with a single process/user. Figure 2 shows the state transitions of a file under concurrent access by multiple processes/users. Without loss of generality, we adopt the policy that the first process/user accessing the file object becomes the invariant holder. (Intuitively, we consider the invariant as an exclusive lock.) The goal of our protection mechanism is to reject any changes to the invariant except by the invariant holder.

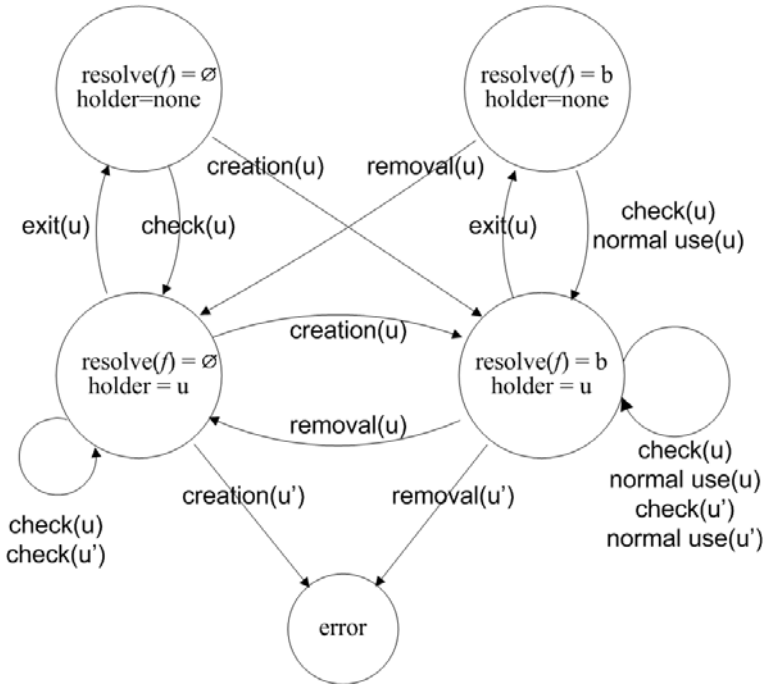

Figure 2: The Enhanced State Transition Diagram with Two Users

The main difference between Figure 1 and Figure 2 is the addition of three states. Two of the states (on the top part of Figure 2) are due to the explicit representation of the cases of invariants with a holder (same as Figure 1) and without a holder (new states). These transitions are allowed, since the pathname is free and the invariant holder is not in competition with any other process/user. The third new state is at the bottom of Figure 2, representing a potential attack since those transitions would change the invariant for the holder. These transitions are rejected as an error. The original invariant holder maintains the hold on the invariant and the invariant remains unchanged.

The implementation of invariant holder lock relies on a lock table and maps the invariant holder id to the invariant across all TOCTTOU pairs. Consider a TOCTTOU pair $<o_1, o_2>$. When a process u accesses a pathname f through $o_1(f)$, u becomes the invariant holder, moving from the top states of Figure 2 to one of the middle states. (Note that all four AbsFS operations are allowed in this step.) Our protection mechanism uses the lock table to remember this invariant/holder mapping. The lock is released when

the invariant holder process ends. These state transitions are denoted as *exit(u)*, in which case *u* releases the invariant.

While the pathname *f* is in one of the middle states, with invariant holder *u*, another process/user (*u'*) may attempt to change the invariant, which will result in "error". Other operations that do not affect the invariant (e.g., *check* and *normal use*) are allowed, as shown in Figure 2. Thus this mechanism implements the assumption required in Proposition 2 to protect the invariants across TOCTTOU pairs.

For practical purposes, we note that our protection mechanism does not require explicit request and release of invariant-related locks. The management of invariant locks can be done automatically on behalf of applications. Furthermore, the implementation can be simplified with the following proposition.

Proposition 3: Blocking the *creation* and *removal* of a file object *f* across a sequence $o_1(f), o_2(f), ..., o_n(f)$ is sufficient to make the sequence safe.

By Definition 5, a sequence of execution $o_1(f), o_2(f), ..., o_n(f)$ is safe if $\forall i, 1 \le i \le n-1$, *resolve*$(f)$ is an invariant between $o_i(f)$ and $o_{i+1}(f)$. If we forbid any *creation* or *removal* of *f* across $o_1(f), o_2(f), ..., o_n(f)$, we forbid *creation* or *removal* of *f* between $o_i(f)$ and $o_{i+1}(f)$, and since *creation* and *removal* are the only operations that can change *resolve*(f), *resolve*(f) must be an invariant between $o_i(f)$ and $o_{i+1}(f)$. So $o_1(f), o_2(f), ..., o_n(f)$ is guaranteed to be a safe sequence of execution.

This proposition is the basis for the EDGI defense in Section 2.5.

Proposition 4: Making all exploitable TOCTTOU pairs safe is sufficient to make all file access sequences safe and prevent TOCTTOU attacks.

Proof: Proposition 3 shows the preservation of invariants through a file operation sequence suffices in making the sequence safe. Proposition 2 shows that all exploitable TOCTTOU pairs have been enumerated. Combining the two propositions we have the

assurance that making all file operation sequences safe (for each process/user) can prevent all TOCTTOU vulnerabilities from being exploited.

2.2.3. Concrete File System Examples

2.2.3.1. Exclusion of Careless File Attribute Settings

The AbsFS contains a simplified model of file system objects, with a very simple mapping of pathname to logical disk blocks, without any additional file system attributes such as access permission. In concrete file systems, appropriate access control attributes need to be set to prevent trivial unauthorized file access. For example, Unix files with world writable settings can be easily exploited by many kinds of attacks. In our modeling and analysis of TOCTTOU attacks, we assume that appropriate file access control settings are being used by careful system administrators.

2.2.3.2. Unix-Style File Systems

Table 2 gives a complete list of TOCTTOU pairs in AbsFS. Now we map the AbsFS into Unix-style file systems. The first observation in the mapping is that Unix-style file systems have several kinds of file system objects: regular files, directories, and links. The second observation is that the abstract operations of *check, creation, normal use,* and *removal* may be implemented by several system calls. Therefore, we map these abstract operations into sets of system calls (CreationSet, NormalUseSet, RemovalSet and CheckSet) and divide these sets into operations on each kind of file system objects.

CreationSet = FileCreationSet $\cup$ DirCreationSet $\cup$ LinkCreationSet

NormalUseSet[2] = FileNormalUseSet $\cup$ DirNormalUseSet

---

[2] On Unix-style file systems, the normal use of a link (symbolic or hard) is actually on the regular file or directory that the link refers to, so we do not need to define a separate NormalUseSet for link.

RemovalSet = FileRemovalSet ∪ LinkRemovalSet ∪ DirRemovalSet

CheckSet = FileCheckSet ∪ LinkCheckSet ∪ DirCheckSet

Table 3: Enumeration of exploitable TOCTTOU pairs (Unix-Style file systems)

Invariant	Exploitable TOCTTOU Pairs
$resolve(f) = \varnothing$	(FileCheckSet × FileCreationSet) ∪ (FileRemovalSet × FileCreationSet) ∪ (DirCheckSet × DirCreationSet) ∪ (DirRemovalSet × DirCreationSet) ∪ (LinkCheckSet × LinkCreationSet) ∪ (LinkRemovalSet × LinkCreationSet)
$resolve(f) = b$	(FileCheckSet × FileNormalUseSet) ∪ (FileCreationSet × FileNormalUseSet) ∪ (LinkCreationSet × FileNormalUseSet) ∪ (FileNormalUseSet × FileNormalUseSet)∪ (DirCheckSet × DirNormalUseSet) ∪ (DirCreationSet × DirNormalUseSet) ∪ (LinkCreationSet × DirNormalUseSet) ∪ (DirNormalUseSet × DirNormalUseSet)

The third observation is that the *removal* operation in Unix-style file systems does not produce any useful results for the attacker. This is because in Unix-style file systems, under the assumption of careful file attribute settings (Section 2.2.3.1), there are only two ways for the attacker to make $resolve(f) = resolve(t)$ in a TOCTTOU attack (*t* is an existing security sensitive file object such as /etc/passwd and *f* is the file object accessed by a TOCTTOU pair $< o_1, o_2 >$ in the victim application): via symbolic link or hard link. If the attacker replaces *f* with a symbolic link to *t*, then the victim's *removal* operation on *f* only removes *f* itself, but not *t*; If the attacker replaces *f* with a hard link to *t*, this will increase the number of hard links of *t* by 1, and when the victim performs the *removal* operation on *f*, it decreases the number of hard links of *t* by 1 (restores the original hard link number of *t*, but never decreases it). Since *t* is physically removed only when its hard link number becomes 0, given *t*'s initial hard link number is nonzero, the attacker can not cause *t* to be removed.

Thus for Unix-style file systems we can eliminate those TOCTTOU pairs with *removal* as the second term from Table 2. The remaining AbsFS TOCTTOU pairs can be

mapped to Unix-style file systems as shown in Table 3. For an actual file system, we can map the actual file system calls to these sets to obtain the concrete TOCTTOU pairs.

FileCreationSet = {**creat, open, mknod, mkfifo, rename**} DirCreationSet = {**mkdir, rename**} LinkCreationSet = {**link, symlink, rename**} FileNormalUseSet = {**chmod, chown, truncate, utime, open, fopen, fdopen, popen, execl, execle, execlp, execv, execve, execvp, pathconf**} DirNormalUseSet = {**chmod, chown, utime, chdir, pathconf**} FileRemovalSet = {**remove, unlink, rename**} DirRemovalSet = {**remove, rmdir, rename**} LinkRemovalSet = {**remove, unlink, rename**} FileCheckSet = {**access, stat**} DirCheckSet = {**access, stat**} LinkCheckSet = {**lstat, readlink**}

Figure 3: POSIX File Operations

FileCreationSet = {**creat, open, mknod, rename**} DirCreationSet = {**mkdir, rename**} LinkCreationSet = {**link, symlink, rename**} FileNormalUseSet = {**chmod, chown, truncate, utime, open, execve**} DirNormalUseSet = {**chmod, chown, utime, mount, chdir, chroot, pivot_root**} FileRemovalSet = {**unlink, rename**} DirRemovalSet = {**rmdir, rename**} LinkRemovalSet = {**unlink, rename**} FileCheckSet = {**stat, access**} DirCheckSet = {**stat, access**} LinkCheckSet = {**stat, access**}

Figure 4: Linux File Operations

2.2.3.3. Study of POSIX and Linux

We focus on POSIX [30] and Linux as representative examples of Unix-style file systems with TOCTTOU vulnerabilities. We believe the same mapping can be done with the other flavors of Unix file systems. The POSIX mapping is shown in Figure 3 and the Linux mapping is shown in Figure 4. Compare Figure 4 to Figure 3 we can see that the sets are almost the same due to the fact that Linux is POSIX-compliant. We do see some discrepancy though, notably the FileNormalUseSet. For example, POSIX has 6 different system calls on executing a program (**execl, execle, execlp, execv, execve, execvp**), but Linux only has one (**execve**). A closer look at the Linux implementation reveals that Linux implements only **execve** as a system call and uses library calls to implement the remaining 5 POSIX interfaces, which are different wrappers on top of this basic system call.

Applying the mapping of Figure 3 to the mapping in Table 3, we have identified 485 exploitable TOCTTOU pairs for POSIX. Similarly, by applying Figure 4 to the mapping in Table 3, we get 224 exploitable TOCTTOU pairs for Linux

Proposition 5: If the classification of a concrete file system's operations is complete, then the enumeration of exploitable TOCTTOU pairs is complete for the concrete file system. By complete we mean the classification contains all the concrete file system calls that operate on file objects, and all the concrete file system calls are classified into *check*, *creation*, *normal use*, and *removal* functions on the file objects. (File system calls that have multiple functions appear in multiple categories.)

Proof: The Proposition 2 guarantees the completeness of exploitable TOCTTOU pairs for the AbsFS. Assuming that we have exhaustively analyzed the concrete file system calls and classified them, Proposition 5 follows from Proposition 2.

By exhaustively analyzing the POSIX file system calls (Figure 3), we can apply Proposition 5 to the enumeration of exploitable TOCTTOU pairs based on Table 3 and Figure 3 and conclude that we have enumerated all the exploitable TOCTTOU pairs in POSIX. Analogously, we apply Proposition 5 to the enumeration of exploitable TOCTTOU pairs in Linux, based on Table 3 and Figure 4, and the result is in Table 28 of the Appendix.

2.2.3.4. Example of TOCTTOU Attacks

We have studied some real world programs with known TOCTTOU vulnerabilities on Unix-style systems. The results are shown in Table 4. For example, in *sendmail*, the TOCTTOU vulnerability is a <**stat, open**> pair, the invariant is $resolve(umbox) = b$, and the attack is first removing *umbox* and second creating a symbolic link under the name *umbox.*

Logwatch Vulnerability. *logwatch* is an open-source script for monitoring log files in Linux. *logwatch* 2.1.1 running as root was reported [52] to allow a local attacker

to gain elevated privileges, e.g., write access to /etc/passwd. This attack consists of the following steps:

1) Get the running process ID {pid} of *logwatch*;
2) Create a temporary directory named /tmp/logwatch.{pid};
3) Create a symbolic link with a specific name in the temporary directory, which points to /etc/log.d/scripts/logfiles/samba/`cd etc;chmod 666 passwd #`
4) Wait for *logwatch* to use the temporary symbolic link. Although *logwatch* only opens it for writing, the tricky file name causes the shell to execute it as a command line later.

Table 4: Some existing TOCTTOU vulnerabilities on Unix-style systems

Applications	TOCTTOU pair	Classification and Invariant
BitKeeper, Cscope 15.5, CUPS, getmail 4.2.0, glint, Kerberos 4, openldap, OpenOffice 1.0.1, patchadd, procmail, samba, Xinetd	<stat, open>	FileCheckSet × FileCreationSet $resolve(f) = \varnothing$
Rational ClearCase, pppd	<stat, chmod>	FileCheckSet × FileNormalUseSet $resolve(f) = b$
Sendmail	<stat, open>	
logwatch 2.1.1	<stat, mkdir>	DirCheckSet × DirCreationSet $resolve(f) = \varnothing$
bzip2-1.0.1, gzip, SAP	<open, chmod>	FileCreationSet × FileNormalUseSet $resolve(f) = b$
Mac OS X 10.4 – launchd	<open, chown>	
StarOffice 5.2	<mkdir, chmod>	DirCreationSet × DirNormalUseSet $resolve(f) = b$

The TOCTTOU pair in *logwatch* is <**stat, mkdir**>. *logwatch* first checks whether the directory /tmp/logwatch.{pid} exists (**stat**) before creating it. However, an attacker may create that directory (as shown above) between the **stat** and **mkdir** system calls. In this case, *logwatch*'s **mkdir** fails, but since *logwatch* does not check the return value of its **mkdir**, it continues blindly and uses the temporary directory. The invariant in *logwatch* is $resolve(tmpdir) = \varnothing$ and the attack is a *creation* operation (**mkdir**) by the attacker. (Here the *tmpdir* is /tmp/logwatch.{pid})

Table 4 summarizes the TOCTTOU pairs and their associated invariants for a number of known TOCTTOU vulnerabilities.

2.3. Detection of TOCTTOU Vulnerabilities

In this part of the dissertation, we implement CUU model-based software tools that systematically search for potential TOCTTOU vulnerabilities in Linux system utility programs. They are able to detect previously reported TOCTTOU vulnerabilities as well as finding some unknown ones (e.g., in the *rpm* software distribution program, the *vi/vim* and *emacs* editors). We conduct a detailed experimental study of successfully exploiting these vulnerabilities and analyze the significant events during a TOCTTOU attack against the native binaries of *rpm* and *vi*. By repeating the experiments, we also evaluate the probability of these events happening, as well as the success rate of these non-deterministic TOCTTOU attacks. These analyses provide a quantitatively better understanding of TOCTTOU attacks.

2.3.1. Model-Based TOCTTOU Detection

2.3.1.1. Components of Practical Attacks

An actual TOCTTOU vulnerability consists of a victim program containing a TOCTTOU pair (described in Section 2.2.2.3) and an attacker program trying to take advantage of the potential race condition introduced by the TOCTTOU pair. The attacker program attempts to access or modify the file being manipulated by the victim through shared access during the vulnerability window between the "Check" call and the "Use" call. For example, by adding a line to an unintentionally shared script file in the *rpm* attack (Section 2.3.2.2), the attacker can trick the victim into executing unintended code at a higher privilege level (root). In general, we say that a TOCTTOU attack is profitable if the victim is running at a higher level of privilege. In Unix-style OSs, this means the victim running as root and the attacker as normal user.

An important observation is that even though the victim is running at a higher level of privilege, the attacker must have sufficient privileges to operate on the shared file attributes, e.g., creation or deletion. This observation narrows the scope of potential TOCTTOU vulnerabilities. Table 5 shows a list of directories owned by root in Linux. Since normal users cannot change the attributes or content of files in these directories, these files are safe.

Table 5: Directories immune to TOCTTOU

/bin	/root	/usr/dict	/var/db
/boot	/proc	/usr/kerberos	/var/empty
/dev	/sbin	/usr/libexec	/var/ftp
/etc	/usr/bin	/usr/sbin	/var/lock
/lib	/usr/etc	/usr/src	/var/log
/misc	/usr/include	/usr/X11R6	/var/lib
/mnt	/usr/lib	/var/cache	/var/run
/opt			

2.3.1.2. CUU Model-Based Detection Tools

Based on the CUU model, we design a software framework and implement software tools to detect actual TOCTTOU vulnerabilities in Linux. Figure 5 shows the four components of our detection framework, based on dynamic monitoring of system calls made by sensitive applications (e.g., those that execute with root privileges). The first component of our framework is a set of plug-in Sensor code in the kernel, placed in file-related system calls such as those in Figure 4. These Sensors record the system call name and its arguments, particularly file name (full path for unique identification purposes). For some system calls, other related arguments are also recorded to assist in later analysis, e.g., the *mode* value of **chmod**(*path, mode*). Some environmental variables are also recorded, including process id, name of the application, user id, group id, effective user id, and effective group id. This information will be used in the analysis to determine if a TOCTTOU pair can be exploited. We do not use standard Linux trace facilities such as *strace* for two reasons: First, *strace* does not output full pathname for

files referred to using relative pathnames; second, *strace* does not give enough environmental information such as effective user id.

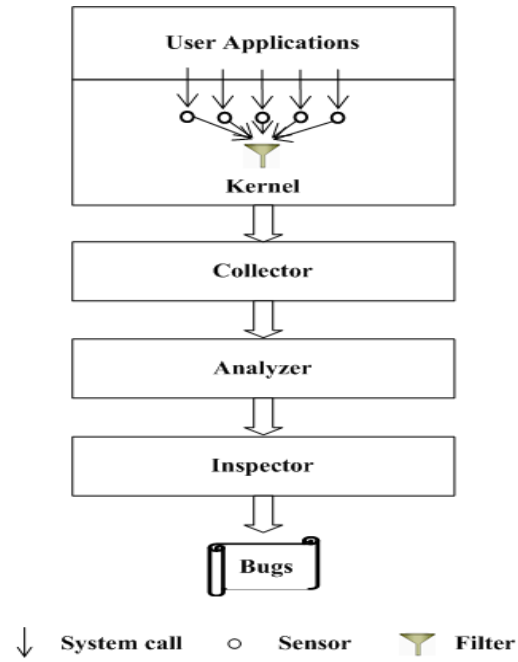

Figure 5: Framework for TOCTTOU detection

The Sensors component also carries out a preliminary filtering of their log. Specifically, they identify the system calls on files under the system directories listed in Table 5 and filter them out, since those files are immune to TOCTTOU attacks. After this filter, remaining potentially vulnerable system calls are recorded in a circular FIFO ring buffer by **printk**.

The second component of our framework is the Collector, which periodically empties the ring buffer (before it fills up). The current implementation of the Collector is a Linux daemon that transforms the log records into an XML format and writes the output to a log file for both online and offline analysis.

The third component of our framework is the Analyzer, which looks for TOCTTOU pairs (listed in Table 3) that refer to the same file pathname. For offline analysis, this correlation is currently done using XSLT (eXtensible Stylesheet Language Transformations) templates. This analysis proceeds in several rounds as follows.

Round 1: First, the Analyzer sorts the log records by file name, grouping its operation records such as the names and locations (sequence numbers) of system calls.

Round 2: Second, system calls on each file are paired to facilitate the matching of TOCTTOU pairs.

Round 3: Third, system call pairs are compared to the list in Table 3. When a TOCTTOU pair is found, an XSLT template is generated to extract the corresponding log records from the original log file.

Round 4: Fourth, the log records related to TOCTTOU pairs found are extracted into a new file for further inspection.

The fourth component of our framework is the Inspector, which identifies the actual TOCTTOU vulnerability in the program being monitored. The Inspector links the TOCTTOU pair with associated environmental information, including file pathname, related arguments, process id, program name, user id, group id, effective user id, and effective group id. The Inspector decides whether an actual exploitation can occur.

For each TOCTTOU pair, the Inspector does the following steps:

- Check the arguments of the calls to see if these calls can be profitable to an attacker. For example, if the "Use" call is **chmod**, then a value of 0666 for the *mode* argument falls into this category because this **chmod** can be used to make /etc/passwd world-writable. On the other hand, a *mode* value of 0600 is not profitable because it will not give the attacker any permission on a file that he/she does not own. In this case the TOCTTOU pair in question is not a TOCTTOU vulnerability.
- Check the file pathname. For the **chmod** example, if the file is stored under a directory writable by an ordinary user, like his/her home directory, then continue to the next step; otherwise the TOCTTOU pair is not a TOCTTOU vulnerability.

- Check the effective user id. Continuing with the **chmod** example, if the effective user id is 0 (root), then report this TOCTTOU pair as a vulnerability; otherwise, the TOCTTOU pair is not a vulnerability.

It should be noted that the steps described above give only an outline of the Inspection process based on one attack scenario for one particular TOCTTOU pair. For different TOCTTOU pair and different attack scenario, the details of these checks can be different. For example, the same TOCTTOU pair as the above with a *mode* value of 0644 and the same other conditions is also considered a vulnerability because it can be exploited to make /etc/shadow readable by an attacker. Thus the Inspector requires a template (or signature) for each kind of attack scenario. Table 6 shows the set of templates used by the current implementation of the Inspector. For brevity, this table does not show the file pathname and effective user id which are checked in every template. This set may be expanded as new attack scenarios are found.

Table 6: Templates used in the Inspector

"Use" call	Arguments to check	Sample attack scenarios
chmod	*mode*	Gain unauthorized access rights to /etc/passwd
chown	*owner, group*	Change the ownership of /etc/passwd
chroot		Access information under a restricted directory
execve		Run arbitrary code
open	*mode, flag*	Mislead privileged programs to do things for the attacker, or steal sensitive information
truncate	*length*	Erase the content of /etc/passwd

2.3.2. Analysis of Real TOCTTOU Attacks

2.3.2.1. Experimental Setup

We applied our detection framework and tools to find previously unreported TOCTTOU vulnerabilities in Linux. Although the CUU model describes all the TOCTTOU pairs in Linux file systems, it is impractical to test all the execution paths of

all the system software (or even a single program of any complexity). Our intent is to learn as much as possible about real TOCTTOU vulnerabilities through a detailed analysis. The experiments show that significant weaknesses can be found relatively easily using our framework and tools.

From the discussion in Section 2.3.1.1, we focus our attention on system software programs that use file system (outside the directories listed in Table 5) as a root. Each program chosen is downloaded, installed, configured, and deployed. Furthermore, we also build a testing environment which includes the design and generation of a representative workload for each application, plus the analysis of TOCTTOU pairs observed. Although this is a laborious process that requires high expertise, one could imagine incorporating such testing environments into the software release of system programs, facilitating future evaluations and experiments.

Our tools were implemented on Red Hat 9 Linux (kernel 2.4.20) to find TOCTTOU vulnerabilities in about 130 commonly used utility programs. The script-based experiments consist of about 400 lines of shell script for 70 programs in /bin and /sbin. This script takes about 270 seconds to gather approximately 310K bytes of system call and event information. The other 60 programs were run manually using an interactive interface. From this sample of Linux system utilities, we found five potential TOCTTOU vulnerabilities (see Table 7).

The experiments were run on an Intel P4 (2.26GHz) laptop with 256M memory. The Collector produces an event log at the rate of 650 bytes/sec when the system is idle (only background tasks such as daemons are running), 11KB/sec during the peak time a large application such as OpenOffice is started, and 2KB/sec on average. The Analyzer processes the log at the speed of 4KB/sec.

From the list in Table 7, we wrote simple attack programs that confirmed the TOCTTOU vulnerabilities in *rpm, emacs* and *vi*. We discuss the attack on *rpm* and *vi* in

detail (Sections 2.3.2.2 and 2.3.2.3, respectively), and outline the others in Section 2.3.2.4.

Table 7: Potential TOCTTOU vulnerabilities

Application	TOCTTOU errors	Possible exploit
vi	**<open, chown>**	Changing the owner of /etc/passwd to an ordinary user
rpm	**<open, open>**	Running arbitrary command
emacs	**<open,chmod>**	Making /etc/shadow readable by an ordinary user
gedit	**<rename, chown>**	Changing the owner of /etc/passwd to an ordinary user
esd (Enlightened Sound Daemon)	**<mkdir, chmod>**	Gaining full access to another user's home directory

2.3.2.2. rpm 4.2 Temporary File Vulnerability

rpm is a popular software management tool for installing, uninstalling, verifying, querying, and updating software packages in Linux. When *rpm* installs or removes a software package, it creates a temporary script file in directories such as /var/tmp or /var/local/tmp. This shell script is used to install or remove help documentation of the software package. Since the access mode of this file is set to 666 (world-writable), an attacker can insert arbitrary commands into this script. Given the privileges required for installing software (usually root), this is a significant vulnerability. The TOCTTOU pair involved is <**open**, **open**>: the first **open** creates the script file for writing the script; and the second **open** is called in a child process to read and execute the script.

2.3.2.2.1 Baseline Analysis of rpm

In our evaluation of the TOCTTOU vulnerability in *rpm*, we start by measuring the total running time of *rpm* (denoted by *t*) and the window of vulnerability (the time interval between the two **open**s, denoted by *v*). We ran *rpm* (as root) 100 times, alternatively installing and uninstalling a package named *sharutils-4.2.1-14.i386.rpm*, and measured *t* and *v* for each invocation. From Table 8 we can see that the window of

vulnerability is relatively narrow (less than 5%), since the two **open**s are separated only by a few milliseconds.

Table 8: Baseline vulnerability of rpm

Package Operation	Install (rpm -i)		Uninstall (rpm -e)	
	Average	Stdev	Average	Stdev
t (μsec)	125,188	9,930	110,571	10,961
v (μsec)	5,053	20	4,218	102
v/t	4.1%	---	3.8%	---

2.3.2.2.2 An Experiment to Exploit rpm

The second part of our evaluation is to measure the effectiveness of an attack trying to exploit this apparently small window of vulnerability. This experiment runs a user-level attack process in a loop. It constantly checks for the existence of a file name with the prefix "/var/tmp/rpm-tmp". A victim process (*rpm* run by root) installs a software package and creates a script file of that name. Note that *rpm* inserts a random suffix as protection against direct guessing, but a directory listing command bypasses the need to guess the full pathname. If a file name of the expected prefix appears, the attacker appends the command "*chown* attacker:attacker /etc/passwd" to it. If the append happens during the window of vulnerability, then the child process of *rpm* will execute the script and the inserted command line, making the attacker the owner of /etc/passwd. When *rpm* finishes, the test program checks whether the attacker has become the owner of /etc/passwd.

Due to the non-deterministic nature of these experiments, we ran the experiment 100 times in a batch. After running several batches, we found a surprisingly high average number of 85 successful attacks per batch, considering the apparently narrow window of vulnerability shown in Table 8.

2.3.2.2.3 Event Analysis of rpm Exploit

To fully understand what happened during the TOCTTOU attack, we analyze the important system events during the experiment. Figure 6 shows the events in a successful

exploit of *rpm*. In Figure 6, the dark (upper) line shows the events of the *rpm* process, and the lower line shows the events of the attacker process. The attacker process stays in a loop looking for file names of interest. When the *rpm* process creates the file (just before the 200 msec clock tick), the attacker detects it and appends the *chown* line to the temporary script and goes back to the loop.

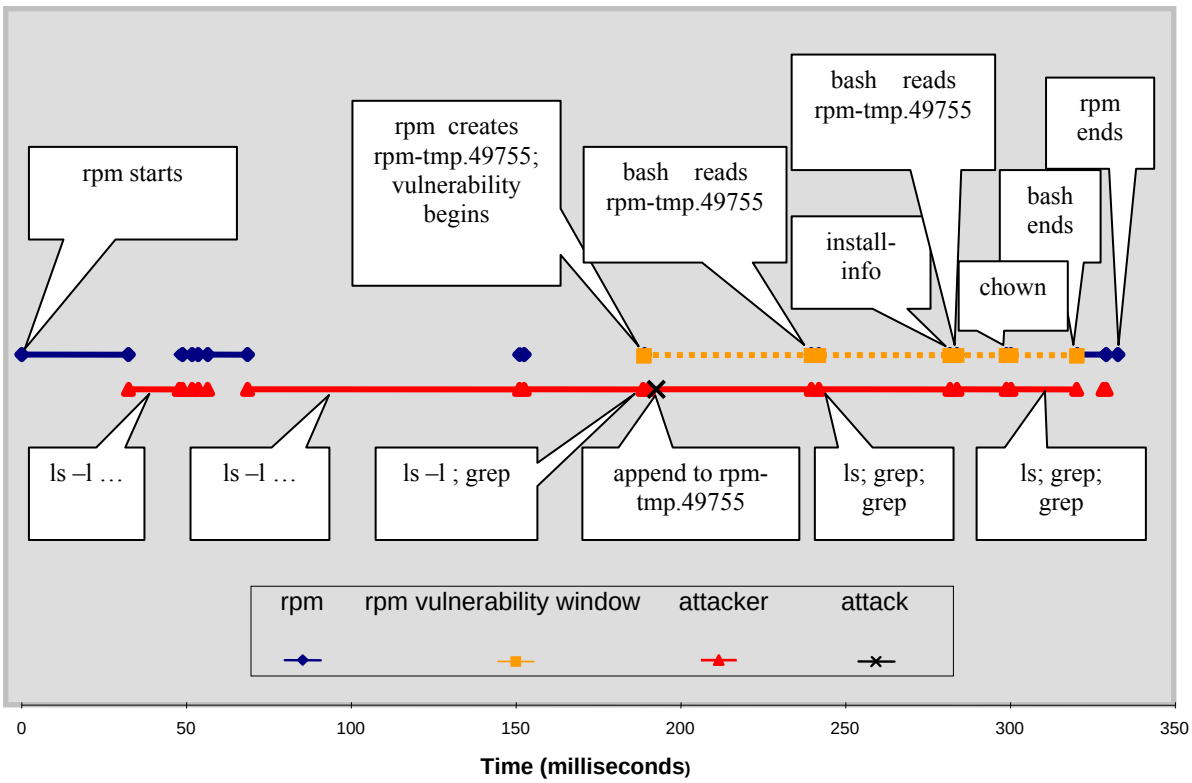

Figure 6: Event Analysis of *rpm* Exploit

The two timelines show that even though the CPU consumption during the window of vulnerability is relatively small, the *rpm* process causes interrupts that lengthen the window, represented by dotted upper line. Specifically, there are at least two scheduling actions within the *rpm* vulnerability window: *rpm* creates a new process to execute *bash*, which creates another new process to execute an external executable file (/sbin/install-info). Each process creation causes *rpm* to yield CPU to the scheduler. Figure 6 shows that the attacker process is scheduled as a result and the attack succeeds.

Consequently, the two scheduling actions created by *rpm* make the attack more likely to succeed because *rpm* yields the CPU in the window of vulnerability.

In our experiments, we also found another reason more attacks succeed than indicated by the short window of vulnerability. Specifically, we observed that in some cases the appending to the script file by the attacker happened after the second open of *rpm* (outside the window), but the attack still succeeds. In these cases, we believe that append started after *bash* opened the script file (the second **open** of *rpm*), but it finished before *bash* reached the end of the script. Since *bash* interprets the script line by line, there is a good chance of executing the newly appended line. These two explanations (CPU yielding and slow interpretation of the script) help explain the lengthening of the window of vulnerability and the high attack success rate of 85%.

2.3.2.3. vi 6.1 Vulnerability

The Unix "visual editor" *vi* is a widely used text editor in many UNIX-style environments. For example, Red Hat Linux distribution includes *vi* 6.1. Using our tools, we found potential TOCTTOU vulnerabilities in *vi* 6.1. Specifically, if *vi* is run by root to edit a file owned by a normal user, then the normal user may become the owner of sensitive files such as /etc/passwd.

while ((fd = **mch_open**((char *)*wfname*, …) …… **chown**((char*)*wfname*, st_old.st_uid, st_old.st_gid); (a)	if (**rename** (temp_filename, *real_filename*) != 0) { … } chmod (*real_filename*, st.st_mode); **chown** (*real_filename*, st.st_uid, st.st_gid); (b)

Figure 7: (a) *vi* 6.1 vulnerability (fileio.c), (b) *gedit* 2.8.3 vulnerability (gedit-document.c)

The problem can be summarized as follows. When *vi* saves the file being edited, it first renames the original file to a backup, then creates a new file under the original name (*wfname* in **Figure 7**(a)). The new file is closed after all the content in the edit buffer has been written to it. If *vi* is running as root, the initial owner and group of this new file is root, so *vi* needs to change the owner and group of the new file to its original

owner and group. This forms an <**open, chown**> window of vulnerability every time *vi* saves the file (**Figure 7**(a)). During this window, if the file name can be changed to a link to /etc/passwd, then *vi* can be tricked into changing the ownership of /etc/passwd to the normal user.

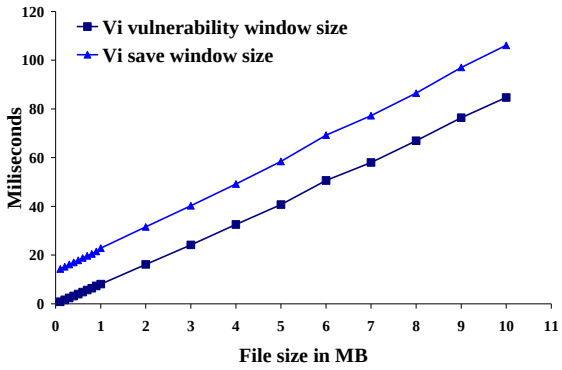

Figure 8: Vulnerability and Save Window Sizes of *vi*

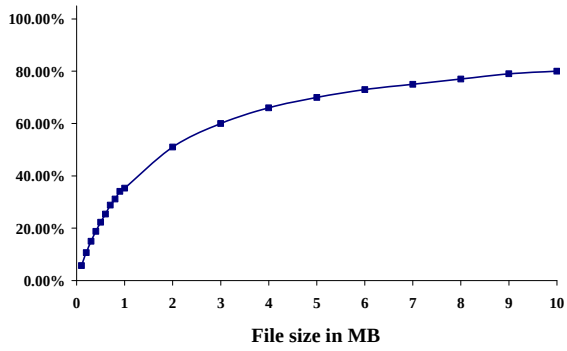

Figure 9: Window of Vulnerability Divided by Total Save Time, as a Function of File Size

2.3.2.3.1 Baseline Analysis of vi

Using the same method of the *rpm* study, we measured the percentage of time when *vi* is running in its vulnerability window as it saves the file being edited. In *vi*, this depends on the edited file size. In our experiments, we bypass the user typing time to avoid the variations caused by human participation.

We define the save window *t* as the time *vi* spends in processing one "save" command, and the vulnerability window *v* during which TOCTTOU attack may happen. We measured 60 consecutive "saves" of the file for *t*, and timestamp the **open** and **chown** system calls for *v*. Since the "save" time of a file depends on the file size, we did a set of experiments on different file sizes. Figure 8 shows the time required for a "save" command for files of sizes from 100KB to 10MB. We found a per file fixed cost that takes about 14msec for the small (100KB) file and an incremental cost of 9msec/MB (for files of size up to 10MB).

Since **chown** happens after the file is completed, the window of vulnerability *v* follows approximately the same incremental growth of 9msec/MB (see Figure 8). Figure 9 shows the window of vulnerability to be relatively long compared to the total "save" time. It gradually grows to about 80% of the "save" total elapsed time for 10MB files. This experiment tells us that *vi* is more vulnerable when the file being edited is larger. For a small file (100KB size) the window of vulnerability is still about 5% of the "save" time.

```
1  while (!finish){
2    if (stat(wfname, &stbuf) == 0){
3      if ((stbuf.st_uid == 0) && (stbuf.st_gid == 0))
4        {
5          unlink(wfname);
6          symlink("/etc/passwd", wfname);
7          finish = 1;
8        }
9    }
10 }
```

Figure 10: A program to attack *vi*

2.3.2.3.2 An Experiment to Exploit vi

Unlike a batch program such as *rpm*, which is easily run from a script, *vi* is designed for interactive use by humans. To eliminate the influence of human "think time" in the experiments, we wrote another program to interact with *vi* by sending it commands that simulate human typing. This reduces the runtime and the window of vulnerability to minimum. The experiment runs a *vi* (as root) editing a file owned by the attacker in the attacker's home directory. The editing consists of either appending or deleting a line from the file and the experiment ends with *vi* exiting.

The attack (Figure 10) consists of a tight loop constantly checking (by **stat**-ing) whether the owner of the file has become root, which signifies the start of the window of vulnerability. Once this happens, the attacker replaces the file with a symbolic link to /etc/passwd (as shown in Figure 10 and Figure 11). When *vi* exits, it should change the

ownership of /etc/passwd to the attacker. If *vi* finishes and /etc/passwd is still owned by root, the attack fails.

Contrary to the surprisingly high probability of success in the *rpm* case, we found a relatively low probability of success in the *vi* case (see Figure 12 and Figure 13), despite a relatively wide window of vulnerability. This leads to a more careful analysis of the system events during the attack.

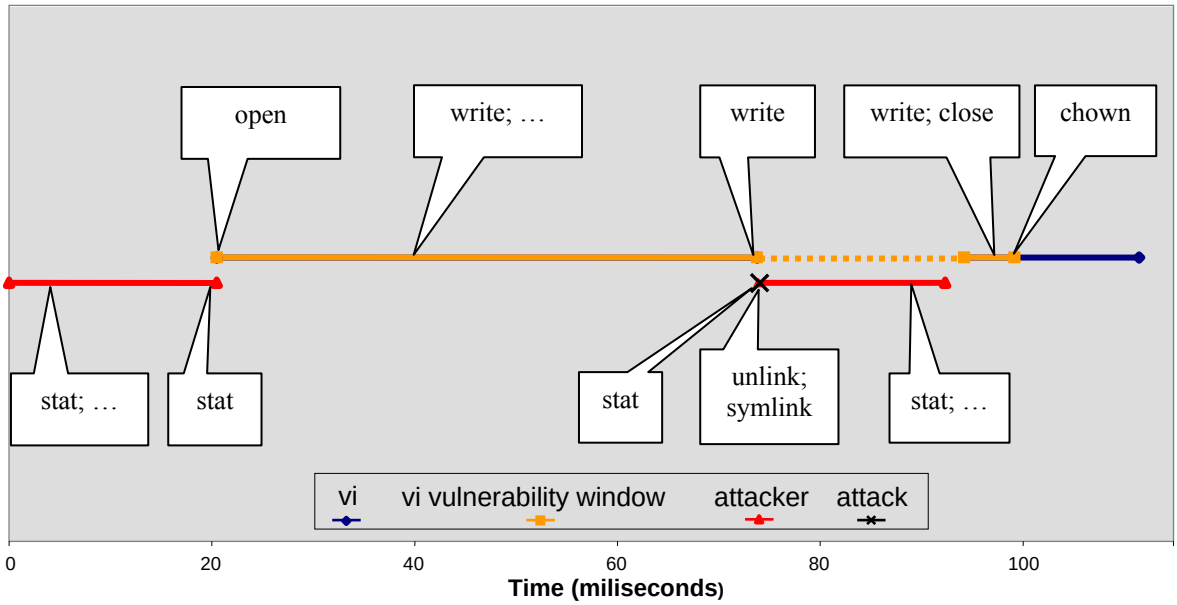

Figure 11: Event Analysis of the *vi* Exploit

2.3.2.3.3 Event Analysis of vi Exploit

Although the window of vulnerability may be wide, an attack will succeed only when:

1. *vi* has called **open** to create the new file,
2. *vi* has not called **chown,**
3. *vi* relinquishes CPU, voluntarily or involuntarily, and the attacker is scheduled to run, and
4. the attacker process finishes the file redirection during this run.

The first two conditions have been studied in the baseline experiment. The fourth condition depends on the implementation of the attacker program. For example, if the attacker program is written in C instead of shell script, it will be less likely to be interrupted.

The third condition is the least predictable. In our experiments, we have found several reasons for *vi* to relinquish CPU. First, *vi* may suspend itself to wait for I/O. This is likely since the window of vulnerability includes the writing of the content of the file, which may result in disk operations. Second, *vi* may use up its CPU slice. Third, *vi* may be preempted by higher priority processes such as *ntpd*, *kswapd*, and *bdflush* kernel threads. Even after *vi* relinquishes CPU, the second part of the condition (that the attacker process is scheduled to run) still depends on other processes not being ready to run.

This analysis illustrates the highly non-deterministic nature of a TOCTTOU attack. To achieve a statistically meaningful evaluation, we repeat the experiments and compute the probability of attack success. To make the experimental results reproducible, we eliminated all the confounding factors that we have identified. For example, in each round of experiments, we ran *vi* at least 50 times, each time on a different file, to minimize file caching effects. We also observed memory allocation problems after large files have been used. To relieve memory pressure, we added a 2-second delay between successive *vi* invocations.

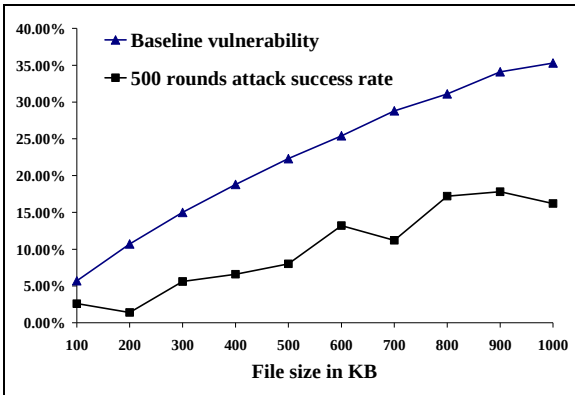

Figure 12: Success Rate of Attacking *vi* (small files)

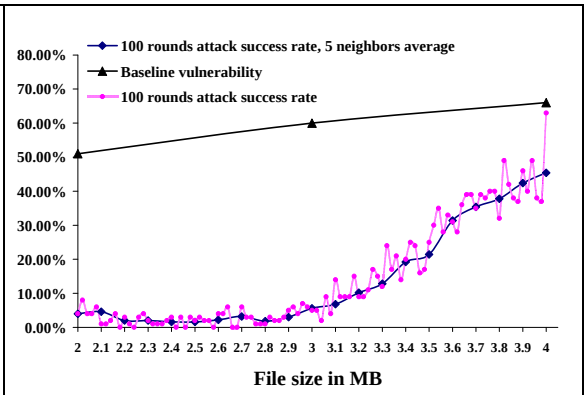

Figure 13: Success Rates of Attacking *vi* (large files)

Figure 12 shows the success rate for file sizes ranging from 100KB to 1MB averaged over 500 rounds. We see that for small files, there is a rough correlation between the size of window of vulnerability and success rate. Although not strictly linear, the larger the file being edited, the higher the probability of successfully attacking vi.

Figure 13 shows the results for file sizes ranging from 2MB to 4MB, with a stepping size of 20KB, averaged over 100 rounds. Unlike the dominantly increasing success rate for small file sizes, we found apparently random fluctuations on success rates between file sizes of 2MB and 3MB, probably due to race conditions. For example, files of size 2MB have success rate of 4%, which is lower than the 8% success rate of file size 500KB in Figure 12. The growing success trend resumes after files become larger than 3MB.

2.3.2.4. Other Vulnerabilities

In our experiments, we identified 5 TOCTTOU pairs (see Table 7) and confirmed 3 of them through direct attacks (*rpm*, *vi*, and *emacs*). Due to its similarity to the *vi* experiments (Section 2.3.2.3), the analysis of the attack of *emacs* is omitted here.

We also tried to attack *gedit*, the fourth vulnerability discovered. *gedit* [25] is a text editor for the GNOME desktop environment, and version 2.8.3 of *gedit* has a <**rename**, **chown**> TOCTTOU vulnerability (See **Figure 7**(b)). Like *vi*, *gedit* becomes vulnerable when it is run by root to edit a file (*real_filename*) owned by a normal user (also the attacker), and it saves the file. Unlike *vi*, *gedit* writes to a temporary scratch file, then renames the scratch file to the original file name, and calls **chown**. Thus the window of vulnerability is between the **rename** and the directly following **chown**, a very short time that reduces the probability of successful attack. Not surprisingly, our attack experiment (using the program in Figure 14) found no success on a uniprocessor.

However, as we will discuss in more detail in section 2.4.4, this is not the case once *gedit* is running on multiprocessors.

The fifth vulnerability is the Enlightened Sound Daemon (*esd*), which creates a directory /tmp/.esd and then changes the access mode of this directory to 777, giving full permissions (read/write/execute) to all users. Besides, this directory is under /tmp, a place where any user can create files or directories. So a possible attack is to create a symbolic link /tmp/.esd before the **mkdir** call of *esd* and let the link point to some directories owned by the running user (such as his/her home directory). If *esd* does not check whether its **mkdir** call succeeds, then it will change the access mode of the running user's home directory to 777. Then an attacker has full access to the running user's home directory. We postponed our experiments on *esd* since this TOCTTOU vulnerability has been reported in BUGTRAQ [12].

```
1  while (!finish){
2    if (stat(real_filename, &stbuf) == 0){
3      if ((stbuf.st_uid == 0) && (stbuf.st_gid == 0))
4        {
5            unlink(real_filename);
6            symlink("/etc/passwd", real_filename);
7            finish = 1;
8        }
9    }
10 }
```

Figure 14: gedit attack program version 1

Overall, we consider the CUU model-based detection framework to be a success. With a modest number of experiments, we confirmed known TOCTTOU vulnerabilities and found several previously unreported ones. However, this offline analysis only covers the execution paths exercised by the workloads, so it cannot guarantee the absence of TOCTTOU vulnerabilities when none is reported.

2.3.3. Evaluation of Detection Method

2.3.3.1. Discussion of False Negatives

As mentioned in Section 2.3.2.1, our tools are not designed for exhaustive testing. While we attempted to generate representative workloads for the 130 programs tested, we cannot guarantee coverage of all execution paths. The coverage problem may be alleviated by improvements in the testing technology and documentation.

More fundamentally, the CUU-Model covers pairs of file system calls, assuming that a precondition is established by the "Check" call before the "Use" call relies on it. In programs where preconditions are not explicitly established (a bad programming practice), e.g., a program creates a temporary file under a known name without first **stat**-ing the existence of the file, exploits may happen outside the CUU model. The problem of complex interactions among more than a pair of system calls is an open research question. (Currently, there are no known examples of such complex vulnerabilities.)

2.3.3.2. Discussion of False Positives

Tool-based detection of vulnerabilities typically does not achieve 100% precision. The framework described in Section 2.3.1 is no exception. There are some technical sources of false positives:

1. Incomplete knowledge of search space: The list of immune directories (Table 5) is not complete because of the dynamic changes to system state (e.g. newly created root-owned directories under /usr/local), which leads to false positives.
2. Artifacts of test environment: If the test cases themselves uses /tmp or the home directory of an ordinary user, our tools have to report related TOCTTOU pairs, which are false positives. For example, the initial test case for *cpio* uses a temporary directory /tmp/cpio, so the tools reported a <**stat, chdir**> on this directory.
3. Coincidental events: Because our tools do system-wide monitoring, they capture file system calls made by every process. Sometimes two unrelated processes happen to make system calls on the same file that appear to be a TOCTTOU pair.

4. Incomplete knowledge of application domain: Not every TOCTTOU pair is profitably exploitable. For example, the application *rpm* invoked by "--addsign" option contains a <**stat, open**> pair, which can open any file in the system for reading, such as /etc/shadow. However, *rpm* can not process /etc/shadow because it is not in the format recognizable by *rpm*. So it is unlikely that this pair can be exploited to undermine a system.

By improving the kernel filter (source 1), re-designing test cases (source 2), and reducing concurrent activities (source 3), we reduced the false positive of our tools; for example, in one experiment testing 33 Linux programs under /bin, the false positive rate fell from 75% to 27%. However, source 4 is hard to remove due to the differences among application domains.

2.3.3.3. Overhead Measurements

To evaluate the overhead of our detection framework, we ran a variant of the Andrew benchmark [28]. The benchmark consists of five stages. First, it uses **mkdir** to recursively create 110 directories. Second, it copies 744 files with a total size of 12MB. Third, it **stat**s 1715 files and directories. Fourth, it *grep*s (scan through) these files and directories, reading a total amount of 26M bytes. Fifth, it does a compilation of around 150 source files. For every stage, the total running time is calculated and recorded. We run this benchmark for 20 rounds and get the average. To mitigate the interference from other processes during the run, we start Red Hat Linux in single-user mode (without X window system and daemon processes such as *apmd, crond, cardmgr, syslogd, gpm, cups* and *sendmail*). To get an estimation of the overhead of our system, we run this experiment on a Linux box without modifications to get the baseline results, and then a Linux box with our monitoring tools (without the Analyzer and the Inspector which are used offline). For the latter case, we ran the experiment under two different directories to

see the influence of file pathname to the overhead. The total running time of these five stages for the experiments is shown in Figure 15 and Table 9.

Table 9: Andrew Benchmark Results (msec)

Functions	Original Linux	Modified Linux Immune Dir		Modified Linux Vulnerable Dir	
		Time	Overhead	Time	Overhead
mkdir	2.8 ±0.06	3.0 ±0.10	7.1%	4.1 ±0.05	46%
copy	59.2 ±0.49	64.8 ±2.2	9.5%	80.8 ±0.46	36%
stat	61.1 ±0.55	69.4 ±0.41	14%	149.3 ±3.5	144%
grep	543.1 ±2.4	576.2 ±5.9	6.1%	645.3 ±3.7	19%
compile	20,668 ±66	20,959 ±90	1.4%	21,311 ±195	3.1%

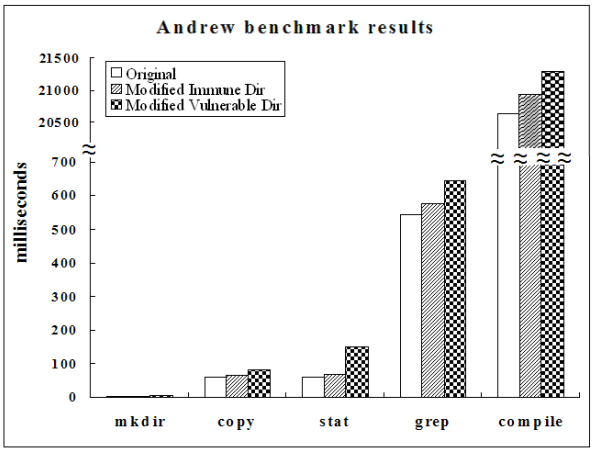

Figure 15: Andrew Benchmark Results

The results show a relatively higher overhead for **mkdir, copy** and **stat** when the benchmark is run under an ordinary user's home directory (denoted Vulnerable Dir in Figure 15 and Table 9). But when the benchmark is run under /root (denoted Immune Dir in Figure 15 and Table 9), the overhead becomes much lower (dropping from 144% to 14% for **stat**). This difference shows that **printk**s in the kernel and the Collector daemon process contribute significantly to the overhead, because the filter in kernel suppresses most log messages caused by the benchmark when it runs in a directory

immune to TOCTTOU (Table 5), therefore the **printk**s and Collector have much less work to do. The other source of overhead comes from the Sensor (including the filter and a query of the internal /proc file system data structure to map a process id to the complete command line to assist the Inspector). However, the overhead of our detection tools is amortized by application workload, as shown for compilation.

PostMark benchmark [43] is designed to create a large pool of continually changing files and to measure the transaction rates for a workload approximating a large Internet electronic mail server. Since mail server software such as *sendmail* had well known TOCTTOU problems, PostMark seems to be another representative workload to evaluate the performance overhead of our software tools.

When PostMark benchmark is running, it first tests the speed of creating new files, and the files have variable lengths that are configurable. Then it tests the speed of transactions. Each transaction has a pair of smaller transactions, which are either read/append or create/delete.

On the original Linux kernel the running time of this benchmark is 30 seconds. On our modified kernel, with all the same parameter settings, the running time is 30.35 seconds when the experiment is run under /root (an immune directory), and 35 seconds when the experiment is run under a vulnerable directory. So the overhead is 1.17% and 16.7% for these two cases, respectively. This result also shows that the **printk**s and the Collector contribute significantly to the overhead.

2.4. Probabilistic Analysis of TOCTTOU Attacks

Traditionally, attacks exploiting race conditions such as TOCTTOU have been condidered rare and “low risk”. Our TOCTTOU attack experiments against *vi* on uniprocessors (Section 2.3.2.3) seem to support this belief. However, one major reason for the low attack success rate is that the CPU is a bottleneck – the attacker simply cannot get a chance to run. Once the CPU is no longer the bottleneck, the situation may change.

For example, a multiprocessor will give the attacker the option of running on a dedicated processor and actively seeking attack opportunities. So the attacker may achieve a higher rate of success on a multiprocessor.

In this section, we present a probabilistic analysis of TOCTTOU attacks taking multiprocessors into account. We first propose a probabilistic model which shows that multiprocessors increase the success rate of TOCTTOU attacks, especially when the victim program is rarely suspended in the vulnerability window. Then we perform a detailed experimental and event analysis of TOCTTOU attacks on multiprocessors, to confirm the applicability of our model. We use *vi* and *gedit* as the victim programs in the attack experiments, which contain new TOCTTOU vulnerabilities that were found by our detection tools (Section 2.3.2).

2.4.1. A Probabilistic Model for Predicting TOCTTOU Attack Success Rate

2.4.1.1. The Basic General Model

A TOCTTOU attack succeeds when the attacker is able to modify the mapping from file name to disk block within the vulnerability window. In order to succeed, the attacker must first find the vulnerability window, and then change the file mapping. Therefore, our model divides the attacker program into two parts: (1) a detection part that finds the beginning of the vulnerability window, and (2) an attack part that modifies the file mapping.

One of the critical issues is whether the victim is suspended within the vulnerability window, since the suspension increases substantially the success rate. Based on the law of total probability, the attack success rate:

P(attack succeeds) = P(victim suspended) * P(attack succeeds | victim suspended) +
P(victim not suspended) * P(attack succeeds | victim not suspended)

In addition, in order for the attack to succeed, the attacker program must be scheduled within the vulnerability window and the attack must finish within the vulnerability window, so

P(attack succeeds | victim suspended) = P(attack scheduled * attack finished | victim suspended)
= P(attack scheduled | victim suspended) * P(attack finished | victim suspended)

We can derive P(attack succeeds | victim not suspended) in a similar way and get the refined probability in Equation 1.

In Equation 1, all the events are under the context of the victim vulnerability window. e.g. "attack finished" means "attack finished within the vulnerability window".

P(attack succeeds) = P(victim suspended) * P(attack scheduled | victim suspended) * P(attack finished | victim suspended)
+ P(victim not suspended) * P(attack scheduled | victim not suspended) * P(attack finished | victim not suspended)

Equation 1: The probability of a successful TOCTTOU attack

2.4.1.2. Attack Success Rate on a Uniprocessor

On a uniprocessor, P(attack scheduled | victim not suspended) = 0 since it is impossible to schedule the attacker when the victim is running. Therefore on a uniprocessor the second part of Equation 1 contributes nothing to the success rate. I.e., P(attack succeeds) = P(victim suspended) * P(attack scheduled | victim suspended) * P(attack finished | victim suspended).

Several observations can be made about P(attack succeeds) on a uniprocessor:

- P(attack succeeds) ≤ P(victim suspended). This means that the probability that the victim is suspended within its vulnerability window gives an upper bound for the attack success rate. If the victim is always suspended (e.g. *rpm* in 2.3.2.2), the attacker can achieve a success rate as high as 100%. In contrast, if the victim is rarely suspended (e.g. *gedit* in Section 2.3.2.4), the attack success rate can be near zero.

- P(attack scheduled | victim suspended) is the probability that the attacker process gets scheduled when the victim relinquishes CPU. This value depends on several factors such as the readiness of the attacker, the system load (if round-robin scheduling is used), or the priority of the attacker (if priority-based scheduling is used). Typically in a lightly loaded environment this value can be nearly 100% if the attacker program uses an infinite loop actively looking for the exploit opportunity.
- P(attack finished | victim suspended) is the probability that the attacker successfully modifies the file mapping while the victim is suspended. Since there is only one CPU, as long as the attack part is not interrupted, this probability can be 100%. Typically this is the case because modifying the file mapping requires very short processing time and needs not block on I/O.

Based on the above analysis, the attack success rate is mainly determined by P(victim suspended) on a uniprocessor system, and the implementation of the attack part is relatively less critical.

2.4.1.3. Attack Success Rate on Multiprocessors

On multiprocessors, the attacker can run on a different processor than the victim when the victim is running within its vulnerability window. This makes the second part of Equation 1 non-zero, i.e., P(attack scheduled | victim not suspended) > 0. This fact increases the success rate of TOCTTOU attacks on multiprocessors as compared to uniprocessors. If P(victim suspended) is relatively large, then the success rate on multiprocessors may not increase significantly. However, if P(victim suspended) is very small (approaching 0), then P(victim not suspended) approaches 1, and the gain due to the second part of P(attack succeeds) may become very significant.

Therefore for an attacker, the benefit of having multiprocessors is maximized when the victim is rarely suspended in the vulnerability window. An analysis of the second part of Equation 1 shows that:

- P(attack scheduled | victim not suspended) is similar to P(attack scheduled | victim suspended) discussed in Section 2.4.1.2. The conclusion is that it can be as high as 100%.
- P(attack finished | victim not suspended) is the probability that the attack is finished within the vulnerability window. Since the victim is running concurrently with the attacker, the result of the attack depends on the relative speed of the attacker and the victim, a more detailed analysis is needed (next Section).

2.4.1.4. Probabilistic Analysis of P(attack finished | victim not suspended)

In order to predict P(attack finished | victim not suspended) in more detail, we analyze the race condition at different levels: the first level is CPU, which is the main contention in uniprocessor attacks; the next level is file object, because the file system already has a synchronization mechanism to regulate shared accesses. In Unix-style file systems, the modifications to an inode are synchronized by a semaphore. Since the operations of the victim and the attacker on the shared file modify the same inode, they both need to acquire the same semaphore. In this case, the race is reduced to the competition for the semaphore and we can model the success rate of the attack in the following way.

In this model, we assume that the attacker runs in a tight loop (the detection part), waiting for the vulnerability window of the victim to appear. Let D be the time consumed by each iteration of detection part, and let t_1 be the earliest start time for a successful detection and t_2 be the latest start time for a successful detection followed by a

successful attack (e.g. the attacker acquires the semaphore first). t_1 and t_2 are determined by the victim process. Some observations can be made as follow (Figure 16):

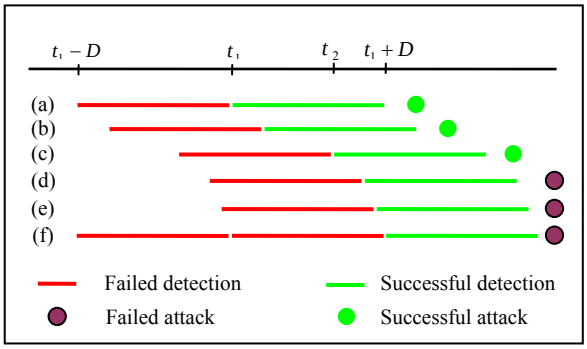

Figure 16: Different attack scheduling on a multiprocessor

A successful attack starts with a successful detection as its precondition. This successful detection may start as early as t_1 (Figure 16, case (a)), and as late as $t_1 + D$ (Figure 16, case (f)). Then the interval $[t_1, t_1 + D)$ is our sample space. Out of this interval $[t_1, t_1 + D)$, if the detection is started before t_2, the attack succeeds (Figure 16, cases (a) through (c)); otherwise the attack fails (Figure 16, cases (d) through (f), because the attack is launched too late). Let's assume a uniform distribution for the start time of the detection part, the success rate is thus $\frac{t_2 - t_1}{D}$.

In Figure 16 we assume that $t_2 \in [t_1, t_1 + D)$. Two other cases are:

- If $t_2 < t_1$, then the success rate is 0;
- If $t_2 \geq t_1 + D$, then the success rate is 1.

Let $L = t_2 - t_1$, and we get:

$$\text{The success rate} = \begin{cases} 0, if\ (L < 0) \\ L / D, if\ (0 \leq L < D) \\ 1, if\ (L \geq D) \end{cases} \qquad (1)$$

In formula (1), L measures the laxity of the successful attacks, which is a characterization of the victim: the larger L, the more vulnerable the victim. D is a characterization of the detection part of the attacker: the smaller D, the faster the attacker,

and the higher success rate. So L/D gives a very useful measurement of the relative speed of the victim and the attacker.

It should be noted that L and D in formula (1) are not strictly constant, because the executions of the victim as well as the attacker are interleaved with other events (e.g. kernel timers) in the system. That is, the running environment imposes variance on these parameters. So formula (1) only offers a statistical guidance about the attack success rate.

2.4.2. Baseline Measurements of TOCTTOU Attacks on Uniprocessors

For comparison purposes, in this section we summarize the measured success rates of *vi* and *gedit* TOCTTOU attacks on uniprocessors from Section 2.3.2.

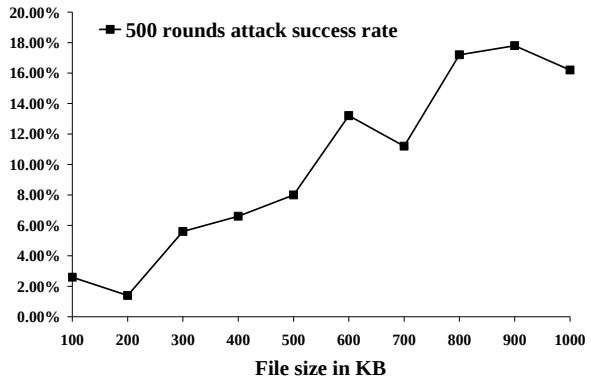

Figure 17: Success rate of attacking vi (small files) on a uniprocessor

2.4.2.1. vi Attack Experiments on Uniprocessors

Since the *vi* vulnerability window includes the writing of a whole file, the size of the window naturally depends on the file size. The measured success rates for file sizes ranging from 20KB to 10MB are the following:

- When the file size is small (from 100KB to 1MB), there is a rough correlation between attack success rate and file size, as shown in Figure 17. However, the correlation disappears for larger file sizes (e.g., between 2MB to 3MB), showing that file size alone does not determine the success rate completely.

- Besides file size, we studied other factors (e.g., I/O operation, CPU slicing, and preemption by higher priority kernel threads) that corroborate the non-deterministic nature of TOCTTOU attacks on a uniprocessor (Section 2.3.2.3).

From Figure 17 we can see that for normal file sizes (Using *vi* to edit a 2MB text file is considered rare in real life), the success rate can be as low as 1.5% and as high as 18%. Furthermore, when the file size approaches 0, the success rate also approaches 0.

2.4.2.2. gedit Attack Experiment on Uniprocessors

The experiments in which a TOCTTOU attack was carried out against the *gedit* vulnerability saw no successes. This is because the *gedit* vulnerability window (Figure 7(b)) does not include the writing of the new file as in *vi*, so it is much shorter and bears no relationship to the file size. These factors reduced the success rate for *gedit* attacks to essentially zero on a uniprocessor.

2.4.3. vi Attack Experiments on SMP

We repeated the *vi* attack experiments described in Section 2.3.2.3 on a SMP machine (2 Intel Xeon 1.7GHz CPUs, 512MB main memory, and 18.2GB SCSI disk with ext3 file system).

First we tried different file sizes ranging from 20KB to 1MB with a stepping size of 20KB, and observed the success rate of 100% for all file sizes. This confirms the probabilistic predictions in Section 2.4.1.3 and shows that a multiprocessor greatly increases the attacker's chance of success compared to a uniprocessor (Figure 17 in Section 2.4.2.1). We did a detailed event analysis to confirm the attacker and victim processes ran on separate CPUs during the vulnerability window. We also eliminated the possibility that the attack success is due to the victim being blocked on I/O operations (which would have made the attack easier). Consequently, we conclude that the attack

success is due to the length of *vi* vulnerability window being much larger than the time it takes the attacker to finish the attack steps (file name redirection).

Figure 18 shows the L and D values (Section 2.4.1.4) for the *vi* attack experiments that we conducted on the SMP. We can see that L >> D when the file is large (e.g.1MB); and the difference (L – D) decreases as the file size decreases. But (L – D) is always positive, even when the file size becomes very small. Therefore we can say with almost certainty that for *vi* attack experiments, L > D. By formula (1) we know that the success rate of *vi* attacks is almost 100% all the time.

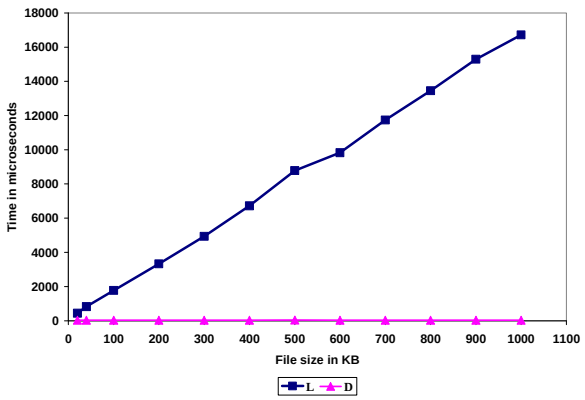

Figure 18: The L and D values for vi SMP attack experiments

One thing to notice from Figure 18 is that as the file size approaches 0, the difference (L – D) also approaches 0. Is it possible that L becomes smaller than D? Then according to formula (1) the attack success rate will be smaller than 100%.

To see this we run the experiment again with the smallest files (only 1 byte each). And the success rate we get is around 96%. Again we did a detailed event analysis of this experiment. We measure the average L and D values and put them in Table 10. We can see that although L > D in these attacks, they have become very close. If we consider the fact that the values for L and D are not strictly constant due to the environmental influence, we realize that whether L > D all the time becomes questionable when they are close enough (When L >> D the inaccuracy introduced by the environment does not

change the relationship). This helps to explain why the success rate can not be 100% when the file contains only 1 byte.

Table 10: The average L and D values (in microseconds) for *vi* SMP attack experiments (file size = 1 byte)

	Average	Stdev
L	61.6	3.78
D	41.1	2.73

Another point is that so far we actually treat P(attack finished | victim not suspended) in Section 2.4.1.4 as the sole basis for predicting the success rate, which is not always accurate (Equation 1). The justification is that when the *vi* vulnerability window is large enough, the effect of other factors in Equation 1 is negligible. For example, P(attack scheduled | victim not suspended) < 100% in general which means that the attacker may not be scheduled during sometime in the vulnerability window. However, if the vulnerability window is very large, the attacker is still within it when he/she is scheduled eventually. That is, the temporary suspension does not affect the result of the attack. However, when the vulnerability window becomes small enough (e.g. L and D become close enough), the suspension may cause the attacker to miss the vulnerability window. In such a case the attack fails, thus the suspension changes the attack result.

In several of the failed 1-byte *vi* experiments, we find that some other processes prevents the attacker from being scheduled on another CPU during the *vi* vulnerability window.

This analysis tells us that although using a multiprocessor can greatly increase the attack's chance of success, the success is still not guaranteed: the attack is still influenced by other environmental factors such as kernel activities and system load. However, 96% is more than enough for an attacker.

2.4.4. *gedit* Attack Experiments on Multiprocessors

2.4.4.1. gedit SMP Attack Event Analysis

As mentioned in Section 2.4.2.2, our attack experiments against *gedit* on uniprocessors saw no successes. However, when we try this attack on a SMP (the same machine as in Section 2.4.3), we get roughly 83%, a surprisingly high success rate. A detailed event analysis is thus conducted to understand this result.

For the *gedit* attack, we have observed that if the attacker's **unlink** is invoked before *gedit*'s **chmod** (Figure 7(b) and Figure 14), then attack succeeds. This is because these two system calls compete for the same semaphore, so if **unlink** wins, **chmod** as well as the following **chown** will be delayed. As a result the attacker's **unlink** and **symlink** can have enough time to finish before *gedit*'s **chown**. On the other hand, if **unlink** loses, **unlink** and the following **symlink** of the attacker will be delayed, so the attack will fail. So there is an interesting cascading effect in *gedit* attack experiment. Therefore, for *gedit* attacks, t_1 is somewhere within the execution of **rename** (the attacker does not need to wait until the end of **rename** to see that *real_filename* has been created), D is the interval between the start of **stat** and the start of **unlink**. Let t_3 be the start of **chmod**, then $t_2 = t_3 - D$, and $L = t_2 - t_1 = t_3 - D - t_1$. We experimentally get the L and D values as in Table 11.

Table 11: L and D values for *gedit* attacks on a SMP (in microseconds)

	Average	Stdev
L	11.6	3.89
D	32.7	2.83

The calculation of L here is not accurate because the estimation of t_1 is not accurate. Currently t_1 is established as the earliest observed start time of **stat** which indicates a vulnerability window. So it may not be optimal. An earlier (thus smaller) t_1 will result in a larger L. So the success rate indicated by Table 11 (35%) may be overly conservative compared to the observed success rate.

An important contributing factor to L is the computation time between the end of **rename** and the start of **chmod**. The average length of this computation is 43 microseconds. As we will see in Section 2.4.4.2, this factor is very important for the high success rate of *gedit* attack on the SMP.

There is another contributing factor. Usually when *gedit*'s **chmod** is blocked, the Linux kernel will try to schedule something else to run (e.g. internal kernel events such as soft IRQs, kernel timers and tasklets), which further lengthens **gedit** vulnerability window (but this contributes just a little to the delay compared with that due to the semaphore).

2.4.4.2. gedit Multicore Attack Experiment

2.4.4.2.1 Attack one

We repeat the *gedit* attack (Figure 14) on a multi-core (Dell Precision 380 with 2 Intel Pentium D 3.2 GHz dual-core and Hyper-Threading CPUs, 4GB main memory, and 80GB SCSI disk with ext3 file system). We get very different result: now we see almost no success in the same attack experiment. The main change in the situation is that the victim spends much less time between **rename** and **chmod** (3 microseconds vs. 43 microseconds), so **chmod** happens before **unlink** of the attacker, but in the SMP experiment (Section 2.4.4.1) situation is the opposite.

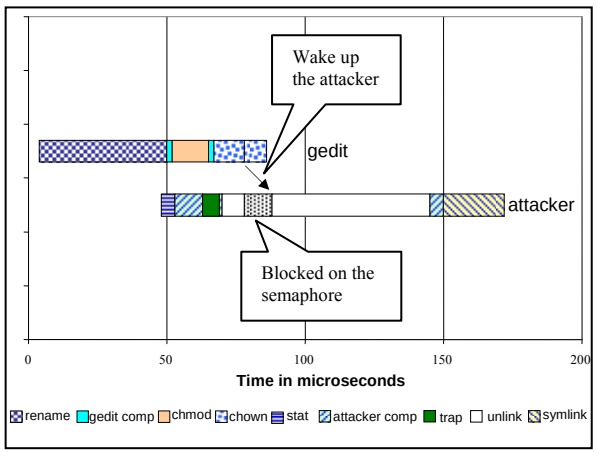

Figure 19: Failed *gedit* attack (program 1) on a multi-core

Figure 19 shows the important system events during one failed attack on the multi-core. The upper bar corresponds to the execution of *gedit* (**rename**, **chmod**, **chown**) and the lower bar corresponds to that of the attacker (**stat**, **unlink**, **symlink**). Notice that the gap (the computation) between **rename** and **chmod** of *gedit* is only 3 microseconds, but the gap between **stat** and **unlink** of the attacker is 17 microseconds. It is because of this relatively larger gap that the attacker's **unlink** is called later than the victim's **chmod**. Actually we can see that **unlink** is called later than **chown** and as a result **unlink** has to wait on the semaphore during its execution. The 17 microsecond gap of the attacker includes 11 microseconds of computation and 6 microseconds of system trap processing (page fault). Speaking in terms of D, these 17 microseconds are counted so D is around 22. On the other hand L is around $3 - D = -19$, so according to formula (1) the attack success rate is probably 0. Putting this in another way, the victim is now much faster than the attacker, so it is very difficult for the attacker to win the race.

2.4.4.2.2 Attack Two

We think that the 17 microsecond gap in **Figure 19** is mainly responsible for the low success rate. If we could reduce the length of this gap then the situation may change. A source code analysis tells us that before the vulnerability window the true branch of statement 3 in Figure 14 (statements 5 to 7) is never taken. Once the vulnerability window starts, the true branch of statement 3 is taken, and then statement 5 (**unlink**) is about to be executed. Right at this point the attacker program encounters a trap (page fault). We figure out that this effect is due to the memory management for shared libraries in Linux. Specifically, in Linux all system calls are through *libc*, which is a dynamic library shared among user-level applications. To save physical memory, Linux kernel keeps only one copy of *libc* in physical memory, and its virtual memory mechanism maps the pages of this copy to the address space of an application on demand. For example, the physical page containing the wrapper for **unlink** is mapped into an

application's address space when this application first invokes **unlink**. This mapping is preceded by a trap (page fault) and the corresponding handler routine carries out the mapping. This is exactly what happens in Figure 14, where **unlink** is first invoked when the true branch of statement 3 is taken. As a consequence, if we intentionally invoke **unlink** (and **symlink** although it seems to be on the same page as **unlink**) before the true branch of statement 3 is taken, we may remove the trap (page fault).

So we re-implement the attacker program as shown in Figure 20. Now **unlink** and **symlink** are called no matter the vulnerability window appears or not. The only trick is to switch in the correct file name when it does appear.

Then we perform the *gedit* attack experiment again using the program in Figure 20. And we begin to see many successes!

```
1   while (!finish){  /* argv[1] holds real_filename */
2     if (stat(argv[1], &stbuf) == 0){
3       if ((stbuf.st_uid == 0) && (stbuf.st_gid == 0))
4         {
5           fname = argv[1];
6           finish = 1;
7         }
8       else
9           fname = dummy;
10
11      unlink(fname);
12      symlink("/etc/passwd", fname);
13    }//if stat(argv[1] ..
14  }//while
```

Figure 20: *gedit* attack program version 2

We plot the important system events during one successful *gedit* attack in Figure 21, similar to Figure 19. We can see that now the gap between **stat** and **unlink** of the attacker has decreased to 2 microseconds: the trap has disappeared. On the other hand, the gap between **rename** and **chmod** of *gedit* is 2 microseconds. So the attacker has a very narrow chance of winning the race. In this particular case, the attacker wins because his/her **stat** starts well before the end of **rename**, so he/she identifies the vulnerability

window at the first moment, and invokes **unlink** ahead of **chmod**. Has the attacker been 2 microseconds later, the attack would fail.

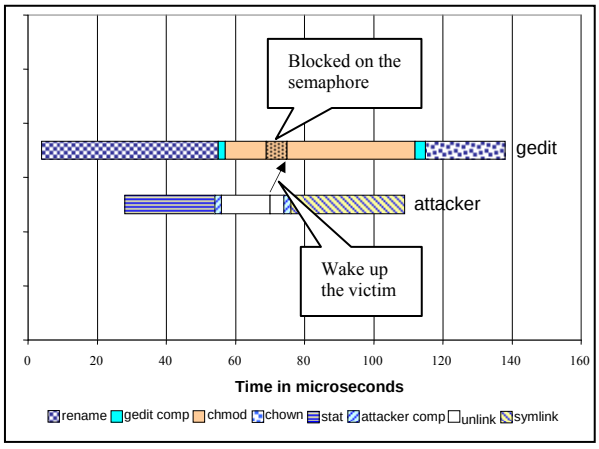

Figure 21: Successful *gedit* attack (program 2) on a multi-core

Notice that during this attack the running time of **stat** has been lengthened to 26 microseconds (typically it needs 4 microseconds), probably due to some other more complicated race condition (For example the contention for directory entries along the path name). We are not quite clear about the reason but this does not change the applicability of formula (1) because now we have a much earlier t_1 (27 microseconds into **rename**), which makes a L value of at least 1 microseconds.

This experience tells us that on multiprocessors the implementation of the attacker program can be very critical in determining the attack success rate, especially when the vulnerability window is very narrow.

2.4.5. Pipelining Attacker Program

The multi-core *gedit* experiment highlights the importance of the implementation of the attacker program. Concretely, we found that among the three steps of the attack (**stat**, **unlink**, **symlink**), **unlink** is the most time-consuming. A closer look into the file system source code shows that actually **symlink** needs not wait on the completion of **unlink**. Instead **symlink** can begin once the inode has been detached from the directory by **unlink**, which happens relatively early. (The main part of **unlink** is spent physically

truncating the file.) This observation shows that on a multiprocessor, the attacker can distribute its attack steps to multiple CPUs to speed up the attack part and increase its success rate.

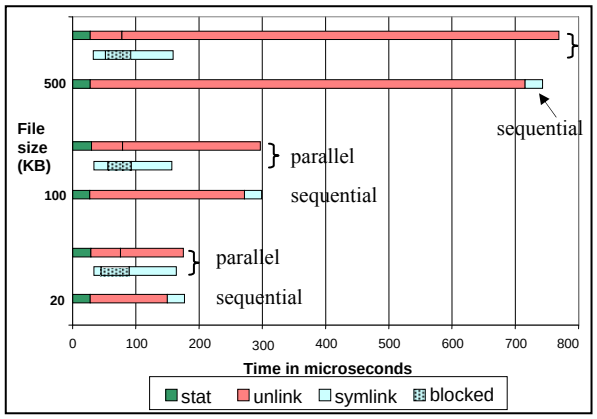

Figure 22: The effect of parallelizing the attack program

To confirm this hypothesis, we implemented a multithreaded *gedit* attack program with two threads: the first thread carries out the **stat**, **unlink** steps and the second thread carries out the **symlink** step asynchronously. Figure 22 shows the effect of parallelizing the attack program for three different file sizes. For each file size (e.g. 500KB), there are three bars: the first two bars correspond to the execution of the two threads in a parallelized attack program, and the third bar corresponds to the execution of the normal sequential attack program. In the parallelized attack, **symlink** can finish (and so does the attack) well before the end of **unlink.** This is in contrast to the sequential attack, where **symlink** has to wait until **unlink** finishes. The comparison between the end times of **symlink** shows that leveraging on the parallelism provided by a multiprocessor can greatly reduce the amount of time needed for a successful attack. This is especially important when the vulnerability window is very narrow so the attacker needs to be very fast. This experiment shows one feasible way of doing it.

2.5. A Methodical Defense against TOCTTOU Attacks: The EDGI Approach

In this part of the dissertation, we present the design, implementation, and evaluation of an event-driven defense mechanism (called EDGI) that prevents

exploitation of TOCTTOU vulnerabilities. The EDGI defense has several advantages over previously proposed solutions. First, based on the CUU model (Section 2.2), EDGI is a systematically developed defense mechanism with careful design (using ECA rules) and implementation. Assuming the completeness of the CUU model, EDGI can stop all TOCTTOU attacks. Second, with careful handling of issues such as inference of invariant scopes and time-outs, EDGI allows very few false positives. Third, it does not require changes to applications or file system API. Fourth, our implementation on Linux kernel and its experimental evaluation show that EDGI carries little overhead.

2.5.1. The Design of EDGI

2.5.1.1. Overview

We propose an event driven approach, called EDGI (Event Driven Guarding of Invariants), to defend applications against TOCTTOU attacks. The design requirements of EDGI are:

1. It should solve the problem within the file system, and does not change the API, so that existing or future applications need not be modified.
2. It should solve the problem completely, i.e., no false negatives.
3. It should not add undue burden on the system, i.e., very low rate of false positives.
4. It should incur very low overhead on the system.

EDGI consists of three design steps (described in the rest of this section), a concrete implementation (Section 2.5.2), and an experimental evaluation (Section 2.5.3). The first design step is to map the CUU model into invariants in a concrete file system (Linux in our case) and the kernel calls that preserve the invariants. The second design step uses ECA (event-condition-action) rules [26, 36] to model the concrete invariant preservation methods, so we can have reasonable assurance the invariants are indeed preserved. The third design step completes the design by addressing the remaining issues

such as the automated inference of invariant scope and inheritance of invariants by children processes.

Under the CUU model's assumption, the "Check" part of a sequence of operations on a file object creates an invariant that should be preserved through to the corresponding "Use" part. Specifically, a file certified to be *non-existent* ($resolve(f) = \varnothing$) by the "Check" operations should remain non-existent until the "Use" operations create it. Similarly, a file certified to be *existent* ($resolve(f) = b$) by the "Check" operations should remain the same file until the "Use" part (by the same user) accesses it. Identifying and preserving these two invariants ($resolve(f) = \varnothing$ or $resolve(f) = b$) are the main goals of EDGI approach.

The EDGI design treats an invariant as a sophisticated *lock*. The user invoking a "Check" call becomes the owner of the lock, and the lock is usually held by the same user through the "Use" call. Due to the complications of Unix file system, the invariant handling is more complicated than a normal lock compatibility table. Therefore, we represent the invariant handling using ECA rules, as explained in the following section. We note that we only use ECA rules as a model, since our implementation does not support general-purpose rule processing.

2.5.1.2. Invariant Maintenance

The EDGI approach adopts a modular design and implementation strategy by separating the EDGI invariant processing from the existing kernel. The invariant-related information is maintained as extra state information for each file object. When an invariant-related event is triggered, the corresponding set of conditions is evaluated and if necessary, appropriate actions are taken to maintain the invariant.

The invariant-related information for each file object includes its state (free or actively used), a tainted flag, invariant holder user id and a process list. In detail:

- refcnt – the number of active processes using the file object. When refcnt = 0, the file object is free.
- tainted – when refcnt > 0, this flag means whether the name to disk object binding can be trusted.
- fsuid – the user id of the processes that are actively using the file object.
- gh_list – a doubly-linked list, in which each node contains a process id and the timestamp of the last system call made by the process on the file object.

Two kinds of events trigger condition evaluation:

- File system calls such as **access, open,** and **mkdir**.
- Process operations: **fork, execve, exit**.

The conditions evaluated by each event and their associated actions are summarized in Table 12 (f denotes the file object). The conditions refer to the file object status (whether the invariant is $resolve(f) = \varnothing$ or $resolve(f) = b$), and actions include the creation, removal and potentially more complex invariant maintenance actions.

2.5.1.3. Inferring Invariant Scope

EDGI prevents TOCTTOU attacks by making the sequence of system calls on a file object safe. As suggested by Proposition 3 (Section 2.2.2.4), the invariant maintenance rules in Table 12 are not restricted to a TOCTTOU pair, but extend to a sequence of file system calls. During the time such a sequence of accesses exists, the file object is said to be *actively used*. Otherwise the file object is said to be *free*.

The interval during which the file object is actively used forms the scope of its invariant. The scope varies in length, depending on the number of consecutive "Use" calls made by the application. Consequently, a significant technical challenge is to correctly identify this scope - the boundaries of the TOCTTOU vulnerability window of the application. Since current Unix-style file systems are oblivious to such application-

level semantics, we need to *infer* the scope, so no changes are imposed on the applications or the file system interfaces.

Table 12: Invariant Maintenance Rules in EDGI

Name	Event	Condition	Action
Incarnation rule	Any system call on f	refcnt == 0	Set f's state as actively used (refcnt++); set its tainted flag as false, fsuid as current user id, record current pid and current system time in the gh_list.
Reinforcement rule	Any system call on f	refcnt > 0 and fsuid == current user id and tainted == false	Record current pid and current system time in the gh_list.
Abort rule	Any system call on f	refcnt > 0 and fsuid == current user id and tainted == true	Record current pid and current system time in the gh_list. Return an error.
Root preemption rule	Any system call on f	refcnt > 0 and fsuid != current user id and current user id == root	Remove all invariant holders information from the gh_list; set f's fsuid as current user id, set refcnt as 1, tainted as false, record current pid and current system time in the gh_list.
Owner preemption rule	Any system call on f	refcnt > 0 and fsuid != current user id and current user id != root and fsuid != root and current user is the owner of f	Remove all invariant holders information from the gh_list; set f's fsuid as current user id, set refcnt as 1, tainted as false, record current pid and current system time in the gh_list.
Invariant maintenance rule 1	Any system call in the RemovalSet (Section 2.2.3.2) on f	refcnt > 0 and fsuid != current user id	Traverse the gh_list to get the latest timestamp t, compute the interval between t and current time, if it is less than threshold MAX_AGE, deny the current request, otherwise grant the current request and set tainted as true.
Invariant maintenance rule 2	Any system call in the CreationSet (Section 2.2.3.2) on f	refcnt > 0 and fsuid != current user id	Traverse the gh_list to get the latest timestamp t, compute the interval between t and current time, if it is less than threshold MAX_AGE, deny the current request, otherwise grant the current request and set tainted as true.
Clone rule	Fork (parent, child)	True	For each file object that has parent in its gh_list, record child and current system time, and increment the refcnt.
Termination rule	Exit	True	Remove current pid from the gh_list of each file object that has it on its gh_list, and decrement the corresponding refcnt.
Distract rule	Execve	True	Remove current pid from the gh_list of each file object that has it on its gh_list, and decrement the corresponding refcnt.

The inference of invariant scope is guided by the CUU model, which specifies the initial TOCTTOU pair explicitly. The "Use" call of the initial pair becomes the "Check" call of the next pair, completed by the following "Use" call. According to Proposition 2, the CUU model correctly captures the TOCTTOU problem. The invariant of the initial pair is maintained from the "Check" call through the "Use" call, and then to the additional "Use" calls. The sequence continues until the program ends, a time-out or

preemption occurs (see Section 2.5.1.4). In summary, the scope of an invariant is a safe sequence of system calls (Definition 5 in Section 2.2.1.2).

2.5.1.4. Remaining Issues

There are some additional issues that need to be resolved for an actual implementation. First, if we consider the invariants as similar to locks, then the question of dead-lock and live-lock arises. For example, it is possible that an invariant holder is a long-running process which only touches a file object at the very beginning and then never uses it again. Consequently, a legitimate user may be prevented from creating/deleting the file object for a long time, resulting in denial of service. This problem can be addressed by a time out mechanism. If an invariant holder process does not access a file object for an exceedingly long time, the invariant will be temporarily disabled to allow other legitimate users to proceed. (Timeout is discussed in Section 2.5.3.2.)

If the time-out results in simple preemption (i.e., breaking the lock), then this method may be used to attack very long application runs. To prevent the preemption-related attack, we use a *tainted* bit to mark the preemption. After a preemption-related file creation or deletion, the invariant no longer holds. EDGI marks the file object as *tainted*, so the next access request from the original invariant holder will be aborted.

The second and related problem is the relationship between the current invariant holder and the next process attempting to access the file object. Up to now, we have assumed a symmetric relationship, without distinguishing legitimate users from attackers. In reality, we know some processes are more trustworthy than others. Specifically, in Unix environments we trust the file object owner and root processes completely. Consequently, we allow these processes to "break the lock" by preempting other invariant holders. Concretely, when the file object owner or root process attempt to

access a file object, they immediately become the invariant holder, and the invariant for the former holder is removed.

The third issue is the inheritance of invariants by children processes. For example, after a user process checks on a file object and becomes an invariant holder, it spawns a child process, and terminates. In the mean time, the child process continues, and uses the file object. In the simple solution, the invariant is removed when the owner (parent) process terminates. In this case an attacker can achieve a TOCTTOU attack before the child process uses the file. Thus we must extend the scope of invariants to the child process at every process creation. This invariant inheritance extension is analogous to the invariant scope extension discussed in Section 2.5.1.3.

A final question is whether the EDGI approach is a complete solution, capable of stopping all TOCTTOU attacks. For every file system call, the rules summarized in Table 12 are checked and followed. The first time a "Check" call is invoked on a file object, that user becomes the file object's invariant holder. At any given time there is at most one invariant holder for each file object. Users other than the invariant holder are not allowed to create or remove the file object (including changes to mapping between the name and disk objects). Therefore, the EDGI defense is able to stop all TOCTTOU attacks identified by the CUU model.

2.5.2. Linux Implementation of EDGI

We have implemented the design described in the previous section in the Linux file system. The implementation consists of modular kernel modifications to maintain the invariants for every file object and its user/owner. We outline the process that remembers the invariant holder of each file object (Section 2.5.2.1) and then the maintenance of the invariants (Section 2.5.2.2).

2.5.2.1. Invariant Holder Tracking

Invariant holder tracking is accomplished by maintaining a hash table of pathnames that keeps track of the processes that are actively using each file object. The index to this hash table is the file pathname, and for each entry, a list of process ids is maintained. Our modular implementation augments the existing directory entry (dentry) cache code and extends its data structures with the fields introduced in Section 2.5.1.2: fsuid, refcnt, tainted, gh_list.

Before a system call uses a file object by name, it first needs to resolve the pathname to a dentry. Our implementation instruments the Linux kernel to call the invariant holder tracking algorithm after each such pathname resolution. There are two possible approaches to implementing this algorithm. The first is to instrument the body of every system call (e.g., **sys_open**) that uses a file pathname as argument. The second is to instrument the pathname resolution functions themselves (in the Linux case, **link_path_walk** and **lookup_hash**).

The first approach has the disadvantage that instrumented code has to spread over many places, making testing and maintenance difficult. Although techniques such as Aspect Oriented Programming (AOP) [31] could help, we were unable to find a sufficiently robust C language aspect weaver tool that can work on Linux kernel. The second approach has the advantage that only a few (in the Linux case, exactly two) places need to be instrumented, making the testing and maintenance relatively easy. We chose the second approach for our implementation.

The invariant holder tracking algorithm GH is shown in Figure 23. This algorithm effectively implements the rules summarized in Table 12, and it is called right before **link_path_walk** and **lookup_hash** successfully returns.

Line 1-2 of the invariant holder tracking algorithm addresses the situation where a new invariant holder is identified: invariant related data structure is initialized, including the invariant holder user id (fsuid), the invariant holder process id, the tainted flag, and a timestamp. After these steps, the invariant maintenance part (Section 2.5.2.2) will start

applying this invariant. We can see that the same sequence also occurs in Lines 10 and 16, where a new invariant holder is decided due to preemption.

```
        Input: dentry d
        Output: 0 – succeed, -1 – the binding of d is tainted.
1       if d.refcnt = 0
2       then d.fsuid ← current user id, record current pid and current time in d.gh_list,
        d.refcnt++, d.tainted ← false, return 0.
3       else
4          if d.fsuid = current user id
5          then record current pid and current time in  d.gh_list, if d.tainted = false
6                          then return 0
7                           else  return -1.
8          else
9            if current user id = root
10           then remove all invariants on d.gh_list,  d.fsuid ← root, record current pid
        and current time in d.gh_list, d.refcnt ←1, d.tainted ← false, return 0.
11           else
12               if d.fsuid = root
13               then return 0.
14               else
15                   if current user id is the owner of d
16                   then remove all invariants on d.gh_list, d.fsuid ← current user id,
        record current pid and current time in d.gh_list, d.refcnt←1, d.tainted ← false,
        return 0.
17                   else return 0.
```

Figure 23: Invariant Holder Tracking Algorithm

Lines 4-7 address the situation in which an existing invariant holder accesses the file object again. Notice that the tainted flag is checked to abort the invariant holder process if the name to disk binding of the file object has been changed by another user's process (Section 2.5.2.2).

Lines 9-10 correspond to the preemption of invariant from a normal user to the root discussed in Section 2.5.1.4. Similarly, lines 15-16 handle the preemption by file object owner.

The invariant holder tracking algorithm needs the current process id and current user id runtime information, which are obtained from the *current* global data structure maintained by the Linux kernel.

2.5.2.2. Invariant Maintenance

The second part of implementation is invariant maintenance by thwarting the attacker's attempt to change the name to disk binding of a file object, which in turn is achieved by deleting or creating a file object. We instrumented two kernel functions to perform invariant checks:

- **may_delete**(d): this function is called to do permission check before deleting a file object d. We add invariant checking after all the existing checks have been passed: If d.refcnt > 0 and the current user id is not the same as d.fsuid, traverse d.gh_list to get the last access timestamp; if it is younger than MAX_AGE, return –EBUSY (file object in use and cannot be deleted). Otherwise set d.tainted as true and return 0.
- **may_create**(d): this function is called to do permission check before creating a file object, similar invariant checking is added after all the existing checks have been passed.

The **may_create** kernel function is called by all the system calls in the CreationSet (Section 2.2.3.2) and the **may_delete** function is called by all the system calls in the corresponding RemovalSet. These invariant checks implement the Invariant Maintenance Rules 1 and 2 in Table 12.

2.5.2.3.Engineering of EDGI Software

Table 13 shows the size of EDGI implementation in Linux kernel 2.4.28. The changes were concentrated in one file (dcache.c), which was changed by about 55% (LOC means lines of code). The other changes were small, with less than 5% change in one other file (namei.c), plus single-line changes in three other files. This implementation of less than 1000 LOC was achieved after careful control and data flow analysis of the kernel, plus some tracing. We consider this implementation to be highly modular and relatively easily portable to other Linux releases.

From top-down point of view, the methodical design and implementation process benefited from the CUU model as a starting point. Then, the ECA rules facilitated the reasoning of invariant maintenance. The rules were translated into the Invariant Holder Tracking algorithm. These steps give us the confidence that the invariants are maintained by EDGI software.

Conversely, from a bottom-up point of view, the Linux kernel was organized in a methodical way. For example, it has exactly two functions (**may_delete** and **may_create**) controlling all file object status changes. By guarding these two functions, we were able to guard all 224 TOCTTOU pairs identified by the CUU model. This kind of function factoring in the Linux kernel contributed to the modular implementation of EDGI.

Table 13: Linux Implementation of EDGI

Source File	Modified Places	Original LOC	Added LOC
fs/dcache.c	4	1,307	749
fs/namei.c	5	2,047	84
fs/exec.c	1	1,157	1
kernel/exit.c	1	602	1
kernel/fork.c	1	896	1

2.5.3. Experimental Evaluation of EDGI

2.5.3.1. Discussion of False Negatives

The EDGI system design follows the CUU model. In Section 2.5.1.4 we included an informal argument for the completeness of the CUU model, details of which can be found in Section 2.2.2. If the ECA rules summarized in Table 12 captures all the TOCTTOU pairs identified by the CUU model, and the invariant holder tracking algorithm in Figure 23 implements all the rules in Table 12, and our Linux kernel implementation (Section 2.5.2) is correct, then our implementation should have zero false negatives.

We have run all the attack experiments we could find, including known TOCTTOU vulnerabilities such as *logwatch* 2.1.1 [52] and new vulnerabilities recently detected, including *rpm*, *vi/vim*, and *emacs*. In all the experiments the EDGI system is able to stop the attacker program.

One exception to the invariant maintenance rules is the preemption by programs running as root, which are allowed to gain the invariant and change file object status at will. We consider this exception to be safe, since if an attacker has already obtained root privileges, there is no further gain for using TOCTTOU attacks.

2.5.3.2. Discussion of False Positives

As mentioned in Section 2.5.1.4, our conservation maintenance of invariants may introduce long delays, if an invariant holder runs for a long time. These long delays can be considered a kind of false positives, since they may or may not be necessary. Our implementation introduces a time-out mechanism to mitigate this problem. If another user's process wants to create/delete the file object and encounters the last access time by the invariant holder to be older than the time-out period, the new process is allowed to preempt the invariant and the file object is marked as tainted. If the original invariant holder attempts to use the file object again, then we have found a real conflict. The current implementation aborts the original invariant holder, although other design choices are possible.

The determination of a suitable time-out period, called MAX_AGE in Table 12, is probably dependent on each specific workload and a research question. If it is too short, an attacker may use it to abort a long running legitimate process by attempting to create/delete a shared file. If it is too long, another legitimate process may be delayed for a long time. We have experimentally chosen a MAX_AGE of 60 seconds.

2.5.3.3. Overhead Measurements

To evaluate the overhead introduced by our EDGI defense mechanism, we run the same variant of the Andrew benchmark as used in Section 2.3.3.3. The experiments were run on a Pentium III 800MHz laptop with 640MB memory, running Red Hat Linux in single user mode. We report the average and standard deviation of 20 runs for each experiment in Table 14, which compares the measurements on the original Linux kernel and on the EDGI-augmented Linux kernel. The same data is shown as bar chart in Figure 24.

Table 14: Andrew Benchmark Results (in milliseconds)

Functions	Original Linux	Modified Linux	Overhead
mkdir	6.35 ±0.21	6.43 ±0.19	1.3%
copy	217.0 ±1.5	218.6 ±1.4	0.7%
Stat	132.0 ±1.9	193.6 ±0.8	47%
grep	777.0 ±4.3	870.1 ±5.3	12%
compile	53,971 ±434	55,615 ±367	3.0%

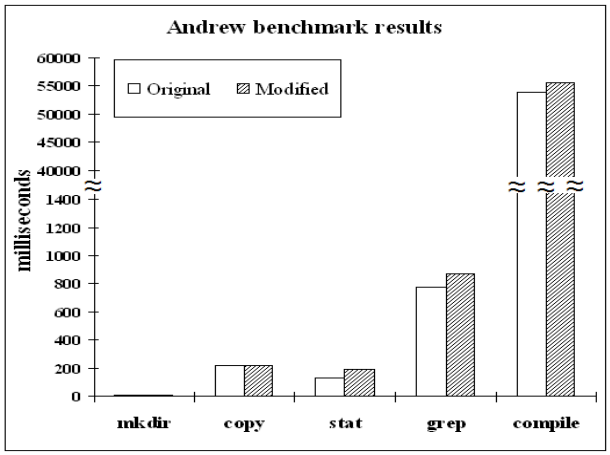

Figure 24: Andrew Benchmark Results

The Andrew benchmark results show that EDGI generally has a moderate overhead. The only exception is **stat**, which has 47% overhead. The explanation is that **stat** takes less time than other calls (such as **mkdir**), but the extra processing due to

invariant holder tracking (now part of pathname resolution) has a constant factor across different calls. This constant overhead weighs more in short system calls such as **stat**. Fortunately, **stat** is used relatively rarely, thus the overall impact remains small.

We also evaluate the overhead of EDGI using the PostMark benchmark mentioned in Section 2.3.3.3. On the original Linux kernel the running time of this benchmark is 40.0 seconds. On EDGI-augmented kernel, with all the same parameter settings, the running time is 40.1 seconds (Again these results are averaged over 20 rounds). So the overhead is 0.25%. This result corroborates the moderate overhead of EDGI.

2.6. Related Work

Bishop and Dilger [6, 7] were the first to explore the TOCTTOU problem and developed a prototype analysis tool that used pattern matching to look for TOCTTOU pairs in the application source code. They suggested several solutions to the TOCTTOU problem, including modifications of file system interfaces. Several research projects have tried to prevent subsets of TOCTTOU vulnerabilities. RaceGuard [18] prevents the temporary file creation race condition in UNIX systems, specifically the <**stat, open**> TOCTTOU pair. Dean and Hu [19] proposed a probabilistic approach to another specific TOCTTOU pair: <**access, open**>. Interestingly however, Borisov [8] described an effective attack that can defeat Dean and Hu's approach, which demonstrates the challenging nature of the TOCTTOU problem.

Tsyrklevich et al. proposed a more generic defense mechanism called pseudo-transactions [57], which can be used to prevent some classes of TOCTTOU vulnerabilities from being exploited. Pseudo-transactions work by wrapping *known* susceptible TOCTTOU pairs inside pseudo-transactions. Their implementation of pseudo-transactions supports a flexible specification of allowed and denied file system call pairs. However, they were only able to generate a set of specifications from

empirical refinement through practical use. The main difference between the CUU model and pseudo-transactions is the complete enumeration of exploitable TOCTTOU pairs by the CUU model. To the best of our knowledge, this complete enumeration has not been achieved before.

Static analysis of source code has recently shown some success in finding bugs in systems software. For example, Meta-compilation [20] and RacerX [21] use compiler-extensions to find software bugs, and MOPS [13, 51] uses model checking to verify that a program preserves certain security properties. These static analysis tools could be used to detect TOCTTOU pairs in programs. However, they are limited in the detection of real TOCTTOU problems because of dynamic states (e.g., file names, ownership, and access rights).

In contrast to static analysis, dynamic detection monitors application execution to find software bugs without access to source code. These tools can be further classified into dynamic online analysis tools such as [34, 50] and post mortem analysis tools such as the one proposed by Ko et al. [33]. However, [33] can only detect the result of exploiting a TOCTTOU vulnerability, but not locate the error.

The difficulty of detection contrasts with the simplicity of some of the technical suggestions in advisories and reports on TOCTTOU exploits from US-CERT [58] and BUGTRAQ [11], including setting proper file/directory permissions and checking the return code of function calls. However, some other suggested programming fixes are varied and non-trivial: using random numbers to obfuscate file names, replacing mktemp() with mkstemp(), and using a strict umask to protect temporary directories. More significantly, none of these fixes can be considered a comprehensive solution for TOCTTOU vulnerabilities.

2.7. Discussion

Our solution to the TOCTTOU problem illustrates one example of para-transactional invariants (PTIs) that the mapping from the file pathname to the disk block number must remain invariant between the check call and the use call. The EDGI defense against TOCTTOU attacks preserves this invariant by wrapping the check and use operations into an atomic execution unit similar to a database transaction, which guarantees that the invariant is preserved despite any concurrent processes (including the attack process).

CHAPTER 3

K-QUEUE DRIVEN TRANSIENT KERNEL CONTROL FLOW ATTACKS

The second contribution of this dissertation research is a solution to K-Queue-driven transient kernel control flow attacks. We identify such attacks as a new hiding technique that can be used by an attacker to *maintain* stealthy control of the kernel (Section 3.1). Having addressed a representative subclass of such attacks (Section 3.3), we solve the complete class of K-Queue-driven attacks as a final step of this thesis (Section 3.4).

3.1. Overview

Internet-scale attacks, such as botnets, often utilize malicious software (malware) to hide their presence and extract information from their host systems. Rootkits, for example, are a common type of kernel-level malware that intercept and modify system events with the goal of hiding illicit activity [10, 29]. Other kernel-level malware can collect sensitive data, cause a denial of service, or open backdoors into the system. In this chapter we present an attack technique that allows an attacker to execute kernel-level malware while evading detection from existing defensive tools. We then focus on techniques for detecting and mitigating the attack.

We divide attacks designed to *maintain* stealthy control of the victim kernel into three broad and sometimes overlapping categories: (1) detour attacks, (2) persistent kernel control flow attacks, and (3) transient kernel control flow attacks. The first category consists of malware (malicious software) that changes code on a disk or in memory. These changes can be detected by trusted security tools that compare the current state of the system code against a known good state (e.g., a "gold" distribution version). The second category consists of attacks that are capable of invoking malicious

functions during execution by changing data (e.g., function pointers in the interrupt handler table). The attacks in this category do not make any changes to the kernel code, but they can be detected by control flow integrity (CFI) [1] and state-based control flow integrity (SBCFI) [42]. However, the attacks in the third category are capable of evading current defensive techniques.

This category, *transient* kernel control flow attacks, can achieve continual malicious function execution without persistently changing either kernel code or data (from the "gold" distribution). One class of transient kernel control flow attacks is *K-Queue-driven attacks* that use existing kernel interfaces (called K-Queues) to dynamically *schedule* executions of malicious functionality in the kernel space. K-Queues are dynamic schedulable queues in the kernel that can be used to inject transient control flows. All instances of K-Queues share some common properties. For example, they all provide APIs for submitting an execution request for some callback function, and they all have a dispatch engine that takes the request from the queue and invokes the callback function. A kernel level malware can abuse these APIs to request that its malicious code be invoked as a callback function. We have confirmed that this is feasible for the soft-timer queue, one type of K-Queue (Section 3.2.3). K-Queue-driven attacks are difficult to detect because malicious requests of the malware are hidden among the many other requests from legitimate kernel components, which prevents CFI and SBCFI from detecting them in this scenario.

To defend against K-Queue-driven transient control flow attacks, we verify and preserve a class of para-transactional invariants (PTIs) at runtime: A legitimate K-Queue callback function and its callees (functions it calls) should always target trusted code of the kernel during the execution of the callback function. In other words, the control flow resulting from a legitimate K-Queue request should never include the malware code. If we can preserve these invariants, we can guarantee that control will never go to the malware code as a result of invoking a K-Queue callback function, which suggests that

K-Queue-driven transient control flow attacks can be defeated by preserving the related invariants.

Our defense encodes the PTIs associated with K-Queues into a whitelist of K-Queue *summary signatures*. Each K-Queue summary signature is a two-element tuple: <*function*, *assertion*>. *function* represents a legitimate K-Queue callback function, and *assertion* represents properties of the legitimate *data* passed to the legitimate callback function as input. We add a reference monitor to the system that verifies each pending K-Queue request (represented by a *function* attribute and a *data* attribute) before invoking the callback function. Specifically, the *function* attribute is used to look up a summary signature database. If a matching signature is found and the *data* attribute satisfies the matching *assertion*, the verification is successful, and the callback function is invoked. Otherwise, the callback function is not invoked.

Although our basic idea is straight-forward, completely implementing it is challenging. The first challenge is the building of the summary signature database. In order to find out all legitimate uses of a particular K-Queue, the entire code base of the kernel, including device drivers, needs to be studied. However, a modern kernel is very complex, which means that hundreds of places may submit requests to a particular K-Queue. Obviously, it is impractical to find all such K-Queue uses manually. Fortunately, significant amount of information is already embedded in the kernel source code concerning the uses of K-Queues. For example, there are certain "contract" (e.g., calling conventions, APIs, or helper functions) between the K-Queues and their requesters. So we can infer K-Queue usage by searching for such "contract" patterns in the kernel source code. By applying static code analysis, we perform this kind of inference in an automated way.

Another challenge is the development of a checker program based on the result of the static analysis. Again it is impractical to write the code manually because the verification can be very complex. For example, a top-level K-Queue callback function

may need to be compared against hundreds of candidate functions, and depending on the candidate function the *data* attribute may need to be tested in many different ways. Obviously, manually writing and maintaining such code is tedious and error prone. Fortunately, this tediousness can be alleviated by applying automated code generation. The observation is that most of the checker code can be generated as a by-product of the static analysis process.

Our approach significantly reduces the amount of human labor in the summary signature generation and coding of the K-Queue checker. For example, out of 46 legitimate soft timer callback functions in a particular kernel of 482,369 lines of code, only one callback function is missed by the static analyzer. If the static analyzer is not used, all 46 callback functions would have to be manually recognized and their checker code manually written.

3.2. K-Queue Driven Transient Control Flow Attacks

3.2.1. Overview of Kernel Control Flows

We use Linux as a concrete and representative multi-threaded kernel. The Linux kernel can have a number of control flows (listed in Figure 25): exception handlers, interrupt service routines, Softirqs, and kernel threads such as work queues [9].

Of the various kinds of kernel control flows, exception and interrupt handlers execute at the highest priority, usually with interrupts disabled. Exceptions such as system calls are a result of process invocation. Interrupts are used by hardware (e.g., I/O devices) to notify the kernel of urgent events (for example, arrival of a packet in the network interface card that needs to be copied to kernel/user buffers).

Some exception and interrupt handler operations are interruptible and executed in Softirqs, for example, sending the keyboard line buffer to the terminal handler process. Softirqs are invoked in interrupt context (e.g. when the service routine for an I/O interrupt

is finished), but with interrupt enabled. Softirqs reduce the kernel response time to exceptions and interrupts.

Furthest from hardware, kernel threads execute in process context and are therefore fully interruptible. They are interleaved with user processes, with the main difference being that kernel threads execute in kernel context while user processes execute in user process context.

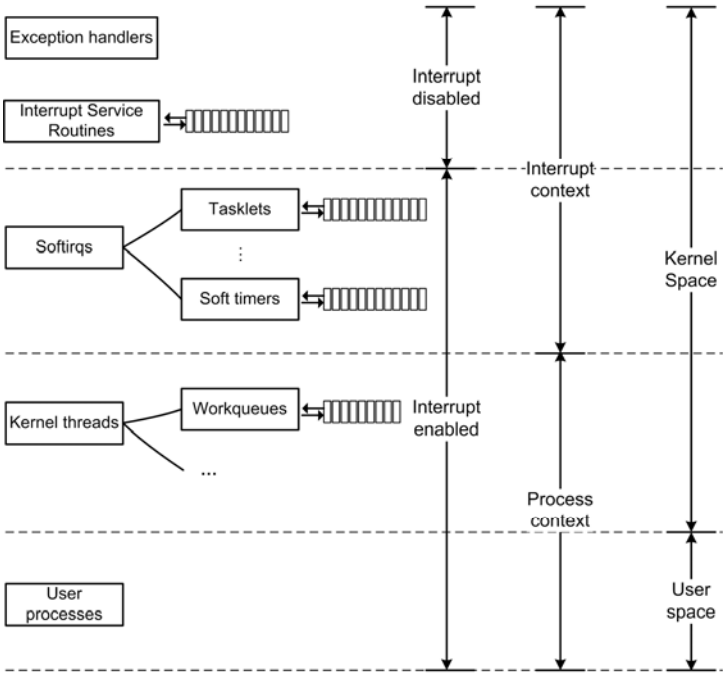

Figure 25: Kernel Control Flows with Schedulable Queues (Linux Kernel 2.6)

3.2.2. K-Queues in the Linux Kernel

The kernel control flows outlined in Figure 25 are executed by the kernel through *kernel schedulable queues* or *K-Queues* for short. These K-Queues are implemented as linked lists. Representative K-Queues (with descending execution priorities) include IRQ action queues, tasklet queues, soft timer queues, and work queues.

```
struct irqaction {
        irqreturn_t (*handler)(int,
void *, struct pt_regs *);
        unsigned long flags;
        cpumask_t mask;
        const char *name;
        void *dev_id;
        struct irqaction *next;
        int irq;
        struct proc_dir_entry *dir;
};
```

Figure 26: The Definition of irqaction in Linux Kernel 2.6

```
struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        void (*func)(unsigned long);
        unsigned long data;
};
```

Figure 27: The Definition of tasklet_struct in Linux Kernel 2.6

```
struct work_struct {
        unsigned long pending;
        struct list_head entry;
        void (*func)(void *);
        void *data;
        void *wq_data;
        struct timer_list timer;
};
```

Figure 28: The Definition of work_struct in Linux Kernel 2.6

3.2.2.1. IRQ Action Queues

When an interrupt happens, the Interrupt Descriptor Table (IDT) is used to find the corresponding Interrupt Service Routine (ISR), which may in turn delegate the interrupt handling to several IRQ actions. This is because multiple I/O devices can share an interrupt pin; therefore each of them may have its own way of handling the shared interrupt. The Linux kernel uses IRQ action queues to support such interrupt sharing. Each element of an IRQ action queue is a structure **irqaction** (Figure 26), which contains a *handler* field, a *dev_id* field, a pointer to the next element in the queue (the *next* field), and other information. The *handler* field is a function pointer to the handler routine, and the *dev_id* field is used to uniquely identify the device that provides the handler routine. When an interrupt happens, the ISR invokes all handler routine in the corresponding IRQ action queue.

3.2.2.2. Tasklet Queues

Compared to Interrupt Service Routines, tasklets are the preferred way to implement deferrable functions in I/O device drivers. For example, a gigabit network interface card driver may dynamically adjust the size of receive buffers according to their fill level (e.g., allocate more buffer space when the fill level exceeds certain threshold). The expansion of receive buffers can be time consuming due to allocation of more kernel

memory, and it should be interruptible, being an optimization that does not affect the correct reception of packets. Consequently, the device driver can request a tasklet to expand receive buffers, instead of doing it in the receive interrupt handler.

As Figure 27 shows, a tasklet request contains a callback function pointer (in the *func* field) and a *data* pointer. In Linux, the tasklet request is inserted into one of two tasklet queues, implemented by two Softirqs (numbers 0 and 5). When the **do_softirq** function comes across a tasklet structure (Figure 27) during the traversal of the two queues, it invokes the callback function and passes on the *data* field as the input parameter.

3.2.2.3. Work Queues

Work queues are used to schedule kernel threads that interleave with user processes. Compared to tasklets, which execute in interrupt context, work queues execute kernel threads in kernel context. Consequently, work queues run at lower priority than tasklets.

A work queue is a linked list of work requests (Figure 28), dynamically inserted through functions such as **queue_work**. Similar to a tasklet, each work request has a callback function (the *func* field) and a *data* field. The server for a work queue is a kernel thread such as *events/0,* which executes each element in the list by invoking its callback function with the *data* field passed on as the input parameter.

Linux kernel may have multiple work queues. Two predefined work queues are the *events* work queue that can be used by all device drivers and the *kblockd* work queue used by the block device layer. Additional work queues can be created at runtime.

3.2.2.4. Soft Timer Queues

Since the attack scenarios described in Section 3.2.3 use soft timer queues, we provide more background information here. Dynamic soft timer is a well-established

mechanism used by many kernel components to schedule the execution of timed-event handling functions. Common uses of soft timers include retries when polling a physical device, retransmission of data, and handling of network protocol timeouts. Figure 29 shows one concrete example in Linux kernel 2.6.16, where a soft timer interrupt is used to implement the retransmission of data when the device is temporarily not ready.

```
static struct timer_list tx;
static void isicom_tx(unsigned long _data)
{     …..
      init_timer(&tx);
      tx.expires = jiffies + HZ/100;
      tx.data = 0;
      tx.function = isicom_tx;
      add_timer(&tx);
      return;
}
static int __devinit isicom_setup(void)
{
      ……
      init_timer(&tx);
      tx.expires = jiffies + 1;  tx.data = 0;
      tx.function = isicom_tx;
      add_timer(&tx);
      ……
}
```

Figure 29: Use of soft timer in Linux-2.6.16/drivers/char/isicom.c; the function isicom_tx may be periodically invoked as a result.

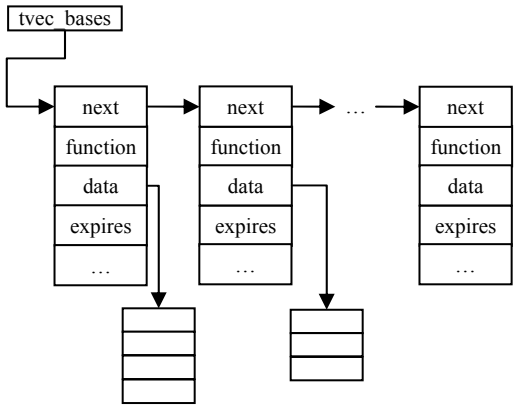

Figure 30: A simplified view of the data structures related to soft timers

In the Linux kernel, the requester of a soft timer first prepares an instance of soft timer interrupt request (STIR) of type **struct timer_list** (such as **tx** in Figure 29), which contains information about the callback function (the *function* field), a data pointer (the *data* field), and the expiration time, among others. The **add_timer** function is invoked to add this instance of STIR into a linked list of pending timers: **tvec_bases** (Figure 30).

The soft timer queue is implemented by a Softirq (number 1) and STIRs executed in interrupt context (Figure 25). When a STIR in the linked list expires, it is removed from the list, its callback function is invoked, and the *data* pointer is passed along to the callback function as the input parameter. Typical callback functions also create the next STIR at the end of request processing (e.g., **isicom_tx** in Figure 29).

3.2.3. Example Attacks Driven by K-Queues

A common feature among the K-Queues described in Section 3.2.2 is that they all contain some callback functions, and upon invocation such functions inject control flows into the main kernel control loop. Under the assumption that everything in the kernel space is equally trusted, such transfers of control are acceptable. However, if one of the requesters is malicious, the K-Queue mechanism can be turned into a reliable way of maintaining stealthy control: an attack can be divided into a sequence of K-Queue requests and executed using successive callback functions. In this section, we demonstrate that such an attack is possible by leveraging the soft timer queue (Section 3.2.2.4).

For ease of presentation, we adopt a simple and informal model of kernel-level malware that executes useful work for a botnet owner or renter. Under this model, the malware's lifecycle can be divided into three steps: (1) system penetration, (2) interpose on the kernel control flow, and (3) continually execute malicious functionality. Penetration methods (step 1) such as buffer overflows [17] are well known and omitted from this discussion. Previous persistent kernel control flow attacks (e.g., the rootkits listed in [42]) change kernel data structures (step 2) to force the kernel to branch/jump to malicious functionality (step 3). Like persistent attacks, our new transient attacks interpose on the kernel's control flow (step 2) at the time of the attack. However, unlike persistent kernel control flow attacks, which typically replace a permanent function pointer in the kernel, a transient kernel control flow attack simply installs a malicious STIR (Section 3.2.2.4). In our demonstration, malicious functionality is implemented using a Linux loadable kernel module (LKM) initialization function that requests the first STIR. When the malicious LKM is loaded, the kernel invokes its initialization function, and step 2 is completed. The malware's persistent execution (step 3) is possible because each STIR can request the next STIR that references the callback function. For added stealth, the location of this callback function can change with each STIR execution.

To understand the effectiveness of transient kernel control flow attacks, this section outlines the design of three soft timer driven attacks to show that they can perform a wide variety of malicious objectives. These attacks are implemented as LKMs and run through the soft timer facility. More specifically, they invoke the kernel API **add_timer** to request a STIR in their initialization function. **add_timer** takes as input a parameter that points to a data structure of type **struct timer_list**, and the *function* field of this structure is set to a callback function. A callback function is specific to the corresponding malware, but all such functions request the next STIR before they return, e.g., by calling **add_timer**.

The three soft-timer based malware examples below demonstrate violation of the three basic security properties: the stealthy key logger violates confidentiality, and the cycle stealer and the alter-scheduler violate both availability and integrity.

3.2.3.1. Stealthy Key Logger

A typical class of malware steals sensitive information from the host node. A straightforward but easily detected malware implementation intercepts the kernel functions that process such sensitive information. For example, a key logger [45] can replace the keyboard interrupt handler (e.g., IRQ 1) with a malicious handler that records the keyboard input. The following implemented example shows that persistent kernel modifications are not needed for this type of malicious functionality.

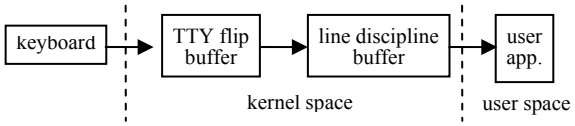

Figure 31: Flow of keyboard input information in Linux

A timer-driven key logger keeps kernel code and interrupt-related data structures intact. It periodically looks at various buffers in the kernel, where the keyboard input information is stored. As Figure 31 shows, when a key is pressed, the keyboard hardware

generates an interrupt. The keyboard interrupt handler fetches the key stroke information and temporarily stores it in the TTY flip buffer before transferring it into the TTY line discipline buffer. Finally, when a user-level application reads from the standard input device, the keystroke information is copied into the user's buffer.

The sampling rate determines whether or not a timer-driven key logger can capture every keystroke. The key logger can obtain keystroke information from the TTY flip buffer, the TTY line discipline buffer, or the user's buffer. The TTY flip buffer has a very short retention time relative to the TTY line discipline buffer, which is a large circular buffer (normally 4096 bytes). Since each keystroke generates 2 bytes of information, the TTY line discipline buffer can keep information on up to 2048 keystrokes. Since it can take several minutes for the average user to fill up the line discipline buffer, the key logger malware only needs to inspect the buffer periodically (e.g., once per minute should be good enough) to collect all of the user's keystrokes. In the event that more frequent sampling is required, the key logger can request faster soft timer interrupts. In this case, techniques for hiding the higher resource consumption should be employed (see Section 3.2.3.2) to keep the key logger stealthy.

We have implemented the sampling key logger on Linux to collect key strokes from an X Window desktop. It captures keystrokes entered into X Window applications, including the gedit editor, the Firefox web browser, and terminal window emulators. These applications handle many security-critical keystrokes including usernames, passwords, and credit card numbers.

3.2.3.2. Stealthy Denial of Service Attack (CPU Cycle Stealer)

A second common type of attack causes a denial of service (DoS) or lowered quality of service. In a soft timer-driven attack, the call back function can perform computationally intensive work to steal system resources thereby slowing down or halting any legitimate application. One simple CPU cycle stealer has been implemented

by inserting a program to compute the factorial of a given number in the call back function. By adjusting the value of the number and the timer's period, different slowdown factor can be obtained. We measure the CPU usage during such an attack where the timer's period is fixed at one second, as shown in Figure 32. When the value of the number is below 25, the CPU consumption by the malware is negligible. As the value becomes larger, there is an exponential increase in the CPU consumption by the malware. For example, when the value is 36, the CPU consumption is about 54%, and when the value grows to 42, the CPU consumption is close to 100%. Note that the actual algorithm used to steal CPU cycles is irrelevant to the attack. Instead, this attack shows that a resource-exhaustion attack can be stealthily deployed, preventing the system from performing its intended tasks.

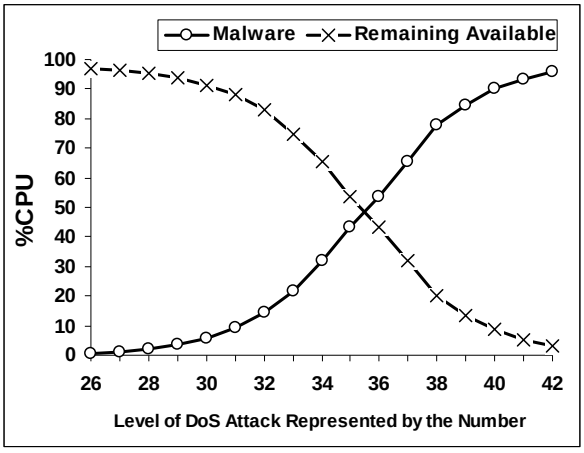

Figure 32: CPU Consumption by Computing Factorials of Different Numbers

The attack becomes effective when the malware is able to hide itself and its effects from detection for a significant amount of time. One problem with typical DoS attacks is that the wasted CPU cycles are detectable by system tools such as *top*. This is because the kernel maintains performance accounting information for different sources of computation. For example, the CPU time consumed by the above malicious call back function is attributed to "software interrupt". To hide this attack, the malicious call back function further manipulates the kernel accounting data (e.g., kstat_cpu(0).cpustat) such

that the CPU time used by the malicious STIR is attributed towards the idle CPU time. Therefore, it is not immediately obvious why the system performance is degrading.

Our CPU cycle stealer violates the availability of CPU resources and the integrity of the performance accounting information. However, since the performance accounting information is dynamic, there is no easy notion of what is normal. Under such attacks, a system may report slowdown of a service, but there can be many other reasons for poor performance (network congestion, server overload, retries due to device error, etc). Therefore, this type of attack is not easily discovered.

3.2.3.3. Running a Hidden Process: the Alter-Scheduler

A third kind of malware, called alter-scheduler, is capable of running a malicious process without relying on the legitimate kernel scheduler. Some existing malware can hide a malicious process by removing its entry from the all-task linked list of the kernel. However, this malware must leave the malicious task structure in the run queue in order for it to be scheduled. Therefore, a detection tool such as [41] that cross-checks the all-task linked list and the run queue is able to detect the malicious process.

The alter-scheduler malware implements a special scheduler exclusively for the malicious process. It keeps a record of the malicious process structure and detaches it from both the all-task list and the standard run queue. Within the STIR call back function, the alter-scheduler preempts the currently running task, as if a higher-priority process has become runnable. Then it forces a context switch to the malicious process, as if the malicious process has been chosen as the new task to run. The standard scheduler is resumed when the malicious process surrenders the CPU.

This style of attack is very powerful because the malicious process is made independent of (and thus invisible to) the legitimate kernel scheduler and other relevant routines, and the malicious alter-scheduler instead supplies the missing functionality

(e.g., giving the malicious process opportunities to run). Therefore, malware based on the alter-scheduler can remain stealthy against state-of-the-art detectors such as [41].

3.3. A Specialized Defense against Soft-Timer-Driven Transient Kernel Control Flow Attacks

3.3.1. Introduction

In this section, we discuss the design, implementation, and evaluation of a static analysis based tool that detects soft-timer driven attacks. Under our security assumptions, this tool detects all soft-timer attacks with less than 7% performance overhead.

The static analysis tool uses *summary signatures* to differentiate STIRs from legitimate and malicious software. Summary signatures characterize legitimate STIRs using callback functions and other constraints, and are derived through automated static analysis of the kernel source code. At runtime, a reference monitor mediates STIR execution based on the summary signatures. We take several measures to protect the reference monitor, including executing it in a different virtual machine and using memory protections to prevent an attacker from bypassing the mediation step. Section 3.3.2 provides a complete discussion of our architecture and its security properties.

In the rest of this section, we present our defense mechanism against such attacks. We describe the Xen-based prototype implementation of the defense and its evaluation in terms of effectiveness and performance overhead.

3.3.2. Soft Timer Attack Detection and Defense

As described in Section 3.2.3, a soft timer based attack must usurp kernel control flow in order to execute malicious code. Soft timers can be leveraged to do this in one of two ways: (Type 1) supply a malicious timer callback function, or (Type 2) supply a legitimate timer callback function but a *malicious data pointer* such that the control flow

of the legitimate callback function is modified to invoke malicious functionality as a subroutine (similar to the "jump-to-libc" style attacks [55]). The latter option is possible because when a STIR callback function is invoked, a data pointer embedded in the STIR is passed as the input parameter. In some cases, the STIR callback function may derive a function pointer from this input, thereby allowing the data to alter the control flow.

3.3.2.1. Security Assumptions and Threat Model

Our defensive techniques against soft timer attacks are based on four standard security assumptions. First, since we use a virtualization-based architecture, we assume that the virtual machine manager (VMM) and the security virtual machine (VM) are part of the trusted computing base. This assumption is based on the idea that the VMM code base can be small, and therefore auditable, and the interface between the guest VM and the VMM can be narrow and protected. Our second assumption is that the legitimate kernel code in the guest VM's memory can not be tampered with by malicious code. In a production setting, this must be enforced by existing security tools such as Copilot [40] or SecVisor [53]. Third, we assume that the source code of the kernel and all kernel extensions are available for the static analysis portion of our tool. Note that closed source operating system vendors could perform the static analysis and make the results available to the end-users. For open source operating systems, the entire procedure can be performed by end-users. Lastly, in order to provide protections for this system, we require that the system can be booted into a known good state (i.e., secure boot [3]). We then perform a brief initialization phase to setup our defensive system and then the guest VM is open to outside events and may be placed under attack at any time.

Our threat model allows an attacker to install malicious code on this system running at the highest privilege level. The attacker is able to perform kernel-level attacks, but we assume that protections are in place to prevent tampering with kernel code as described above. Under this model, the attacker is powerful and able to run soft timer

attacks unless our defensive system prevents them. This is a realistic threat model and no more constraining to an attacker than previous work in this space [42].

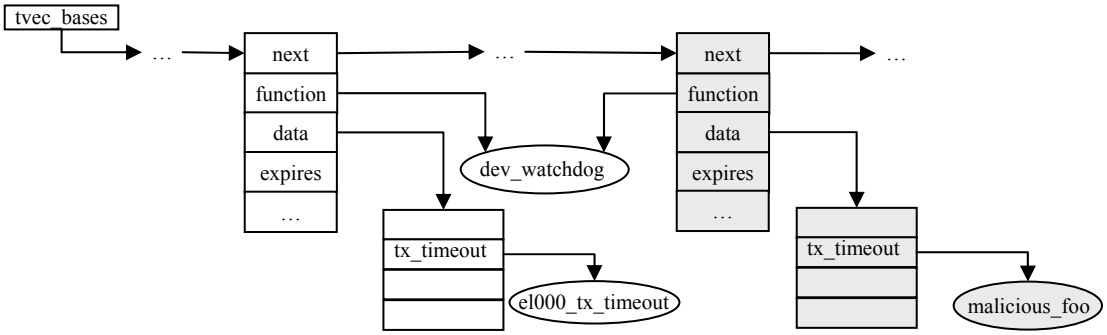

Figure 33: Illustration of a malicious STIR with a legitimate callback function (dev_watchdog in Linux kernel 2.6.16) and a malicious data pointer (Shaded area means malicious). Here dev_watchdog may invoke a function pointer derived from the *data* field of the STIR.

3.3.2.2. Legitimate STIR Identification

The basic idea of our proposed defense is to validate each STIR before its execution, thereby preventing the execution of malicious STIRs. Based on the "fail-safe defaults" principle [49], we use a white list of STIR *summary signatures* to distinguish legitimate STIRs from malicious ones. An unknown STIR that does not have a matching STIR summary signature is considered suspect and denied execution.

3.3.2.2.1 STIR Summary Signatures

Recall that a malicious STIR can induce kernel control flow in two ways: (1) supply a malicious timer callback function, or (2) supply a legitimate timer callback function but a malicious data pointer. In order to detect type 1 malicious STIRs, we only need to check their callback functions against a white list of legitimate timer callback functions. However, in order to detect type 2 malicious STIRs, we must check the data pointer in addition to checking the callback function. Figure 33 illustrates a type 2 malicious STIR (in shaded color). This figure shows that the `tx_timeout` field of the data structure (in shaded color) referenced by the data pointer of the malicious STIR is set to a malicious function (e.g., `malicious_foo`). Therefore, we can detect this

malicious STIR by comparing the `tx_timeout` field against a white list of legitimate functions (for example `e1000_tx_timeout`) that can be assigned to this field for the legitimate STIRs.

Consequently, we choose the STIR summary signature as a two-element tuple <*function*, *assertion*>, where *function* represents a legitimate timer callback function (e.g., `dev_watchdog`), and *assertion* represents properties of legitimate data passed to the legitimate callback function as input. Specifically, an assertion is the logical AND of 0 or more parameterized predicates. Each predicate has the form "*deref* equals *functionlist*", where *deref* specifies a way to dereference a function pointer (e.g., `data->tx_timeout`), and *functionlist* is the logical OR of one or more legitimate functions that can be assigned to the dereferenced function pointer. An example assertion associated with `dev_watchdog` is:

(`data->tx_timeout` equals (`e1000_tx_timeout OR xircom_tx_timeout`))

Figure 34 shows the overall processing of the STIR summary signatures, divided into three phases corresponding to compile time, initialization time and evaluation time, respectively. In the first phase, Linux kernel source code is statically analyzed by the STIR Analyzer to generate the symbolic STIR summary signatures. These signatures are symbolic because the addresses of the functions in them may be unknown at compile time (e.g., due to dynamic kernel module loading). The actual mappings of these functions to their runtime addresses happen in the second phase, when the symbolic summary signatures become *resolved summary signatures*. This process is in some way similar to partial evaluation [16]. Finally, during the normal operation of the guest VM (e.g., the evaluation time), the STIR Checker (Section 3.3.2.3) uses the resolved summary signatures to prevent control transfers due to malicious STIRs.

In the first phase, the STIR Analyzer performs a top-level analysis to derive the *function* part of the STIR summary signatures and a transitive closure analysis to generate the *assertion* part of the STIR summary signatures. The latter analysis identifies all

function pointer dereferences of the input parameter in the legitimate STIR callback functions, as well as all legitimate functions that they target.

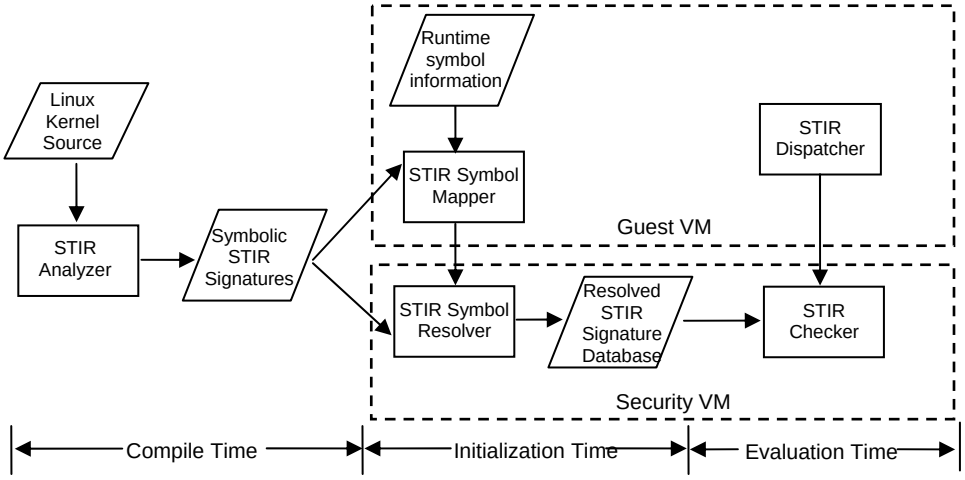

Figure 34: Overall processing of the STIR summary signatures

3.3.2.2.2 Top-Level Analysis

We first consider the collection of legitimate STIR callback functions, which we call `LegitTimerfuncs`. These are the top-level functions that require validation before execution. Each function in `LegitTimerfuncs` will become the *function* part of a STIR summary signature after the transitive closure analysis.

Table 15: Different ways of assigning timer callback functions in the Linux kernel

t.function = fn;
t = TIMER_INITIALIZER (fn, expires, data);
DEFINE_TIMER(t, fn, expires, data);
setup_timer(&t, fn, data);

`LegitTimerfuncs` is constructed by scanning the kernel source code to identify all legitimate instances of soft timer callback functions. Table 15 shows the four techniques to link soft timer callback functions, denoted `fn`, to the `timer_list` structure, denoted `t`. The first is by assignment. The second and third techniques are macros that actually expand to assignment. Therefore the first three cases are analyzed in

the same way: the STIR Analyzer traverses each assignment statement (`lval = rval`) of each function in the Linux kernel, and if `lval` ends with a field named *function* within a structure of type `timer_list`, then `rval` is recognized as a soft timer callback function. The last technique to link a soft timer callback function is to use the `setup_timer` procedure. This technique is handled by traversing each function call to `setup_timer` and collecting the second parameter in the function call.

We assume that benign programmers follow the standard APIs in Table 15 to request STIRs. Since the top-level analysis considers all 4 ways in Table 15, it can capture all legitimate STIR callback functions.

3.3.2.2.3 STIR Callback Transitive Closure Analysis

Verification of the top-level `LegitTimerfuncs` is insufficient to guarantee defense because it only addresses type 1 malicious STIRs and not type 2. To detect potential attacks in lower level subroutines, the second part of the STIR Analyzer checks the function calls within each callback function in `LegitTimerfuncs` to see if any of them allows indirect control transfers. Concretely, if function pointers are derived from the input parameter of a callback function and the callback function further branches to one of those pointers, then the analyzer raises a flag to indicate that the callback function needs a transitive closure analysis of all such pointers.

Figure 35 shows the high-level algorithm for the transitive closure analysis. Given a callback function `fn` with parameter `arg`, the STIR Analyzer first traverses each assignment statement of `fn` to compute the set of variables (`tainted_vars`) whose value can be influenced by `arg`, directly or indirectly. Next, the STIR Analyzer searches every function call statement of `fn` to see if the target function or its parameter is influenced by any variable in `tainted_vars`. Existence of such a function call means that control can go to places decided by `arg`, which could be exploited by malware.

If the STIR Analyzer does not raise a flag for a callback function in the transitive closure analysis, a signature <*function*, *assertion*> is completed where *function* is the name of the callback function, and *assertion* is simply the boolean value *true* (which means that no further check is needed on the data parameter `arg` of the callback function).

```
Transitive closure analysis of fn(arg):
  - Initially arg is added to tainted_vars;
  - For each assignment statement lval = rval or lval = f'(rval) in fn:
      If any part of rval is in tainted_vars, then lval is added to tainted_vars.
  - For each function call statements f(params) in fn:
      If any part of f(params) is in tainted  vars, then raise a flag for fn.
```

Figure 35: Analysis of each STIR callback function

If the STIR Analyzer raises a flag, a further step is performed to compute the *assertion*. This step can be further subdivided into three cases.

Case 1: Only the function name part of a function call statement (e.g. `f` in `f(params)` of Figure 35) is influenced by the input parameter (`arg`), which means that `arg` is used to derive a function pointer. In this case, the third step decides the legitimate functions that can be assigned to the function pointer derived from `arg`. For each distinct way of dereferencing `arg`, a predicate "*deref* equals *functionlist*" is generated, where *deref* specifies the way to dereference `arg`, and *functionlist* is the logical OR of legitimate functions that can be assigned to the dereferenced function pointer. The *assertion* then is the logical AND of all such predicates. The process of deriving legitimate functions in a predicate is similar to the top-level analysis (section 3.3.2.2.2) which identifies the timer callback functions.

Case 2: Only the parameter part of a function call statement (e.g. `params` in `f(params)` of Figure 35) is influenced by the input parameter `arg`. In this case, the same analysis in Figure 35 is applied to `f`, and all resultant predicates are appended (ANDed) to the *assertion*.

Case 3: Both the function name and the parameter of a function call statement are influenced by `arg`. The third step treats this case as a composition of case 1 and case 2. E.g., it first processes the function name part to derive the legitimate functions and then processes the parameter part on each of the identified legitimate functions.

The STIR Analyzer relies on accurate type information to recognize function pointer dereferences. In the Linux kernel (written in C), addresses could be calculated by pointer arithmetic operations. In practice, we have found no such unsafe pointer arithmetic operations in all of the STIR related kernel functions we have inspected. Due to the threat represented by kernel control flow attacks (both persistent and transient), we encourage kernel developers to continue avoiding pointer arithmetic operations in legitimate kernel functions. This will help to support comprehensive kernel code analysis that depends on type information.

3.3.2.2.4 Generation of Resolved STIR Summary Signatures

The outcome of the STIR Analyzer is the symbolic STIR summary signatures. These contain symbols (e.g., STIR callback function names) whose runtime *addresses* may not be determined statically. Specifically, Linux supports loadable kernel modules (LKMs) that can be added to the kernel at runtime. If a legitimate LKM uses a soft timer, the address of its callback function cannot be known until after the module is loaded (at runtime). Therefore, we provide a mechanism for registering such symbol-address mappings at runtime.

Because we employ a VMM-based detection architecture (described in Section 3.3.2.3), the registration mechanism is split into two components: a guest VM component (called a STIR Symbol Mapper) and a security VM component (called a STIR Symbol Resolver), as shown in Figure 34. At the guest VM initialization time, the STIR Symbol Mapper first generates mappings from function names in the symbolic STIR summary signatures to virtual addresses in the guest kernel's address space. It then informs the

STIR Symbol Resolver about these mappings through an inter-VM communication. When the STIR Symbol Resolver receives the mapping list, it merges the addresses with the corresponding symbolic STIR summary signatures, which become resolved STIR summary signatures that can be used to check the legitimacy of pending STIRs.

3.3.2.3. The STIR Checker

Because soft timer attacks are at the kernel-level, a defense mechanism inside the same kernel would be vulnerable to tampering by an attacker. Consequently, an effective defense must be isolated from the victim kernel. Virtual machine managers (VMMs) are one environment that provides such isolation, allowing us to run the defensive mechanism in a VM that is isolated from the guest VM. Our implementation uses Xen [4] for these isolation properties.

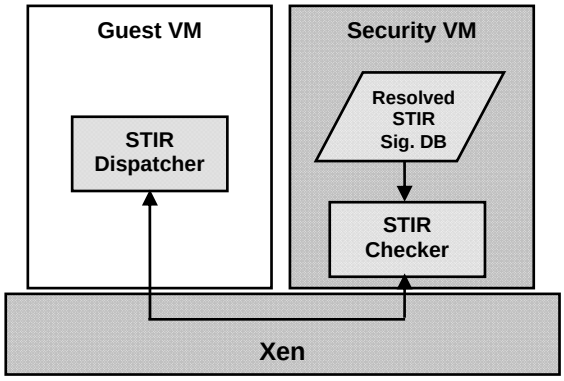

Figure 36: Defense against soft timer attacks

As shown in Figure 36, our architecture places the STIR Checker outside of the victim guest kernel in a separate domain (called the security VM). The purpose of the STIR Checker is to prevent control transfers from the guest kernel to malicious functionality such as those outlined in Section 3.2.3. Specifically, the software timer dispatcher of the guest kernel is modified to inform the STIR Checker about the callback function and related *data* when a pending STIR expires, and invoke the callback function only if the STIR Checker returns *true* (yes). During the time when the STIR Checker is

making a decision, the guest kernel is suspended waiting for the decision. The communication between the STIR Checker in the security VM and the guest VM is facilitated by an inter-VM communication channel. The small modification to the guest kernel is protected from tampering using the memory-protection capabilities from the Lares architecture [39]; therefore the STIR Checking cannot be trivially bypassed.

The STIR Checker module compares the next STIR to be dispatched against a list of resolved STIR summary signatures (Section 3.3.2.2). As Figure 36 shows, all STIR summary signatures are stored in a database, indexed by their *function* element (Section 3.3.2.2). Given a STIR, the STIR Checker first uses its function field as the index to look up the summary signature database. If a signature is found, and the located *assertion* evaluates to *true* on the data field, the STIR is considered legitimate. Otherwise it is considered malicious.

3.3.3. Linux Implementation and Evaluation

3.3.3.1. Implementation and Evaluation of the STIR Analyzer

We use the Common Intermediate Language (CIL) [37] to implement a prototype STIR Analyzer, which consists of several program analysis modules that implement the algorithms in section 3.3.2.2. These modules receive high-level representations of the kernel source files generated by CIL, analyze them, and output the results into a text file.

The STIR Analyzer can analyze the entire Linux kernel 2.6.16 in about one hour on our test system (a 2.4 GHz Intel Core 2 Duo with 2 GB of RAM). The analyzer found a total of 365 legitimate STIR callback functions in the 3688 kernel source files analyzed.

A majority of these STIR callback functions (333 out of 365) do not derive function pointers from the input parameter; therefore they can not be used to construct type 2 malicious STIRs (Section 3.3.2.2).

On the other hand, 32 of the 365 top-level callback functions do derive function pointers from their input parameter. Transitive closure analysis was carried out for these 32 functions to identify the legitimate subroutines to which the derived function pointers can point. We describe them in some detail, since they represent potential vulnerabilities (e.g., type 2 malicious STIRs) that are difficult to defend against.

Table 16: Representative STIR callback functions that need transitive closure analysis (Linux-2.6.16)

Source file	Timer Callback Function	Function Pointers Derived From Input
drivers/input/joystick/db9.c	db9_timer(struct db9 *private)	private->pd->port->ops->read_data, private->pd->port->ops->read_status, private->pd->port->ops->write_control
drivers/input/gameport/gameport.c	gameport_run_poll_handler(struct gameport *d)	d->poll_handler
drivers/isdn/hisax/isdnl3.c	l3ExpireTimer (struct L3Timer *t)	t->pc->st->lli.l4l3
drivers/scsi/scsi_debug.c	timer_intr_handler (unsigned long indx)	queued_arr[indx].done_funct
net/sched/sch_generic.c	dev_watchdog (struct net_device *arg)	arg->tx_timeout

Table 16 lists some of the 32 STIR callback functions that derive function pointers from the input parameter. From these functions, we can make the following observations. First, the dereferences in some functions are complicated. For example, the input parameter `private` in `db9_timer` goes through 4 layers of indirection before reaching a function pointer (`private->pd->port->ops->read_data`). Second, it is normal for a STIR callback function (such as `db9_timer`) to dereference the input parameter in multiple ways. Correspondingly the *assertion* part of the STIR summary signature for such a function will have multiple predicates (Section 3.3.2.2). Finally, most of the callback functions interpret the input parameter as a pointer to a structure. The only exception is `timer_intr_handler` in drivers/scsi/scsi_debug.c, which uses the input parameter as an index into a global array of structures. A function pointer is in turn retrieved from the array element indexed by the input parameter.

When a callback function such as `dev_watchdog` is encountered, the STIR Analyzer goes through a further step of transitive closure analysis. For `dev_watchdog`, the STIR Analyzer reveals 113 functions in the Linux kernel that can

be assigned to `dev–>tx_timeout`. Due to space limitations, only 4 of them are shown in Table 17.

Table 17: A sampling of legitimate functions that can be assigned to dev–> tx_timeout in dev_watchdog

Function	Location
ace_watchdog	drivers/net/acenic.c
ariadne_tx_timeout	drivers/net/ariadne.c
arlan_tx_timeout	drivers/net/wireless/arlan-main.c
e1000_tx_timeout	drivers/net/e1000/e1000_main.c

Uses of the Symbolic STIR Summary Signatures. As shown in Figure 34, the output of the STIR Analyzer is the symbolic STIR summary signatures. We use this information to implement the rest of our defense. The usage falls into two categories: first, the function names in the symbolic summary signatures are retrieved and incorporated into the STIR Symbol Mapper in the guest kernel and the STIR Symbol Resolver (Section 3.3.3.2) in the security VM; second, the function pointer dereference information in the symbolic summary signatures are transformed into offsets within data structures (through an offline type analysis) and then incorporated into the STIR Checker (Section 3.3.3.2).

3.3.3.2. Implementation of the STIR Defense

Our implementation uses the Lares architecture [39] to transfer control to the STIR Checker in the security VM and to ensure that the STIR Dispatcher cannot be circumvented. Lares provides the infrastructure needed to place hooks into the guest kernel, which divert execution into the security VM. Lares also provides protections to ensure that the hooks in the guest VM are not tampered or circumvented.

This functionality is supported, in part, by a new hypercall (`lares_op`) that is effectively a system call from an operating system kernel into the VMM. The security VM first invokes `lares_op` to register a shared memory region for exchanging information between itself and the VMM. When the hook in the guest VM is triggered, a

VMCALL to `lares_op` is made with input parameters that contain the hook's location and function arguments. Upon receiving the VMCALL, `lares_op` pauses the guest VM, copies the parameters from the guest VM to the memory region shared with the security VM, and triggers a virtual IRQ. The security VM handles the virtual IRQ by copying the event context from the guest into its address space. It then performs its monitoring function and places the response in the shared memory block. Next, `lares_op` is invoked again to inform the VMM that the response is ready. Upon receiving this hypercall, the VMM unpauses the guest and enforces the response from the security VM in the guest VM.

For this work, we extended Lares by defining a new parameter structure passed through the VMCALL from the guest kernel to the security VM. Two commands are defined in this structure: REGISTER_STIR_SYMBOLS, and CHECK_STIR. The first command is used by the STIR Symbol Mapper, and the second command is used by the modified soft timer dispatching logic.

3.3.3.2.1 Implementation of the STIR Symbol Mapper

The STIR Symbol Mapper is implemented in the guest VM as a loadable kernel module that notifies the STIR Symbol Resolver about symbol-address mappings through a VMCALL with the command REGISTER_STIR_SYMBOLS, and the address and length of an array of <symbol id, address> tuples. The return value of this VMCALL is a boolean (success or failure).

Our implementation of the Symbol Mapper first performs a filtering of available kernel and module symbols before invoking the VMCALL, such that only STIR-related symbol-address mappings are passed to the Symbol Resolver. In order to perform the filtering, the STIR Symbol Mapper is initialized with a static list of STIR-related symbols, which is derived from the symbolic STIR summary signatures generated by the STIR Analyzer (Section 3.3.3.1).

3.3.3.2.2 Implementation of the STIR Symbol Resolver

The STIR Symbol Resolver is the security VM component to support STIR related symbol registration. The main task of this component is to handle the REGISTER_STIR_SYMBOLS command from the guest VM. It first copies the STIR-related symbol mappings (in a list of <symbol id, address>) from the guest kernel using the XenAccess [38] virtual machine introspection library. Next, it merges the addresses in the mappings to the STIR summary signature database (Figure 34) for that guest, using the symbol id as a search index.

In our implementation, each guest has its own instance of the STIR summary signature database. This database is initialized by a template generated from the STIR Analyzer (Section 3.3.3.1), where the addresses of the function symbols are undefined (therefore the signatures are initially symbolic signatures). When the REGISTER_STIR_SYMBOLS command is executed, these symbols are resolved, and the corresponding signatures become resolved STIR summary signatures.

The symbol-address mapping registration must be carried out in a secure way, to ensure that the malware is unable to register a malicious callback function. Therefore we assume that some other measure is taken to ensure that this registration is performed only when the guest OS is in a "known good" state. Since a guest OS is less likely to be compromised in the early stage of its life (e.g., during a secure boot [3]), our current implementation approximates this requirement by dividing the life time of a guest OS into a *symbol registration phase* (e.g., the initialization time in Figure 34) followed by a *guarding phase* (the evaluation time in Figure 34), where symbol mappings can be registered only in the *symbol registration phase* (during this phase the guest OS is assumed to be in a "known good" state). We further perform the phase transition automatically for the guest kernel when it performs such registration for the first time, which is intended to minimize the attack window where a malware can misuse the VMCALL interface to insert malicious address mappings. However, a side effect of this

particular implementation decision is that all legitimate LKMs that use soft timers must be loaded prior to the registration phase.

We note that this requirement may be undesirable for on-demand kernel module loading, but it can be resolved by other implementation options, such as verifying the runtime integrity of the guest kernel using Copilot [40] before allowing symbol mappings to be registered for a second time. Addressing these issues would improve the usability of the system, but security is already assured based on our assumptions. For these reasons, the usability issues are beyond the scope of this dissertation, and we leave them as future work.

3.3.3.2.3 Implementation of the STIR Checking

As shown in Figure 36, the current STIR Checker is implemented in a security VM running on Xen. Its core function is check_stir, which performs verification of pending STIRs. As Figure 37 shows, check_stir takes as input two integer parameters: *function* and *data*, and returns true (success) or false (failure). It uses the resolved STIR summary signatures that are transformed from symbolic STIR Signatures by the STIR Symbol Resolver.

```
boolean check_stir (unsigned long function, unsigned long data){
    Use function as index to look up the resolved STIR summary signature database.
    If no signature is located, return false.
    Otherwise, if the assertion part of the located signature is empty, return true.
                Otherwise, return assertion (data).
}
boolean assertion (unsigned long data){
    for each predicate (deref equals functionlist){
        if deref(data) matches no address in functionlist
        return false.
    }
    return true.
}
```

Figure 37: Pseudocode of check_stir

The function `deref` in Figure 37 uses the APIs provided by XenAccess [38] to dereference the data pointer (`data`) passed from the guest kernel (e.g.,

`data->tx_timeout`). The offset information is statically computed by using the output of the STIR Analyzer. For example, in order to dereference `data->tx_timeout`, where data is of type `struct net_device *`, we statically compute the offset of the field `tx_timeout` by analyzing the definition of `struct net_device`.

Finally, the soft timer dispatching logic of the guest Linux kernel is modified to make a VMCALL into Xen. Specifically, when a STIR in the pending timers queue expires, the guest kernel invokes a VMCALL, with the command CHECK_STIR, plus the *function* and *data* fields of the STIR as parameters. If the VMCALL returns true, *function* is called as normal. Otherwise, a warning message is generated and *function* is not invoked.

3.3.3.3. Evaluation of Linux Case Study

3.3.3.3.1 Effectiveness of Malicious STIR Detection

To evaluate the efficacy of our approach, we experimentally confirmed that our implementation of the STIR Checker is able to detect the key logger, the CPU cycle stealer and the alter-scheduler discussed in section 3.2.3. We first installed our three "malware" kernel modules into an unprotected guest Linux kernel and confirmed that they are able to achieve their intended malicious purposes (e.g., stealing key strokes). We then activated the STIR-Aware environment containing the modified guest kernel, the Lares-patched Xen VMM, and the security VM running the STIR-Checker. We first instructed the STIR Symbol Mapper in the guest kernel to register symbols with the STIR Symbol Resolver; currently this is initiated by loading the Symbol Mapper LKM. Then we installed the malware kernel modules. The STIR Checker is able to immediately generate warnings about the suspicious STIRs used by the newly loaded modules, and the malware functions are not invoked by the guest kernel as a result. The "malware"

modules have been implemented using both attack techniques mentioned in Section 3.3.2. These results confirm that our approach can stop both types of STIR attacks.

False Positives. Under the assumption that the STIR Analyzer processes the complete source code of the guest kernel (including all legitimate modules), the STIR Analyzer correctly carries out function pointer analysis, and the guest kernel installs all necessary and legitimate modules before registering symbol-address mappings, our detection can have no false positives. This is because all potential legitimate STIRs have been captured in the resolved STIR summary signature database before the guest Linux enters the *guarding phase* (Section 3.3.3.2.2).

False Negatives. Due to our detection methodology, in order to obtain control, the malware must reuse legitimate STIR callback functions (such as `dev_watchdog` in Figure 33), and manipulate the parameter passed to the STIR callback function in such a way that control will eventually go to its malicious code. One way to leverage `dev_watchdog` has been shown in Figure 33. However, our detection techniques counter this type of attack by calculating and verifying the legitimate functions that can be assigned to `dev->tx_timeout` as shown in Table 17, thus closing this possibility.

However, it is possible for the malware to search deeper in the control flow for opportunities, such as looking at the function `ace_watchdog` in Table 17, since `ace_watchdog` takes `dev` as the input parameter. This approach will also fail because the transitive closure analysis covers this case.

In summary, we believe that our detection can have no false negatives under the threat model in Section 3.3.2.1. However, since we may be facing a powerful adversary, our detection is not a panacea. A determined attacker may find a way not covered by our threat model to evade detection, although the STIR checking clearly raises the bar for an attacker.

Attacks on the STIR checking mechanism and our counter-measures. We anticipate that attackers may use either of two different kinds of attacks in an attempt to

defeat the STIR checking. (1) The malware may disable the modification to the soft timer dispatcher so that it does not make the VMCALL, or ignores the return value. We protect against this by using Lares to make the code page of the soft timer dispatcher read-only. (2) The malware may try to register false mappings for legitimate symbols. By performing the phase transition (Section 3.3.3.2.2), such actions are ignored and therefore have no effect.

3.3.3.3.2 Performance Overhead

In order to measure the performance overhead of the STIR Checker, we ran a set of synthetic workloads: *cat* - read and display the content of 8000 small files (with size ranging from 5K to 7.5K bytes) in a complicated directory tree. *ccrypt* - encrypt a text stream of 64M bytes, where ccrypt[3] is an open source encryption and decryption tool. *gzip* - compress a text file of 64M bytes using the --best option. *cp* - recursively copy a Linux kernel source tree. *make* - perform a full build of the Apache HTTP server (version 2.2.2) from source.

Table 18: Overhead measurement of the STIR Checker in execution time (seconds)

	cat	ccrypt	gzip	cp	make
Original	20.85	3.30	5.92	43.95	217.95
STIR-aware	20.96	3.30	6.01	46.61	218.58
Overhead	0.52%	0%	1.52%	6.05%	0.29%
Callbacks/Sec	46.9	46.3	47.3	61.4	81.6

Table 18 shows the execution times of the workloads under the original Linux and the modified Linux (denoted STIR-aware). The VMM used in these experiments is Xen 3.0.4, and the guest Linux kernel is version 2.6.16. The host CPU is an Intel Core 2 Duo

[3] http://sourceforge.net/projects/ccrypt/

running at 2.4 GHz with VT-x enabled. The host is allocated 1.5 GB of memory and the HVM (i.e., fully virtualized) guest is allocated 512 MB of memory.

From Table 18 we can see that the performance overhead of the STIR Checker on the synthetic workloads is low (less than 7%). Our testing found that out of the 365 STIR callback functions identified by the STIR Analyzer, only 74 are present in the guest kernel at runtime, and the majority of these STIR callbacks are dormant most of the time (although there may be multiple STIRs sharing the same call back function), therefore the frequency that a STIR actually expires (e.g., the frequency of the callbacks) is not high. For example, the baseline frequency of callbacks is around 45 per second. Table 18 shows the average frequency of callbacks during the experiment, which is similar to the baseline frequency.

We also evaluate the overhead of the STIR Checker by running the Iperf-2.0.2 benchmark[4]. In this experiment the security VM ran the Iperf server, and the guest VM ran the Iperf client. Iperf is used to measure the maximum throughput between the virtual NIC in the guest VM (the front end) and the virtual bridge in the security VM (the backend). The experiment is run for 60 seconds, using 64KB buffers and 10 concurrent connections. The average throughput in the original environment is 717.9 MB/s, and it is 688.4 MB/s in the STIR-Aware environment. This suggests a performance drop of 4.1% (decrease in network throughput). In addition, we measured the frequency of STIR callbacks during the Iperf experiment and found that it increased to 287 per second, which explains the slightly higher overhead of the STIR Checker compared to the synthetic workloads.

In summary, the performance overhead for the STIR-Aware environment is small compared to the added security benefit that it provides.

---

[4] http://dast.nlanr.net/Projects/Iperf/

3.4. A General Defense against K-Queue Driven Transient Control Flow Attacks

While our defense in Section 3.3 addresses the soft-time-driven attacks, there are other instances of K-Queues (Section 3.2.2) that can be leveraged in a similar way to the soft timer by an attacker to maintain stealthy control of the victim kernel. Therefore, we have extended our defense against soft-timer-based attacks to address the more general class of K-qeueue-driven control flow attacks. Our contributions are (1) a unified static analysis framework and a set of tools that can generate summary signatures and the corresponding checking code for different K-Queue instances. (2) A runtime reference monitor that validates K-Queue invariants and guards such invariants against tampering. (3) A comprehensive experimental evaluation of our tools on a series of Linux kernel configurations.

3.4.1. A Unified Static Analysis Framework and Tool Set

We build a unified static analysis framework (Figure 38) and develop a set of tools that can be used to derive specifications of legitimate K-Queue requests, based on the following observation: although details of different K-Queues may vary, their specifications can be derived by a *common* set of analysis tasks. For example, the top-level legitimate K-Queue callback functions can be derived by a *points-to* analysis of the function pointer embedded in the respective K-Queue request data structures (Figure 26 – Figure 28); and every legitimate K-Queue callback function that takes a data parameter needs to go through a *transitive closure* analysis. It is more scalable and efficient to separate such analysis requirements out and allow the different K-Queue analyzers to share them. This way, future K-Queue analyzers will not have to repeat the effort of existing K-Queue analyzers. Therefore, we develop basic analysis tools and an analysis engine that composes these basic tools to carry out the analysis for each K-Queue instance.

Our analysis framework has the following advantages:

- General: the same framework engine can be used by any K-Queue instance, with only different starting *seed* analysis tasks. Other than that, all K-Queue analysis proceeds in a similar fashion. This ensures that our framework can handle future K-Queue instances not covered in Section 3.2.2.
- Incremental: our framework uses a database to store basic analysis results. This database enables accumulation of static analysis results over time, and more importantly, it facilitates sharing of basic analysis results among the different K-Queue analyzers.
- Automated: we develop a set of static analysis tools that can process the kernel source code and generate stubs of the corresponding checker code. Such automation greatly simplifies the job of a human analyzer.
- Tunable: our analysis framework leverage persistent data (e.g., a file-based work list and database tables) that can be easily modified offline, which offers an opportunity for correcting errors or omissions made by the analysis tools. Then we can have improved analysis precision.

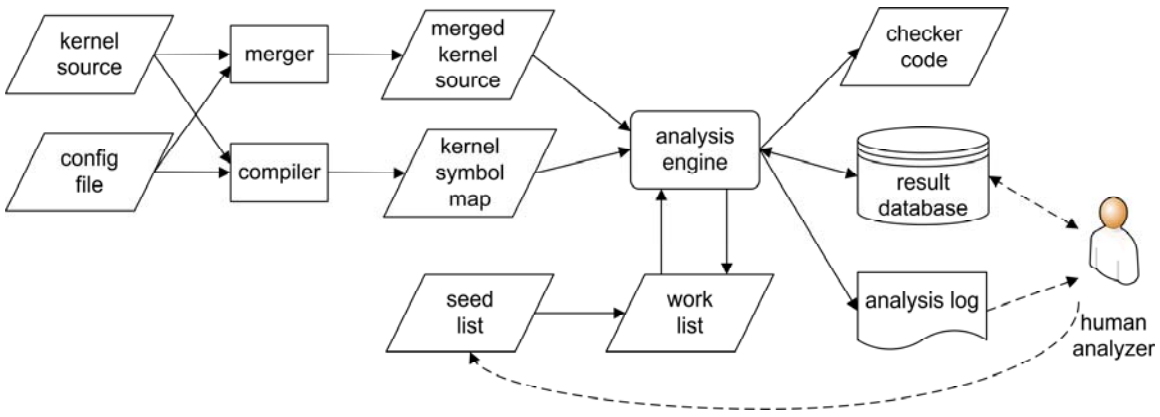

Figure 38: K-Queue static analysis framework

3.4.1.1. Basic Analysis Tasks

One of the basic analysis tasks is points-to analysis of function pointers, since our defense needs to know the legitimate targets of function pointers. For example, the top-level legitimate K-Queue call back functions can be recognized in this way. We observe that there are certain "contracts" between a K-Queue and its requesters that enable automated inference of legitimate K-Queue callback functions. Figure 26 – Figure 28 show the data structures in certain K-Queues that need to be initialized by a requester and read by a K-Queue dispatcher. Among the fields that must be initialized is the callback function. Table 19 summarizes possible ways that a callback function field can be assigned for different K-Queues.

- Direct assignment (DA). This is an unstructured way of assignment. The requester needs to be aware of the K-Queue data structure, allocate memory for it, and initialize its fields. But here a callback function is directly assigned to the appropriate field, as oppose to the indirect assignment case below.
- Indirect assignment (IA) through an intermediate variable. This way is also unstructured, but different from the DA case above, a function pointer variable is assigned to the callback function field. For example, `do_floppy` in Table 19 may point to several possible functions under different conditions.
- Assignment through a function parameter (PA). This is a structured way of assignment in which a requester can call a wrapper function which in turn initializes a K-Queue data structure. The actual callback function is passed in as a parameter to the wrapper function. For example, the task queue callback function can be assigned indirectly through a parameter to the function `schedule_bh`, which in turn assigns it to the `routine` field of a task queue structure. In order to capture this kind of functions, the analyzer must recognize every function call to `schedule_bh` and record the corresponding actual parameter.

Accordingly, we can decompose the points-to analysis task into three kinds of simpler tasks: direct assignment analysis, indirect assignment analysis, and parameter assignment analysis.

Table 19: Possible ways that a call back function can be assigned in different K-Queues

	Direct Assignment		Parameter Assignment		Indirect Assignment
	Structure	Field	Function	Index (from 0)	
Soft timer	timer_list	function			
IRQ action queue	irqaction	handler	request_irq	1	
Tasklet queue	tasklet_struct	func	tasklet_init	1	
Task queue[5]	tq_struct	routine	schedule_bh	0	do_floppy

Another basic analysis tasks is transitive closure analysis, which identifies constraints on the data parameters passed on to a legitimate target function. For example, the K-Queue instances discussed in Section 3.2.2 all pass a requester-supplied data parameter to the callback function. If the callback function makes control transfer decisions based on the data parameter, we must make sure that the attacker cannot supply a malicious data parameter to induce kernel control flow to the malware (e.g., via a type 2 attack as the one in Figure 33).

3.4.1.2. The Analysis Engine

The heart of the K-Queue static analysis framework is the analysis engine, which repeatedly consumes individual analysis tasks (Section 3.4.1.1) from a work list. This work list is dynamically changing: on one hand, tasks are removed from it by the analysis engine; on the other hand, new tasks may be appended to it as a result of performing an analysis task. The analysis process is bootstrapped by some *seed* tasks inserted to the work list by a human analyzer, and it finishes when the work list becomes empty.

---

[5] Task queue in Linux kernel 2.4.32 is the predecessor of work queue found in Linux kernel 2.6

During each individual analysis task, the analysis engine runs one or more of the basic tools on the merged kernel source file as necessary and generates three kinds of output: (1) source code for the runtime checker to verify the integrity of a pending K-Queue request, (2) static analysis results that are stored to a database for reuse, and (3) detailed logs for in-depth diagnosis by a human analyzer.

3.4.1.3. The Work List

The work list contains pending static analysis tasks. Each element in this list specifies the type of analysis (direct assignment, parameter assignment, or transitive closure) and the corresponding input parameters. One example is <DA, tasklet_struct, func, 1>, which instructs the analysis engine to invoke the direct assignment collector for the *func* field of kernel data structure *tasklet_struct*. This task can bootstrap an analysis for the tasklet queue (section 3.2.2.2), one of the K-Queues.

3.4.1.4. Basic Tools

These are the building-blocks of the static analyzer that carry out the basic analysis tasks discussed in Section 3.4.1.1.

3.4.1.4.1 Direct Assignment Collector

This tool takes as input the name of a structure (e.g., `irqaction`) and the name of a field (e.g., `handler`) within that structure, and outputs kernel functions that can be assigned to such a field. It traverses each assignment statement (`lval = rval`) of the kernel. If `lval` ends with a field with the specified name, this field belongs to a structure with the specified name, and `rval` is an actual function, the tool collects `rval` as a legitimate function. If `rval` is not an actual function (e.g., a formal parameter), the tool dumps the exact expression of `rval` to the log file, which can help a human analyzer find corner cases that need other means of analysis (e.g., the Parameter Collector).

3.4.1.4.2 Parameter Collector

This tool collects target functions that are passed to a wrapper function as an actual parameter and later assigned to a function pointer (i.e., in the PA case in Table 19). It takes as input the name of a wrapper function and the index of the parameter of interest. It traverses the entire kernel searching for each invocation to the specified function, and collects the actual values of the parameter at the specified index.

3.4.1.4.3 Transitive Closure Analyzer

This tool is a major component of the tool set. It takes as input the name of a function and a list of its formal parameters that are tainted, i.e., these parameters can influence the control flow of the given function. This tool performs a flow-sensitive and intra-procedural transitive closure analysis, starting from the given function and descending into functions called by the given function and so on. It is flow-sensitive because it propagates taint to downstream functions through parameters. It is intra-procedural because only downstream functions defined within the same source file as the given function are analyzed. In case that a downstream function is located in a different source file, an external transitive closure analysis task is scheduled for execution later.

This tool builds a hash table of all functions defined in the given kernel source file, so that it can quickly navigate to any function to continue the analysis. It also maintains a list of functions that needs to be analyzed (called a work list). Initially, the work list contains only the function given as the input of this tool. As this tool processes the given function, it may recognize more functions that need to be analyzed; then it adds such functions to the work list. The main body of this tool is a loop over the work list until it becomes empty. For each function in the work list, this tool performs two kinds of tasks: taint propagation and new analysis task recognition.

Taint propagation. The tool traverses each assignment statement (`lval = rval`) in the function and taints the variable `lval` if any part of `rval` is already tainted.

New analysis task recognition. The tool traverses each function call statement `fn(args)` in the function to see if any part of `fn` or `args` or both is a tainted variable. If `fn` is tainted, a new points-to analysis task is generated for `fn` after the corresponding structure name and field name are derived from `fn`. If `args` is tainted and `fn` is an actual function, a transitive closure analysis task is generated for `fn` with the list of tainted arguments.

The work list maintained by the transitive closure analyzer is called an *internal work list* to differentiate it from the *external work list* used by the static analysis engine in Figure 38. New points-to analysis tasks are added to the external work list. New transitive closure analysis tasks are first added to the internal work list, and if they can not be handled because the corresponding function is not defined within the given kernel source file, they are added to the external work list with the hope that they will be found in some other source file.

3.4.1.5. Kernel Merging

One challenge of transitive closure analysis is how to continue analysis on downstream functions invoked by the current function – if such downstream functions are not in the current source file, the analysis task needs to be recorded somewhere and later tried on a different source file. Although our external work list can satisfy this requirement, this kind of interprocedural transitive closure analysis can be very inefficient, because many kernel source files may need to be sifted through before the body of a function is found. To optimize transitive closure analysis, we merge the entire kernel (given a configuration) into a single source file, so that the interprocedural analysis tasks are all turned into intraprocedural analysis. We test our analysis engine on a series of merged kernel source files in Section 3.4.6.2.

3.4.1.6. Result Database

In our initial implementation, each kind of K-Queue analyzer runs independently. We quickly find out that there are redundant analysis tasks among the K-Queue analyzers that can be avoided. For example, the fact that the task queue analyzer has performed points-to analysis on structure *scsi_cmnd* and field *done* is agnostic to the IRQ action queue analyzer even if the latter needs to perform points-to analysis on the same pointer. A more specific measurement of redundancy is presented in Table 20, which shows the number of common analysis tasks among pairs of the K-Queue analyzers. When such sharing is significant, an analyzer may waste much time processing analysis that has been handled. As Table 21 shows, without sharing, the soft timer analyzer runs for 284 minutes on a kernel of 482,369 lines of code, but with sharing it only needs 127 minutes on the same kernel. This means that enabling sharing among the K-Queue analyzers can have significant time savings. Therefore, we introduce a database of individual points-to and transitive closure analysis results that is shared among the different K-Queue analyzers. This database contains two tables: pointsTo and transClosure, the formats of which are shown in Table 22 and Table 23, respectively.

Table 20: Number of common analysis tasks among different K-Queues

	Transitive Closure	Points-to
IRQ action queue vs. soft timer queue	97	51
IRQ action queue vs. task queue	4	2
Task queue vs. soft timer queue	4	3

Table 21: Benefit of sharing on the K-Queue analysis time

K-Queue	Without sharing	With sharing	% time savings
Task queue	155 minutes	147 minutes	5.2
Tasklet queue	360 seconds	314 seconds	12.8
IRQ action queue	175 minutes	166 minutes	5.1
Soft timer queue	284 minutes	127 minutes	55.3

When the analysis engine (Section 3.4.1.2) sees a points-to analysis task, it first uses the structure and field names as a key to query the pointsTo table. If a row is found, it directly uses the returned points-to set. Otherwise, it invokes the points-to analysis

tools (e.g., the Direct Assignment Collector and the Parameter Collector) and inserts the results to the pointsTo table. The analysis engine uses the transClosure table in a similar fashion except that it uses the function name and the list of tainted arguments as search keys.

Table 22: Format of the table pointsTo

Field	Type	Meaning
sname	varchar[40]	Structure name
fldname	varchar[40]	Field name
p2set	blob	Set of function names

Table 23: Format of the table transClosure

Field	Type	Meaning
Fname	varchar[40]	Function name
Arglist	varchar[20]	List of tainted arguments
pointer_set	blob	Set of function pointers used

3.4.2. Code Generation for the K-Queue Checkers

The analysis tool generates code stub for the runtime checker. The generated code includes two kinds of functions: those for verifying the control flow integrity of a function pointer and those for verifying the control flow integrity of a real function. Figure 39(a) shows the function pointer checker code for structure *irqaction* and field *handler*, and Figure 39(b) shows the checker code for the real function *rtl8139_interrupt*.

The main body of the code in Figure 39(a) performs a series of comparisons to match the runtime value of a function pointer to a real function in its points-to set. If a match is found, the integrity of the function pointer is reduced to that of the matching real function. If no such match is found, the function pointer has no integrity because it points to something unexpected (e.g., the malware code). In other words, the integrity of a function pointer is the disjunction (logical OR) of the integrity of all its legitimate targets (real functions).

```
int  check_pointer_irqaction_2_handler_01(unsigned
int data){
  unsigned int fp;
  int offs[1] = {-1};

/* Fetch the function pointer value into fp */
  fp = deref_data(offs, data);
  if (fp == 0) return 1;

  if (fp == 0xc010c6cc)
    return check_func_math_error_irq_01(data);
  …
  if (fp == 0xc01b4f50)
    return check_func_rtl8139_interrupt_01(data);
  …
  unlock_kqueue_regions();
  return 0;
}
```

(a)

```
int check_func_rtl8139_interrupt_01(unsigned int
data){

 return 1
       &&
check_pointer_mii_if_info_2_mdio_read_110(data)
       &&
check_pointer_pci_ops_2_read_word_100(data)
       &&
check_pointer_pci_ops_2_write_word_100(data);
}
```

(b)

Figure 39: Generated code for a function pointer (a) and a real function (b)

Similarly, the integrity of a real function is the conjunction (logical AND) of the integrity of all function pointers that it transfers control to, and if no such function pointers are used, the function has integrity by default. For example, *rtl8139_interrupt* in Figure 39(b) invokes three function pointers. One of these function pointers has structure name *mii_if_info* and field name *mdio_read*.

Note from Figure 39(a) that the function pointer checker uses constant address (e.g., 0xc010c6cc) to recognize actual target functions at runtime. This is an optimization performed by the static analyzer to avoid address translation at runtime, which is possible for target functions built into the kernel. Specifically, the code generator looks up the kernel symbol map generated by the normal kernel compiler to carry out such translations.

Also note from Figure 39(a) that before comparison the code fetches the runtime value of the function pointer by calling `deref_data` (Figure 40). `deref_data` takes as input an array of integers (i.e., `offsets`) representing byte offsets. The exact content of this array is not supplied by the current implementation, because the automatic derivation of a function pointer expression by dereferencing the data variable is still an

ongoing research problem. For now, the output of the transitive closure analyzer contains enough information for a human analyzer to derive such an expression. Once that is done, our offset analyzer (Section 3.4.3) can automatically analyze the given pointer expression and generate offset information to fill in the offset array in Figure 39(a).

```
unsigned long deref_data(int *offsets, uint32_t data){
  unsigned long i = 0;
  uint32_t tmpval, uint32_t guest_p = data;

  while (offsets[i] != -1){   /*offsets array ends with -1*/
    if (guest_p == 0)   return -1;          /* refuse to dereference null pointer */

    lock_kqueue_region(guest_p + offsets[i], guest_p + offsets[i] + sizeof(uint32_t));

    if (lares_copy_from_guest(&tmpval, guest_p + offsets[i], sizeof(uint32_t)))
       return -1;

    guest_p = tmpval;
    i++;
  }
  return (unsigned long) guest_p;
}
```

Figure 40: Source code for retrieving the value of a function pointer from a guest VM

3.4.3. The Offset Analyzer

Since our runtime K-Queue checker employs the same architecture as the STIR Checker discussed in Section 3.3.2.3, there exists a semantic gap between the K-Queue checker and the guest kernel that resides in a different address space. Specifically, before the K-Queue checker can evaluate a pointer expression in the guest kernel, it needs to convert the structure field information into byte offset information; because what the K-Queue checker can access are just raw memory pages of the guest kernel. Therefore, we provide an offset analyzer for this purpose.

3.4.3.1. Computing the Byte Offsets for Individual Fields

Given the definition of a structure, in order to compute the byte offset of a particular field within this structure, a naïve offset analyzer would just sum up the bytes occupied by the fields preceding the given field. However, this approach may give wrong

results because of the padding of structure fields by the compiler. Specifically, a modern compiler can pad additional bytes between a field and its successor so that the latter will be properly aligned in memory [47]. Normally, such padding is invisible to the programmer. However, when we want to fetch the value of a field from the raw memory, we must have the correct offset information taking the padding into account.

Although the C standard[6] specifies the expected way of structure alignment and padding, statically computing the padding is error prone and non-portable because the exact number of bytes to pad depends on several factors, mainly the compiler and the machine architecture. On the other hand, at runtime, we can reliably get the offset of a field by using the '&' operator. I.e., to compute the offset of field `f` within a structure `s`, we can simply compute `&s.f-&s`. Therefore, we take a hybrid approach which proceeds in several steps:

(1) Statically generate code that defines a variable for each structure type in the kernel and calculates the offset of each field within its structure when running. For example, for struct `s` and field `f` defined in Figure 41(a), our approach generates the code snippet in Figure 41(b).

(a)	(b)
struct s{ … struct foo f; …}; struct foo{ … int bar; … };	struct s s_v; printf("s f %d foo 0\n", (unsigned int) &s_v.f - (unsigned int) &s_v);

Figure 41: Code generation for offset analysis

The code in Figure 41(b) also displays the type name of field `f` (i.e., foo) and whether `f` is a pointer ("0" means "no", and "1" means "yes"). Such information is used to make further dereference starting from `f` (e.g., `s.f.bar`), which will be discussed in more detail shortly;

---

[6] ISO/IEC Standard 9899:1999

(2) Compile the generated code on the target architecture. The major caveat is to include proper header files of the kernel in the generated code so that it can compile, which can be tricky. We solve this problem by merging all type definitions of the kernel into the generated code.

(3) Run the generated code on the target architecture and collect the output.

(4) Transform and store the output into the *offset database table* for future inquiries. This table has five attributes: structure name, field name, field offset, field type (structure name), and whether the field is a pointer.

3.4.3.2. Computing Offset Information for Arbitrary Pointer Expressions

Given a function pointer expression $S_0.f_1.f_2.\cdots.f_n$, the offset analyzer returns an array of non-negative offset integers using the information in the offset database table. It first uses (S_0, f_1) to query the offset database table, if the result is not empty, it should contain the byte offset of f_1 within structure S_0, and the type name of f_1 if it is a structure type (let's call the type S_1 for now). If f_1 is a structure type, the offset analyzer uses (S_1, f_2) to query the offset database table to obtain the byte offset of f_2 within S_1. This process continues until the offset of f_n within structure S_{n-1} is found from the database.

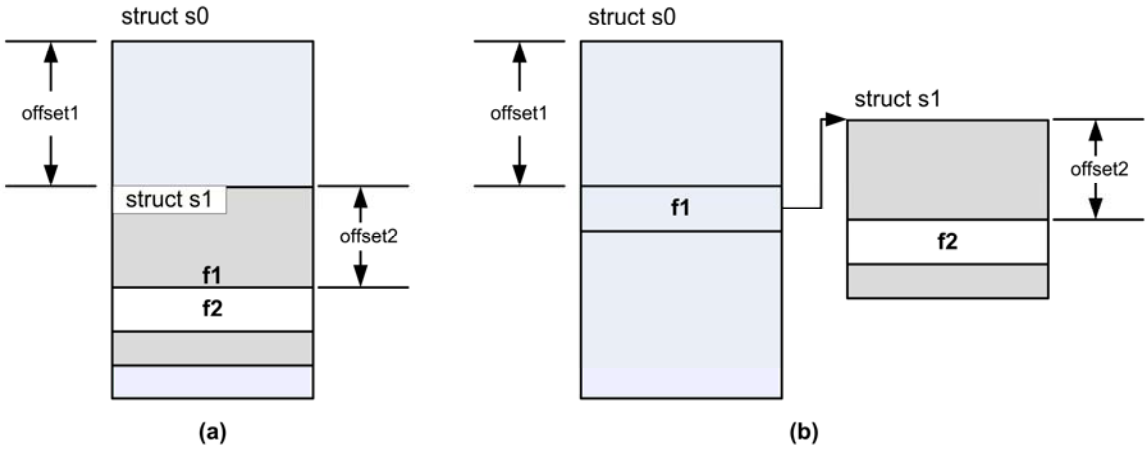

Figure 42: Dereferencing of complex function pointers

The offset database table records whether a field is a pointer, because this attribute influences the number of memory accesses when we evaluate a function pointer expression at runtime (i.e., the length of the array returned by the offset analyzer). Consider the above example again, if f_1 is a pointer as shown in Figure 42(b), in order to retrieve the value of f_2 we need to have two memory accesses, the first one retrieves S_0 to get f_1, and the second one retrieves S_1 to get f_2. However, if f_1 is not a pointer as shown in Figure 42(a), we only need one memory access because f_1 is a structure embedded in S_0, so f_2 can be obtained by reading directly from the byte offset offset1+offset2 of structure S_0.

3.4.4. Guarding of K-Queue PTIs at Run-time

3.4.4.1. TOCTTOU Attack against the K-Queue Defense

So far our defense validates (checks) a K-Queue PTI before the guest kernel uses it (i.e., invokes a K-Queue callback function). However, if the guest kernel is multi-threaded, which is the case for the current Linux kernel, such a defense is vulnerable to a TOCTTOU (Time-Of-Check-To-Time-Of-Use) attack: right after the K-Queue PTI is checked, but before the K-Queue call back function finishes, a malicious control flow in the guest kernel can potentially modify a function pointer involved in the PTI, so that the call back function transfers control to the malware. This constitutes a TOCTTOU race condition. Such attacks may be hard to mount and succeed, but they are possible.

3.4.4.2. Countermeasures to the TOCTTOU Attacks

To counter the TOCTTOU attack against our defense, we protect the function pointers participated in a K-Queue PTI from tampering during the execution of the K-Queue call back function. Specifically, we temporarily write-protect memory regions in the guest domain that hold such function pointers until the guest kernel finishes the K-

Queue call back function. To support this kind of write-protection, we extend the shadow page table manager of the hypervisor so that it masks a memory page as read-only if it contains a protected memory region. In case there is a legitimate write to the same page but out of the protected region, we use emulations. We add a hyper call for the security VM to request that a memory region for any guest VM be protected or unprotected.

During the K-Queue PTI checking, each participating structure field is first write-protected (i.e., locked by `lock_kqueue_region` in Figure 40) and then checked. The addresses of the structure fields protected so far are recorded in an array, so that when the checking fails at any point, the already protected structure fields can be unlocked (i.e. by `unlock_kqueue_regions` in Figure 39(a)). If the checking succeeds, the unlocking is deferred until an acknowledgement is received from the guest VM that the call back function has finished.

We take careful measures to unlock structure fields as soon as possible. This is because of the performance penalty caused by page-level write-protection. Since hardware support for fine-grained memory protection is not widely available, we have to satisfy with a suboptimal page-level protection.

Our memory region protection can defeat the TOCTTOU attacks mentioned above. Since we lock a structure field before using it, the attacker cannot change the verification result. If the attacker modifies a legitimate function pointer before it is locked (and thus verified), the checker will discover this modification and not follow (use) the function pointer. On the other hand, the attacker cannot modify the function pointer immediately after it is verified, because it has been locked. By the time the attacker can modify the function pointer, it has been used (i.e., the callback function has finished), so the modification is harmless. Note that if we first verify then lock, then there is a possibility that an attacker modifies the verified function pointer before it is locked, thus the attack can still succeed. So verifying then locking is a wrong design.

3.4.5. Implementation

3.4.5.1. The K-Queue Analyzers

We implement the static analyzers for the IRQ action queue, the tasklet queue, the task queue, and the soft timer queue based on our static analysis framework. Such analyzers are extended from the STIR analyzer (Section 3.3.3.1), so they use the same CIL [37] tool. We implement the analysis engine in Shell scripts, which invokes the CIL modules that implement the basic analysis toos (Section 3.4.1.4). These modules are written in Objective Caml[7]. We use MySQL[8] (version 5.1.34) to store the result database (Section 3.4.1.6), and write a Java program to insert into or query the result database.

3.4.5.2. The K-Queue Defense

We implement runtime defense for the IRQ action queue, the tasklet queue, and the task queue, based on the STIR Checker in Section 3.3.3.2, which includes a security VM component and a guest VM component. The STIR Checker also has a better implementation because of the code generation. We define several new commands in the parameter structure passed through the VMCALL from the guest kernel to the security VM; these commands correspond to invocations and completions of the new K-Queue requests.

The K-Queue checkers in the Security VM are implemented based on the code stubs generated by the K-Queue analyzers. They inspect the runtime status of the guest kernel via the XenAccess [38] library, and they use the offset information returned by the Offset Analyzer (Section 3.4.3). We implement the Offset Analyzer in a mixture of CIL module, Shell script, and Java program, and the offset result database is again MySQL.

---

[7] http://caml.inria.fr/ocaml/index.en.html
[8] http://www.mysql.com

Finally, we modify the dispatching logic of the IRQ action queue, the tasklet queue, the soft timer queue, and the task queue in the guest kernel (version 2.4.32), so that a VMCALL is made into Xen to start the K-Queue checking before a pending K-Queue request is invoked. The guest kernel is suspended until the result comes back from the security VM. Table 24 summarizes the modifications to the guest kernel.

Table 24: Modifications to the guest kernel

K-Queue instance	Kernel function(s) modified	Location
Tasklet queue	tasklet_action, tasklet_hi_action	kernel/softirq.c
IRQ action queue	handle_IRQ_event	arch/i386/kernel/irq.c
Task queue	__run_task_queue	kernel/softirq.c
Soft timer queue	run_timer_list	kernel/timer.c

3.4.6. Evaluation of the K-Queue Defense

3.4.6.1. Security Properties

Our K-Queue defense has no false negatives because all function pointers occurring in the control flow of the callback function are validated no matter if they are actually invoked by the callback function or not. Specifically, the transitive closure analyzer searches through every possible execution path (starting from the callback function) and recognizes function pointers along the way. Some of the function pointers may not be called in a particular invocation, but the analyzer conservatively reports all such function pointers for points-to analysis.

On the other hand, our implementation of the K-Queue defense may have false positives, due to the limitations of the points-to analysis module in our K-Queue analyzer. Our current implementation of the K-Queue analyzers covers direct assignment (DA) and some cases of parameter assignment (PA), but not indirect assignment (IA). Therefore, a real function assigned through IA is not collected into the points-to set by the tools

automatically. This will result in a false alarm at runtime should such a function is invoked.

One example of IA is through `do_floppy`, a global function pointer variable that can point to different functions (`main_command_interrupt`, `seek_interrupt`, `recal_interrupt`, or `reset_interrupt`) under different situations. To fully capture the IA case would require an alias analysis which is by itself an open research area [27].

Our current implementation does not support function pointer arrays, either. For example, `bh_action` in kernel/softirq.c invokes `bh_base[nr]` where `nr` is the data associated with `bh_action` in a tasklet request. Fortunately, the integrity of `bh_base[nr]` can be verified by finding the *known-good* values of the global array `bh_base`, which is assigned by the `init_bh` function call. By analyzing all calls to `init_bh` and collect the second parameter, we can figure out the known-good values of this array.

Table 25: Complicated function pointers encountered by the task queue analyzer

Pointer specification		Parameter Assignment		Indirect Assignment
Structure	Field	Function	Index (from 0)	
acpi_os_dpc	function	acpi_os_queue_for_execution	1	n/a
ata_queued_cmd	scsidone	ata_scsi_qc_new	3	n/a
buffer_head	b_end_io	init_buffer	1	"callback" in fs/xfs/linux-2.4/xfs_buf.c, function _pagebuf_page_io
pci_socket	handler	pci_register_callback	1	n/a
scsi_cmnd	done	scsi_do_cmd	4	"SRpnt->sr_done" in drivers/scsi/scsi.c, function scsi_init_cmd_from_req

Despite the limitations, our implementation can satisfy the majority of pointer analysis tasks in the K-Queues that we found. For example, out of 55 points-to analysis tasks for the task queue, 50 use direct assignment (DA). The corner cases include five parameter assignments (PA) and two indirect assignments (IA), as listed in Table 25. The

most complicated case is a PA analysis for `ata_queued_cmd->scsidone`, which involves multiple levels of PAs. But our manual analysis only took several minutes to find that the points-to set is {`scsi_done, scsi_old_done, scsi_eh_done`}.

3.4.6.2. Performance and Scalability of the K-Queue Static Analyzer

We test the performance of our K-Queue static analyzer on a series of 10 configurations of a Linux kernel with increased complexity. The kernel version used in the evaluation is 2.4.32. The first configuration is a minimal kernel that can boot the guest virtual machine. It contains 482,369 lines of code, with essential support for IDE disk, ext3 file system, and TCP/IP networking. Each successive configuration includes more device drivers. The most complex kernel configuration contains 1,010,196 lines of code. A summary of these 10 kernel configurations is presented in Table 26.

Table 26: Configurations and complexity of the kernels used in the evaluation

Configuration	Description	Lines of code (LOC)
1	Baseline, minimal configuration	482,369
2	+ Multi-device support + Networking options + Telephony Support	563,944
3	+ ATA/IDE/MFM/RLL support	592,472
4	+ SCSI support (part I)	633,021
5	+ SCSI support (part II)	685,526
6	+ SCSI support (part III)	765,729
7	+ e100 network device support (part I)	820,610
8	+ e100 network device support (part II)	882,138
9	+ e1000 network device support	948,183
10	+ wireless network device support	1,010,196

Each experimental run covers four kinds of K-Queues in the following order: task queue, tasklet queue, IRQ action queue, and soft timer queue. Initially the analysis result database is empty. As the analysis proceeds the analysis results are accumulated in the database. Each K-Queue instance takes advantage of analysis that has finished, including its own analysis tasks and the K-Queue instance(s) ahead of it. For example, the analysis

for the soft timer queue uses some results of the IRQ action queue, so it takes less time than if it has no existing results to use.

The experimental run for each kernel configuration proceeds as follows. Each K-Queue analysis starts with a points-to analysis task. When the points-to set is determined, a round of transitive closure analysis is performed, one for each function in the points-to set. As the result of the transitive closure analysis, new points-to analysis tasks may be recognized. If this is the case, another round of points-to analysis is performed, which may lead to one more round of transitive closure analysis. This iterative process continues until the last round of transitive closure analysis recognizes no new points-to analysis tasks.

All the experiments run on a 3.0 GHz Intel Pentium 4 with 1 GB of RAM.

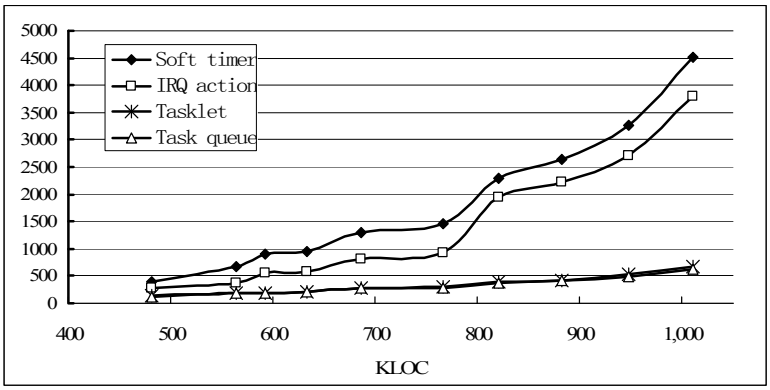

Figure 43: Cumulative Analysis Time (in minutes)

The first thing that we measure is the execution time of the K-Queue analysis. Figure 43 shows the cumulative execution time at four milestones for different kernel configurations. For example, the curve marked as “IRQ action” represents the total analysis time for the task queue, the tasklet queue, and the IRQ action queue. The X-axis is the complexity of the kernel configurations measured in KLOC or “thousand lines of code”, and the Y-axis is the cumulative execution time in minutes. The ten points on each curve correspond to the measurements for the ten kernel configurations, the left-

most point corresponds to configuration 1, and the right-most point corresponds to configuration 10.

From Figure 43 we can see that in general the analysis time increases as the complexity of the kernel increases. However, it seems that the execution time is not a simple function of the kernel size. In fact, we can see flat segments as well as steep slopes on the curves, suggesting a non-uniform distribution of the K-Queue requesters in the kernel. For example, the first steep slope occurs on the IRQ action queue curve from configuration 2 to configuration 3. This is because configuration 3 requires more analysis tasks. For example, from configuration 2 to configuration 3, the points-to analysis for structure `hwif_s` and field `ide_dma_test_irq` returns six more actual functions. These functions belong to the device drivers for several kinds of IDE controller chipsets (including the CMD64 series of chipsets and the HPT36X/37X chipset) that are added in configuration 3. These new actual functions demands more transitive closure analysis than configuration 2. However, from configuration 3 to configuration 4 the IRQ action queue curve is pretty flat, because there are few new analysis tasks.

The way that the execution time curves look like is expected, because our choice for new kernel configurations is agnostic to K-Queue usage.

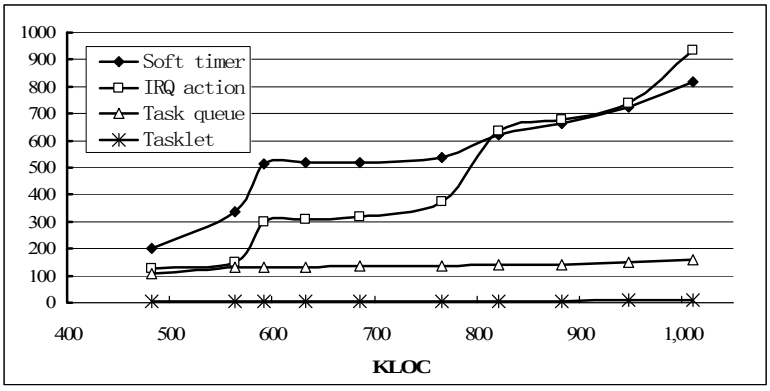

Figure 44: Number of External Transitive Closure Analysis

Figure 44 shows the number of external transitive closure analysis for the four kinds of K-Queues and different kernel configurations. Since we use merged kernel

source files, all such analysis is due to new results from points-to analysis. Clearly, this number increases as the kernel size increases. The reasoning is as follows. As the size of the kernel grows, more source code is analyzed; then the number of requesters for a particular K-Queue is potentially increased. This leads to a larger points-to set for the top level function pointers, thus more functions that need transitive closure analysis. The new transitive closure analysis may lead to new points-to analysis tasks, which result in more transitive closure analysis, and so on.

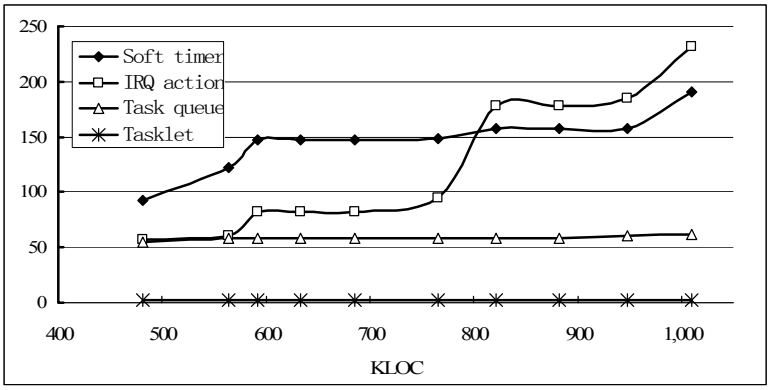

Figure 45: Number of Points-to Analysis

The above reasoning is supported by Figure 45, in which we show the measurement of the number of points-to analysis during the experiments. We can see that for all four kinds of K-Queues, the number of points-to analysis tasks indeed increases with the size of the kernel.

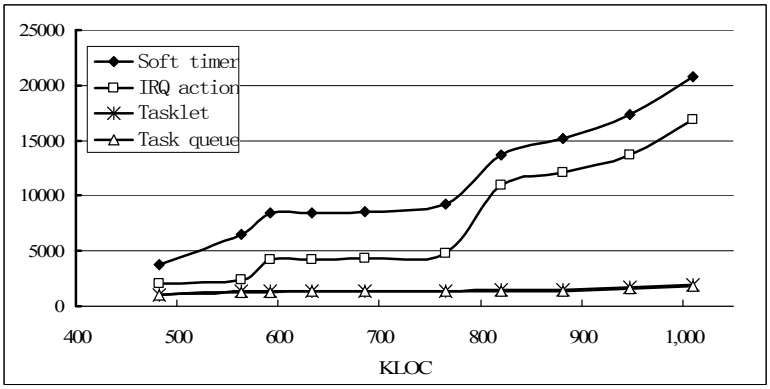

Figure 46: Number of Cumulative Internal Transitive Closure Analysis

Figure 46 shows the cumulative number of internal transitive closure analysis during the experiments. The curves have a similar trend as the number of external transitive closure analysis and points-to analysis, but at a much larger scale (20x). This demonstrates the benefit of kernel merging: if it is not used, a large number of such internal transitive closure analyses would become external transitive closure analyses; then the total analysis time would increase dramatically. This is because an external transitive closure analysis is more time-consuming than an internal transitive closure analysis. Each external transitive closure analysis has a constant overhead of preprocessing and parsing the entire kernel source code, while internal transitive closure analysis does not incur such overhead. As the kernel becomes more complex, such overhead becomes more and more significant.

One interesting point on Figure 46 is that up until configuration 6 the soft timer queue accounts for the most internal transitive closure analysis among the four K-Queues. But starting from configuration 7, this dominance is lost to the IRQ action queue, and the number of internal transitive analysis for the soft timer queue even drops from 4,457 in configuration 6 to 2,715 in configuration 7. This is a correct behavior, because the number of internal transitive closure analysis for the IRQ action queue increases dramatically from 3,449 in configuration 6 to 9,485 in configuration 7, in such a way that they cover a significant portion of the analysis for the soft timer queue. As a supporting evidence, the analysis for the IRQ action queue took 1,548 minutes in configuration 7, which is significantly longer than that for configuration 6 (630 minutes), as shown in Figure 43.

The Simplest K-Queue

From the evaluation, it seems that the tasklet queue is the simplest K-Queue. The numbers of points-to and transitive closure analysis stay very low until the nineth configuration. The entire analysis can be finished in less than 15 minutes in most cases. This suggests that tasklet is not heavily used in Linux kernel 2.4.32.

3.4.6.3. Benefit of the Code Generation

Our K-Queue static analysis tools generate the corresponding checker code as a by-product. For example, they generate over 5,800 lines of code for kernel configuration 1 in Table 26. Without automated code generation, it would be very time-consuming to develop such checker programs manually.

3.4.6.4. Performance Overhead of the K-Queue Checker

To measure the runtime overhead of our K-Queue Checker, we run the five benchmarks used to measure the overhead of the STIR Checker (Section 3.3.3.3.2). These benchmarks run on a 2.66 GHz Intel Core 2 Duo with VT-x support, the security VM (Domain 0) is allocated 512 MB of RAM, and the guest VM is allocated 256 MB of RAM. The hypervisor is Xen 3.3.0, and the guest kernel is Linux 2.4.32 with configuration 1 (Table 26). Each experiment is run 10 times and the mean and standard deviation of the measurements are computed. Table 27 shows the preliminary results.

Table 27 contains three kinds of results. The "Original" results are collected on unmodified Xen and guest kernel and serve as the baseline. The results marked as "K-Queue-aware, no lock" are collected on the modified Xen and guest kernel, but with the page-level memory protection (Section 3.4.4.2) turned off. Finally, the results marked as "K-Queue-aware, lock" are collected on the full-fledged defense mechanism including the modified Xen, the modified guest kernel, and the page-level memory protection.

From Table 27, we can see that our implementation of the K-Queue Checker incurs performance overhead ranging from 11% to 25 times slow down. This is much higher than the overhead measurement for the STIR Checker (Table 18).

In order to understand the result, we carry out an event analysis of the K-Queue runtime defense and identify three reasons for the high overhead: (1) the K-Queue defense needs to protect more K-Queue instances (four instead of one) and some K-Queue call backs happen at a high frequency. For example, IRQ action call back

functions happen at the rate of roughly 136 per second, and tasklet call back functions have a similar frequency, not to mention the other two K-Queues. (2) More importantly, some K-Queue call back functions are very complicated, so they require a large number of function pointer verification. For example, *ide_intr*, the IRQ action call back function for the IDE disk, requires a total of 192 function pointers to be verified, which leads to 648 cross-domain introspections. On the other hand, the most complicated soft timer call back function *dev_watchdog* needs only three function pointer verification and five cross-domain introspections, and the second most complicated soft timer call back function *neigh_periodic_timer* needs only two function pointer verification and two cross-domain introspections. This explains why the overhead measurements in Table 18 are much lower than those in Table 27. For example, the *cp* benchmark has the highest performance penalty in Table 27, because *cp* demands frequent disk operations and accordingly frequent callbacks to *ide_intr* (42 times per second), and we know that the verification of *ide_intr* is very complicated. (3) The coarse-grain locking of memory pages by our defense causes a large number of legitimate memory write operations to be emulated in software, which adds more performance overhead. This is consistent with the results in Table 27. For example, the *cp* benchmark sees a 25 times slowdown with the memory protection turned on, but once the memory protection is turned off, the slowdown drops to 14 times.

Table 27: Overhead of the K-Queue Checker

	cat	ccrypt	gzip	cp	make
Original	13.06 ±1.61	3.05 ±0.27	5.35 ±0.10	50.00 ±4.35	128.78 ±3.91
K-Queue-aware, no lock	16.66 ±0.38	3.89 ±0.50	5.93 ±0.12	749.99 ±42.97	175.92 ±5.24
Overhead	28%	28%	11%	1,400%	37%
K-Queue-aware, lock	17.15 ±1.68	4.28 ±0.99	6.12 ±0.59	1,309.73 ±100.03	210.44 ±25.16
Overhead	31%	40%	14%	2,519%	63%

Complexity of the K-Queue Checker

To further understand the performance overhead, we define and measure two complexity metrics of the K-Queue Checker program: layer and fanout. First we give an informal definition of the *layer* of verification: each layer is associated with a function pointer. The checker starts in layer 1, where the associated function pointer is the top-level K-Queue function pointers embedded in the K-Queue data structures. At layer i the value of the function pointer is first verified against a white list; if the verification is successful then the integrity of the target function itself needs to be verified, which may require the checking of a new function pointer. In this case the verification enters a new layer i + 1. When the verification for a target function completes, the checker returns to the previous layer (i.e., layer i). We also define the *fanout* of a function as the number of function pointers whose integrity needs to be checked for that function.

For our K-Queue Checker program, the maximum layer during the verification of the IRQ action queue is seven, which happens when the top-level call back function is *ide_intr* (linux-2.4.32/drivers/ide/ide-io.c). And during the checking of the IRQ action queue, the maximum fanout is 15 (for the function *idedisk_error* in linux-2.4.32/drivers/ide/ide-disk.c).

3.5. Related Work

Defenses against Stealthy Attacks. Defense techniques against attacks that change kernel code include Tripwire [32], a file system integrity checker, IMA [48], a load-time kernel and application code integrity checker, and Copilot [40] and Pioneer [54], runtime kernel code checkers. Representative defenses for attacks that change kernel data include CFI [1] and SBCFI [42].

To the best of our knowledge, there have been few concrete instances of attacks that do not change kernel code or data, but insert transient execution units into a schedulable queue. The Blue Chicken [46] uses a KTIMER in a Windows Vista kernel to

reinstall the Blue Pill hypervisor, which is an example of how a kernel-level malware can use soft timer to maintain control of the victim platform. The "cheat" attack described in [56] may be regarded as a user-level example, since it uses the to-be-scheduled task queue. Known malware detection methods have difficulties with transient kernel control flow attacks. For example, signatures of known malicious STIRs can be created by reverse engineering the malware. This approach suffers from the same problems seen in the anti-virus community. Specifically, they are unable to detect or prevent zero-day attacks, and the process of finding appropriate signatures is difficult and error prone. For these reasons, signature checking alone is insufficient to mitigate this threat.

Another possible approach for detecting these attacks is to extend control flow integrity techniques such as SBCFI [42] and CFI [1]. SBCFI is a checker for persistent kernel control flow attacks. It starts by looking at kernel global variables and performs a garbage-collection style traversal of kernel data structures to verify that all of the function pointers target trusted addresses in the kernel. SBCFI can potentially catch a type 1 malicious STIR, since the function pointer targets can be validated when SBCFI scans the kernel variables. However, SBCFI can not detect type 2 STIRs because it does not follow the uninterpreted data field included as part of the callback: it is not defined as a pointer type. The definition of data is intended to allow maximum flexibility for different call back functions. In order to make SBCFI work on type 2 STIRs, accurate type information for the data field in each call back function must be added, which would require a static analysis of all STIR callback functions. Such an approach would then be similar to our STIR Analyzer (Section 3.3.2.2).

A more general approach, CFI [1] uses inline reference monitors [22] to compare the dynamic execution flow of a program against a statically computed control flow graph (CFG). CFI is a general framework that can be instantiated into an alternative implementation of the STIR Checker, however the exact checks that must be performed

against the STIR callback functions would still need to be constructed by tools such as the STIR Analyzer.

Secure Kernel Extensions. K-Queue driven malware exploits an interface exposed by the core kernel to its extensions. There has been some effort to achieve finer-grained divisions within a monolithic kernel, with the goal of improving security. For example, Palladium [15] demotes the privileges of the kernel extensions so that misbehaving or malicious extensions cannot harm the core portion of the kernel. However, such approaches can only prevent the malicious extensions from corrupting the core kernel, but cannot prevent sensitive information stealing (section 3.2.3.1) and denial of service attacks (Section 3.2.3.2).

Points-to analysis. There has been a large body of research work on points-to analysis. However, this problem has not been completely solved yet because in general points-to analysis is undecidable. As a result, a large number of approximation algorithms have been proposed, with various trade-off between efficiency and precision. Interested readers are referred to a survey by Hind [27]. Our K-Queue analyzer provides specialized points-to analysis algorithms (e.g., direct assignment collector and parameter collector) for the Linux kernel, which are not intended for a general solution to the points-to analysis problem.

Applications of Static Analysis in Systems and Security Research. In recent years, static analysis of software has been used for many purposes including deriving application behavior models for intrusion detection systems [60], building control flow graphs of an application [1], and determining type and global variable information for the Linux kernel [42]. This technique has also been applied to finding bugs in both kernel and application code [13, 21, and 35]. In this dissertation, we add one more use case by applying this technique to derive summary signatures for legitimate K-Queue requests.

3.6. Discussion

The K-Queue case study demonstrates another example of para-transactional invariants (PTIs): that the control flow resulting from a legitimate K-Queue callback function should always target trusted code of the kernel. The scope of such K-Queue invariants is from the verification of the K-Queue request to the end of the execution of the callback function.

CHAPTER 4

CONCLUSION AND FUTURE WORK

Attacks exploiting inherent shortcomings of today's operating systems (e.g., missing transactional support) are the most difficult to defend because they are often stealthy and non-obtrusive. Yet such attacks are on the rise to become a major security threat. This dissertation argues that we can defend these attacks by identifying and guarding specific correctness models. To exemplify our approach, we solve two classes of important security problems: TOCTTOU and K-Queue. TOCTTOU is a file-based race condition that represents a high security risk due to the wide-scale deployment of multiprocessors, and K-Queue driven attacks misuse the schedulable queues interface to inject transient and malicious control flows in the victim kernel and can evade the detection of state-of-the-art kernel integrity checkers. We propose the CUU model that is capable of enumerating all potential TOCTTOU vulnerabilities and our CUU-guided defense mechanism and implementation are also complete. We apply automated static analysis and code generation that infer the correct usage model of the K-Queues (called summary signatures) and generate the corresponding guards that enforce the usage model at runtime. Our work suggests that improving the correctness of operating systems enable powerful defense against certain classes of malicious attacks, and that automated software engineering techniques are very helpful in increasing the productivity of such efforts.

4.1. Future Work

Reduce the runtime overhead of the K-Queue Checker

The current K-Queue Checker incurs unacceptable overhead in some cases. But the overhead can be reduced in at least two ways. First, we can optimize the Checker software to verify different function pointers in the same data structure together and

avoid redundant pointer verifications. This is because the function pointers often cluster in a small number of data structures (such as *hwif_s* and *tty_driver*), and different target functions often require the same function pointer to be verified. Second, we can employ architectural support for fine-grain memory protection (e.g., Mondrian [64]) once they are available, to reduce the overhead of our defense against the TOCTTOU attacks on our K-Queue Checkers.

Support for polymorphic or obfuscated code. Our current design assumes that there is a fixed memory layout for the guest kernel; therefore, it cannot support obfuscated guest kernels which apply techniques such as address space randomization. Under the current assumption, the runtime addresses of the kernel functions are known in advance and therefore can be built into the K-Queue checker. However, when the address space of the guest kernel is obfuscated, the runtime addresses of the kernel functions can not be known in advance. In order to support such guest kernels, we need to generate checker code that refers to kernel functions by name rather than address, and we need to add a runtime service in the guest kernel that maps function names to their actual addresses.

Support for loadable kernel modules. Due to a constraint imposed by the CIL merger, our current implementation of the K-Queue defense does not support loadable kernel modules. We plan to make CIL merger run over individual kernel modules, so that we can capture K-Queue requests made by them. These results can then be merged into the result database for the core kernel.

APPENDIX A

Table 28: Exploitable TOCTTOU Pairs in Linux

Invariant	TOCTTOU Pairs
resolve(*f*) = ∅	<stat, creat> <stat, open> <stat, mknod> <stat, rename> <access, creat> <access, open> <access, mknod> <access, rename> <unlink, creat> <unlink, open> <unlink, mknod> <unlink, rename> <rename, creat> <rename, open> <rename, mknod> <rename, rename> <stat, mkdir> <stat, rename> <access, mkdir> <access, rename> <rmdir, mkdir> <rmdir, rename> <rename, mkdir> <rename, rename> <stat, link> <stat, symlink> <stat, rename> <access, link> <access, symlink> <access, rename> <unlink, link> <unlink, symlink> <unlink, rename> <rename, link> <rename, symlink> <rename, rename>
resolve(*f*) = *b*	<stat, chmod> <stat, chown> <stat, truncate> <stat, utime> <stat, open> <stat, execve> <access, chmod> <access, chown> <access, truncate> <access, utime> <access, open> <access, execve> <creat, chmod> <creat, chown> <creat, truncate> <creat, utime> <creat, open> <creat, execve> <open, chmod> <open, chown> <open, truncate> <open, utime> <open, open> <open, execve> <mknod, chmod> <mknod, chown> <mknod, truncate> <mknod, utime> <mknod, open> <mknod, execve> <rename, chmod> <rename, chown> <rename, truncate> <rename, utime> <rename, open> <rename, execve> <link, chmod> <link, chown> <link, truncate> <link, utime> <link, open> <link, execve> <symlink, chmod> <symlink, chown> <symlink, truncate> <symlink, utime> <symlink, open> <symlink, execve> <rename, chmod> <rename, chown> <rename, truncate> <rename, utime> <rename, open> <rename, execve> <chmod, chmod> <chmod, chown> <chmod, truncate> <chmod, utime> <chmod, open> <chmod, execve> <chown, chmod> <chown, chown> <chown, truncate> <chown, utime> <chown, open> <chown, execve> <truncate, chmod> <truncate, chown> <truncate, truncate> <truncate, utime> <truncate, open> <truncate, execve> <utime, chmod> <utime, chown> <utime, truncate> <utime, utime> <utime, open> <utime, execve> <open, chmod> <open, chown> <open, truncate> <open, utime> <open, open> <open, execve> <execve, chmod> <execve, chown> <execve, truncate> <execve, utime> <execve, open> <execve, execve> <stat, chmod> <stat, chown> <stat, utime> <stat, mount> <stat, chdir> <stat, chroot> <stat, pivot_root> <access, chmod> <access, chown> <access, utime> <access, mount> <access, chdir> <access, chroot> <access, pivot_root> <mkdir, chmod> <mkdir, chown> <mkdir, utime> <mkdir, mount> <mkdir, chdir> <mkdir, chroot> <mkdir, pivot_root> <rename, chmod> <rename, chown> <rename, utime> <rename, mount> <rename, chdir> <rename, chroot> <rename, pivot_root> <link, chmod> <link, chown> <link, utime> <link, mount> <link, chdir> <link, chroot> <link, pivot_root> <symlink, chmod> <symlink, chown> <symlink, utime> <symlink, mount> <symlink, chdir> <symlink, chroot> <symlink, pivot_root> <rename, chmod> <rename, chown> <rename, utime> <rename, mount> <rename, chdir> <rename, chroot> <rename, pivot_root> <chmod, chmod> <chmod, chown> <chmod, utime> <chmod, mount> <chmod, chdir> <chmod, chroot> <chmod, pivot_root> <chown, chmod> <chown, chown> <chown, utime> <chown, mount> <chown, chdir> <chown, chroot> <chown, pivot_root> <utime, chmod> <utime, chown> <utime, utime> <utime, mount> <utime, chdir> <utime, chroot> <utime, pivot_root> <mount, chmod> <mount, chown> <mount, utime> <mount, mount> <mount, chdir> <mount, chroot> <mount, pivot_root> <chdir, chmod> <chdir, chown> <chdir, utime> <chdir, mount> <chdir, chdir> <chdir, chroot> <chdir, pivot_root> <chroot, chmod> <chroot, chown> <chroot, utime> <chroot, mount> <chroot, chdir> <chroot, chroot> <chroot, pivot_root> <pivot_root, chmod> <pivot_root, chown> <pivot_root, utime> <pivot_root, mount> <pivot_root, chdir> <pivot_root, chroot> <pivot_root, pivot_root>

REFERENCES

1 Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (Nov. 2005). Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*.

2 Abbott, R. P., Chin, J.S., Donnelley, J.E., Konigs-ford, W.L., Tokubo, S., and Webb, D.A. (April 1976). Security analysis and enhancements of computer operating systems. *NBSIR 76-1041*, Institute of Computer Sciences and Technology, National Bureau of Standards.

3 Arbaugh, W. A., Farber, D. J., and Smith, J. M. (May 1997). A secure and reliable bootstrap architecture. *IEEE Symposium on Security and Privacy*, Oakland, CA.

4 Barham, P., Dragovic, B., Fraser, K., et al. (Oct. 2003). Xen and the art of virtualization. *ACM Symposium on Op-erating Systems Principles (SOSP)*, Bolton Landing, NY.

5 Bisbey, R., Hollingsworth, D. (May 1978). Protection Analysis Project Final Report. *ISI/RR-78-13, DTIC AD A056816*, USC/Information Sciences Institute.

6 Bishop, M. and Dilger, M. (1996). Checking for race conditions in file accesses. *Computing Systems,* 9 (2), 131–152.

7 Bishop, M. (September 1995). Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux. *Technical Report 95-8*, Department of Computer Science, University of California at Davis.

8 Borisov, N., Johnson, R., Sastry, N. and Wagner, D. (2005). Fixing races for fun and profit: how to abuse atime. In *Proceedings of the 14th USENIX Security Symposium*.

9 Bovet, D., Cesati, M. (Dec. 2002). *Understanding the Linux Kernel*, Second Edition. O'Reilly.

10 Brumley, D. (Nov. 16, 1999). Invisible intruders: rootkits in practice. ;login:. http://www.usenix.org /publications/login/1999-9/features/rootkits.html.

11 BUGTRAQ archives. Retrieved on May 30, 2009, from http://securepoint.com/lists/html/bugtraq/

12 BUGTRAQ report RHSA-2000:077-03: esound contains a race condition. Retrieved on June 16, 2009, from http://seclists.org/bugtraq/2000/Oct/0105.html.

13 Chen, H., Wagner, D. (November 2002). MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC.

14 Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. (August 2005). Non-Control-Data Attacks Are Realistic Threats. *USENIX Security Symposium*, Baltimore, MD.

15 Chiueh, T.-c., Venkitachalam, G., and Pradhan, P. (Dec. 1999). Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *17th ACM Symposium on Operating Systems Principles (SOSP),* Charleston, SC.

16 Consel, C., Danvy, O. (Jan. 1993). Tutorial notes on partial evaluation. *20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Charleston, SC.

17 Cowan, C., Pu, C., Maier, D., et al. (Jan. 1998). StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. *7th USENIX Security Symposium*, San Antonio, TX.

18 Cowan, C., Beattie, S., Wright, C., and Kroah-Hartman, G. (August 2001). RaceGuard: Kernel Protection from Temporary File Race Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington DC.

19 Dean, D., and Hu, A. (August 2004). Fixing Races for Fun and Profit: How to use access(2). In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA.

20 Engler, D., Chelf, B., Chou, A., and Hallem, S. (September 2000). Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*.

21 Engler, D., and Ashcraft, K. (2003). RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*.

22 Erlingsson, U., Schneider, F. IRM enforcement of Java stack inspection. *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

23 Espiner, Tom. (November 2007). Microsoft exec calls XP hack 'frightenin'. In *CNET News*. Retrieved on June 22, 2009, from http://news.cnet.com/Microsoft-exec-calls-XP-hack-frightening/2100-7349_3-6218238.html.

24 F-Secure Weblog. Calculating the Size of the Downadup Outbreak. Retrieved on June 22, 2009, from http://www.f-secure.com/weblog/archives/00001584.html.

25 gedit. http://projects.gnome.org/gedit/

26 Harel, D. (June 1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

27 Hind, Michael. (2001). Pointer analysis: haven't we solved this problem yet? *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 54-61

28 Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N. and West, M. J. (February 1988). Scale and performance in a distributed file system, *Transactions on Computer Systems, vol. 6*, pp. 51-81.

29 Hultquist, S. (Apr. 2007). Rootkits: The next big enterprise threat? http://www.cso.com.au/article/184125/rootkits_next_big_enterprise_threat, Retrieved on July 3, 2009.

30 IEEE Std 1003.1, 2004 Edition. Retrieved on July 9, 2009, from http://www.unix.org/single_unix_specification

31 Kiczales, G., Lamping, J., et al. (1997). Aspect-Oriented Programming. *Proceedings European Conference on Object-Oriented Programming*.

32 Kim, G. H. and Spafford, E.H. (1994). The design and implementation of Tripwire: A file system integrity checker. *2nd ACM Conference on Computer and Communications Security (CCS)*.

33 Ko, C., Fink, G., Levitt, K. (Dec. 1994). Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. *Proceedings of the 10th Annual Computer Security Applications Conference*, page 134-144.

34 Lhee, K., and Chapin, S. J. (2005). Detection of File-Based Race Conditions, *International Journal of Information Security*.

35 Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., and Zhou, Y. (Oct. 2007). MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA.

36 McCarthy, D. R., Dayal, U. (1989). The Architecture of an Active Data Base Management System. *SIGMOD Conference* 1989: 215-224

37 Necula, G. C., McPeak, S., Rahul, S. P. and Weimer, W. (Apr. 2002). CIL: Intermediate language and tools for analysis and transformation of C programs. *Conference on Compiler Construction (CC)*, Grenoble, France.

38 Payne, B. D., Carbone, M., and Lee, W. (Dec. 2007). Secure and flexible monitoring of virtual machines. *23rd Annual Computer Security Applications Conference (ACSAC)*. Miami, FL.

39 Payne, B. D., Carbone, M., Sharif, M., and Lee, W. (May 2008). Lares: An architecture for secure active monitoring using virtualization. *IEEE Symposium on Security and Privacy*, Oakland, CA.

40 Petroni, N., Fraser, T., Molina, J., and Arbaugh, W.A. (Aug. 2004). Copilot—a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium.*

41 Petroni, N., Fraser, T., Walters, A., Arbaugh, W. A. (2006). An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. *15th USENIX Security Symposium*.

42 Petroni, N. and Hicks, M. (Oct. 2007). Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*.

43 PostMark benchmark. http://www.netapp.com/tech_library/3022.html

44 Pu, C. and Wei, J. (March 2006). A methodical defense against TOCTTOU attacks: the EDGI approach. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*.

45 rd. (June 2002). Writing Linux kernel keylogger. *Phrack Vol. 11, Issue 59*.

46 Rutkowska, J. and Tereshkin, A. IsGameOver() Anyone? *Black Hat USA 2007*. Retrieved on June 16, 2009, from http://bluepillproject.org/stuff/IsGameOver.ppt

47 Saks, D. (June 2009). Padding and rearranging structure numbers. *Design Perspectives Blog*. http://www.techonlineindia.com/blog/news/archives/2009/06/padding_and_rea.html;jsessionid=C1F2IHTEKRR4IQSNDLPCKHSCJUNN2JVN?loc=design_perspectives

48 Sailer, R., Zhang, X., Jaeger, T., and Doorn, L. V. (Aug. 2004). Design and implementation of a TCG-based integrity measurement architecture. *13th USENIX Security Symposium*, San Diego, CA.

49 Saltzer, J. H., and Schroeder, M.D. (Sept. 1975). The protection of information in computer systems. *Proceedings of the IEEE 63(9)*.

50 Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (November 1997). Eraser: a dynamic data race detector for multithreaded programs. In *ACM Transactions on Computer Systems*, 15 (4).

51 Schwarz, B., Chen, H., Wagner, D., Morrison, G., West, J., Lin, J., and Tu, W. (December 2005). Model checking an entire Linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*.

52 Security holes in logwatch. Retrieved on May 31, 2009, from http://xforce.iss.net/xforce/xfdb/8652.

53 Seshadri, A., Luk, M., Qu, N., Perrig, A. (2007). SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

54 Seshadri, A., Luk, M., Shi, E., et al. (Oct. 2005). Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. *ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom.

55 Solar Designer. (Aug. 1997). Getting around non-executable stack (and fix). *Bugtraq*.

56 Tsafrir, D., Etsion, Y., and Feitelson, D.G. (Aug. 2007). Secretly monopolizing the CPU without superuser privileges. *16th USENIX Security Symposium*, Boston, MA.

57 Tsyrklevich, E., and Yee, B. (2003). Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*.

58 United States Computer Emergency Readiness Team. Retrieved on October 22, 2008, from http://www.kb.cert.org/vuls/

59 U.S. Department of Energy Computer Incident Advisory Capability. http://www.ciac.org/ciac/

60 Wagner, D., and Dean, D. (May 2001). Intrusion detection via static analysis. *IEEE Symposium on Security and Privacy*, Oakland, CA.

61 Wei, J. and Pu, C. (December 2005). TOCTTOU vulnerabilities in UNIX-style file systems: an anatomical study. In *Proceedings of the 4th Usenix Conference on File and Storage Technologies (FAST'05)*.

62 Wei, J. and Pu, C. (2007). Multiprocessors may reduce system dependability under file-based race condition attacks. In *Proceedings of the 37th IFIP/IEEE International Conference on Dependable Systems and Networks (DSN 2007)*.

63 Wei, J., Payne, B. D., Giffin, J., and Pu, C. (December 2008). Soft-timer driven transient kernel control flow attacks and defense. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008)*.

64 Witchel, E., Cates, J., and Asanovic, K. (October 2002). Mondrian memory protection. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*.