Improving System Security and Reliability with OS Help

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Noam Shalev

Submitted to the Senate of the Technion – Israel Institute of Technology

Av 5778 Haifa July 2018

The research thesis was done under the supervision of Prof. Idit Keidar in the Viterbi Faculty of Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel.

The generous financial help of the Technion, the Meyer Foundation, the Leonard and Diane Sherman Family Foundation, Intel Israel, the Hasso-Plattner Institute, the Hiroshi Fujiwara Cyber Security Research Center and the Israel Cyber Bureau is gratefully acknowledged.

This research thesis is dedicated my parents. For their endless love, devotion, sacrifice, support and encouragement.

Acknowledgments

I would like to sincerely thank the many people who made this research possible.

- I was fortunate to have Prof. Idit Keidar as my advisor. I wish to thank Idit for believing in me, for teaching me how to conduct professional research, for always saying the right words at the right time, and for giving me the freedom to explore while providing guidance when needed.

  I thank you for your support and encouragement at hard times. This period of my life was an amazing experience that will always be with me, and I deeply thank you for giving me the opportunity and for introducing me to the world of research.

- I heartily thank Dr. Yaron Weinsberg for his mentorship, for his endless ideas, for our fruitful collaboration and for numerous meetings and priceless advises. I have no doubt that our best collaborations are yet to come.

- I wish to thank Prof. Mark Silberstein and Prof. Uri Weiser for insightful feedbacks, important advises, connections and help.

- I with to thank the researchers with whom I collaborated: From the Technion, I thank Dr. Tomer Y. Morad, Dr. Nimrod Partush and Prof. Avinoam Kolodny; from Microsoft Research, I thank Dr. Ofer Dekel and Chuck Jacobs; from IBM Research, I thank Dr. Dalit Naor, and Dr. Yosef Moatti; and from Yahoo! Labs, I thank Dr. Edward Bortnikov, Dr. Iftah Gamzu and Dr. Lev Korostyshevsky. I learned a lot from each one of you, and I hope to collaborate again in the future.

- I would like to thank our research group's team members for the fun times together. I enjoyed our group meetings, the lunch times talks, and the professional discussions. Your feedbacks, advices, and support were always valuable. In particular, I would like to thank Dr. Dmitry Perelman, Dr. Naama Kraus and Prof. Ittay Eyal for many discussions and encouragements.

- I thank my students Eran Harpaz and Hagar Porat for their hard work. Furthermore, I thank the rest of my undergrad students that contributed to my research in their final projects: Ron Blechner, Merav Natanson, Guy Barshatski, Bassam Yassin, and Hezi Banda.

- I wish to thank Orly Babad-Tamir and Danit Cohen for their administrative support. Thank you for efficiently assisting in any need that arises, and always doing it with a smile.

- I wish to express my deepest gratitude to my parents, Dorit and Haim, for their continuous support and encouragement. Thanks for teaching me to always strive, excel and never give up.

- I thank my sisters, Racheli and Yael, who supported me in a way that only sisters can.
- I thank my sister Hodaya, who thought me more than anyone in the world, about responsibility, independence, tolerance, acceptance, happiness and proportions.
- Last though most important – I thank Shannah Zilcha, my fiancée and my best friend, for all the love and support. I am more than fortunate to have you in my life me and I wish us many happy years together.

List of Publications

1. Tomer Y. Morad, Noam Shalev, Idit Keidar, Avinoam Kolodny, Uri Weiser. "EFS: Energy-Friendly Scheduler for Memory Bandwidth Constrained Systems", *2016 Journal of Parallel and Distributed Computing (JPDC 2016).*[1]

2. Noam Shalev, Eran Harpaz, Hagar Porat, Idit Keidar, Yaron Weinsberg. " CSR: Core Surprise Removal in Commodity Operating Systems", *The 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016).*

3. Noam Shalev, Idit Keidar, Yaron Weinsberg, Yosef Moatti, Elad Ben-Yehuda. "WatchIT: Who Watches Your IT Guy?", *The 26th ACM Symposium on Operating Systems Principles (SOSP 2017).*

4. Noam Shalev, Nimrod Partush. "Binary Similarity Using Machine Learning", *The 13th Workshop on Programming Languages and Analysis for Security (PLAS 2018).*

[1] Not included in this thesis.

Table of Contents

List of Figures

List of Tables

Abstract

In this thesis we research key security and reliability challenges in modern computer systems that arise due to today's growth in cyber security threats and economies of scale. We tackle three problems: securing organizations from privileged insiders, detecting similarities in binary codes, and providing tolerance to commodity systems in face of hardware faults. Our proposed solutions include new operating system kernel abstractions and mechanisms, unique use of existing software and hardware components, and machine learning aided strategies.

We present WatchIT, a strategy that constrains IT personnel's view of the system and monitors their actions, thus allowing organizations to mitigate the security threats that their system administrators pose. We propose Lava, a tool for detecting similarities between binaries that were compiled differently. By drawing concepts from image processing and using machine learning-based predictors, we provide a scalable solution that can take a vulnerable code snippet and look for that vulnerability in other platforms in binary form. We introduce CSR, a strategy for overcoming hardware faults that happen in the CPU. By exploiting hardware transactional memory, our mechanism can tolerate faults that happen in any execution context, and keep the system alive and running while virtually incurring no overhead.

We implement our approaches and evaluate them by empirically experimenting on real systems and by conducting a case study in a real production environment. The results show that our proposed solutions are effective and significantly improve state-of-the-art systems and tools.

List of Acronyms

AR	Action Required
CSR	Core Surprise Removal
CPU	Central Processing Unit
FDU	Failure Detection Unit
FS	File-System
FUSE	File-system in user space
HPC	High Performance Computers
HTM	Hardware Transactional Memory
ITFS	IT File-System
IPI	Inter-Processor Interrupt
ML	Machine Learning
NN	Neural Network
OS	Operating System
proc2vec	Procedure to Vector
RCU	Read-Copy Update
RWQ	Recovery Work Queue
TCB	Trusted Computing Base
TLB	Translation Lookaside Buffer
TM	Transactional Memory
TSX	Transactional Synchronization Extension
VFS	Virtual Filesystem Switch
VM	Virtual Mahicne

List of Symbols

P	a project in a training set
p_i	binary procedure i
L_1	Size of first neural network hidden layer
L_2	Size of second neural network hidden layer
C_F	Core that fails
C_D	Core that performs recovery process
C'_D	Core that performs cascading failure recovery process
T_e	Time period for executing recovery process
T_q	Time period for queuing recovery work
t_{start}	Starting time of a recovery process
t_{acq}	Time in which the FDU receives an ack
t_{end}	End time of a recovery process

Chapter 1

Introduction

Recent developments in computer systems have introduced new security and reliability challenges that have not been addressed before. In this thesis, we research three of these challenges: mitigating privileged insider threats within organizations, detecting similarities between binary codes in a scalable way, and providing hardware fault tolerance to commodity systems.

We begin in Chapter 2 by introducing *WatchIT*, a new approach for protecting organizations from privileged insider threats. The permission mechanisms in commodity operating systems are coarse-grained, and as a result, system administrators have unlimited access to system resources. As the Snowden case highlighted, these permissions can be exploited to steal valuable personal, classified, or commercial data. This problem is exacerbated when a third party administers the system. For example, a bank outsourcing its IT would not want to allow administrators access to the actual financial data. Though we are entering an era of big data and information security, current systems still do not propose any remedy for such threats.

For answering this need, we introduce the abstraction of *perforated containers* – while regular Linux containers are too restrictive to be used by system administrators, by "punching holes" in them, we strike a balance between information security and required administrative needs. Following the principle of least privilege, our system predicts which system resources should be accessible for handling each IT issue, creates a perforated container with the corresponding isolation, and deploys it as needed for fixing the problem. Under this approach, the system administrator retains superuser privileges, however only within the perforated container limits. We further provide means for the administrator to bypass the isolation, but such operations are monitored and logged for later analysis and anomaly detection. We provide a proof-of-concept implementation of our strategy, which includes software for deploying perforated containers, monitoring mechanisms, and changes to the Linux kernel. Moreover, we analyze the threats that our system faces and detail the security measures we take in order to mitigate them. We further present a case

study conducted on the IT database of IBM Research in Israel, showing that our approach is feasible.

In Chapter 3, we propose a novel technique for detecting similarities between code binaries, which achieves both high accuracy and peerless throughput. As the Heartbleed case demonstrated, numerous products can be affected by a single vulnerability, including operating system distributions. In such scenarios, a security-aware company would want to use a sample of the vulnerable product (in binary form) to search for the vulnerability across all the software installed in its system. Unfortunately, though there are tools for performing binary similarity search, none of them is fast and accurate enough for dealing with the huge code corpora that both individuals and enterprises usually possess.

For tackling this challenge, we propose a novel methodology for detecting similarity between code binaries. To this end, we employ machine learning alongside similarity by composition. We present the *proc2vec* algorithm, and use it to decompose binary code into smaller comparable fragments, transform these fragments to vectors, and build machine learning-based predictors for detecting similarity between vectors that originate from similar procedures. Our technique is able to detect similar pieces of codes even when they have been compiled using different compilers, in different versions and for different architectures. We present a tool called *Lava* implementing this technique, and evaluate it by searching similarities in open source projects that we crawl from the world-wide-web. Our results show that Lava performs 250X faster than state-of-the-art algorithms without harming accuracy.

In Chapter 4 we explore the uncharted field of reliability in face of hardware faults in commodity systems. With the shrinking of transistor size and voltage supply, the probability of physical flaws, induced by thermal changes or variability in the manufacturing process significantly rises. Thus, the chances for soft errors or permanent faults during system runtime are growing and become more evident. Nevertheless, current commodity operating systems are not designed to survive any hardware fault.

In light of the above, we present *CSR*, a strategy for recovery from unexpected permanent processor faults in commodity operating systems. Our approach overcomes surprise removal of faulty cores. It is scalable, incurs low overhead, and is designed to integrate into modern operating systems. When a core fails in user mode, CSR terminates the process executing on that core and migrates the remaining processes in its run-queue to other cores. We further show how hardware transactional memory may be used to overcome failures in critical kernel code and to assist in tolerating cascading core failures. Moreover, we elaborate how CSR can be leveraged to survive chip-originated soft errors. We have implemented our approach in the Linux kernel, using Haswell's Transactional Synchronization Extension, and tested it thoroughly on real Linux systems.

Finally, in Chapter 5 we discuss future work and conclude.

Chapter 2

WatchIT: Who Watches Your IT Guy?

According to IBM's annual cyber-security report [59], 44% of cyber attacks detected last year were carried out by malicious insiders; this positions malicious insiders as one of today's biggest security threats. Among a company's employees, IT workers pose the greatest risk to the organizational information security, since a system administrator typically has unlimited permissions and access to system resources. The famous Edward Snowden [135] case, among others [75, 47], has again brought to public attention the danger arising from giving bulk permissions to system administrators, which can be abused by greedy or disgruntled employees to steal valuable, classified, or commercial data.

The problem is aggravated in cases where a third party handles the organizational IT. For instance, a health clinic may acquire cloud storage services with a maintenance contract. In such cases, the service provider's IT personnel has access to the clinic's information, which is to be kept confidential by law.

Ideally, IT workers should have access only to resources they actually need, while any mechanism for isolating them from other resources should not interfere with their work. The challenge in creating appropriate isolation for system administrators lies in the coarse-grained permission model employed in commodity operating systems like Linux. Better defense against privileged insider attacks therefore requires finer-grained control over privileges.

We introduce *WatchIT* which implements such fine-grained control in the context of containers, a virtualization method already widely deployed in cloud systems and development environments today. A container envelops a set of processes and gives them an isolated view of the operating environment. On the face of it, the isolation that containers provide goes against the very nature of system administration – indeed, IT personnel get permissions for a reason.

Perhaps counter-intuitively, we propose in this work to exploit containers for system administration. Our idea is to put holes in the isolation of traditional containers in order to

create a middle-ground that is useful for administrators and yet controlled. We introduce the abstraction of a *perforated container* – a container that may share system resources with the host, under logging and monitoring. This perforated container is essentially a sandbox that constrains IT personnel's view of the system. It provides a way to monitor their actions, yet preserves their superuser privileges within specified boundaries.

The solution we devise is based on three principles: (1) isolating the system administrator from resources that are expected to be irrelevant (using a perforated container abstraction), (2) enabling her to perform actions beyond the isolation boundaries under monitoring and logging, and (3) optionally monitoring the allowed operations executed inside the perforated container. To support the second, we introduce a *permission broker* – a software service that can change the boundaries of the perforated container and perform operations on the administrator's behalf.

Current container software such as Docker [87] and others [81, 102, 97] does not provide the required flexibility for configuring and deploying containers for administration. For example, a container cannot share the host's view of the filesystem or provide access to certain file types while excluding others.

We implement *ContainIT*, dedicated container software for building perforated containers. ContainIT can share the host's filesystem view as well as other namespaces (e.g., processes or network). It can further log and monitor network activity and filesystem operations. Moreover, it allows for login of privileged users but can block access to specific files even if the contained administrator can see that they exist. Via the permission broker, a ContainIT perforated container can obtain additional filesystem and network views on-demand, without restarting. Furthermore, ContainIT is able to utilize a new namespace we introduce, *exclusion namespace*, which excludes access to parts of the filesystem for the processes it contains, even if its associated container shares the host's filesystem table.

We complement our container software with a framework that orchestrates our approach, and is deployed within the trusted computing base of each organizational machine. The framework includes a permission broker and a software system that can receive a free-text IT problem description, classify it, and deploy a corresponding perforated container at the target machines.

To illustrate the feasibility of our approach, we present a case study conducted in the IT department of IBM Research, Israel. We used the department's database, which includes tens of thousands of user reported *tickets* and dozens of maintenance scripts; we further collected hundreds of IT tickets in real time during a three-month evaluation period. The case study shows that IT tasks can indeed be divided into categories and classified using common classification algorithms. By adjusting a perforated container for each ticket class, we were able to accurately capture the needs of 92% of the 398 cases observed during the evaluation period. We used the permission broker to complete the remaining 8% of the cases. Our results showed that our solution denied full filesystem view in 62% of the cases, and isolated the network view in 98% of the cases, while all

filesystem operations and network traffic were monitored. Overall, we compartmentalized IT personnel from resources that were indeed proved to be irrelevant to the ticket solving in all of the observed cases, thus reducing the IT attack surface. Following the success of the case study, IBM is working to integrate a WatchIT-based solution in its products whose customers are required by law to preserve their clients' information confidentiality.

Our contributions in this chapter are as follows:

1. We devise a novel container-based approach to protect organizations from their system administrators.

2. We provide a proof-of-concept implementation of our approach, which includes new container software, a framework for deploying perforated containers, and required changes to the Linux kernel.

3. We present a real-world case study, based on the IT department database and workflow of IBM Research, Israel.

The rest of this chapter is organized as follows: We state our threat model and detail the current state-of-the-practice in Sections 2.1 and 2.2, respectively. We describe the main concepts of our approach in Section 2.3, and give a detailed overview of WatchIT and its implementation in Section 2.4. Next, in Section 2.5 we discuss ways to circumvent WatchIT and the measures we take to mitigate them. In Section 2.6 we present a case study, performed using a real IT department, through which we evaluate our approach. Finally, in Sections 2.7 and 2.8 we discuss related work and conclude.

2.1 TCB and Threat Model

We utilize standard techniques and build upon a Trusted Computing Base (TCB) to boot the WatchIT framework into a trusted initial state. We consider a TCB that provides a secure environment, which includes the system hardware, the operating system with all its built-in security controls, system drivers, system services, and WatchIT components. An example commodity product that provides such TCB functionality is BitLocker [74], which validates the integrity of boot and system files before decrypting a protected volume.

Given such a TCB, our threat model is a single rogue IT employee who wishes to access confidential data, directly or indirectly by installing malicious software, that is considered irrelevant for the specific ticket context. We assume a workflow in which clients report trouble tickets which are later assigned to specific IT personnel, based on expertise or required permissions. The assignment creates a certificate that allows the designated system administrator to login into a perforated container deployed on the target machine for a limited time. System administrators do not otherwise have access to the machines in question, and they cannot create trouble tickets on their own initiative.

Once an administrator is logged into a perforated container, she can modify all system resources to which she is exposed, as long as she does not change the TCB. Our system

blocks all actions that may change the TCB signature; thus, an IT person cannot change the OS kernel, install unauthorized drivers or kernel modules, or install non-certified services. These special actions require escalation, provided by the permission broker, and thus allow WatchIT to audit the change and make sure it is signed by the organizational policy system. Note that these assumptions are supported by the case study presented in Section 2.6, as less than 1% of the tickets we encountered involved driver updates.

2.2 State-of-the-Practice

In Section 2.2.1 we provide the motivation for our work by discussing the state-of-the-practice in IT workflow and pinpointing the danger that IT personnel poses to organizational information security via three use-cases. Next, in Section 2.2.2, we give essential background on container technology.

2.2.1 IT Vulnerability

IT personnel usually have superuser privileges on the machines in their responsibility domain. Indeed, superuser permissions are essential for system administration. However, due to the coarse-grained permission granularity in traditional Linux systems, an IT person practically has access to any file or configuration stored in these machines. We identify three main scenarios in which IT employees can take advantage of their permissions to steal classified data.

Daily IT Support IT workflow usually consists of three main steps: (1) The end user fills out a ticket that describes the request, usually in free text. (2) The IT department receives the ticket and dispatches it to an appropriate IT specialist. (3) The IT specialist gains access to the computer in question, usually remotely, and attends to the request under superuser privileges on the target machine.

The last step constitutes a major security breach. While attending to the ticket, the IT specialist has unsupervised access to all the resources of the target machine, even though most of these are irrelevant to the ticket at hand. This can be exploited to steal classified commercial data, copy personal information, or install malware on the end-user's computer.

Third-Party Support Such threats come not only from the organizational IT department, but also from external service providers, e.g., a company that provides storage solutions for various customers such as banking systems, health companies, etc. Beyond providing the product or service itself, the company also remotely provides support, which requires superuser privileges on the customer's machines. Furthermore, the service provider is logged into the customer's organizational network, and is exposed to data on the target machine as well as data on other machines in the network. Thus, the service provider might be exposed to data that must remain confidential by law. For example,

this may include credit card information that must comply to the PCI data security standard and medical records subject to HIPPA compliance. Moreover, if the service provider is not an employee of the storage company, but a third-party IT contractor, the leakage can be very difficult to trace.

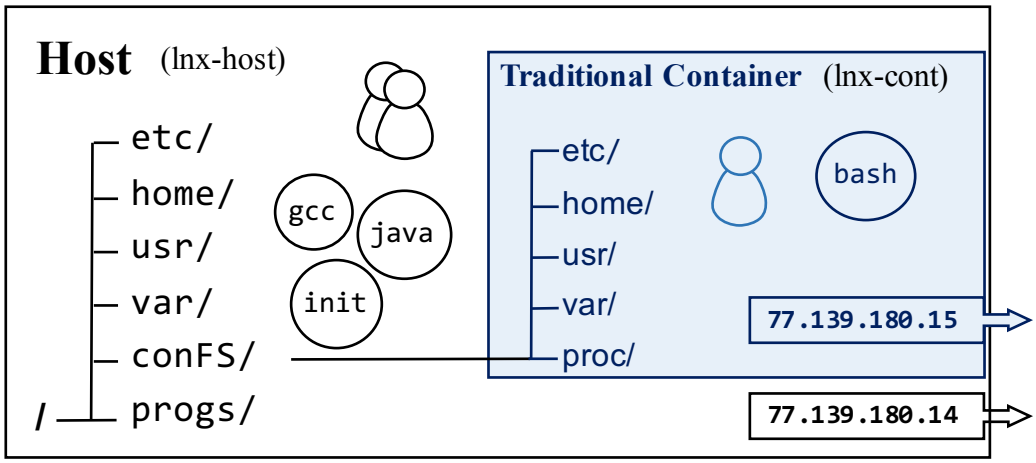
Automatic Management Tools IT departments usually employ automatic management tools, such as Chef [27] and Puppet [104], in order to perform various administration and maintenance tasks. These tools operate using dedicated scripts, which are written by the IT personnel and designed to verify system configurations and properties. These scripts may run periodically or be executed manually as part of handling specific IT requests.

Automatic management tools run with superuser privileges, without any sandbox to monitor the script's operations. An IT insider can tamper with these scripts, causing them to install malware or leak classified data, thus compromising many machines at once.

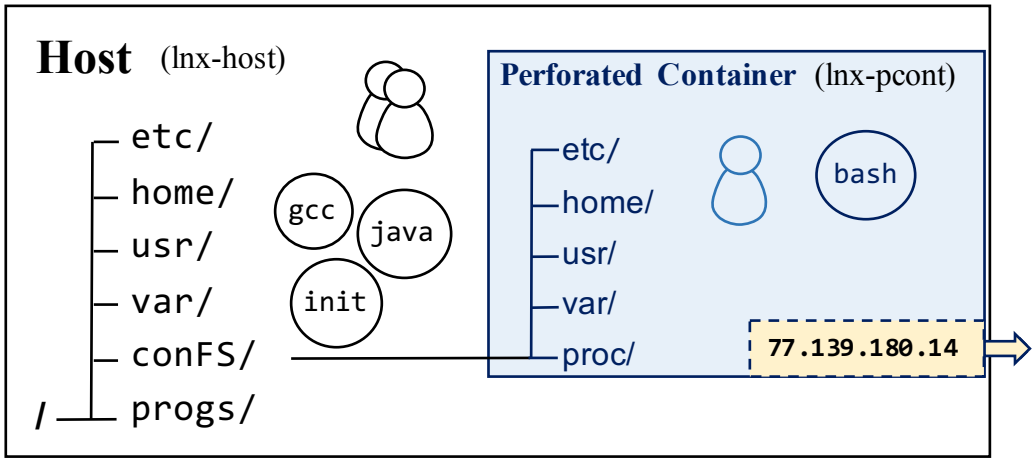
2.2.2 Containers

Containers offer an operating system level virtualization technique for running multiple isolated systems (containers) on a host running a single Linux kernel. Virtually, a container is a group of processes, isolated in various resources from the host system on which they run. The resource isolation in modern Linux-based containers is provided by the *namespaces* mechanism [83]; when a process is associated with a different namespace from the host, it has a different view of the corresponding resource. There are six namespaces in Linux; these provide per-container views of the following resources (see Figure 2.1):

1. UNIX Time Sharing (UTS) – affects the view of the host-name. For example, in Figure 2.1a, processes inside the container read the host-name of the machine that they run as `lnx-cont`.
2. Mount namespace (MNT) – provides different views of the mounted filesystem table to different processes running on the same Linux host. For example, in Figure 2.1a, we see a container that is able to see only the files mounted under `/conFS/` directory.
3. Network namespace (NET) – controls the network view of the contained processes. Processes that belong to the same NET share routing tables, firewall rules, and network devices (represented by IP arrow boxes in Figure 2.1a).
4. Process ID namespace (PID) – provides different views of the processes in the system. Processes that are associated with a given PID namespace can only see each other and processes in child PID namespaces.
5. Inter-process communication (IPC) – affects IPC objects that are identified by mechanisms other than filesystem pathnames, such as shared memory.



(a) Traditional container example.



(b) Perforated container example.

Figure 2.1: Traditional vs. perforated container example. A traditional container is associated with a different namespace of each available type, while a perforated container may share a namespace with the host (the network namespace in this example).

6. User ID namespace (UID) – provides different views of the IDs of the users in the system. This enables mapping of contained users to host users, thus giving corresponding permissions to contained users on resources in their view.

As demonstrated in Figure 2.1a, a traditional Linux container associates the processes that it contains with a new namespace of each available type.

Overall, a container offers an environment with the salient benefits one gets from a VM, but without the overhead that comes with running a separate kernel and simulating the hardware. As a result, the deployment time of containers is a matter of seconds (if all

the required files are available locally), and significantly shorter than virtual machines.

2.3 Perforated Containers

Perforated Containers Protecting a system from IT insider threats goes through fine-grained control over privileges along with monitoring. To satisfy these demands, we exploit the isolation properties of traditional Linux containers, and contrary to their nature, puncture this isolation – thus introducing *perforated containers*.

Perforated containers allow us to associate a process with only a subset of the available namespaces, thus allow it to share the remaining resources with the host. For example, Figure 2.1b illustrates a container that associates the processes it contains with all the available namespaces except for the network namespace, which is shared with the host. Thus, processes running inside the perforated container can only see the files under `/conFS/`, read the host-name as `lnx-pcont`, and so forth; this is similar to the traditional case. However, containerized processes have the same network view as non-containerized ones; namely, the perforated container and the host share routing tables, firewall rules, and network devices. Such a perforated container might be useful for repairing connectivity problems.

In some cases, the perforation might be too permissive. To this end we would like to inspect the operations done from within the perforated container, and the information that flows through these holes. Therefore, alongside the resource sharing, the perforated container's boundaries should be monitored; thus turning it into a sandbox environment. Such monitoring allows us to perform network packet inspection for a container's network traffic, filter filesystem accesses by content or file type, log various operations, and more.

Custom Tailoring Our proposed approach is to deploy perforated containers as a sandbox mechanism for IT operations. We adopt the Multics [116] principle of least privilege, and thus for each IT request, we sew a custom-made perforated container that allows access only to resources that are relevant to that request. We follow another Multics security principle and base the protection on permission rather than exclusion. For example, as illustrated in Figure 2.2, if the request requires fixing an expired Matlab license, the deployed custom-made perforated container should have filesystem access only to the Matlab directory, and its network view should include only the organizational license server. The view of other filesystem parts, as well as other network nodes, and even other system processes, should be unavailable. We configure the perforated container boundaries, as explained next, by processing the user request and predicting the maximal isolation under which the IT requests can be handled.

Bypassing and Monitoring By perforating the isolation provided by traditional containers, we create a supervised, controlled, and flexible sandbox environment for each IT request. Nevertheless, our prediction cannot be perfect, and there are bound to be cases in

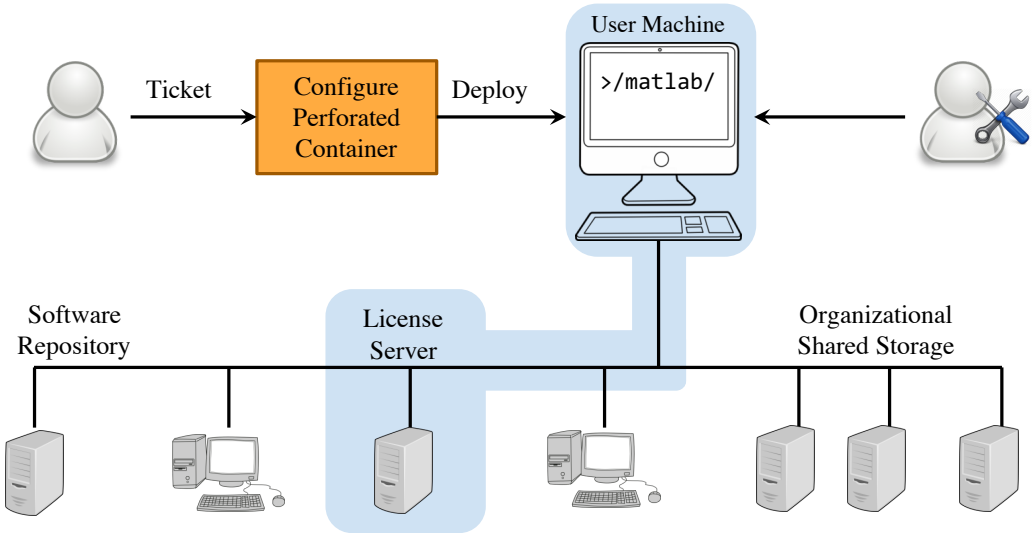


Figure 2.2: Example of deploying a perforated container designed to handle Matlab licensing problems.

which the configured isolation is too restrictive, and not sufficient for completing the designated IT request. To this end, we propose a *permission broker*. The permission broker is a software service that runs on the host and provides two complementary mechanisms. First, it allows administrators to bypass the container boundaries by executing commands on behalf of the perforated container. Second, it can grant the perforated container additional permissions (system views), thus effectively expanding its boundaries. Requests sent to the permission broker can be denied or approved, according to a predefined policy. Moreover, all such requests are logged for future analysis, thus enabling monitoring, better adjustment of container limits for future tickets, and improved investigation capabilities in case of security breach.

Benefits Exploiting containers for administration has the following benefits: First, our approach supports the creation of various perforated containers, each with a different isolation set. This allows for fitting a container for each IT mission, in a fine-grained granularity. Second, a superuser inside the container may indeed have the privileged capabilities it needs. Nevertheless, these privileges are effective only on the resources that are visible to the container. Third, containers can be deployed within seconds, given that all needed files exist on the target machine; this is valuable for administration tasks, which usually should be done in a timely manner. Fourth, namespaces are supported in all the latest Linux distributions. In addition, as explained later, our approach enables efficient monitoring and auditing of IT, including filesystem operations and network traffic.

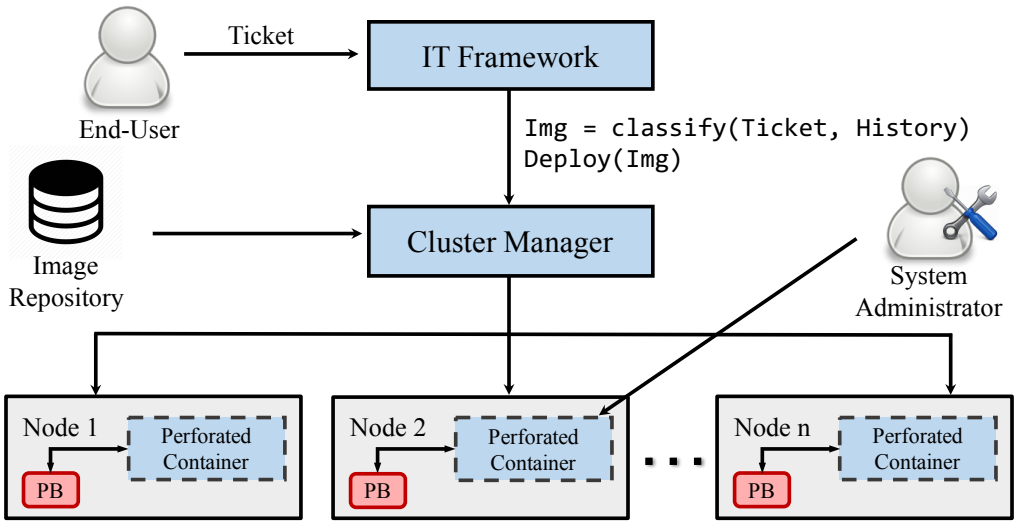


Figure 2.3: WatchIT architecture overview. PB is the permission broker.

2.4 WatchIT

In this section we present *WatchIT*, and dive into its implementation. In Section 2.4.1 we give an overview of the system structure. Next, in Section 2.4.2 we present our perforated container software, ContainIT, and discuss its architecture. We then present two mechanisms that ContainIT uses: ITFS (Section 2.4.3) for inspecting IT file operations and the permission broker (Section 2.4.4). Next, we discuss in Section 2.4.5 our technique for changing a running container filesystem view. Finally, in Section 2.4.6 we propose a new namespace for the Linux kernel, motivated by this work.

2.4.1 Architecture Overview

The architecture of our proposed system is depicted in Figure 2.3. The workflow begins with the submission of a request to the IT department by an end-user; we refer to this request as a *ticket*. Tickets are written in free text, and describe software problems, networking issues, expired licenses, and so on.

The system is pre-configured with a number of ticket *classes*, each associated with a dedicated perforated container encapsulating the privileges required in order to attend to tickets of this class (via namespace settings, installed software, and monitoring configuration). Like the Docker architecture [87], the various container images and configurations are held in a dedicated image repository for quick deployment. New tickets are submitted to an organizational *IT framework*. The framework analyzes the tickets and classifies each ticket to one of the predefined ticket classes, based on history and ticket parameters. The classification is performed automatically, and reviewed by the user or a supervisor.

Upon classifying the ticket, the framework asks the cluster manager to deploy the cor-

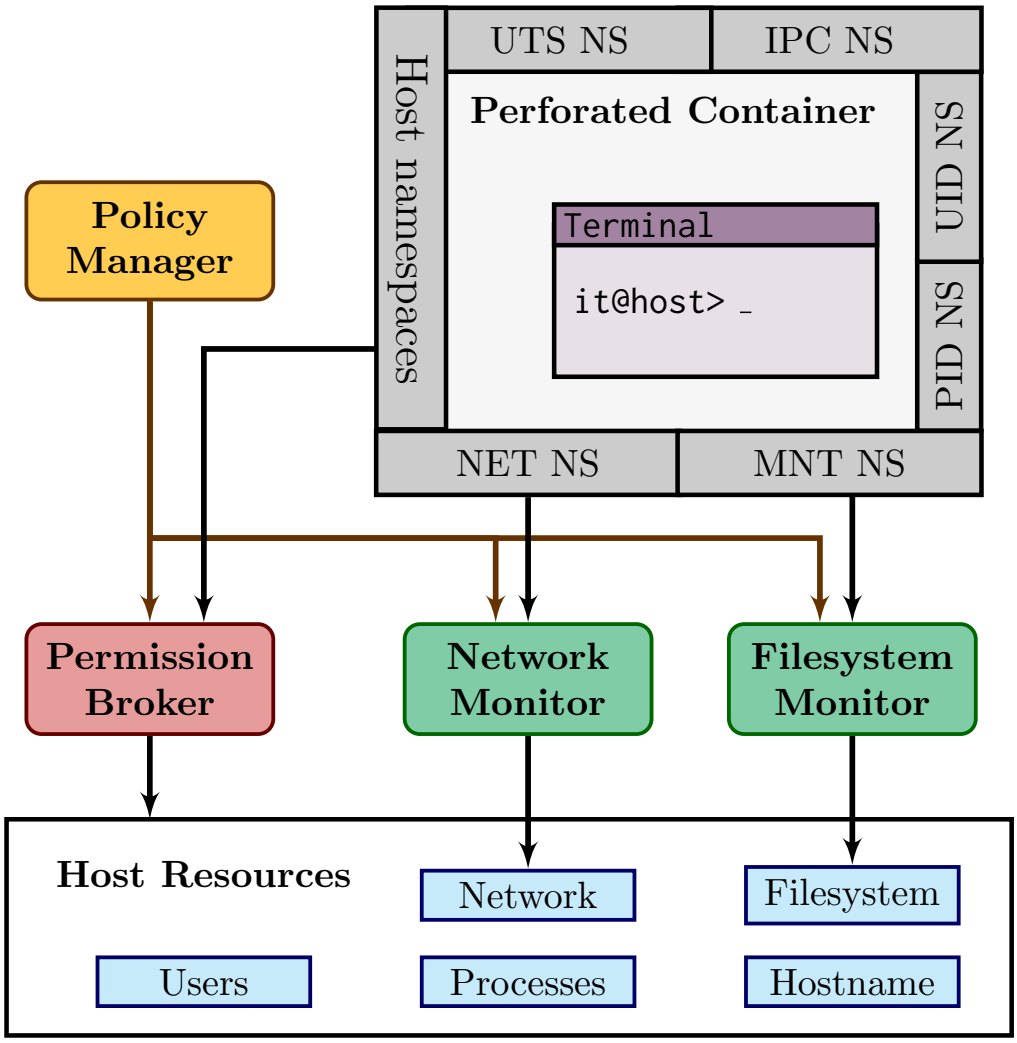


Figure 2.4: ContainIT software architecture. NS stands for namespace.

responding perforated container image on the target machines. Following the deployment, IT personnel can log into the deployed containers in the target machines and attend to the ticket. As proposed in previous works [132], connecting to the deployed perforated containers is enabled via a temporary certificate, which is revoked once the ticket time expires. Thanks to the isolation provided by the namespace subsystem of the OS kernel, the administrator is confined to the limitations dictated by the perforated container, and cannot operate on system parts to which the container is not exposed.

2.4.2 ContainIT

We now describe *ContainIT*, our software for deploying perforated containers. Commodity container software (such as Docker or LXC) does not support many of the features we require, including custom made perforated containers, monitoring container filesystem operations and network traffic, flexible on-line file-sharing, and more. Therefore, we build ContainIT, our own container software, which supports these features. ContainIT is written in C.

Figure 2.4 presents the software architecture of ContainIT. First, the boundaries of a perforated container are determined by its attributed namespaces (abbreviated NS). As explained in Section 2.3, not all available namespaces must be used, and access to resources governed by missing namespaces is unconstrained. For example, if the perforated container shares the host's PID namespace (because this namespace is excluded), then the IT person may freely communicate with processes on the host.

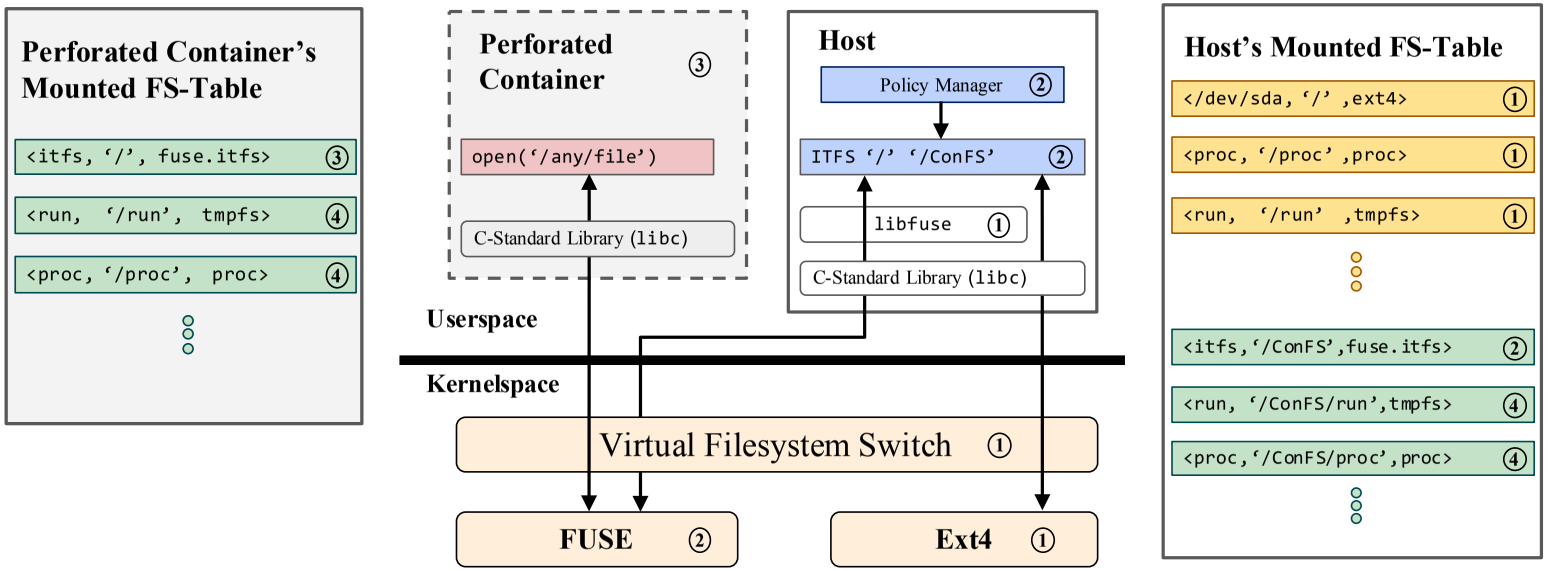
Second, accesses to network and filesystem resources through the corresponding namespace are monitored (green boxes). For example, network traffic going through network devices associated with the perforated container's NET namespace is tapped, analyzed, and can be blocked if necessary. To implement this feature, we make use of existing sniffing mechanisms [110, 98]. To monitor filesystem operations, we build ITFS – a FUSE-based filesystem that traps file system calls, allowing the inspection, blocking, and logging of file operations, as detailed in Section 2.4.3. Conceptually, this monitoring approach may be applied to additional namespaces, but we did not find a need for it.

Finally, for cases in which a contained user needs to access the host's resources, she can contact the permission broker (pink box) – a software service running on the host with unlimited access to the host's namespaces. Requests sent to the permission broker are logged, inspected, and can be denied or accepted. The policies of the permission broker and the namespace monitors are dictated by the *policy manager* (yellow box).

This software architecture makes our container software highly configurable. Examples include a perforated container that shares the host's root filesystem but cannot access document files; a perforated container that cannot see the host's filesystem, yet is able to see and kill the running processes on the host; a perforated container that can be accessed via SSH, but has no access to the world-wide web.

2.4.3 ITFS – Filesystem Monitor

We now describe the mechanisms we provide for monitoring file accesses by IT personnel as well as for sharing the host's root filesystem with a perforated container. For these purposes we build *IT File-System (ITFS)*, a FUSE-based [108] monitoring filesystem that can deny or log accesses to the underlying filesystem according to configurable rules. Note that current container software (Docker, LXC) does not provide the ability to deploy a container that has the same view of the filesystem as the host's. Since this is an imperative feature in administration, we provide it here, alongside the monitoring mechanism.



(a) Perforated container's mounted filesystem table.

(b) FUSE software architecture. Arrows represent the flow of file system calls.

(c) Host's mounted filesystem table.

Figure 2.5: Using ITFS in order to monitor a perforated container's filesystem operations. The perforated container's mount point is `/ConFS/`. The run-time flow is depicted in Figure 2.5b.

We build our ITFS solution in two steps. First, we mount the host's root filesystem at a dedicated mountpoint, while trapping all the file system calls targeted at that mountpoint. Second, we `chroot()` the perforated container to the dedicated mountpoint. Namely, we change the filesystem view of the container by making it see the dedicated mountpoint as the root of the system. Assuming that the perforated container has a different MNT namespace than the host, this mechanism ensures that all filesystem operations performed within the perforated container are monitored.

Figure 2.5 depicts how our mechanism inspects filesystem accesses initiated from within a perforated container. (1) In the initial state, the host only has the Ext4 filesystem registered in its VFS, while its mounted filesystem table (Figure 2.5c) only contains the yellow entries. (2) Next, we prepare the ground for deploying the filesystem-monitored perforated container by invoking ITFS on the host and mounting the root of the host's filesystem on a dedicated mount point (the directory `/ConFS/` in this example). This invocation adds an entry to the mounted filesystem table of the host (first green entry the in Figure 2.5c), and forwards all future file system calls targeted at this mountpoint to the FUSE kernel module, which registers a new filesystem in VFS.

(3) When deploying the perforated container, we configure its filesystem to be rooted at `/ConFS/`. Since the perforated container has a separate MNT namespace, it is only aware of the ITFS filesystem mounted on its root address. This is the first entry of the perforated container's mounted filesystem table shown in Figure 2.5a. (4) Additional future filesystems mounted in the perforated container's namespace scope (`run` and `proc` green entries in Figures 2.5a and 2.5c) are rooted at `/ConFS/`. As shown in the figure, all the entries of the perforated container's filesystem table have counterparts in the host's filesystem table.

Once deployed, every file system call issued from the perforated container is forwarded through VFS, by the FUSE kernel module, to the libfuse library running on the host. Hence, as illustrated by the arrows in Figure 2.5b, when a containerized application calls `open()` (pink box), the call is forwarded to ITFS, which invokes the corresponding callback defined in its policy (blue boxes) for the `open()` operation. The callback may take *any action*, and return the desired data in a supplied buffer. For instance, it can read the file from the underlying filesystem (such as Ext4), detect its type according to its signature, and deny access if the file is a picture or a document. Alternatively, it can allow access to the file, but log this access for later analysis. Finally, the return value is propagated back by libfuse, through the kernel, to the application that issued the `open()` inside the container.

We further provide an API for configuring the monitoring rules and actions. The API supports forbidding and/or logging access to a file according to its signature or extension. This allows, for example, control over access to documents – files that are usually irrelevant to IT work. In addition, ITFS exposes an API for integrating user-supplied detection rules, as scripts or programs, so that each organization can create customized file filtering.

Under the approach described in this section, the user logged in to the container

inherits the privileges of the user that invokes the ITFS on the host; this happens in all FUSE-based filesystems by design. Thus, if ITFS is mounted with superuser privileges, the user inside the container also has superuser privileges for all the files that are exposed to the container through ITFS. Indeed, this is the desired behavior for administration tasks.

Note that the presented approach is possible only if the perforated container has a different MNT namespace from the host, which is usually the case. As shown in the case study in Section 2.6, none of the examined tickets required sharing the MNT namespace of the host. Under this condition, ITFS can inspect the file accesses of the contained IT person, without allowing her to bypass this mechanism, even if the perforated container is admitted to the host's root filesystem itself. For cases in which the administrator does need access to the host's entire MNT namespace, we provide the XCL namespace, as discussed in Section 2.4.6.

2.4.4 Permission Broker

On some occasions, the permissions assigned to a container may be insufficient for handling the ticket. To address such cases, we augment the system with a *permission broker*. The permission broker is a software service that runs on the host and can grant a running container additional permissions, change its filesystem and network view, and provide it with information regarding the host system, while logging all accesses.

For example, if the IT specialist attends to a network problem and during the work wishes to see the list of running processes in the system (from which she is compartmentalized), she can submit a request to the permission broker, as shown in Figure 2.6. In the example, executing "`ps -a`" from the container shows only processes that belong to the container's PID namespace. By requesting the command from the permission broker (using the `PB` prefix), the contained user may see the host's processes.

The permission broker grants a request if it follows the security policy corresponding to the specific ticket class and IT specialist, and can refuse otherwise. Either way, these requests are logged in real-time to a secure append-only storage device, and can be monitored and analyzed later for anomaly detection. As a consequence, our permission broker logs only IT activities that diverge from the predefined isolation, which captures the expected behavior for handling a specific ticket. Hence, the permission broker's log is sufficiently succinct to be inspected and analyzed for anomaly detection [24, 115, 145, 29], where one of the major challenges is handling enormous amounts of data. Moreover, if certain permissions are repeatedly requested, they can be added to the ticket class's perforated container, thus further reducing the amount of gathered data, and in turn facilitating future data analysis.

We implement the permission broker in Python, using a client-server architecture, where the server side is installed on the host and the client side is invoked from the container. The communication goes through the tcp/ip stack, and we use Google's Protocol Buffers and gRPC [53] for serializing and streaming the data. In order to prevent regular

```
root@ITContainer:/home/itsupport# ps -a
  PID TTY          TIME  CMD
    1 pts/4    00:00:00  ContainIT
   32 pts/4    00:00:00  bash
   71 pts/4    00:00:00  testscript
   73 pts/4    00:00:00  ps
root@ITContainer:/home/itsupport# PB ps -a
  PID TTY          TIME  CMD
 1023 pts/14   00:00:00  PermissionBroker
 1075 pts/4    00:00:00  sudo
 1077 pts/4    00:00:00  ContainIT
 1080 pts/4    00:00:00  itfs
 1081 pts/17   00:00:00  snort
 1139 pts/4    00:00:00  bash
 1272 pts/18   00:00:00  testscript
 1276 pts/19   00:00:00  ps
root@ITContainer:/home/itsupport#
```

Figure 2.6: Example of using the permission broker.

users from contacting the permission broker, we configure the permission broker client to accept only requests from privileged users. If one wishes to further secure the communication between the perforated container and the permission broker, one can employ SSL.

In order to change the system view of the deployed perforated container, the permission broker performs operations on the routing tables and firewall rules of the container's namespaces and uses the `nsenter` tool as detailed next in Section 2.4.5.

2.4.5 Online File Sharing

Commodity container software such as Docker provides means for mapping directories from the host filesystem into a non-root directory in the container filesystem. However, such mapping must be requested at launch-time; once the container has been deployed, there is no support for exposing additional host directories to the container.

In WatchIT, on the other hand, we would like to allow the permission broker to map additional directories on-the-fly; namely, expose additional directories to the perforated container while it is running, without requiring it to restart.

Hence, we equip ContainIT with the ability to perform on-the-fly file sharing, while ensuring that subsequent contained accesses to newly shared files are monitored by our ITFS mechanism. Mounting additional directories into the container filesystem is not trivial. Due to the namespace hierarchy, mount operations on the host are not visible in the container; hence, the mounting should be executed from within the perforated container. On the other hand, the perforated container is not aware of the files from which it is isolated; hence the operation cannot be purely executed from within the container.

Our solution employs nsenter [12], a Linux tool for entering the namespaces of a given process and executing programs within them.

The full process of adding a new volume to a running container begins by issuing a request from the container to the permission broker. Next, the permission broker logs the request and turns to execute it under superuser privileges. The implementation of the feature itself is comprised of three main stages: (1) extracting the full real path to the host directory and device ID on whose filesystem the directory resides. (2) using nsenter in order to infiltrate the namespaces of the running perforated container; and (3) creating an ITFS bind mount to the host directory in the target path from within the container's namespace.

Since we create an independent ITFS bind mount, accesses to the newly mounted filesystem are supervised by ITFS, but can have different rules. Moreover, even if one chooses not to employ ITFS monitoring on the originally deployed perforated container, one can still employ it only on the newly mounted files.

Finally, we note that our online file sharing technique does not constitute a security breach in the Linux namespace mechanism. It is possible only because it requires superuser privileges on the host. In our case, these privileges are provided by the permission broker.

2.4.6 Exclusion Namespace

The mechanism described in Section 2.4.3 for sharing the host's underlying filesystem with the perforated container under supervision and monitoring is only possible when the container has a different MNT namespace from the host. However, it may be the case that the administrator needs to access the host's entire MNT namespace, for example, if the system administrator needs to handle problems with the filesystem itself or mount more filesystems on the machine. In these cases, the perforated container and the host have the same view of the mounted filesystem table, and there is no guarantee that each filesystem operation of the containerized superuser is monitored.

To solve this problem, we introduce a new namespace to the Linux kernel – *exclusion (XCL) namespace*. The XCL namespace has a table of excluded directories – filesystem sub-trees that cannot be accessed by processes that belong to that namespace, disregarding the user privileges. Thus, even if a containerized superuser has access to the underlying filesystem, she will not be able to access the parts of the filesystem that are specified in the exclusion table.

We implementat the XCL namespace in version 4.6.3 of the Linux kernel. Following Linux conventions, one can associate a newly created process with a new XCL namespace instance by executing the *clone()* system-call with the flag CLONE_XCL. Entries can be added to and removed from the excluded directory table using dedicated system-calls. A newly created namespace instance inherits its parent's exclusion table.

ID	Attack	Defense	Weaknesses
1	Escape perforated container boundaries	Block chroot() capability	
2	Bind shell	Block ptrace() capability	IT cannot perform debugging
3	Raw disk mounting	Block mknod() capability	IT cannot create special files
4	Memory tapping	Block access to /dev/mem & /dev/kmem	
5	Tampering with WatchIT software	(1) Include WatchIT in TCB (2) Block access to WatchIT files	
6	Tampering with log files	(1) Replication (2) Block access by ITFS	
7	Kill WatchIT component	Other components exit and terminate session	
8	Encrypt and exfiltrate	(1) Blocks filesystem access (ITFS) to unencrypted files (2) Sniff or block suspicious network traffic	Requires ITFS and network sniffer rules
9	Fake tickets	IT personnel cannot create trouble tickets	Collusion with non-admin insider
10	Ticket stringing	(1) Permission-based ticket assignment (2) Imposing hard constraints on all perforated containers	
11	Malware installation	(1) Website whitelisting (2) Monitor incoming network traffic	Watering hole attacks, phishing, etc.

Table 2.1: Possible attacks on WatchIT, the measures we take to neutralize them alongside their main weaknesses.

2.5 Threat Analysis

We now discuss the threats facing WatchIT, which we summarize in Table 2.1, and the measures we take in order to neutralize them. In Section 2.5.1 we analyze methods for breaching WatchIT protection from within a perforated container using technical skills. Section 2.5.2 discusses ways to circumvent the WatchIT system.

2.5.1 WatchIT Software Security

Keeping Perforated Container Boundaries Safe Our perforated container is created by punching holes in the isolation provided by traditional containers. Moreover, a perforated container may map a contained user to a privileged one on the host, since it may be required to perform operations like service restarts or system reboots. The holes in perforated containers alongside privileged permissions might be abused to escape the perforated container boundaries.

There are four known techniques to escape a chroot()-ed environment (including containers) [131]. The most common one requires root privileges, and issues a new chroot() command in order to escape the current one. The second technique is based on communicating with a process outside the container, changing its code using ptrace(), and turning it into a bind shell. The third builds on creating raw disk devices and mounting existing filesystems on them. The fourth creates /dev/mem or /dev/kmem devices, thus tapping and modifying system and kernel memory.

Using the Linux capabilities feature [23], we deprive contained users of three capabilities that are essential for employing the first three aforementioned bypassing techniques (Attacks 1-3 in Table 2.1), and are also rarely needed in IT work. These are: chroot(), ptrace(), and mknod(). We further implement a new capability and employ it to block a contained user from opening /dev/mem and /dev/kmem (Attack 4).

Previous work [19] presented nine ways to escape chroot()-ed environments by exploiting privileged permissions, which use similar capabilities as the attacks described in [131]. We tested each of them in containers deprived of the capabilities mentioned above, and verified that they cannot escape the perforated container.

Protecting WatchIT Software We prevent the contained administrator from tampering with our sandbox mechanism in two ways (Attack 5). First, we include WatchIT software (including the permission broker, policy manager, and ITFS software) in our TCB, so the system will not boot if any of its components have been tampered with. Second, we use ITFS to block accesses to all WatchIT files; this way, we deny access to these files even if the contained user can view them. Furthermore, in order to prevent the log files from being compromised, they can be replicated on a remote append-only storage (Attack 6). Finally, we design ContainIT to terminate the session if any of its peer processes (e.g., permission broker service) are killed (Attack 7).

Protecting Files within a View Network sniffer software [98, 110] mostly relies on detecting the signatures of files sent over the network. Hence, a common attack would be to tamper with the victim files or conceal their content by encryption and send them over the network (Attack 8). In order to protect from such an attack, the ITFS blocks the actual access to files that are defined as classified by the policy manager. Since one cannot tamper with a file without reading it, this prevents exfiltration.

2.5.2 Circumventing WatchIT

Collusion. WatchIT is designed to protect against a single adversarial system administrator and is vulnerable to collusion between a user and an IT person (Attack 9). That said, we note that the state-of-the-practice is that administrators can act solo, so making collusion necessary significantly raises the bar for attack; in particular, when IT is managed by a third party (possibly in a different country). Furthermore, filesystem operations performed from within a perforated container are monitored by ITFS and since all tickets are recorded, collusion leaves a trail.

Ticket Stringing. If an administrator is assigned to handle multiple ticket types, a possible attack would be to sequentially string tickets, thus concatenating system views and expanding her effective permissions (Attack 10). An effective solution against such threats is to impose hard constraints on all perforated containers. For example, as presented in the case study in the next section, imposing an ITFS policy (e.g., forbidding access to documents) and network sniffer rules (e.g., disallowing transfer of encrypted files) on all perforated container classes, prevents data leakage even in the face of stringing. Moreover, in large organizations, WatchIT can be protected from such threats by assigning to each IT person only tickets of the same class.

Group-Targeted Attacks Organizations are usually exposed to group-targeted attacks, like phishing and watering hole (Attack 11). WatchIT is not designed to provide protection from such threats. Thus, for example, watering hole attacks on whitelisted sites that are used for software downloading (e.g., Eclipse) remain dangerous. Nevertheless, they affect regular users as well as administrators. Moreover, if WatchIT is not in use, administrators accessing these sites allow the downloaded malware broader system view and permissions than when WatchIT is used.

Finally, we note that WatchIT supervisors who create the perforated containers and define their system view remain omnipotent. However, we narrow the trust group from many administrators, possibly including third-party contractors, to very few.

2.6 Case Study

This section presents a case study of applying WatchIT on the IT database of IBM Research in Israel. The studied IT department consists of about 30 system administrators,

among them 7 Linux specialists, supporting around 600 technical users. We begin in Section 2.6.1 with a thorough ticket analysis, clustering, perforated container tailoring, and testing on real-world workload. Next, in Section 2.6.2, we audit scripts used by the same IT department (e.g. Chef, Puppet) and show that perforated containers can be suited to protect from tampered scripts. Finally, in Section 2.6.3 we evaluate ITFS performance and show that it can indeed be used as a filesystem for administration tasks.

2.6.1 IT Tickets

We apply the WatchIT approach to tickets of a real-world IT department. We employ statistical tools to cluster the tickets into classes and validate the resulting division with IT personnel. Next, we custom-tailor perforated containers and test their compatibility on new tickets collected during a trial period.

Ticket Clustering

For our case study, we analyze an IT database containing about 66,000 tickets, collected during the years 2009 through 2016. Tickets are written in free text by end-users and are manually sent by the IT department to the corresponding IT specialist. The actions taken by IT personnel to handle each of these tickets are not included in the database. From this corpus, we gather the tickets pertaining to Linux machines. The filtering is done by choosing only the tickets that were assigned to IT personnel who specialize in Linux issues. This leaves us with a corpus size of around 17,000 tickets.

To group the tickets into categories, we use topic modeling [13]; such algorithms take a corpus and group the words across it into topics. Before performing topic modeling, we pre-process the corpus by applying word stemming, stop word removal, deletion of common words that do not add information (like 'hello' and 'please'), and obfuscation of confidential information such as server names, addresses, project names, etc. We then use Latent Dirichlet Allocation (LDA) [14] to process the data and group it into a specified number of topics. We run LDA with 7 to 14 topics and choose the most appropriate result, which in our case consisted of ten topics.

Partial results of the ten-topic LDA analysis appear in Table 2.2. LDA represents a topic as a distribution over words; thus, the full results of LDA for each topic include a list of all the words that appear in the corpus. Each word in each list is associated with a number that represents the likelihood of finding that word in a text on the corresponding topic. For illustration purposes, Table 2.2 shows only six representative words for each topic, taken from the top-twenty words of each list. Surrounding angle-brackets represent names or addresses; e.g., <IP> stands for any IP address.

One can infer from the results the main categories that the studied IT department deals with. The category distribution appears in Figure 2.7. For example, topic T-1 constitutes 5% of our tickets and refers to license problems, which are usually associated with Matlab, Matlab toolboxes, and database software; topic T-5 refers to slow or non-

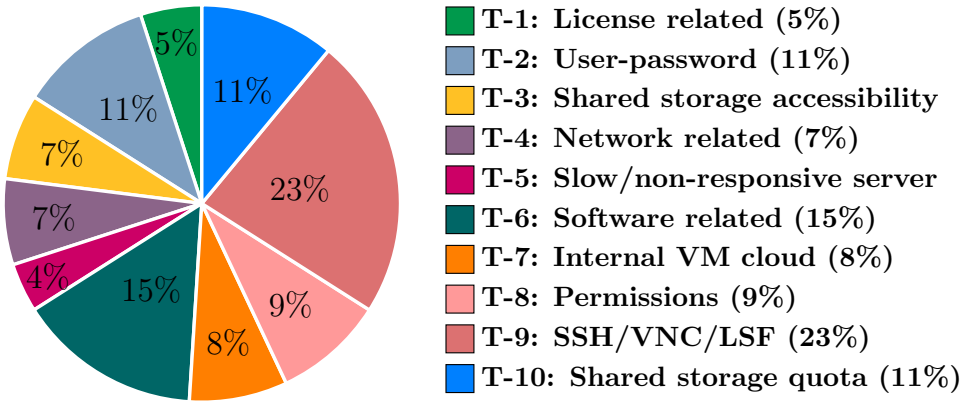


Figure 2.7: Category assignment and distribution.

Topic T-1	**Topic T-2**	**Topic T-3**	**Topic T-4**	**Topic T-5**	**Topic T-6**	**Topic T-7**	**Topic T-8**	**Topic T-9**	**Topic T-10**
license	password	file	connect	work	install	VM	access	connect	space
matlab	user	< Shared Storage >	< IP >	time	< Server >	< VM >	user	< Server >	project
error	connect	access	port	machine	version	GB	add	SSH	< Shared Storage >
DB2	account	SVN	server	slow	< OS >	IP	group	respond	GB
toolbox	login	directory	network	stuck	upgrade	disk	team	VNC	increase
message	locked	git	IP	reboot	< Application >	kvm	permission	LSF	quota

Table 2.2: Results of running 10-topic LDA on our data set. For each topic we present six of its top 20 words.

	Process Management Permission Set	**Filesystem Access**			**Network Access**						
		Home Directory	/etc/	Root Directory	License Server	Batch Server	Shared Storage	Target Machine	Software Repository	Whitelisted Websites	Network Namespace
T-1: License related		X			X						
T-2: User / password			X								
T-3: Storage accessibility	X	X					X				
T-4: Network related	X		X		-	-	-	-	-	-	X
T-5: Slow server	X	-	-	X							
T-6: Software related	X	-	-	X					X	X	
T-7: Internal VM cloud			X								
T-8: Permissions		-	-	X							
T-9: SSH/VNC/LSF	X	X	X			X		X			
T-10: Storage quota			X				X				
T-11: Other											

Table 2.3: Permission and isolation per each container type. We denote by "X" explicitly included resources, and by "-" resources that are implicitly included due to the inclusion of another resource.

responsive servers; and topic T-6 is associated with software related requests, as among its top words we can find application names like eclipse, gcc, and hadoop, and words like "install", "upgrade", "version", "package", and "plugin".

To verify our topic choices, we interview the IT personnel without exposing them to the LDA results. Our interviews with the IT personnel yielded 13 main categories, which are all included in the LDA results, although with some modifications: (1) IT personnel treated SSH, VNC, and Load Sharing Facility (LSF, a batch job execution environment) issues as different categories, while LDA groups them into one (Topic T-9). This happens since the descriptions of such problems often use the same words. (2) IT personnel distinguish between shared storage accessibility problems to SVN and git issues, while LDA mixes them into one. Again, this happens for the same reason – organizational SVN and git repositories reside on the organizational shared storage, and all related IT issues are associated with creating and managing accessibility to these repositories; hence the same words are used to describe these issues.

However, as presented in the next section, in both of these cases, the mixed categories also share the required permissions for handling their associated tickets. Indeed, SSH, VNC and LSF issues always necessitate a connection to a remote server, and involve changing local configuration files. Similarly, SVN and git repositories always reside in the shared organizational storage and are associated with accessibility issues. We conclude that the LDA output is accurate and sufficient for mapping tickets to the needed permissions.

Permission Assignment

Given the above classification, we consulted with the IT personnel and built ten perforated containers, which differ in their permissions and resource isolation. The isolation and permissions for each ticket class appear in Table 2.3; alongside the isolation, filesystem accesses are monitored by ITFS and network traffic is sniffed by IDS software.

For example, the T-1 perforated container for attending to licensing problems can modify the home directory of the user and connect to the organizational license server (a server responsible for company license management, maintained by the IT). Nevertheless, it is compartmentalized from the rest of the filesystem and other nodes in the network.

The process management permission set includes the ability to (1) see and kill the host's running processes, (2) restart host's system services, and (3) reboot the machine. Whereas T-1 (the licensing container) is isolated from these permissions, we do grant this set of permissions to T-5 containers, which target non-responsive/slow server issues, since these usually involve killing resource-consuming processes.

For T-6, covering software issues, we match a perforated container with ITFS-monitored access to the root filesystem of the target machine. The network view of such a container includes only the organizational software repository and monitored access to a whitelist of websites.

The perforated container for T-9, which handles SSH, VNC, and batch computing

problems, shares the corresponding configuration files (located under /etc/ and in the home directory) with the host, and has a limited network view, which includes the target machine (for SSH and VNC) and the organizational batch computing server (for batch computing). To enable service restarts after configuration fixes, it is granted the process management permission set. Note that this container is deployed both on the user and the target machines, since configurations might need to be fixed in both of them.

T-7 is the container for VM cloud issues. The organizational VM cloud is managed using an EC2-style GUI that can create, reboot, terminate, and change resource allocations of the hypervisors. These operations cover most of the VM cloud-related tickets, without the IT personnel having to log into any VMs. However, when creating a new VM from a ready and signed image, the IT person must access it in order to configure its ownership properties. With WatchIT included in the signed initial filesystem image of each VM, the T-7 perforated container is only exposed to the relevant ownership configuration files in /etc/. Thus, if a user requires a new VM with some software installed, she should create two tickets – one for a new VM on her name, and one for software installation in that VM.

For issues that do not match any of the classes, we build a fully isolated container, T-11, thus tracking and logging all operations that are done while attending to the unclassified ticket.

Testing and Results

After extracting the ticket classes from historical data, during three months from December 2016 to February 2017, we collected and audited all 398 Linux-related tickets created in the system, excluding hardware failures and Windows/iOS related tickets. For each ticket, we recorded the operations that were performed while treating it, and asked the IT team to classify it to one of the ten predefined categories. Thus, we created a database that includes for each ticket its free-text description, a classification, and the permissions required for handling it. Next, we check whether we can apply the operations performed for each ticket inside its corresponding perforated container. We also predict the class of each ticket using our LDA model, after applying spelling correction.

We present the results in Table 2.4. The first column shows the category distribution of the collected ticket set, as labelled by the IT department, and the second presents the prediction accuracy for tickets of each class. The third column shows, for each category, the percentage of tickets in this category that were completely satisfied by their corresponding custom-made perforated container. As the table shows, 92% of tickets could be handled from within the corresponding container. Considering the isolation of each container, we prevented full filesystem view in 62% of the cases, compartmentalized the process view in 36% of the tickets, and isolated the network view in 98% of the cases. Moreover, access to the world-wide web was made possible only in 32% of the tickets (T-6), and only to whitelisted websites. Furthermore, all filesystem and network accesses were monitored.

ID	% of Total Tickets	Classif-ication Precision	% of Tickets Satisfied by P.Container	% of Tickets Used Permission Broker		
				Process Management	Filesystem	Network
T-1	9%	94%	94%	3%	-	3%
T-2	7%	95%	86%	-	-	14%
T-3	8%	94%	93%	-	-	7%
T-4	2%	100%	100%	-	-	-
T-5	5%	95%	89%	-	-	11%
T-6	30%	94%	91%	-	-	9%
T-7	10%	100%	97%	3%	-	-
T-8	3%	92%	75%	17%	-	17%
T-9	21%	98%	100%	-	-	-
T-10	3%	92%	100%	-	-	-
T-11	2%	80%	-			
Total	**100%**	**95%**	**92%**	**1%**	**-**	**7%**

Table 2.4: Results of attempting to use our custom-made perforated containers for handling tickets collected during a test period.

Tickets that did not match any of the predefined categories were classified as "Other" (T-11). These include rare IT requests such as partition resizing and driver updates. The latter constitutes only 0.5% of the tickets; they change our TCB and are indeed rare.

The next three columns in Table 2.4 details the causes for which the permission broker was employed. One example of such a ticket was classified as license problem since a user requested a license for a specific Matlab toolbox, but the toolbox was not installed on his machine. Since license-type perforated containers are isolated from the software repository, the IT needed the permission broker in order to install the toolbox.

2.6.2 IT Scripts

Chef and Puppet We review twenty bash scripts used by the IT department. These scripts are intended for various purposes: time synchronization, permission and configuration verification, service restarts, etc. They are executed periodically and before ticket handling, using Chef and Puppet. They execute with root privileges and occasionally resolve the ticket without human intervention.

We examine the required isolation for each script and conclude that most of them

Container		Capabilities			
ID	Dist.	Process Management Permission Set	Home Directory	/etc/	Network Namespace
S-1	60%			X	
S-2	20%	X		X	
S-3	10%		X		
S-4	10%	X		X	X

(a) Custom-made perforated containers for Chef and Puppet scripts.

Container		Capabilities			
ID	Dist.	Process Management Permission Set	Statistic Tools	Filesystem Access	Network Namespace
S-5	80%		X		
S-6	20%	X			

(b) Custom-made perforated containers for cluster management scripts.

Figure 2.8: Perforated container tailoring for IT scripts.

access only specific configuration files, few of them engage with system processes, and a few require sharing the network namespace (for IP table operations). Overall, we group the scripts into four categories. Hence, as listed in Figure 2.8a, we build four different containers and map each of the scripts to a container that can run it under maximal isolation.

Cluster Management We audit thirteen scripts used for automation and management of Apache Spark and IBM Swift clusters. These scripts mainly collect statistics, search for failures by reading system logs, and automate cluster operations like service restarts and system reboots. As presented in Figure 2.8b, we learn that a single, very limited perforated container can answer the needs of 80% of the scripts. This container should have access only to system logs and statistic tools (e.g., mpstat). The rest of the scripts handle system services and reboots, and are thus granted only the process management permission set. Note that these perforated containers are isolated from the network; as a result, tampered scripts can never leak information outside of the cluster.

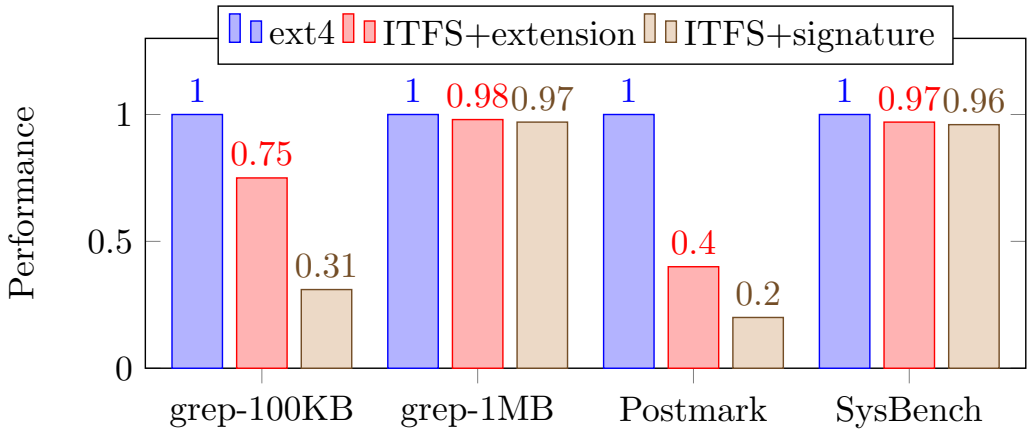


Figure 2.9: ITFS performance evaluation.

2.6.3 ITFS Evaluation

Our container software utilizes the Linux namespace mechanism and therefore enjoys the low overhead attributed to containers. That being said, as [106] shows, the use of ITFS, just like any other FUSE-based filesystem, may incur performance degradation.

We measure the ITFS overhead by performing a typical administration task, grep, on 25GB directories with average file sizes of (1) 1MB and (2) 100KB. We further use two other benchmarks, characterized by different workloads: (1) Postmark [68], configured to access many 5KB-256KB files; and (2) SysBench [3], which accesses a small number of large files. Our system runs Ubuntu 16.04 on Intel Core i7-4790 with 16GB RAM and SSD hard drive.

Figure 2.9 presents the results of executing these benchmarks on our system with three filesystem configurations: ext4 (baseline), ITFS with file-extension monitoring, and ITFS with file-signature monitoring. We see that the ITFS overhead depends on the sizes of the accessed files and on the monitoring rules. Overall, when engaging large files the performance is close to the baseline and under small file workload the ITFS overhead becomes more substantial. Note that ITFS mainly provides permission checking and does not intervene in the actual read or write operations. Therefore, if one wishes to improve its performance, one can employ a pass-through read/write approach as proposed in previous work [95].

2.7 Related Work

The leading solutions for providing protection from insider threats are based on mandatory access control [9, 35], tainting processes that access classified information [69], and defining SDNs within the organization [2]. However, they all trust the system administrators; a rogue IT person can change their configuration and compromise the system.

SELinux [84] and others [20, 78] implement role-based access control models. Contrary to these, WatchIT adopts an ACL-like approach, which proved to be more practical. Furthermore, unlike WatchIT, they do not provide monitoring and network virtualization, and escalation is not possible in cases of insufficient permissions.

Security information management software such as [130, 48, 33, 58, 134, 123] helps organizations collect and analyze log and intelligence data in order to identify malicious activities. However, rather than *proactively* preventing attacks, they only perform after-the-fact analysis to *reactively* detect anomalies.

The Jail [71] and Zone [103] mechanisms are designed to enable multiple root users, each with a different view of the system. However, these are intended for isolating customers in server consolidation scenarios. Contrary to our approach, no actions can be performed on the host from within a Jail or a Zone, and they are not suited for administration.

Santos et al. [117] proposed an OS that suppresses superuser privileges and exposes a narrow management interface, thus protecting systems from untrusted administrators. The WatchIT approach, on the other hand, allows system administrators to retain their superuser privileges, thus causing minimal changes to IT workflow. Moreover, WatchIT does not require changes to the OS, and provides monitoring on all actions performed by IT personnel.

2.8 Summary

We proposed an approach for mitigating insider threats from the organizational IT department. Our strategy exploits containers' properties, but goes against their nature by perforating their isolation, and thus turns them into sandboxes for administration. We further implemented a proof-of-concept of our approach and provided a case study on a real IT department in which we custom-tailored perforated containers to the needs of the studied IT department. The isolation and permissions, as well as the ticket class granularity can be tuned for the needs of any organization that adopts our approach.

Finally, we note that WatchIT is not a panacea. Collusions can bypass WatchIT protection and group-targeted attacks remain efficient in the presence of WatchIT. That said, the mere fact that bypassing WatchIT requires collaboration with another insider or a sophisticated infection of a known website implies that WatchIT significantly raises the bar for attacks by an adversarial administrator.

Chapter 3

Binary Similarity Detection Using Machine Learning

Similarity detection between code binaries has various use cases in the fields of cyber security and intellectual property. Code vulnerabilities are discovered continually, while some of them (e.g. `heartbleed`) were possibly introduced many years ago, so they might have already made their way into various software packages and computer platforms. Such vulnerabilities may have been deployed in a *binary* form to home computers, enterprise servers and device firmwares. A security-savvy individual, or more commonly a security-aware company, would want to use a sample of the vulnerable product (in a binary form) to search for the vulnerability across all the software installed in its system, where the source code is usually not available. Likewise, cloud providers would like to protect the virtual machines of their clients, however without inquiring their software or data in order to keep their privacy. These would like to periodically scan the memory of the guest VMs and make sure that their clients do not use vulnerable code. Furthermore, in an acquisition process, the company being purchased should prove that it does not use any licensed code in its products; preferably, without exposing its source code. All the above-mentioned scenarios share the common interest of having the ability to search for a specific code in a binary form, without possessing its source code.

The main challenge in identifying binary similarities is that the same code can be compiled using different compilers, different optimization levels or for different architectures, thus producing syntactically different binary code. Though this challenge can be solved with near-perfect accuracy using SMT solvers [36], such approaches are infeasible due to the low throughput that they can achieve and the huge amount of code corpora. On the other hand, faster approaches achieve lower accuracy and generate false positives, which cost in waste of human resource that later manually review the detection results.

In this paper, we propose an approach for binary similarity detection which is highly accurate yet faster than any previous binary similarity search technique, thus allows for scanning real-world workloads in practical time. It is based on the similarity by compo-

sition principle [16] alongside machine learning. We introduce the *proc2vec* method for representing procedures (or code sections) as vectors. In proc2vec we decompose each procedure to smaller segments, translate each segment to a canonical form and transform its textual representation to a number, thus finding an embedding in the vector space for each procedure. Next, we design a neural network classifier that detects similarity between vectors that originate from semantically equivalent procedures. In order to train our classifier, we build a database with dozens of millions of examples, generated by compiling various open-source renowned projects that we find in the world-wide-web. We compile each project using different compilers, with different compiler versions, optimization levels and for different architectures.

Finally, we evaluate our approach by predicting similarities in open-source projects that are not included in the training set. We show that our proposed classifier achieves high accuracy and executes more than 250X faster than current state-of-the-art tools.

3.1 Related Work

Similarity Search Previous work in the binary code-search domain mostly employed syntactic techniques, and were not geared or suited to handle the challenges of a cross-compiler search [93, 148]. Others require dynamic analysis in addition to static analysis [42, 120] or do not scale [36, 120] due to computationally heavy techniques.

David et al. [36, 37] presented an approach for reasoning about similarity of binaries based on segmenting code sections into strands. We build upon this approach, however in contrast to [36] which uses a heavyweight SMT solver, we employ machine learning techniques for predicting similarity. Moreover, unlike GitZ [37], we forgo the need for lengthy translations between IR representations and costly statistical reasoning; thus we achieve better throughput of about two orders of magnitude.

ML and Similarity Search Previous work [139, 77, 8, 101, 79] proposed using ML for code clone detection and vulnerability search; however, they all operate on the source code level and not on the binaries. Specifically, [101, 79] proposed techniques for building program vector representations, so these can be fed into deep learning models. In this work, we wish to do the same for binaries.

3.2 Problem Statement

Problem definition. Given a query procedure q and a large collection T of (target) procedures, in a stripped binary form (without debugging information), our goal is to quantitatively define the similarity of each procedure $t \in T$ to the query q. We require a method that can operate without information about the source code and/or tool-chain used in the creation of the binaries. The main challenge is to devise a prediction method that is precise enough to avoid false positives, flexible enough to allow finding the code in

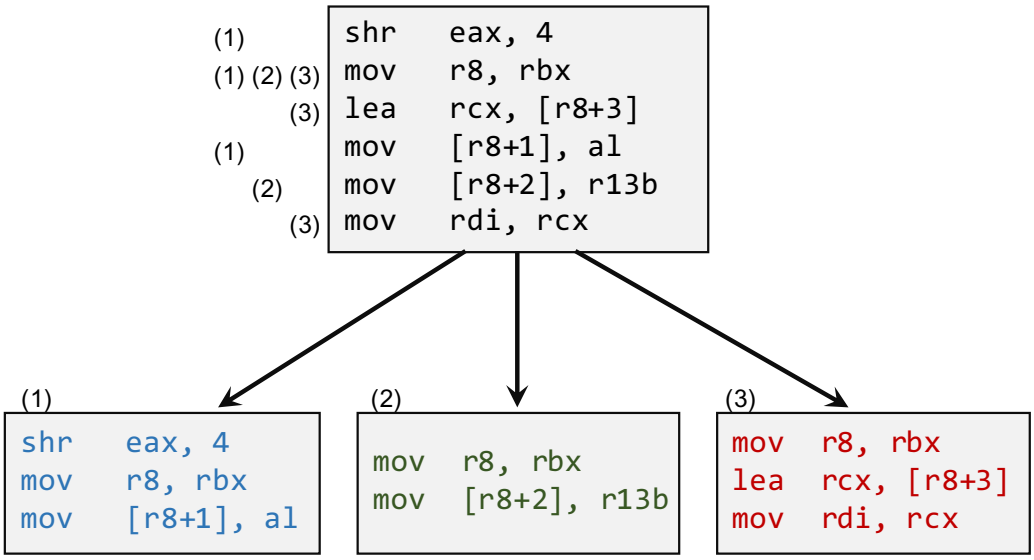


Figure 3.1: Decomposition to strands.

any compilation configuration, and fast enough so similarity search can be performed on huge corpora or memory regions in a timely manner.

Metrics. For evaluation, we conduct all vs. all classification experiments. That is, we compile each project P in our test set using multiple compilation configurations, thus generate the binary procedures $\{p_1, p_2, ..., p_n\}$. Next, we perform n^2 predictions; each prediction corresponds to the probability that procedure p_i is similar to procedure p_j. Overall, for each project we output n lists; each list corresponds to a procedure and contains n probabilities. The accuracy of each of these lists is measured by Concentrated ROC (*CROC*) [127], a standard tool in evaluation of threshold based classifiers.

3.3 Lava Overview

In this section we provide an overview of our algorithm and system design. We begin in Section 3.3.1 by reviewing the concept of a code strand and giving intuition about our approach. Next, in Section 3.3.2, we describe proc2vec, our algorithm for representing code sections as vectors. Finally, in Sections 3.3.3 and 3.3.4 we cover our data generation and ML classifier model design, respectively.

3.3.1 Strands as Features

Strands

We adopt the notion of *strands* introduced in [36] as a building block in our algorithm. A strand is the set of instructions from a code block that are required to compute the

value of a certain variable. Figure 3.1 shows an example of a decomposition of a code block into its composing strands. Note that a single instruction can be associated with multiple strands within a code section. For example, the second instruction of the code block in Figure 3.1 belongs to all three strands that compose the code section. Note that two syntactically different strands can be equivalent, and that a strand is not necessarily syntactically contiguous.

Intuition

The intuition behind our proc2vec algorithm is based on the similarity by composition principle [16], according to which two signals are similar if it is easy to compose one signal from large contiguous chunks of the second signal. The similarity by composition principle has been proven to be valuable for similarity detection in image processing [16] and program analysis [36].

Therefore, in this work we wish to use the strands that compose a code section as its feature set. We transform strands to numbers, and assemble those numbers to form a vector that represents the corresponding code. In order to learn the prevalence and co-occurrence of virtually contiguous chunks of code (strands), we train a machine learning model. Thus, code sections that contain rare strands or rare combinations of strands, are likely to be more unique and get a higher similarity score. Likewise, common strand combinations are likely to have the opposite effect. For example, compiler-induced strands (e.g. stack handling operations) are very prevalent and indeed irrelevant for similarity matters.

Previous work [37, 111] has put effort in detecting compiler-induced binary code and artifacts. In our work, we achieve this by employing ML as detailed in this section.

3.3.2 prov2vec

In this section we describe proc2vec, our algorithm for transforming procedures or code sections to vectors. As described in Figure 3.2, given the assembly code of a procedure (Figure 3.2a), the technique for transforming it to a vector is comprised of five steps:

1. First, we split the procedure to basic blocks (Figure 3.2b). A basic block is determined according to the `jmp` instruction placements in the procedure binary code.

2. As depicted in Figure 3.2c, we further decompose each basic block into *strands*. In the figure, different strands have different colors, while instructions that belong to few strands are marked with all the associated colors.

3. Next, we bring syntactically different strands with same semantic meaning to the same textual representation. For that purpose, we use techniques introduced in previous work [37] and optimize and canonize each of the strands. As shown in Figure 3.2d, this step changes the strands' representation to a canonical one, so

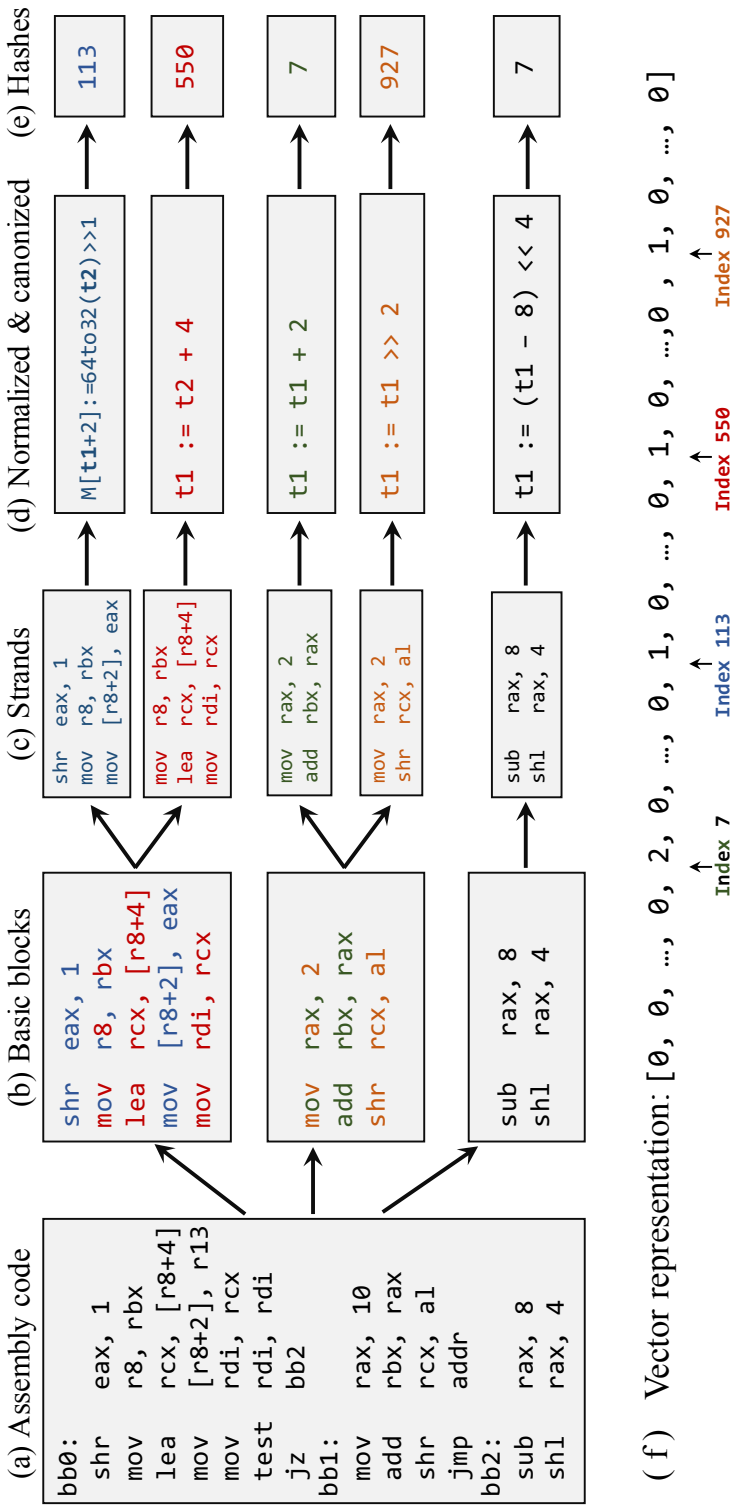


Figure 3.2: Procedure to vector transformation stages.

subsequent additions are grouped, multiplications with power-of-two are replaced by shifts, arithmetic operations are reordered to a consistent representation, etc.

4. We apply b-bit MD5 hashing on the textual representation of the strands; thus translating each strand to an integer (Figure 3.2e) in the range $\{0, 2^b - 1\}$. Note that if b is small then collisions are likely to happen, thus different strands can be mapped to the same hash value.

5. Finally, as depicted in Figure 3.2f, we use the resulting integer set as indexes and build a vector of length 2^b whereby each element equals to the number of times that the index of the element appears in the set. Hence, the vector elements can be larger than one, and the sum of the vector elements is equal to number of strands that the corresponding procedure contains.

For implementing the above algorithm, we use the PyVEX [121] open-source library. We employ its binary lifter in order to lift the assembly code into `VEX-IR` and slice it to strands (step 2). We further exploit the VEX optimizer on each of the strands for bringing them to a normalized representation (step 3).

Note that the lifting process from assembly code to IR may occasionally fail. This mostly happen due to esoteric instructions that do not have representations in VEX-IR, such as Intel's AES instructions. In such cases, we ignore the strand that contains the problematic instruction and resume the transformation to vector without considering that strand.

Moreover, in order to further assist the transition to IR, we sometimes preprocess the assembly code. For example, ARM processors do not provide a fully automatic subroutine call/return mechanism like other processors, but instead use the Branch and Link instruction. This type of function calls in not known to the PyVEX mechanism and prevents it from making a correct transition to intermediate representation. Hence, in such cases we preprocess the code and instrument it with call instructions in order to aid the lifting process.

Finally, we note that we skip basic blocks that contain more than 5000 (empirically set) lines of code. This is done for two main reasons: (1) Skipping such huge blocks improves the throughput of our approach. (2) The chances for similarity between such huge blocks of code are extremely small, while on the other hand, these can contain huge number of strands that may drastically change the values of the resulting vector elements, and as a result harm the data quality.

3.3.3 Data Generation

For generating the data, we take various open-source projects (Table 3.3a) that we find in the wild and compile them for different architectures using different compilers types, versions and optimization levels (Table 3.3b). Next, we apply our proc2vec algorithm on each of the resulting binary procedures, thus creating a list of vectors, each of which

Training Set	
Application Name	**Version**
binutils	2.3
OpenSSL	1.0.1
bash	4.3
httpd	2.4.33
ntp	4.2.8
cURL	7.60.0
Snort	2.9.11.1
Git	2.9.5
util-linux	2.32
Apache Mesos	1.5.0
QEMU	2.12.0

Test Set	
Application Name	**Version**
tar	1.30
FFmpeg	2.7.1
Wireshark	1.10.10
coreutils	8.29
bzip2	1.0.6
wget	1.15

(a) Open source projects used for data generation.

Compiler	Versions	Architecture	Optimization Levels
`gcc`	`4.{7,8,9}`	`x86_64`	`-O{s,0,1,2,3}`
`icc`	14, 15	`x86_64`	`-O{s,0,1,2,3}`
`Clang`	`3.{5,6,7,8}`	`x86_64`	`-O{s,0,1,2,3}`
`gcc`	4.8	`AArch64`	`-O{s,0,1,2,3}`
`Clang`	4.0	`AArch64`	`-O{s,0,1,2,3}`

(b) Compilers and versions used for data generation.

Figure 3.3: Data set properties.

represents a procedure in some specific compilation setting. By stacking all these vectors we build the matrix M.

Using our prior knowledge about the origin of each vector, we build a *match-list*. The match-list contains pair tuples that represent indexes of matching rows in M. That is, if the pair (i, j) appears in the match list, then the corresponding vectors M_i (i'th row of M) and M_j are originated from the source code of the same procedure. Furthermore, in order to teach our classifier that every procedure is similar to itself, and that the input order of a pair has no importance we augment each matching pair (x, y) to a foursome consisting of the pairs (x, y), (x, x), (y, y), (y, x). We label these pairs by 1 (match) and use them to generate our positively labeled dataset.

For generating negative examples, we generate random pairs of vectors that originate

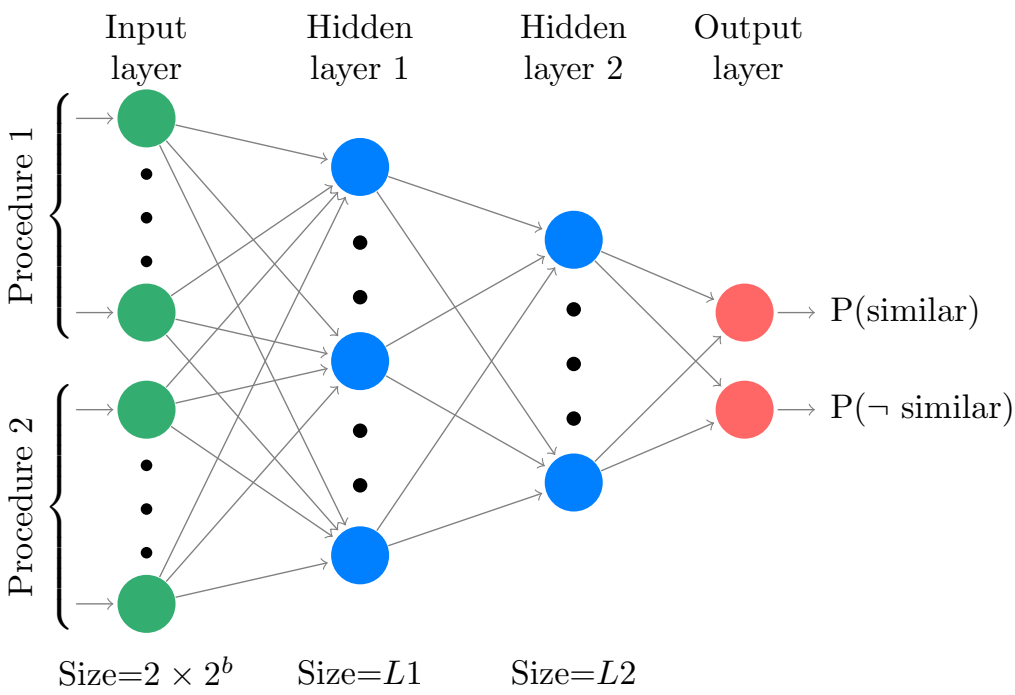


Figure 3.4: Lava's neural network skeleton.

from non-equivalent procedures. We make sure that each of these pairs does not appear in the match-list, label the pairs by 0 (non-match) and use them as our negatively labeled data-set.

A similarity between code sections is relatively a rare event and we address that in two ways: (1) we use the CROC metric, which is designed for evaluating unbalanced classification problems, for estimating the classifier's performance. (2) We generate a strongly unbalances dataset and examine different ratios between the number of the negatively and positively labeled examples. After exploring various ratios, we set the ratio to 6.

Finally, we build our full database by shuffling the union of the positively and negatively labeled data-sets. Overall, we get a database with about twenty million examples.

Note that our data generation process can possibly generate erroneous records, as similar procedures that originate from different projects are labeled as non-matches. This adds noise to our data and might affect our results in two ways. First, in the training phase, it possibly reduces the quality of the resulted model. Second, in the testing phase, we might measure a lower accuracy than the real one, since true similarities that are detected by our model might be labeled as non-matches. In order to minimize the influence of this noise on our model, we train it using numerous amount of data, thus reducing the effect of such outliers.

3.3.4 Designing an NN Classifier

For estimating the similarity score of a pair of vectors, we build a deep learning classifier using TensorFlow. Our model is comprised of an input layer, two fully-connected hidden layers, and a softmax output layer, as shown in Figure 3.4. The input layer is comprised of 2×2^b neurons, corresponding to the two vectors that represent the two input procedures. The first and the second hidden layers are of sizes $L1$ and $L2$, respectively. Finally, the output is a 2-neuron softmax function, representing the probability of similarity between the code sections that generated the input vectors.

Except for the output layer, all the activation functions in the network are `tanh`. We train the model using cross-entropy cost function, dropout regularization of 0.1, batch size of 32 and 3 passes over the data (epochs).

3.4 Evaluation

We evaluate our approach and compare our results with GitZ [37], the fastest state-of-the-art binary similarity search tool [92]. We start in Section 3.4.1 by exploring various hash size values and fit for each value a corresponding classifier. Next, in Sections 3.4.2 and 3.4.3 we present the accuracy and throughput results of our cross platform similarity predictor, respectively. All experiments were performed on a machine with four Intel Xeon E5-2640 processors, 368 GB of RAM, running Ubuntu 16.04.1 LTS.

3.4.1 Hash Size

MD5 processes text into a fixed-length output of 128 bit; however, a 2^{128}-length vector is much longer than we need. Therefore we fold the hash values (using modulo) thus de-facto reducing the size of the hash. The hash size determines a trade-off: high values (longer input vectors) mean longer representation for each procedure, which may cost in over-fitting and increased training time. On the other hand, low values (shorter input vectors) might create collisions in the procedure representation and omit imperative information from the classifier (underfit).

For finding the optimal hash size, we explore various values and fit a classifier that performs best for the each of the examined values on a toy dataset. The toy dataset consists of code drawn from four applications (OpenSSL, git, util-linux and Apache Mesos) which we compile using all our `x86_64` compilers (see Figure 3.3b) with the `-O2` optimization level. The parameters that we fit are $L1$, $L2$, and dropout rate.

We test each classifier on three test projects (coreutils, wget, tar) by conducting an all vs. all experiment and present the results in Figure 3.5. Note that the hash size axis is in units of bits; namely, a value of h implies a vector representation of length 2^h. As can be inferred from the figure, for all of the projects in our test set, using 10 bits for performing the MD5 hashing yields the best results; namely, 2^{10} is the optimal length for a vector representation of a procedure under such a neural network setting.

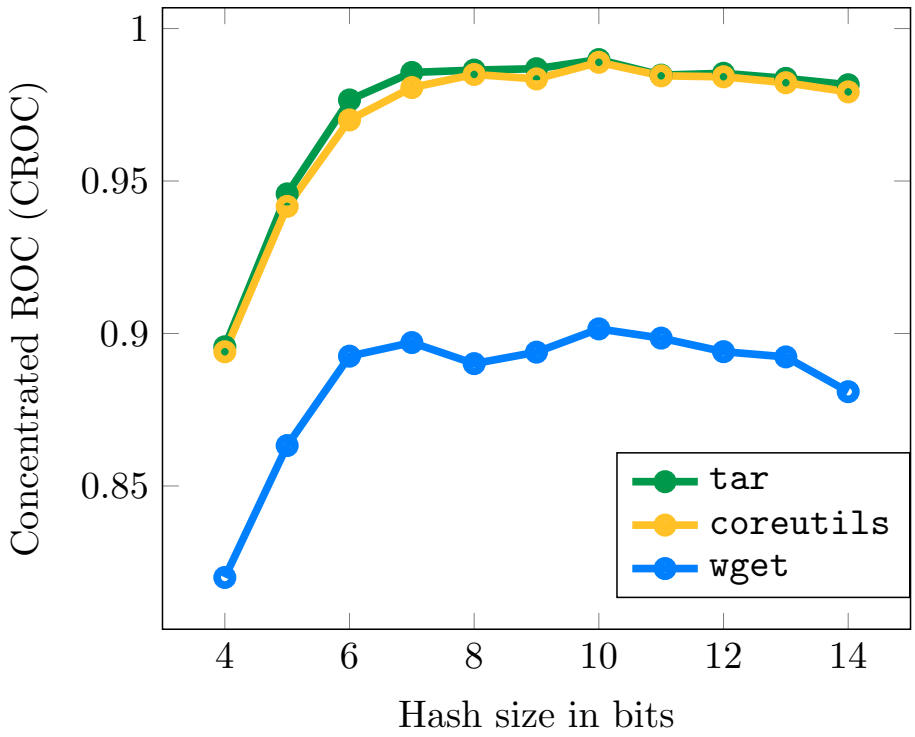


Figure 3.5: Hash size exploration results.

3.4.2 Cross Platform Similarity Predictor

Given the results of the previous section, we choose a 10-bit MD5 hash size, configure our neural network with the parameters that optimized the performance of 2^{10} procedure representation ($L1 = 512$ and $L2 = 128$), use data gathered from all of the projects listed in Table 3.3a, and train a cross-{compiler, version, optimization} binary similarity predictor.

For evaluation, we conduct an all vs. all experiment for each of the projects in out test set and present the results in Figure 3.6. We repeat the all vs. all experiment using GitZ and present the results alongside Lava results. As can be seen, Lava achieves higher average CROC score in all of the tested projects. Note that comparing to the previous section the results for `wget` are improved. We attribute that to the larger and more diverse training set.

As discussed in 3.2, our tool outputs the similarity probability (ranging from 0 and 1) between the query procedure and each of the target procedures. Hence, each query procedure gets a list of similarity grades. The CROC score incorporates the accuracy of our predictor regardless of a specific threshold value, while a score of 1 means that our predictor perfectly managed to rank the similar procedures higher than the non-similar ones. For illustration, by setting a predefined threshold of 0.5 for the `coreutils` experiment, 47% of the query procedures are classified with perfect accuracy. The rest

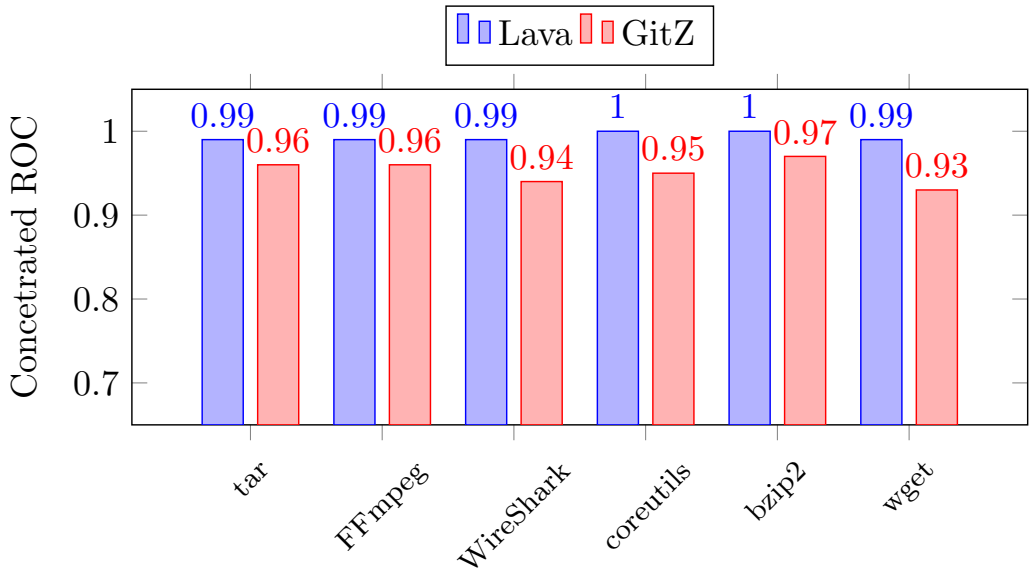


Figure 3.6: CROC score of the all vs. all experiments, for each of the projects in the test set.

of the query procedures achieve accuracy higher than 99%. We stress that the threshold value of 0.5 may change, and that our tool only outputs probabilities which should later be examined by human eyes, as in other early retrieval problems [127].

3.4.3 Throughput and Latency

Our neural network can output approximately 7000 predictions per second on a single core (excluding the training time, which is performed only once). Furthermore, since the procedure vectors are independent and the model is small enough, the prediction generation can scale with the number of available processors. GitZ, on the other hand, is able to output about 25 predictions per second, hence Lava introduces a speedup of more than 250X over the state-of-the-art fast similarity detection tools. Note that approaches that achieve near-perfect accuracy like [36] needs about 5 seconds for a single comparison. In terms of latency, comparing to GitZ, Lava has a shorter preprocessing pipeline, as it does not require lengthy translation from `VEX-IR` to `LLVM-IR`, thus improving also the latency in about 15%.

3.5 Future Work: inst2vec

Our proc2vec algorithm has a noticeable disadvantage – the md5 hash can map relatively close strands to very different hash values, thus possibly losing semantic meaning. We wish to find a way for representing code sections as vectors that also completely encapsulates

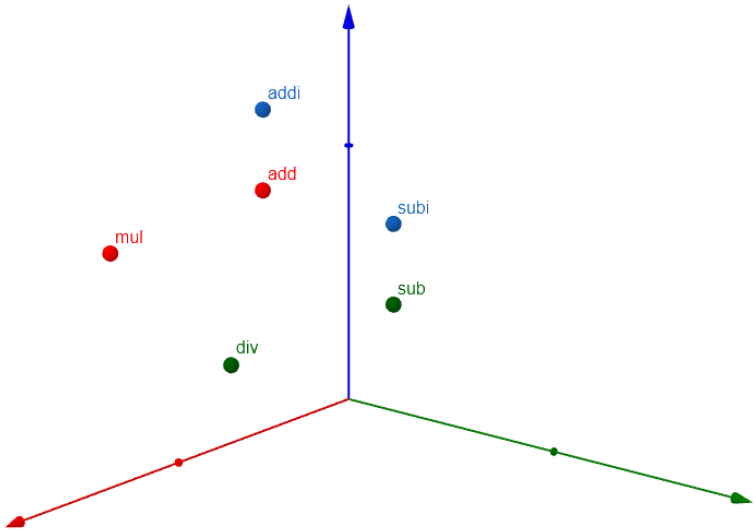


Figure 3.7: Instruction to vector embedding example.

the semantic meaning of the code section (demonstrated in Figure 3.7).

As algorithms for estimating word representations in the vector space (word embedding) [89, 90] are improving and gaining more attention, we believe that such techniques can be applied for assembly instructions. Moreover, having such an embedding for instructions can allow for applying NLP techniques on code sections by treating instructions as words and procedures as sentences. For example, based on the instruction vectors we can use algorithms for sentence embedding for finding efficient representations for whole procedures while also capturing the full semantic meaning of the procedure.

That said, there are major differences between natural language and assembly instructions. For example, in natural language, the number of words in the vocabulary is bounded and known beforehand. On the other hand, assembly instructions may contain constants and memory addresses which practically have infinite number of possible values. Furthermore, a procedure is usually much longer than a sentence.

We believe that any instruction embedding algorithms will have to consider these differences. A possible approach would be to preprocess the instructions by replacing constant values by placeholders and by binning memory addresses. Such a preprocessing will bound the number of possible instructions and mediate the differences between natural language and code. Embedding algorithms like word2vec typically use a surroundings radius of three or five for learning the word representations. For handling the longer procedure size (comparing to sentence size), embedding algorithms might consider using a larger surroundings radius, thus achieving a representation that captures the longer nature of procedures.

3.6 Summary

We present an approach for detecting binary similarity using the similarity by composition principle alongside machine learning. We devise proc2vec, a way to represent code sections by vectors, without applying human-crafted feature extraction processes. We show that our approach achieves high accuracy and throughput, thus it is practical for use in real world scenarios.

Chapter 4

CSR: Core Surprise Removal in Commodity Operating Systems

In today's economies of scale, which rely upon adding more and more servers with cheaper hardware, the observation that failures are the norm rather than the exception [25, 49] has become a guiding light for system designers. Nowadays, constant monitoring, fault tolerance, and automatic recovery are an integral part of any large cloud or data-center system. However, these are mostly focused on machine failures rather than low-level hardware failures such as processor faults.

At the same time, the number of cores per chip constantly increases. Modern consumer machines already contain 32 cores, data center servers include more than 60 cores per machine, while tera-devices with hundreds or even a thousand fault-prone cores are the subject of active research [50, 133]. This was made possible due to constant shrinking in transistor size and voltage supply. However, with hardware shrinking, the probability of physical flaws on the chip significantly rises [17, 34], and chances for processor faults become substantial [125]. With many tens or even hundreds of cores per chip, core failures can no longer be ruled out. Recent studies show that, even on consumer machines, hardware faults are not rare [94], and memory errors are currently dominated by hard errors, rather than soft-errors [119]. We therefore believe that, with technology advances, tolerating failures of individual cores shall become inevitable.

Current operating systems, including Linux and Windows, crash in the face of any permanent core fault and most chip-originated soft errors. As we explain later, the reason for the crash lies in the various kernel mechanisms that require the collaboration of the faulty core. Thus, the failure of a single core, out of hundreds in the foreseeable future, brings down the entire system. Cloud systems, for example, usually consolidate multiple virtual machines on a single server in order to improve its utilization [15, 67, 73, 114, 124]. In such settings, the failure of a single core crashes all the VMs running on the server.

Our goal in this work is to fortify existing operating systems against unexpected core failures, and in particular, allow processes – and VMs – on other cores to operate without

interruption. We further elaborate on our goals and design considerations in Section 4.1.

In Section 4.2, we present *CSR*, a strategy for overcoming *Core Surprise Removal* in commodity operating systems. CSR is designed for multi-core architectures with reliable shared memory, and incurs virtually no overhead when the system is correct. Its primary objective is to keep the system alive and running after a core crash, while terminating at most the one user program that had been running on the faulty core. Recovery from that user program's failure can be handled, e.g., using checkpointing mechanisms [43], by application developers, and is out of this work's scope.

CSR's basic fault recovery function overcomes all core failures that happen when the faulty core is either in user-mode or executing non-critical kernel code. In order to cope with failures inside kernel critical sections, we propose in Section 4.3 a complementary technique based on *lock elision* using *Hardware Transactional Memory (HTM)* [57, 107], which is already incorporated in modern commodity processors [32, 144] and HPC systems [136]. In addition, CSR has a modular design, and it employs deferred execution of the recovery process in order to minimize the vulnerability to cascading failure scenarios, as discussed in Section 4.4.

In Section 4.5 we provide a proof-of-concept implementation of CSR in version 3.14.1 of the Linux kernel. We further exemplify the use of Haswell's HTM support[1] for recovery from failures in critical kernel code. HTM prevents critical sections' intermediate values from being written to shared memory, thus avoiding inconsistent states following a crash.

In Section 4.6 we show how CSR can be leveraged for coping with soft-errors. We analyze the current Linux machine check exception handler and change it, making it revert to CSR upon a fatal soft error detection instead of entering panic mode.

In Section 4.7, we evaluate CSR via three different experiments. First, we use a virtualized environment to inject thousands of permanent failures at random times. Since our emulated VM does not support HTM, we run CSR in this experiment without using HTM protection for critical kernel sections. We show that, while an unmodified kernel persistently fails, CSR successfully recovers from all failures occurring in user or idle mode, and in roughly 70% of the faults injected during execution of kernel code.

Next, we inject permanent core failures on real systems, during the execution of a hundred different critical and non-critical kernel functions. In these experiments, we use CSR with lock elision, and it successfully recovers from 100% of the faults. Finally, we build a fault-resistant cloud setting by installing CSR on the host OS of a 64-core server over which multiple virtual machines run unmodified Ubuntu. We show that following a core fault, only one VM crashes, while the others resume normal execution.

To conclude, our contributions in this work are:

1. We present CSR, a strategy for overcoming permanent core failures.

2. We propose using HTM for tolerating processor faults that happen during execution

[1] The recently found erratum in Intel-TSX instructions did not affect our experiments. Our work comprises a prototype, and the concept of using HTM for our purposes shall remain applicable in future technologies.

of critical kernel code.

3. We implement CSR in Linux, as well as a proof of concept for using HTM for recovery.

4. We evaluate CSR by injecting numerous faults on real and virtualized environments.

4.1 Design Considerations

We design CSR to overcome core failures, whereby a core stops working without advance notice. Our goal is to take a system from an arbitrary state induced by a failure to a *recovered* state, where the following holds: (1) Interrupts are routed to online cores only. (2) All threads, as well as all types of delayed kernel tasks, are affined to online cores only. (3) All kernel services and subsystems operate using only online cores.

In this section we first argue that core failures merit a different treatment from CPU Hot-Plugging (Section 4.1.1), and then detail our failure model (Section 4.1.2) and the extent to which we can overcome failures in kernel code (Section 4.1.3).

4.1.1 Why Not CPU Hotplug?

Modern commodity operating systems usually include a *CPU Hotplug* [88, 91] mechanism, which allows one to dynamically plug or unplug a core. Ideally, we would like to simply unplug a faulty core once a failure is detected. However, there is a profound difference between voluntary CPU unplugging and on-line removal of a faulty core: CPU unplugging can use the cooperation of the victim core, thus enabling a completely supervised removal process, including choosing the correct moment to commence the removal, asking the victim core to perform some local tasks, and reading the values stored in its registers. In contrast, a failure can happen any time, depriving the crashed core of the ability to communicate, and leaving its buffers' and registers' contents unknown. The latter suggests that it is impossible to know what the core was doing at the moment of the failure.

4.1.2 Fault Model

We consider cores prone to *permanent faults* [100] – irreversible physical faults that cause consistent failure. This captures cases such as permanently open or shortcut transistors, slow components that cause timing violations, or a partially burned-out chip [105].

CSR ensures recovery from failures that happen while the faulty core executes in user-mode; other cores can be either in kernel or user mode. Failures in kernel code are partially handled, as we discuss in Section 4.1.3. CSR is only concerned with the system's survival, whereas application-level recovery is out of scope.

When dealing with core failures, one has to define their scope, namely, which hardware components are affected by the failure. Some designs of future many-core machines [50, 137, 138] assume that the entire internal state of a faulty core (including registers and

buffers) may be flushed upon failure. We do not require this in this work, but rather allow a core's internal state (buffers and registers) to remain unavailable following a crash. We do, however, assume that core malfunction does not destroy its cache. Hence, following a core's failure, it is possible to flush its private cache. We use such a flush in order to ensure kernel data consistency in the following scenario: a kernel thread updates some OS data structure; next, the core switches to user code, but the changes it had made to kernel data have propagated from its local cache to the shared memory only partially; then the core fails. If one wishes to forgo triggering a cache flush following failure without sacrificing consistency, other solutions are possible. For example, one can configure kernel pages as non-cacheable, or flush the cache in kernel-to-user mode transitions. Note that fault-tolerant cache technologies are currently emerging [72, 61], and it is not unlikely that future caches will provide fault-tolerance features.

For detecting failures and triggering the recovery process, we assume the existence of a reliable *Failure Detection Unit (FDU)*. This can be implemented in hardware, using heartbeats and Machine Check Architecture [60, 65, 4], as suggested in previous works [44, 137, 138]. Upon fault detection, the FDU (1) halts the faulty core, thus preventing it from corrupting shared memory; (2) triggers a flush of its private cache; and (3) reports the error to the OS. Note that a hardware-based FDU implementation can easily provide these requirements. In this work, we simulate the FDU in software using heartbeats (see Section 4.5.1).

Given such an FDU, we design our core surprise removal strategy to cope with a *fail-stop* [118] model. Namely, a faulty core simply stops executing from some point onward.

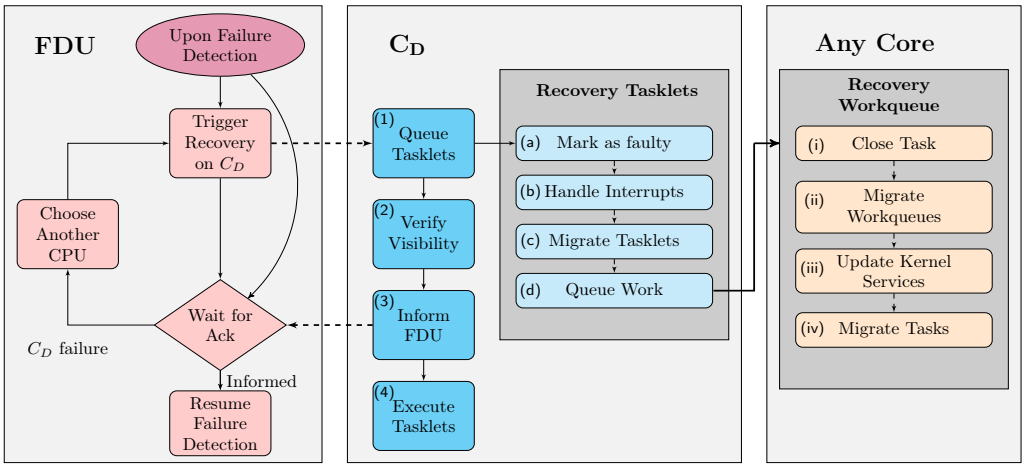
CSR is designed for operating systems running on multi-core architectures with reliable shared memory. Memory failures are typically addressed using other, complementary, mechanisms, such as error-correcting codes and relocating data [28, 113, 143]. By building upon reliable shared memory, we forgo the need for checkpointing.

4.1.3 Failures in Kernel Code

Unlike a failure in user-mode, certain failures in kernel-mode might be impossible to recover from by simply terminating the running thread. For example, failure during execution of a critical section might lead to data inconsistency. Failures in kernel code also include *cascading failure* scenarios, whereby the core that executes the recovery procedure crashes before completing the recovery process.

We address kernel-mode failures in two complementary ways: First, we design our algorithm to withstand cascading failures, while striving to minimize the time interval during which a cascading failure may interfere with an ongoing core removal. To this end, we use a strategy of queueing work for later execution. This allows us to migrate the recovery work to other cores in case the core that executes the recovery also fails.

Second, we propose the use of HTM for executing kernel critical sections. This prevents propagation of partial updates resulting from unfinished execution of critical sections to the shared memory. Recall that upon failure detection, the FDU triggers a flush of the



(a) Failure Detection Unit flow. (b) Recovery procedure on recovery core C_D. (c) Delayed tasks, execute on any core.

Figure 4.1: Recovery flow chart. Dashed lines represent message passing; solid lines represent flow.

faulty core's cache; here, we assume that the system does not flush uncommitted values.

We note that our proposed solution to failures in kernel code is a best-effort one. It guarantees recovery from failures during execution of code that does not access hardware different from RAM, and therefore can be protected with a transaction. Moreover, there are instructions that cannot be executed transactionally, such as a TLB flush, accessing a control register, an IO operation, etc. [64]. We cannot guarantee recovery from failures during execution of code sections containing such instructions.

4.2 Core Surprise Removal

The current section presents the heart of CSR's recovery approach, and focuses on the common case – failure during execution of user-code or non-critical kernel code. The next section addresses failures during critical kernel code, and in Section 4.4 we discuss cascading failures.

4.2.1 Background: OS Mechanisms Used

Our solution utilizes a few common facilities that exist in modern operating systems. We begin with some background and discuss the relevant aspects of these kernel mechanisms.

Deferrable Functions

Operating systems usually support multiple types of delayed tasks for performing work with different urgencies and execution contexts. For readability, we use the Linux termi-

nology for the different delayed tasks types. CSR employs two types of delayed kernel tasks:

Tasklets [83] (Deferred Procedure Calls in Windows), which have three important properties: (1) They always run in interrupt context, and as a consequence, are unable to sleep or block. (2) A tasklet always executes on the core that schedules it. (3) The kernel services all pending tasklets immediately after handling all pending hardware interrupts.

Work-queues [83] (Asynchronous Procedure Calls in Windows), which, in contrast to tasklets, are executed by kernel threads in user context, and therefore are allowed to sleep or block. Furthermore, a work-queue may be bound to a specific processor, or may be unbound, in which case it can be executed by any core.

CPU States and Hotplug

Operating systems manage the state of all cores. For example, Linux uses bitmaps, dubbed *CPU masks*, for each possible state. Each CPU mask represents one state, and contains one bit per core. These masks are accessed globally and used by the kernel in various cases, such as for iterating over per-core data structures.

Linux, for example, defines, among others, an *online* state and an *offline* state. An offline core is not available for scheduling and does not receive or handle interrupts; it is in deep sleep mode, and consumes low power. A core's state can be dynamically changed between online and offline by using the CPU Hotplug mechanism. The CPU Hotplug feature has various uses [6, 91]. However, as explained in Section 4.1.1, CPU Hotplug cannot be used for disabling a faulty core, as it does not consider any reliability issues.

4.2.2 CSR Data Structures

We augment the kernel with two new data structures. First, we define a new processor state, *faulty*, and a corresponding global CPU mask, *faulty_mask*. This mask is the first to be updated upon processor failure, and, as its name suggests, indicates which cores in the system are in a faulty state. The faulty state serves two purposes at two different time intervals:

- While the system is not in a recovered state, namely, following a failure and before CSR's removal procedure has completed, the corresponding bit in the *faulty_mask* indicates to kernel code that hasn't yet adjusted to the failure to treat this core as unavailable for scheduling, migration, updating its data structures, etc.
- After the removal procedure finishes, a core marked as faulty is treated by the kernel as offline, with the exception that it is not permitted to come back online.

Second, we add a new globally accessed work-queue, *recovery-workqueue (RWQ)*. This work-queue is not bound to any processor, namely, different processors can pull its work items and execute them. We use RWQ to defer execution of certain recovery functions to

a later time and a different execution context, in order to shorten the time interval during which a cascading failure may require re-execution of the recovery process. Functions queued to this work-queue are less urgent or might need to sleep, and so they are not queued as tasklets.

4.2.3 Failure Detection Unit

The FDU's operation is shown in Figure 4.1(a). When a failure is detected, the FDU invokes CSR's *recovery procedure* on a designated non-faulty online core, denoted C_D, which in turn handles the crash. The selection of C_D prefers cores that reside in a different socket in order to reduce the probability for a cascading failure. The ensuing flow in C_D, depicted in Figure 4.1(b), is described is Section 4.2.4 below.

After invoking the recovery process on the designated core, the FDU waits for an ack. Meanwhile, if the FDU detects a cascading failure of C_D, it invokes the recovery algorithm again, on another processor, for handling both the first failure and the new one. Once the FDU receives an ack from the designated core, it considers the recovery process done, and resumes normal operation with the faulty core, C_F, removed from the available processors list.

Note that this ack does not function as a keep-alive message. Its purpose is to notify the FDU that the recovery process has reached a point from which its completion is guaranteed; in case of a cascading failure before that point, the FDU cannot know what stages C_D completed before the failure.

4.2.4 Recovery Procedure

High Level Operation

C_D begins the recovery procedure, as presented in Figure 4.1(b). C_D's role consists of four primary steps: (1) First, it enqueues four new tasklets for repairing the system's state. These include Hotplug-like operations of modifying CPU masks, resetting interrupt affinities, and migrating tasklets from the removed core, as well as recovery-specific actions that address the surprise nature of the removal. The tasklets are only queued at this point, and are not executed yet. Queuing the tasklets rather than executing them reduces the time interval during which a cascading failure of C_D causes re-execution of the recovery process. (2) Next, C_D takes actions that assure the visibility of the tasklets to the rest of the cores (e.g., flush its write buffer). Once they are visible, other cores can steal them, and so the recovery operation is guaranteed to complete. (3) At this point, C_D sends an ack to the FDU. (4) Subsequently, C_D naturally turns to execute the tasklets found in its tasklets queue in FIFO order, as part of the kernel's normal behavior.

Recovery Work

Recovery Tasklets. The actual core removal process begins with the execution of four tasklets (depicted in the grey box in Figure 4.1(b)), which we subsequently call *recovery tasklets*. They perform the following:

(a) The first tasklet marks C_F's state as faulty in the corresponding CPU mask. Once a core is marked as faulty, kernel services will treat it as if it is out of the online map. Namely, no tasks will be scheduled or migrated to its run queue, iterations on various CPU masks will skip it, etc. The longer this step's execution gets postponed, the further the system might diverge from a recovered state, e.g., new tasks might get attached to C_F's run-queue.

(b) The next tasklet deals with interrupts. Modern operating systems use SMP IRQ affinity to assign interrupts to specific cores – this means that interrupts that were affined to the faulty core might have been lost. Therefore, to minimize the time interval during which interrupts can be lost, we reset their affinities early in the recovery process.

Next, we deal with interrupts that may have been lost. The loss of an interrupt can cause errors in software components that depend on its arrival. For example, a lost interrupt originating in the NIC may prevent network packets from being delivered to an application. Even worse, a lost Inter-Processor Interrupt (IPI) might prevent imperative kernel procedures from being executed and possibly crash the system. To recover from possible interrupt loss, we send spurious interrupts for all interrupt types that were previously routed to C_F (a technique that was employed in previous Linux versions). In addition, we check if any of the pending functions in C_F's IPI queue need to be migrated, and migrate those that do.

(c) The third tasklet migrates tasklets that are attached to C_F. This has to be performed urgently for two main reasons: first, tasklets embody high priority kernel work that is important to system maintenance and functionality. The second reason lies in our support for cascading failures, as we explain in Section 4.4 below.

(d) The last tasklet queues additional work that is essential to the recovery process but cannot run in interrupt context, or can endure a short delay in its execution. We divide this work into a number of functions, which we queue to our recovery-workqueue. This work will be executed later by dedicated kernel threads. Most of the work items queued at this step are needed in every operating system, but some of them are OS-dependent.

Queued Work. The following work-items are queued:

(i) Close the running task. We close the task that was running on C_F at the moment of the failure and free its resources. Since we are unable to communicate with C_F, significant data such as the instruction pointer and the register file content is lost. Therefore, we cannot recover the running application's state. It is possible to recover from such failures using a checkpointing mechanism, which could be done at application level.

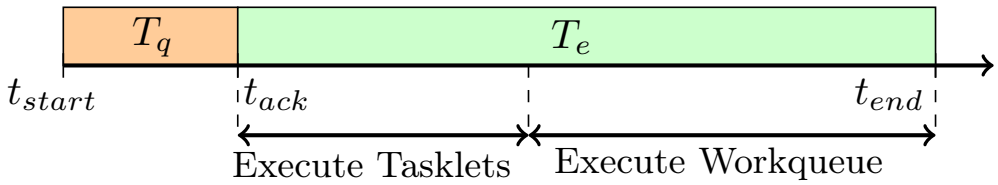


Figure 4.2: Recovery Periods.

(ii) Migrate work-queues affined to C_F. As in the tasklets case, we migrate work-items from the work-queues affined to C_F for later execution on other cores.

(iii) Update kernel services. This step is OS-dependent. Here, we update kernel services that might be affected by the sudden departure of C_F from the online map, such as performance events, synchronization services, and memory allocation.

(iv) Migrate C_F's tasks. User processes and kernel threads that were attached to the run-queue of C_F at the moment of the failure will usually still be there (unless the load balancer moved some of them); these must be migrated to other run-queues. The target run-queues for migrated tasks can be chosen in a variety of ways, but this choice is inconsequential as it only has a short-term effect – until the load-balancer kicks in. For the sake of simplicity, CSR moves these threads to the run-queue of the lowest-id correct core, leaving it up to the load-balancing mechanism to correct the overload that might be temporarily formed at the target core.

As in a regular CPU-unplug process, the migration of the run-queue is performed at the end of the removal process, as it does not affect system correctness. Its delay only blocks the execution of the tasks in C_F's run-queue until the end of the recovery process.

It is worth pointing out that CSR's recovery process subsumes the set of recovery actions performed by CPU-Hotplug. Specifically, the following steps are common, at least partially, to both CSR and CPU-Hotplug: (b), (c), (ii), (iii), and (iv). Nevertheless, tasks that perform analogous logical roles in both mechanisms differ in their implementation and order of execution, as CPU Hotplug exploits the cooperation of the unplugged core, and CSR cannot. After the execution of all the above, the system is again in a correct state, as all actions that would have been needed by a hot-unplug operation have been performed.

4.3 Failures in Kernel Code

Kernel code execution usually comprises a small fraction of the total system's runtime. However, a failure in kernel code might have severe consequences. For example, a failure during a migration operation may cause a task to be dequeued from the source queue without being enqueued at the target one. Moreover, a crash of a core holding a kernel lock may leave the system in an inconsistent, or even unrecoverable, state.

In order to prevent the implications such crashes may inflict, we propose to elide the locks [107] protecting kernel critical sections using HTM; HTM mechanisms cause changes to multiple memory locations to appear to be atomic, while providing isolation between parallel executions [57]. We assume that these properties still hold in the presence of failures, namely, the HTM prevents uncommitted changes from propagating, even when a failure occurs during a transaction. Thus, in case of a failure inside a critical section, no changes are written to shared memory, and the system remains in a consistent and recoverable state. Rossbach et al. [112] have proposed TxLinux, an operating system that exploits hardware transactional memory in kernel code for handling concurrency and improving performance. We, on the other hand, exploit HTM abilities in a similar context, but for the purpose of reliability.

Kernel developers devote effort in order to reduce contention among cores. To this end, each core has its own kernel data structures, and accessing another core's data is relatively rare. Therefore, transaction aborts due to data conflicts are likely to be infrequent. Nevertheless, since Intel Transactional Synchronization Extension (TSX) [64] is a *best effort* HTM, (namely, transactions are not guaranteed to commit because of various architectural reasons), and since kernel code executes many sensitive instructions, such as interrupt masking, interrupt sending, TLB and cache operations, which cause aborts, we expect transactions to abort occasionally. Furthermore, due to architectural reasons, some critical sections cannot be executed transactionally, e.g., a context-switch always causes a TLB flush – an operation that leads to abort. In such cases, our solution reverts to using locks. Using HTM in all remaining critical sections keeps the system fault-resistant a majority of the time. Though reliability is not assured while executing uncommon abort-prone sections as discussed above, the simplicity of this solution, and the very low overhead (or even performance gain) it incurs makes it appealing and easily applicable. We prototype this approach in Section 4.5, for a subset of OS critical sections, to illustrate its feasibility.

An alternative approach to handling failures during critical sections is by using Recovery Domains [76], an organizing principle presented by Lenharth et al., which uses logging and rollback for restoring system state upon failure. Unlike transactional memory systems, the Recovery Domains principle only focuses on recovery from failures and not on isolation between parallel executions. However, this does not constitute a problem, since concurrency in kernel code is already managed by locks. The most prominent advantage of this approach is the ability to cope with critical sections that cannot commit transactionally. However, this approach is likely to incur much higher overhead and complexity, a cost that might not justify the benefits. We do not further explore this path in this work.

4.4 Cascading Failures

Domain	T_q	T_e	
		Tasklets	Workqueue
In-Chip	0.9ms/1.1ms	1.8ms	1.5ms/1ms
Inter-Chip	0.9ms/1.1ms	1.8ms	2.3ms/2.1ms

Table 4.1: T_q and T_e measurements (Busy/Idle).

In this section we elaborate on CSR's tolerance to cascading failures. This relies on the principle that each recovery task is eventually executed by some core exactly once.

We use HTM to avoid inconsistent states resulting from failures during recovery tasks execution. To this end, we have divided the recovery procedure into four tasklets and four work items, each of which is small enough to run as a transaction. We execute each individual tasklet or work item as a separate atomic transaction. Next, we take steps to ensure that recovery items are executed exactly once.

To analyze different cascading failure scenarios, we examine a timeline of the recovery process, as shown in Figure 4.2. The time at which the failure of C_F is detected is denoted t_{start}, and t_{ack} denotes the time at which the FDU receives the ack from C_D. We denote by T_q the interval between t_{start} and t_{ack}, and by T_e the time between t_{ack} and the end of the recovery process.

Measurements of our implementation on a 64-core machine, presented in Table 4.1,

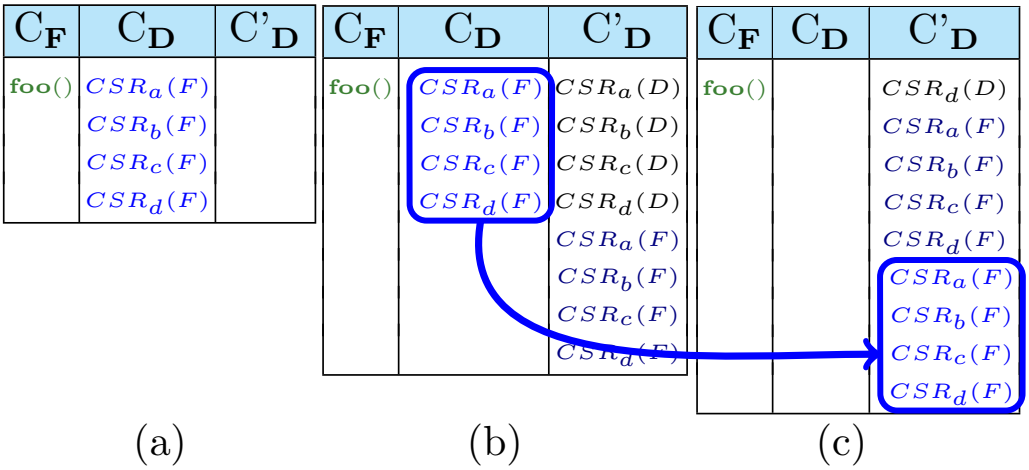


Figure 4.3: Duplicate queuing of recovery tasklets due to failure during T_q. Columns represent tasklet queues.
$CSR_X(Y)$ represents recovery tasklet X for core Y.
(a) C_D queues recovery tasklets for C_F's failure.
(b) C_D fails, C'_D queues recovery tasklets for C_D and C_F.
(c) After executing $CSR_c(D)$, C'_D has duplicates in its queue.

show that T_q lasts about 1ms, in both busy and idle systems. T_e, can last a bit longer, depending on the system load as well as the chips on which C_D and C_F reside.

In case of a cascading failure, the FDU detects and triggers a recovery procedure on another processor, C'_D. We first discuss the case where C_D fails during T_e. Here, the FDU knows that the recovery tasklets are accessible by the rest of the cores, and if necessary, other cores can read, migrate, and execute those tasklets. T_e itself can be divided into two sub-periods: one that consists of the execution of the recovery tasklets and a second where the work-items queued to RWQ are executed. In the latter case – since RWQ is not bound to any core, failure of C_D during that time does not affect the recovery process. In the former case, as part of step (c) of the recovery procedure, C'_D will take over and execute the recovery tasklets among the rest of C_D's tasklets. Thus, in either case, the recovery tasklets related to C_F's failure are executed, either by C_D before it fails, or by C'_D.

If the failure occurs during T_q, it is impossible to know which stages of CSR C_D has completed. Therefore, in case the FDU detects a failure of C_D without having received an ack from it, the FDU invokes a double recovery procedure on C'_D, to handle the failures of both C_F and C_D.

A failure during T_q can cause the recovery functions to appear more than once on the same queue. We give an example in Figure 4.3, where C_D completes stage (2) and crashes just before informing the FDU. C'_D then invokes a double recovery procedure (Figure 4.3(b)), during which it migrates C_D's tasklets. As a result, C'_D has the tasklets for handling C_F's failure twice. To prevent duplicate execution, we give the tasklets unique IDs, and mark completed tasklets in a globally accessed bitmap, as part of the transaction executing them. This bitmap is checked before each tasklet execution to ensure that it has not been executed before.

4.5 Implementation Issues

As a proof of concept, we implemented CSR in version 3.14.1 of the Linux kernel. Our implementation uses high-priority tasklets to queue the recovery tasklets, and allocates a new unbound workqueue to serve as the recovery-workqueue. We exploit, with some modifications, most of the functions that are used by CPU Hotplug. Eighty-one files of the Linux source were changed, for a total of about 4000 changed and new lines. We next explore some issues that arose during the work.

4.5.1 FDU and C_D

We implement the FDU using a periodic timer, which times out in predefined constant-length intervals and checks for heartbeats from all online cores (see Figure 4.4). Online cores send heartbeats to the FDU each timer interrupt. When the FDU detects a core that has stopped sending heartbeats, it considers that core to be faulty, and initiates CSR on another processor, C_D, by sending it an IPI. To simulate the reliability of the FDU,

```
for_each_possible_cpu(cpu){
  if (!heartbeats_received(cpu) &&
          !cpu_is_faulty(cpu)){
    mark_cpu_as_faulty(cpu);
    Cd = choose_recovery_cpu();
    send_IPI(Cd,CSR,cpu);
  }
}
```

Figure 4.4: FDU periodic callback.

we affine the FDU to core 0 solely and do not crash it during the experiments. This is done only for simplicity, as a reliable FDU can be implemented in hardware, even in a fault-prone environment [137, 44].

Upon receiving the IPI, C_D queues the recovery tasklets and verifies their visibility by using the WBINVD [62] instruction, which flushes its write buffer and private caches by invalidating them and performing a write back.

4.5.2 Setting a Faulty State

Linux's supervised core-unplug uses the *stop_machine()* mechanism for setting a core offline. The purpose of this mechanism is to prevent a core from going offline during execution of non-preemptible code on another core. This prevents, for example, changes to the online mask while another core iterates on it, and thus avoids inconsistent updates of the removed core's data. The *stop_machine()* function halts the execution of all online cores before the actual removal of the outgoing core from the online mask; thus, it postpones the actual removal of the victim core to a later time, at which no non-preemptible code is executed.

However, in crash failure cases, the use of *stop_machine()* is inadvisable. This is because the update of *faulty_mask* about the failure, as well as interrupt resetting, should happen fast. Deferring or preventing a core from crashing while another core executes non-preemptible code is, of course, impossible. Moreover, since the core will not get back online, consequences of inconsistent updates of its data are limited. We therefore refrain from using *stop_machine()*.

Note that previous works [51, 126] have also proposed to decouple *stop_machine()* from the CPU Hotplug path, for performance and design considerations.

4.5.3 Handling Lost Interrupts

For handling possible interrupt loss, following resetting the interrupt affinities, we send spurious interrupts for all the interrupt types that were previously routed to the faulty core. However, this does not cover cases of lost Inter-Processor Interrupts: The IPI queue on the faulty core might contain *migratable* callbacks, namely, callbacks that can execute on any core, and their execution is necessary. For example, in order to change the frequency of a chip, an IPI is sent to one of the cores in that chip; if that core fails, the callback needs to be migrated. On the other hand, some functions passed by IPIs should execute exclusively on the core they were sent to (e.g., read MSRs or flush TLB). Only the code that sends the IPI can determine whether the callback should be migrated when its recipient fails.

We therefore provide each callback with a flag indicating whether it is migratable. The flag is set when the IPI is generated and checked during recovery. Linux provides four primitives for broadcasting IPIs, depending on whether the destination is (1) a particular core, (2) a subset of the cores, (3) all of the cores, (4) any of the cores in a subset (*anycast*).

By examining the Linux source, we found that only functions that are sent by the anycast primitive are migratable. Thus, we changed the anycast implementation to queue these functions with flag set to true. By default, the other primitives queue functions as not migratable.

4.5.4 RCU Implications

Read-Copy Update (RCU) [85, 86] is a synchronization mechanism that allows low overhead wait-free reads at the cost of potentially expensive updates – each update must wait for a grace period to elapse before it completes. Specifically, an update thread must wait for a quiescent state to occur on each of the online cores in order to complete its update. As a result, an update operation that begins prior to C_F's crash and does not complete before the crash might wait forever for a quiescent state to occur on the faulty, non-responsive, core. In order to prevent such deadlocks, following a core crash, we explicitly allow all RCU updaters waiting for a quiescent state to occur on C_F to proceed. Likewise, future updates must avoid the waiting for C_F, and stop tracking it, as its future quiescent states will not happen. We therefore iterate over all RCU data-structures and delete all wait list entries corresponding to C_F, thus preventing future RCU updaters from waiting in vain. Figure 4.5 shows the high level operation of the corresponding code, which is executed as part of stage (iii).

```
rcu_report_crash(cpu){
  // Update all RCU trees
  for_each_rcu_tree{
    struct rcu_node* rnp;
    rnp = rcu_tree_leaf_node(cpu);
    while (rnp != NULL){
      // For future grace periods -
      // Exclude cpu from the initial mask
      remove_from_mask(rnp->qsmaskinit,cpu);
      rnp=rnp->parent;
    }
  }
  // For the current grace period -
  // Artificially report a quiescent state
  rcu_report_qs(cpu);
}
```

Figure 4.5: RCU recovery.

4.5.5 Using HTM

We gathered statistics about lock usage in kernel code, and found that the run-queue locks are among the most commonly acquired for the workloads we study and the most commonly occurring in the kernel source. We therefore chose to elide these locks as a case study for using transactions for recovery. We replaced all kernel critical sections protected by run-queue locks with the lock elision code in Figure 4.6, using Intel TSX. Here, we assume a convention in the Linux kernel, whereby all accesses to the run-queues are protected by these locks.

Each critical section is executed transactionally, and, as befits a best effort TM, is provided with a fallback path (line 9) [63]. The fallback path retries to execute the transaction, up to a predefined number of times. If the allowed number of retries has been exhausted, the implementation resorts to regular lock acquisition (line 13). To ensure correctness and reciprocity of a transaction with the fallback path, hardware transactions must read the lock as free (line 4), thus inserting the lock value into their read sets. The transactional memory semantics then guarantee that the transaction commits only when there is no ongoing fallback execution.

Though resorting to regular locking compromises our ability to recover from core failures in critical code, it is necessary to prevent livelock. The retries limit value determines a tradeoff – higher values favor reliability, whereas lower ones favor performance, as they shorten the maximum possible time to spend on retrying. Also, too high values may cause abort-prone sections to retry numerous times, thus harming the commit rates of other sections, causing them to resort to locking. We examined different limit values, and found the sweet spot to be at 10000 retries. Our results in the next section show that more than 99% of the transactions commit successfully in a lock-free manner, making

```
1  RetryTxn:
2  // start the transaction
3  if (_xbegin() == _XBEGIN_STARTED){
4      if (raw_spin_is_locked(&rq->lock)){
5          _xabort(1);
6      }
7      /*** Critical Section Code Here ***/
8      _xend(); // finish the transaction
9  }else{ //Tx failed - fallback:
10     if(retries++ < MAX_RETRIES){
11         goto RetryTxn;
12     }
13     raw_spin_lock(&rq->lock);
14     /*** Critical Section Code Here ***/
15     raw_spin_unlock(&rq->lock);
16 }
```

Figure 4.6: Lock elision using HTM [63].

failures inside a lock-protected critical section extremely improbable.

Among the 47 kernel critical sections protected by the run-queue locks, only three could not commit transactionally. Not surprisingly, one of them contains the context-switch function, which issues a TLB flush – an operation that cannot commit transactionally. The second creates a new kernel thread, and is called only a handful of times during the system's lifetime. We left the first two critical sections with a surrounding lock, compromising our ability to recover from failures in these critical sections. The third critical section has a large data set and could successfully commit after we split it into three smaller sections, after ensuring that such chopping is safe.

Given the above, we conclude that eliding the run-queue locks constitutes a good case study. While the most contended lock in the kernel will be workload-dependent, run-queue locks are commonly used and some of the critical sections they protect have complex behaviors that are challenging for HTM mechanisms. Our work is a proof of concept for using HTM for recovery on a real system. A complete conversion of kernel locking to HTM usage has already been presented on a simulator in TxLinux [112], and in that aspect, TxLinux is complementary to our work.

4.6 Soft Error Recovery

In this section we propose a new approach for addressing chip-level soft errors that leverages the basic CSR strategy. In the new approach, whenever an unrecoverable soft error is detected, the system initiates a surprise removal of the core in question. Thus, instead of the traditional action, a system restart, the core is taken offline and the system resumes execution. The only impact on the system in such cases is the removal of the faulty core from the system's available list of processors, and a termination of the task running on it. Finally, since a soft error is a transient event, following the surprise removal of the faulty core and the survival of the system, a supervised insertion of the core can be performed, using the Linux CPU-Hotplug mechanism. That is, by treating unrecoverable soft-errors as permanent, we elegantly save the system from crashing following a soft error.

We implement our approach in the Linux kernel, and upgrade its soft error handler. We evaluate our implementation and show that by adding only 60 lines of code (additional to the basic CSR code) to the Linux source, our system is able to recover from unrecoverable soft errors.

4.6.1 Background

Machine Check Architecture

In light of the increasing soft-error rate in modern processors (see Figure 4.7), leading chip manufacturers integrate a Machine Check Architecture (MCA) mechanism in their products, a mechanism through which a processor reports hardware errors to the operating system. MCA is already common in Intel and AMD modern processors, and can detect

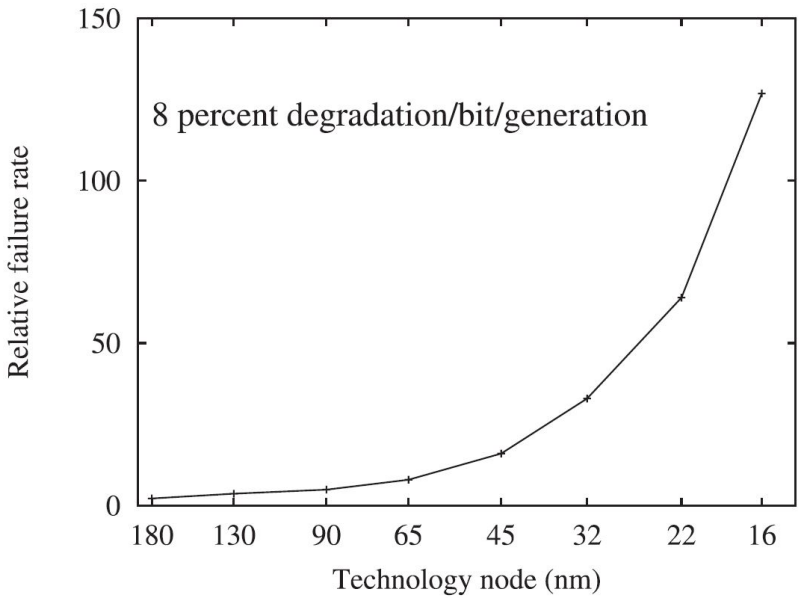


Figure 4.7: Soft error failure-in-time of a chip (logic and memory) [Borkar 2005]

and report hardware errors such as: system bus errors, ECC errors, parity errors, cache errors, and translation lookaside buffer errors. The MCA consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs used for recording errors that are detected. Upon a detection of a hardware error, the MCA broadcasts a Machine Check Exception (MCE) interrupt. The MCE interrupt is caught by all system's cores, and handled by the *MCE Handler*, which is part of the operating system kernel code.

Linux MCE Handler

The purpose of the Linux MCE handler is to manage soft-error cases. This includes conducting a comprehensive search to find the location of the error; classify its severity; take proper actions if the error is correctable; and halt the system if the core's state is unrecoverable.

The handler is executed in NMI context, and therefore cores are not allowed to sleep or block during its execution. Particularly, lock usage is forbidden. The MCE handler is executed by all the cores in parallel, while some of them might be damaged beyond repair, hence, synchronization between the cores is challenging and should be treated with exceptional care. Furthermore, since some of the chip components are shared between the cores, some of the MCA banks are shared. Therefore it is essential to establish an order between the cores for avoiding duplication of event reporting on shared banks; the first core to see an error will clear it.

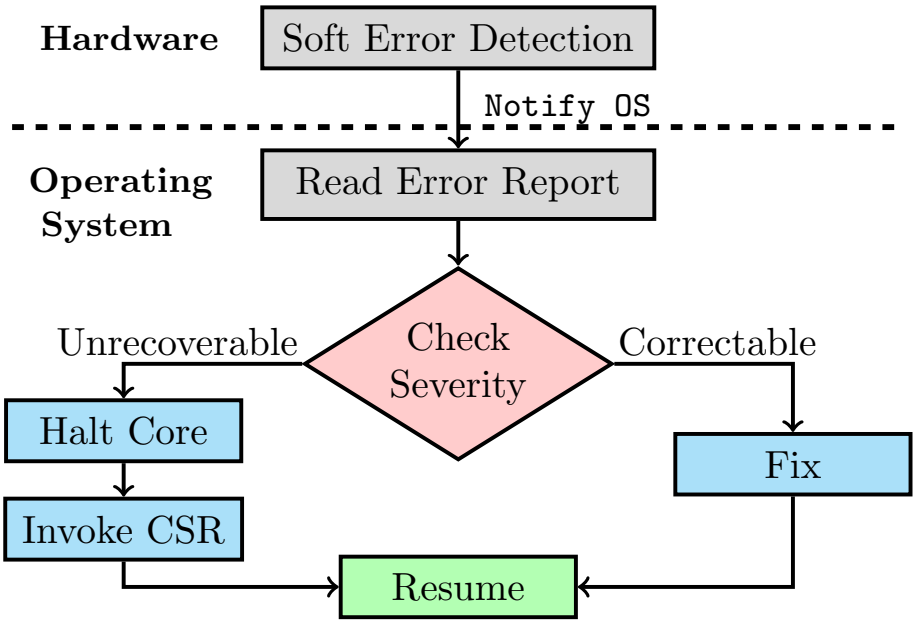


Figure 4.8: Soft Error Handling Recovery Flow

When considering MCE handling, one must always keep in mind that if there is a fatal error in the system that cannot be isolated, code execution should be prevented. Otherwise, this may result in execution of unexpected arbitrary instructions in kernel mode. This, in turn, might lead to permanent data corruption and misconfiguration of the system. For this reason, **`panic`** is used.

4.6.2 Soft Error Recovery Strategy

There are two types of soft errors, chip-level soft error and system-level soft error. Chip-level soft errors occur when particles hit the chip and cause a cell to change state to a different value. System-level soft errors occur when the data being processed is hit with a noise phenomenon, typically when the data is on a data bus. The bad data can even be saved in memory and cause problems at a later time. Nowadays, all system-level and most of the chip-level soft errors are unrecoverable; and if detected, cause the system to panic and require a system restart. Ridiculously, a single soft-error in one of the cores of a thousand core machine, can cause the system to crash (see a sample panic message in Figure 4.9).

We propose to leverage the CSR strategy, which is intended for handling fail-stop core crashes, in order to recover from unrecoverable chip-level soft error cases. Our proposed recovery flow is described in Figure 4.8. The recovery process begins with a detection of a soft error by the designated hardware (e.g., MCA), which in turn inform the operating system about the event. Next the soft error handler reads the reported error, determines

the actions need to be taken by classifying the severity of the error. If the error is found to be recoverable, then, just like current MCE handlers, it takes proper measures to correct the error (e.g., rewrite the faulty data, fix the bit-flip, kill running process if needed, etc.). On the other hand, if the error is found to be unrecoverable (e.g., context corrupt, non-correctable imperative data), then instead of panicking we suggest to halt the core from executing and revert to CSR. Halting the core prevents it from executing possibly wrong instructions or storing wrong data to the memory. Reverting to CSR can be done by issuing an IPI or relying on the FDU to trigger the removal process. By invoking CSR the system may proceed execution with the faulty core turned off.

We further propose to bring the core back online in a later time. Since soft errors are transient and the core is not defected, after resetting the core, it becomes usable again. This requires a hardware support that is still not available on modern processors – a software-driven reset of a core.

Note that since the core that suffers from the soft error is not reliable, chances always exist that it will not be able to execute the handler correctly. In these rare cases, the system's behavior in undefined and cannot be predicted. Applying our proposed changes to the machine check handler does not affect the probability for such cases; hence, overall, our technique only improves the system's behavior following a soft error.

4.6.3 Implementation in Linux

In this section we describe our implementation of our approach in the MCE handler of the Linux kernel. We start by describing the current Linux MCE handler design and proceed by elaborating our changes.

Traditional MCE Handler Flow

The handler flow can be broken down to 5 primary steps:

1. **Quick fatal error check.** Each core reads its error registers (banks). If a fatal error is detected, the local `no_way_out` flag is raised.

2. **Set an order.** The cores are numbered by order of arrival. The first core to enter the handler is dubbed the *monarch*. The rest of the cores, the *subjects*, wait in a barrier and get released one by one for executing the next step. Meanwhile, the

```
CPU 0: Machine Check Exception:
    0000000000000004
Bank 2: f200200000000863
Kernel panic: CPU context corrupt
```

Figure 4.9: Linux kernel message after a soft error.

local `no_way_out` flags are distributed among the cores, in a way that if one core has set its local `no_way_out` flag, then all other cores also raise their flags.

3. **Read and grade errors.** This step is executed by all the cores serially, according to the order dictated by the previous step. Each core in its turn: (a) Reads its banks for reported errors, (b) Grades error severity, (c) Schedules further handling for treating recoverable errors (due to the NMI context, non-urgent actions are postponed for later time), (d) Logs the errors to a file, (e) Saves the worst error in a global structure, and finally (f) Clears its banks.

   When grading error severity, four possible degrees are available: (i) Action optional (ii) Uncorrected (iii) Action required (iv) Fatal error. Among the above, only the fourth severity requires halting the execution of the core and rebooting the system. In our upgraded handler, presented in the next section, we aim to identify some of the cases that are classified as fatal, and take alternative recovery operation in order to avoid kernel panic. We next refer to a core that detected a fatal severity soft error in its hardware as *fatal core*.

4. **Monarch "reign".** This step is executed only by the monarch, while the subjects are halted. The monarch core goes through the worst errors reported by each core and finds the globally worst error. If it finds a fatal severity error, it panics the system. Otherwise, it clears the global error structures and the subjects are released.

5. **Take action.** In this last stage of the handler, the global system status was determined and all errors were already analyzed. Each core checks its `no_way_out` flag, and if set - it panics. If the flag is clear, then the corresponding core performs recovery operations: It (a) kills the current process if needed, (b) schedules additional recovery operations after handler terminates, and (c) reports if more errors require handling. However, if too many concurrent recoverable errors are detected at the same core, the system also panics. Our upgraded handler, presented in the next section, avoids panicking also in these cases.

Upgrading the MCE handler

We improve the MCE handler so that when there are cores that suffer from a fatal error, the handler reverts to CSR, and logically removes the faulty cores from the system. For simplicity, we implement it for cases whereby only one core suffers from a fatal error. Our solution can be extended to support recovery from soft-errors in up to $N-1$ cores, where N is the system core count.

We apply five primary changes to the recovery flow:

1. **Identify the fatal core.** The MCE is broadcasted to all system's cores; namely, non-fatal cores also executes the MCE handler. Therefore, when entering the handler we first check if a fatal error exists in one of the cores. For that, each core reads

its MCA banks and sets its private *no_ way_ out* flag if it detects a fatal error. We add two shared variables (lines 3-4): `num_fatals`, that represents the number of cores that found a fatal error in their hardware, and `faulty_core`, that keeps the identity of the faulty core. The former is atomically incremented by each core that detects a fatal error. The latter is updated by the core that finds the fatal error, and its value is ignored if multiple cores were found to be fatal.

2. **Prevent the fatal core from being the monarch.** The traditional MCE handler panics the system upon any fatal soft error detection, so it does not consider the identity of the monarch as consequential. However, our upgraded MCE handler may evade panicking. The monarch is responsible for exclusively executing part of the MCE handler, and as a result we must ensure that the fatal core is not nominated as the monarch.

   For that purpose, we force the core that corresponds to `faulty_core`, to get the last ordering number; thus prevent it from possibly being the monarch. As described in Figure 4.10, in lines 14-23, if the executing core is the only core in the system that suffers from a fatal error, then it will be delayed, spinning on the `mce_callin` variable until the rest of the cores receive their order. The rest of the cores skip this code section, receive an order in line 25, and resume execution to the barrier in lines 26-36.

3. **Prevent non fatal cores from panicking.** As demonstrated in lines 38-40 of Figure 4.10, we distribute the `no_way_out` flag among the cores only if there is more than one fatal core; thus, we let non-fatal cores get out of the handler without panicking. Furthermore, we prevent the monarch from panicking the system (see stage 4) if it detects only one faulty core.

4. **Trigger Core Surprise Removal instead of panicking.** Finally, each core checks its `no_way_out` flag and if the flag is set, we trigger core removal. For that purpose, as showed in figure 4.11, we lead the core to an infinite loop so it stops executing any real code. Note that the handler is executed in NMI context while interrupts are disabled. Thus, the core stops responding, which is enough for being detected by the FDU.

5. **Too Many Action-Required Errors.** In cases where an action required (AR) error is detected, the core keeps info about this error in a designated data structure for later handling. However, the data structure is allocated from a pool, and if there are too many AR errors (typically 16) the pool gets drained, and the traditional MCE handler panics instead. That is, if there are too many concurrent recoverable errors, the system is configured to panic. Same as before, we change the handler, making it revert to CSR instead of panic (See Figure 4.12).

```
1 shared int mce_callin  =  0;
2 shared int global_nwo  =  0;
3 shared int num_fatals  =  0; //newly added
4 shared int faulty_core = -1; //newly added
5
6 #define IS_SINGLE_FATAL(core)
7 (atomic_read(&num_fatals)==1)&&(core==
    faulty_core)
8
9 int mce_start(int* no_way_out){
10   int me = smp_processor_id(); //this core
11   unsigned long long timeout = TIMEOUT_VAL;
12
13   atomic_add(*no_way_out, &global_nwo);
14   if (IS_SINGLE_FATAL(me)){
15     //wait until all cores take a number
16     while (atomic_read(&mce_callin)!= cpus-1)
         {
17       if (mce_timed_out(&timeout)) {
18         atomic_set(&global_nwo, 0);
19         return -1;
20       }
21       ndelay(SPINUNIT);
22     }
23   }
24   //take a number
25   order = atomic_inc_return(&mce_callin);
26   if (order == 1){ // Monarch:Start executing
       now
27     atomic_set(&mce_executing, 1);
28   }else{ // Subject:spin here till your turn
29     while (atomic_read(&mce_executing)<order)
         {
30       if (mce_timed_out(&timeout)) {
31         atomic_set(&global_nwo, 0);
32         return -1;
33       }
34       ndelay(SPINUNIT);
35     }
36   }
37
38   if (atomic_read(&num_fatals) > 1) {
39     *no_way_out = atomic_read(&global_nwo);
40   }
41   return order;
42 }
```

Figure 4.10: Ordering the faulty core as last while preventing non-fatal cores from panicking.

```
1 if (no_way_out) {
2   int me = smp_processor_id(); //this core
3   if (IS_SINGLE_FATAL(me)){
4     atomic_dec(&mce_entry);
5     //interrupts already disabled
6     while(TRUE);
7   }
8   mce_panic("Multiple␣fatal␣cores.");
9 }
```

Figure 4.11: Triggering CSR, relying on an FDU.

```
1 /*Look for a free mce info resource*/
2 for (mi=mce_info;mi<&mce_info[MCE_INFO_MAX];mi++){
3   if (atomic_cmpxchg(&mi->inuse,0,1)==0){
4     mi->t = current ;
5     mi->paddr = addr;
6     mi->restartable = c ;
7     return;
8   }
9 }
10 /* Not enough resources */
11 if (atomic_cmpxchg(&num_fatals,0,1)==0){
12   atomic_dec(&mce_entry);
13   //interrupts already disabled
14   while(TRUE);
15 }else{
16   mce_panic("Too␣many␣AR␣errors" , NULL, NULL) ;
17 }
```

Figure 4.12: Triggering CSR on too many AR error detection.

Benchmark	Workload Properties			Success Rate	
	user	*system*	*iowait/idle*	*non-system*	*system*
K-means	99%	1%	0%	100%	86%
401.bzip2	99%	1%	0%	100%	72%
410.bwaves	99%	1%	0%	100%	88%
429.mcf	22%	14%	64%	100%	68%
Postmark	5%	21%	74%	100%	70%

Table 4.2: Recovery rates, without HTM, under random fault injections in user and kernel code.

4.7 Evaluation

This section presents our evaluation of CSR. We begin in Section 4.7.1 with a massive random fault injection campaign on a virtualized environment protected by the basic CSR algorithm, without using HTM. Next, in Section 4.7.2, we evaluate CSR with the lock elision code on real systems. In Sec. 4.7.3, we quantify the effects of using HTM on energy and performance using a dedicated scheduler benchmark. Finally, in Section 4.7.4 we discuss evaluation methods for our approach for tolerating soft-errors.

In all experiments, except noted otherwise, the timer interrupt is set to the default period of 4ms and the FDU wakeup period, which can be set to any value higher than the timer interrupt period, is set to 10ms.

4.7.1 Massive Virtualized Fault Injection Without HTM

We install our CSR-enhanced Linux on a 4-core VM emulated by QEMU [11]. As we employ QEMU without HTM emulation, we run a kernel with only the basic CSR functionality and no lock elision. Hence, kernel critical sections are not protected. We run three sets of experiments, with random fault injections during three different execution modes: (1) user mode, (2) kernel mode, and (3) idle/IO-wait mode.

Emulating Failures. We change the QEMU source, causing it to shutdown a random virtual core at a random time when this core is executing in the desired mode (user, kernel or idle), thus simulating a permanent core fault. This is done by abruptly stopping one of the QEMU threads that simulate the virtual cores. We determine the execution mode by examining the CPL and EIP registers of the victim core. Note that an unmodified kernel crashes following a fault injection during any execution mode.

In each experiment, we start a VM running our kernel, run a given workload, wait for a virtual core to crash during the required execution mode, and allow the system time to recover. In order to verify that the system has indeed recovered, we create a new file, write a timestamp into it, and flush it to disk using **`sync`**. Thus, a file is created per each successful recovery. We repeat the experiments thousands of times per workload and per

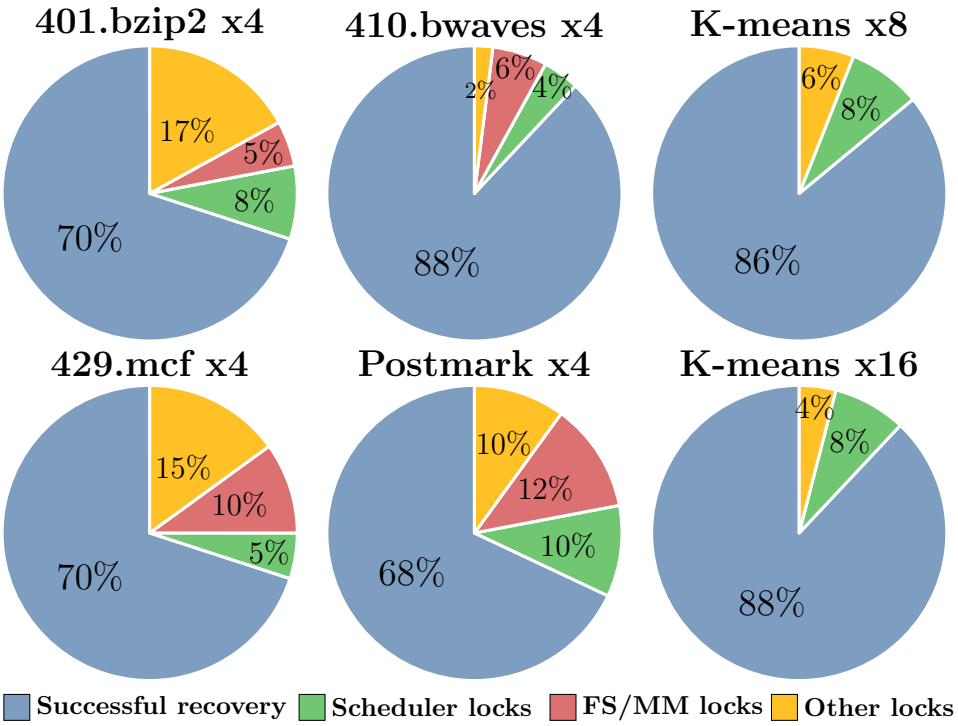


Figure 4.13: Recovery rates and failure locations under random fault injection in kernel code, no HTM in use.

execution mode.

The benchmarks we use are K-means of the Metis [82] in-memory MapReduce library; SPEC-CPU™2006 [55] benchmarks, out of which we choose one data-intensive (429.mcf) and two CPU-intensive (401.bzip2, 410.bwaves) applications; and Postmark [68], an IO-intensive filesystem benchmark. Since Postmark and SPEC benchmarks are single threaded, we run four instances of each application, in order to keep all the cores occupied. In order to exercise the scheduling system, we run K-means with 8 and 16 threads on the four cores. We measure the induced workload properties of each benchmark using SYSSTAT [52].

Results. Our results, which appear in Table 4.2, show that our system recovered in all cases wherein fault injections were performed in user or idle mode, as expected. For fault injections during kernel code execution, the system recovered in about 70% (or more) of the experiments. For each experiment that resulted in a system crash, we extracted the exact code line before which the fault was injected (using the EIP and CPL registers) and analyzed the reason that prevented the system from recovering. We present in Figure 4.13 a breakdown of the failure reasons into three main categories: (1) holding a *scheduler lock*, such as the run-queue locks; (2) holding a *filesystem or a memory lock*; and (3) holding

other locks, such as RCU and timer locks. As can be seen, IO-intensive workloads (mcf, Postmark) tend to crash during filesystem and memory operations more often than their computation-intensive counterparts. In addition, all workloads suffer considerably from crashes due to holding a scheduler lock. These failures occur since our system is tested here without eliding the scheduler locks (see Sec. 4.5.5), which we next evaluate on a real HTM-equipped system.

4.7.2 Experiments on a Real System

We next evaluate our implementation in two physical environments. The first is a PC running Ubuntu 14.04 on a 1x4x2 Intel Core i7-4770, TSX-equipped processor. The second is a server running Ubuntu Server 12.04 on 4x8x2 Intel Xeon E5-4650 processors.

Crash Simulation

For simulating a crash in a real system, we force a core to hang inside various kernel critical and non-critical sections in an un-interruptible state, thus causing it to stop responding. Figure 4.14 shows the code snippet we use for crashing an online core. The return value of *fault_injection()* is negative until set to be some core's id by a system call. Once the code is invoked on the victim core, it hangs in an infinite loop, with interrupts disabled, and therefore no possibility to be preempted. Note that this simulates our failure model, where the values in the crashed core's cache is accessible to other cores following a failure.

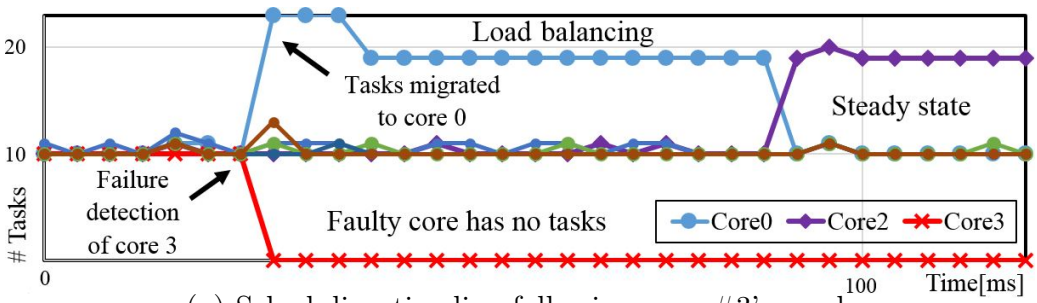
In order to hang a core on a real system, the code snippet (Figure 4.14) has to be hard coded into the kernel source. This does not allow us to perform automatic fault injection at random locations as in the virtualized case. Therefore, we manually perform fault injections in one hundred representative kernel functions (a partial list appears in Table 4.3). For functions that contain critical sections protected by a run-queue lock, we inject a crash in the critical section itself.

Crash and Recovery Timeline

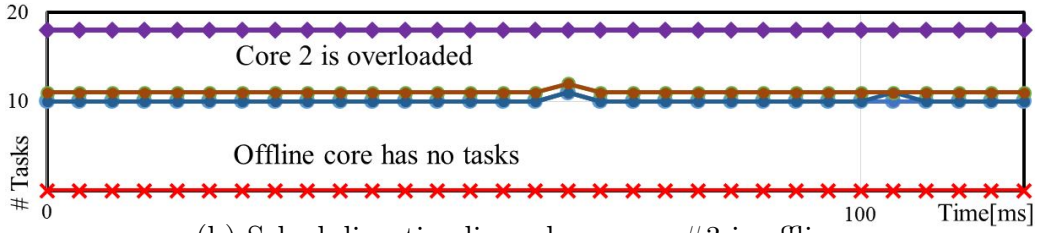
In this set of experiments, we create and affine to each core 10 computation-heavy tasks. These tasks are always hungry for CPU time, and so, at any given moment during the experiment, each correct core should have at least 10 tasks in its run-queue. Next, we crash one of the cores and let CSR recover the system. We repeat the experiment on

```
interrupts_disable();
if (fault_injection() == smp_processor_id())
    while(TRUE);
```

Figure 4.14: Crashing a core.



(a) Scheduling timeline following core #3's crash



(b) Scheduling timeline where core #3 is offline

Figure 4.15: Scheduling timelines for PC system with 8 cores.

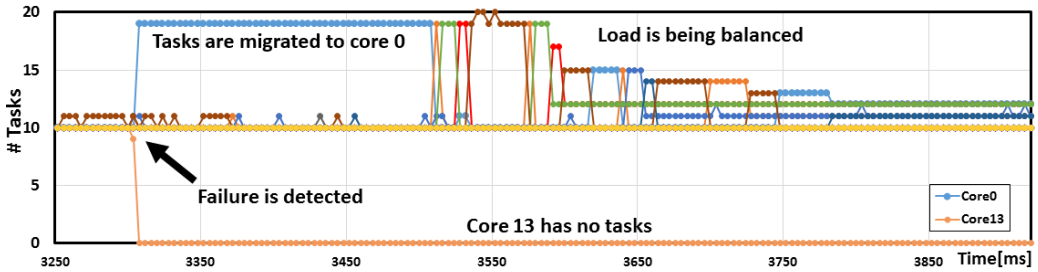


Figure 4.16: Scheduling timeline, on server with 32 cores (16 shown) following the crash of core 13.

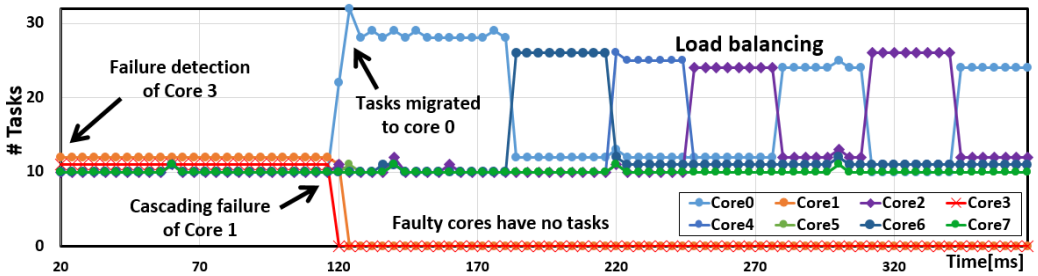


Figure 4.17: Scheduling timeline, with cascading failures on a PC.

the PC by placing the code snippet of Figure 4.14 in the one hundred kernel critical and non-critical sections mentioned above.

Traditional operating systems do not recover from such faults. Linux, for example,

uses a watchdog [31] to detect lock-ups. Following a fault detection, Linux provides two possible actions (determined at system install time). The first is to reboot the system, and the second is to ignore the fault and resume normally. We configured our unmodified Linux system to the second option and used the method described in Figure 4.14 to hang a core. We repeated this experiment multiple times, and got a variety of behaviors, all of which led to a system freeze within a few seconds, possibly due to lost interrupts, unanswered IPIs, synchronization problems (held locks, stuck RCU operations), etc.

On the other hand, with CSR, the system successfully recovered in *all* the experiments. Exemplary results of experimenting on the PC and server are shown in Figures 4.15 and 4.16, respectively. To improve readability, we increase the FDU timeout to 100ms, show only cores 0-15, and disable hyper-threading on the server, just for the depicted experiments. The number of tasks in each run-queue is sampled every clock interrupt by a tool we implemented, and we plot the resulting scheduling timeline. Figure 4.15a shows the recovery timeline following a crash of core 3 (out of 8). We see that the system recovers, and the tasks that belong to the faulty core are migrated to the lowest id correct core. After less than 70ms, the load balancing mechanism kicks in and corrects the overload. Perhaps surprisingly, the tasks are migrated to core 2. The reason for this behavior lies in the scheduling-domains approach [70], according to which, in our case, each pair of logical cores constitute a scheduling domain. After the crash of core 3, core 2 constitutes a domain by itself. Therefore, the balancing among the domains causes core 2 to get a double amount of work. To verify this analysis, we examine the system with the same workload and take core 3 offline using CPU-Hotplug in an unmodified Linux kernel. As can be seen in Figure 4.15b, the load balancing behaves the same in both cases. We conclude that the Linux load balancer is not tuned for considering offline cores, and leave fixing this issue for future work. We further discuss this issue in Appendix A.

Figure 4.16 shows the scheduling timeline of our server platform, after a crash of core 13 among 32. We see that the system recovers, and the overload formed on core 0 is spread among the rest of the cores.

We simulate cascading failures by crashing a core, and immediately afterwards, crashing the core that was nominated by the FDU to execute CSR. Figure 4.17 presents the resulting scheduling timeline on the PC. Here, the FDU detects core 3 as faulty and nominates core 1 to handle the fault. Core 1 crashes before it sends an ack to the FDU, and

File	Functions
`kernel/sched/core.c`	`scheduler_tick(), schedule(), ttwu_queue()`
`kernel/watchdog.c`	`watchdog_timer_fn()`
`kernel/timer.c`	`__run_timers()`
`kernel/workqueue.c`	`__queue_work()`
`kernel/softirq.c`	`wakeup_softirqd(), __do_softirq()`
`kernel/events/core.c`	`update_event_times()`
`kernel/pid.c`	`alloc_pid()`

Table 4.3: Functions injected with faults (partial list).

100ms (the FDU wakeup period) later, the FDU considers core 1 to be faulty as well. It then nominates core 2 to perform CSR for both cores 1 and 3, and the system recovers.

Multiple Virtual Machines

Servers nowadays often run multiple VMs. In the absence of support for core surprise removal, a crash of a single core brings down the entire system, along with all running VMs. To demonstrate that CSR eliminates this problem, we conduct the following experiment: We install CSR on the host OS of our 64 hardware threads server. Using unmodified QEMU, we create and run four VMs. Each of the VMs runs Ubuntu 14.04 with an unmodified kernel. We then set the affinity of VM i to the CPU in socket i, namely, we attach VM i to the cores in the range $[16 \cdot i, 16 \cdot (i + 1) - 1]$. Figure 4.18a shows a screenshot of the system in an initial correct state. Next, we cause a permanent failure to one of the cores in the 4th socket. The result can be seen in Figure 4.18b: Thanks to CSR, only the fourth VM, (which is affined to the fourth chip), suffers from the crash, while the remaining VMs continue normally. Figure 4.18c shows the behavior of the unmodified kernel on the host OS in this scenario, where all VMs hang.

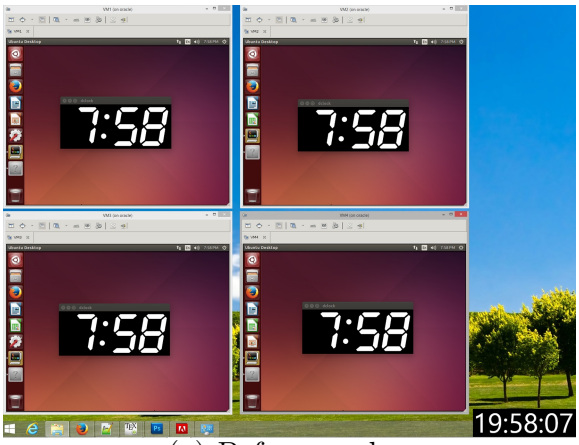
4.7.3 Lock Elision

We now proceed to quantify the energy and performance implications of using HTM instead of locks in critical kernel code. We use the SysBench [3] benchmark tool, and gather performance, energy consumption (using the RAPL [66] interface), and abort statistics under various workloads on our PC system. Since we apply lock elision on the run-queue lock, we use the *threads* test mode of Sysbench, which is intended for measuring scheduler performance. We set the retries limit to 10000 for all critical sections. By examining the commit rates of each critical section, we found one exceptional critical section (*task_tick)*) that was able to commit, but caused, under some workloads, a significant increase to the total system abort rate. In principle, this code section updates only local data, and should abort rarely. However, the best-effort nature of Intel-TSX causes it to abort under certain workloads due to reasons like cache evictions. To avoid performance penalties, we set the retries limit for this critical section to 10, thus prevent it from contending excessive times with other sections and improve the overall commit rate.

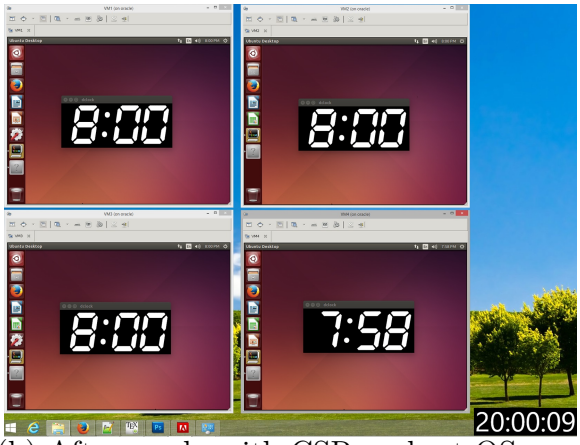
Measurements of our reliable kernel over billions of transactions appear in Table 4.4. The use of lock elision eliminates expensive atomic instructions. This results in small improvements to the energy consumption and performance. As can be seen, commit rates are always higher than 40%, meaning that transactions require less than 3 retries on average to commit successfully. The second column shows that the percentage of critical sections that exhaust all retries and resort to locking is negligible, meaning that the vast majority of critical sections execute in a reliable manner.

Workload	Commit Rate	Fall-Backs on Locking	Performance Gain	Energy Saving
Idle	61%	0%	-	4%
16-threads	93%	0.1%	0%	1%
32-threads	80%	0.1%	3%	3%
64-threads	42%	0.2%	2%	2%

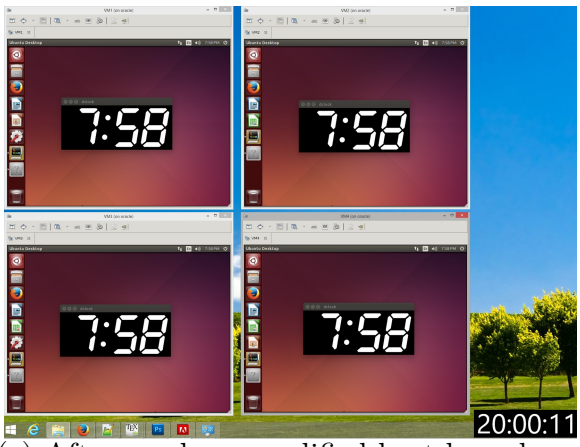
Table 4.4: HTM implications on performance and energy.



(a) Before crash –
4 VMs run concurrently.



(b) After crash, with CSR on host OS –
Only one VM hangs.



(c) After crash, unmodified host kernel –
All VMs hang.

Figure 4.18: Screenshots taken before and after a core crash on a server running four VMs.

4.7.4 Soft-Errors

Unfortunately, all soft error emulation methods and simulation tools do not fully imitate a real soft error on all its effects. Here, we detail about three soft-error emulation approaches that we tried.

MCE-Inject tool. MCE-Inject [1] is an open source Intel-provided tool that is intended for validation of the Linux kernel machine check handler. The tool allows for injecting machine check errors on the software level into a running Linux kernel. It is basically a user process that uses input arguments in order to create a fake MCE status and then issues a system call that loads the MCA banks with this error status. The system call then explicitly invokes the MCE handler as if interrupt was actually raised. The handler decodes the status register and take actions as if it was a real error.

However, this is not enough for our purposes. It is a user application and therefore the randomness nature of soft errors is completely eliminated. It always triggers the handler from the same kernel code, and it doesn't let us test the system at random times, while doing random tasks, e.g., user mode.

QEMU Though extremely useful for evaluating our core mechanism, we do not find QEMU valuable for emulating a soft error. When QEMU injects an MCE, it only simulates the MCE interrupt, while manually writing fictive data to the MCA banks of the faulty core; thus making the operating system detect a bogus soft error.

This does not fully simulate the danger to systems data, cannot cause execution of wrong instructions, etc. We conclude the QEMU merely checks the quality of the MCE handler of the operating system.

Inappropriate Kernel Behavior. We wish to perform an inappropriate kernel behavior in order to trigger an MCE. A known prohibited behavior is a write-back to an MMIO region; this causes an error as devices do not allow to be accessed with any write policy except for write-through. In order to trigger such an event we used Memory Type Range Registers (MTRRs). MTRRs are a set of processor supplementary capabilities control registers that provide system software a control over the way that accesses to memory ranges by the CPU are cached.

However, after implementing the approach, all logical processor contexts got corrupted as MTRRs have a shared state for all processors. Unfortunately, our solution does not support recovery from such cases, because our implementation assumes that there is no more than a single faulty processor. Moreover, in order for CSR to work, we must have at least one correct core. Finally, we note that such an approach enables soft-error injection only in specific locations, and like the rest of the approaches mentioned in this section, is not able to emulate the various effects that soft errors might cause.

We leave the evaluation of CSR's capability for mitigating soft-errors for future work.

4.8 Related Work

To the best of our knowledge, CSR is the first system to address unexpected permanent core failures in commodity architectures and OS.

Reliability Support in OS. A number of works have addressed permanent or intermittent hardware failures: Hive [26], is designed to cope with fail-stop failures. It is built of independent kernels and confines errors to kernels where they occur. However, it is intended for a special architecture, is incompatible with commodity OS, and was not tested on a real system. Dobel and Hartig [40] designed an OS that tolerates soft errors using redundant threads by transferring essential OS code to a dedicated reliable computing base. CSR, on the other hand, does not assume the existence of a reliable core. C^3 [122] provides predictable recovery from intermittent faults in embedded and real-time systems. However, it does not address commodity computers and OSes.

Dolev and Yagel [41] present two self-stabilizing OS principles, which allow an OS to start from any initial state. Unlike CSR, they do not deal with allowing a system to continue to run in case of hardware failures.

Other works have addressed software-induced failures. MINIX3 [56], for example, is a reliable micro-kernel that detects software failures such as deadlocks, and restarts the faulty software component for recovery. Microreboot [22] performs fine-grain rebooting of application components, but is not designed for kernel code. Nooks [128, 129] and SafeDrive [147] provide fault isolation and recovery for device drivers by inspecting their interaction with the kernel, and are thus able to recover from failures in drivers and other kernel extensions without rebooting the OS. Akeso [76] is a system based on the Recovery Domains principle, which uses logging and rollback to tolerate faults in the entire kernel. All efforts mentioned above tolerate software-induced failures only, and except for Akeso [76], none of them tolerates errors in core kernel code.

Transactional Memory. Rossbach et al. have proposed the use of TM in kernel code, and presented TxLinux [112], which exploits TM in order to improve performance. CSR, on the other hand, exploits HTM for reliability. FaulTM [142] utilizes a modified HTM for reducing the overhead of double execution; however, it does not address execution of OS code. Moreover, unlike both of these works, CSR was tested on a real system, taking into account the best-effort nature of real HTM implementations.

CPU-Hotplug. Various works have pointed out CPU Hotplug's shortcomings. Gleixner et al. [51] propose improvements to the CPU-Hotplug path for energy and real-time purposes. Panneerselvam and Swift [99] propose operating system support for dynamic processors, which can dynamically reconfigure the machines' cores at runtime. However, neither of these refers to hardware failures, and both require the cooperation of the victim core.

Many-Core OS. Many-core operating systems is an active area of research. The TeraFlux project [50] encompasses various aspects of teradevice computing, including reliability [44, 45, 46, 137]. These works mainly focus on designing and implementing hardware FDUs, but do not consider the OS implications upon failure detection, and are therefore complementary to our work. Other works on OS for systems with large processor counts mainly focus on improving performance and scalability [10, 18, 54, 80, 109, 146], rather than reliability. IBM BlueGene [96] features fault-aware job scheduling, which uses fault prediction to improve the supercomputer's performance, but without considering recovery methods within a faulty node.

Soft errors. Previous work in the soft error domain mainly focused on detection [21, 7] and injection [38, 30, 141, 1] methodologies, and are complementary to our work. To the best of our knowledge, there is no prior art that proposes a mechanism for recovering from fatal soft errors.

4.9 Summary

Existing operating systems cannot recover from unexpected core failures. Thanks to technology scaling, many-core machines are going to be widely deployed in the foreseeable future. However, hardware shrinking introduces new reliability challenges which cannot be addressed by current software.

In this chapter we have introduced CSR, a new approach for OS reliability in the face of permanent core faults. We further proposed the use of HTM to cope with faults in kernel critical sections. CSR can be integrated with kernels running on servers with multiple coherence domains [5, 50], preventing entire clusters from crashing following a single core fault. Moreover, a combination of CSR and application-level runtime solutions to re-issue work performed by crashing threads [39, 140] can conceal consequences of failures from user applications.

The CSR principle can be extended to cope with unrecoverable chip-originated soft-errors. Such errors, which are common even today, typically indicate that something is wrong with the chip and therefore it is considered good practice to take the malfunctioning core offline. Since the core is already known to be faulty, expecting it to perform the unplug steps is inadvisable. In addition, Machine Check Exceptions that catch such malfunctions today usually result in a kernel panic. Instead, we catch the MCE, halt the target core, and revert to CSR upon such errors.

Chapter 5

Conclusion

In this thesis, we explored few key challenges in systems security and reliability. We presented WatchIT, which provides a comprehensive solution for organizations that wish to mitigate the danger that their IT personnel poses to the organizational information security. We further introduced a unique approach for detecting similarity between code binaries, which uses concepts from image processing and employs machine learning-based classifiers. Finally, we designed CSR, a strategy for overcoming soft errors and hardware faults that are originated in the CPU.

Our proposed solutions comprise of new operating systems abstractions (e.g., perforated container), mechanisms (e.g., core surprise removal), methodologies (e.g., proc2vec), and non-intuitive use of existing software and hardware components (e.g., FUSE, HTM). We evaluated our proposed approaches by implementing and testing them on real systems and by conducting a case study on real production environments.

That said, a lot of work remains for future work. It would be of interest to build on the WatchIT approach and use the data collected by our monitoring mechanisms for performing anomaly detection. Moreover, by designing a machine learning model that can learn from the collected data and automatically adjust the boundaries of the deployed perforated containers we can further improve the efficiency and performance of the approach. For binary similarity detection, it would be of interest to try to improve our proc2vec algorithm by borrowing concepts from natural language processing to design a better algorithm for finding representations for assembly instructions and binary procedures. In the context of CSR, it would be useful to find a reliable way for injecting soft-errors that can capture the random yet destructive nature of such errors. This will allow for evaluating how CSR can mitigate the consequences of soft-errors, and not only fail-stop ones.

We hope that our proposed security approaches will be adopted by companies and organizations, thus making computer systems safer, for both enterprises and end-users. Moreover, as hardware-induced failure rates increase, we hope that our reliability strategies, conclusions, and results, will be integrated into future systems, and will contribute

to the reliability of future computer systems.

Appendix A

Linux Load Balancer Unexpected Behavior

In this chapter we discuss some of the Linux load balancer mechanism properties. We start by giving background on the scheduling domains load balancing approach, and continue by describing a surprising behavior (referred in the following as *bug*) we have encountered while working with it. We further research the source of this behavior and propose a change to the Linux kernel for mitigating it.

A.1 Scheduling Domains

One of the key problems a scheduler must solve on a multiprocessor system is balancing the load across the system cores. Moving processes between processors is not free, and some sorts of moves (across NUMA nodes, for example, where a process could be separated from its fast, local memory) are more expensive than others.

The Linux scheduler handles this problem using the *scheduling domains* approach. A scheduling domain (`struct sched_domain`) is a set of cores which share properties and scheduling policies, and can be balanced against each other. Scheduling domains are hierarchical; a multi-level system will have multiple levels of domains. Each domain contains one or more CPU groups (`struct sched_group`) which are treated as a single unit by the domain. When the scheduler tries to balance the load within a domain, it tries to even out the load carried by each CPU group without worrying directly about what is happening within the group.

As exemplified in Figure A.1, a 4x2x2 machine will have 3 hierarchies: the node level, with four scheduling groups; the physical level, with 2 scheduling groups each, and the core level, also with 2 groups each. The Linux load balancer only balances load within those domains, in a top-down manner. E.g., in the system described in Figure A.1, it first balances the loads between the system nodes.

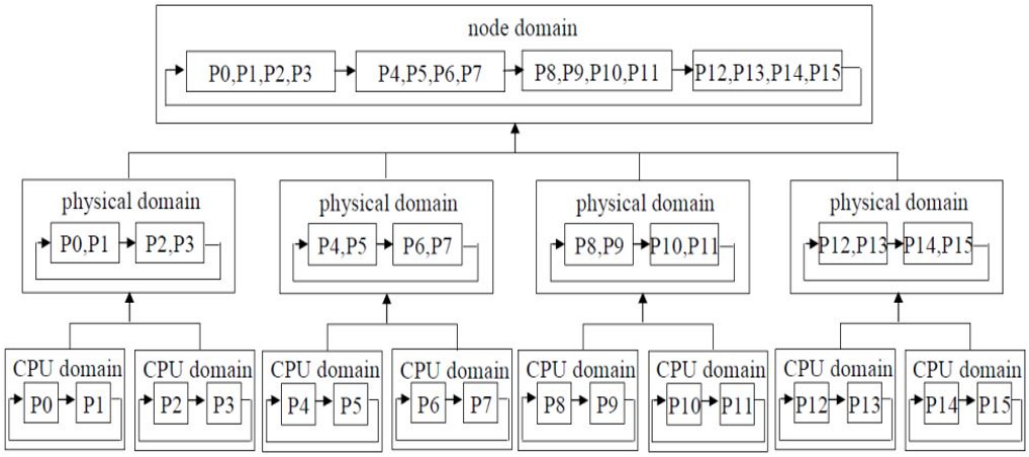


Figure A.1: Scheduling domains on a 16-core machine with four chips, each has two physical hyperthreaded cores (4x2x2).

A.2 Bug Description

For reproducing the bug, we use our 8-core machine (1x4x2) and create 120 tasks; namely fifteen tasks per available core. We group these tasks to 8 groups, and set the affinity of the tasks in each group to a specific core. Overall, we get the following correct initial state:

```
#> ./print_cpus_rq
cpu 0: nr running :   15
cpu 1: nr running :   15
cpu 2: nr running :   15
cpu 3: nr running :   15
cpu 4: nr running :   15
cpu 5: nr running :   15
cpu 6: nr running :   15
cpu 7: nr running :   15
```

In our next step (step 2), we turn-off one of the cores (core 3) by using the CPU-Hotplug mechanism:

```
#> ./print_cpus_rq
cpu 0: nr running :   17
cpu 1: nr running :   17
cpu 2: nr running :   18
cpu 3: nr running :   0
cpu 4: nr running :   17
cpu 5: nr running :   17
cpu 6: nr running :   17
```

```
cpu 7: nr running :   19
```

Note that at this point, except for a slight deviation in core 7, the system still behaves as expected. Next, in our final step (step 3), we use the CPU-Hotplug mechanism again, and bring core 3 back online. However, the resulting state is somehow surprising:

```
#> ./print_cpus_rq
cpu 0: nr running :   15
cpu 1: nr running :   15
cpu 2: nr running :   30
cpu 3: nr running :   1
cpu 4: nr running :   15
cpu 5: nr running :   15
cpu 6: nr running :   15
cpu 7: nr running :   15
```

The resulting state exhibits an unexpected behavior, as the system is not balanced.

A.3 Bug Explained

For explaining the above, we split it to two main questions:

1. Why does core 3 ends up with only 1 task?
2. Why does core 2 ends up with 30 tasks?

Core 3 behavior. Once a task reaches the point where all of its allowed core are offline, the Linux scheduler aggressively changes its affinity, forcing it to contain *all* the online cores in the system. Furthermore, this modification is permanent. Hence, if one of the original allowed cores is brought back online – the mask is not set back to its original state.

We illustrate the chain of events in our scenario in Table A.1. In step 1, a task is created with affinity to core 3 only; indeed, its `cpu_allowed_mask` contains only core 3. In step 2, when core 3 is turned off, the mask remains empty; thus the kernel proactively involves and forces the affinity mask to include all the online cores. Note that now, core 3 is offline and therefore is not included in that mask. Finally, in step 3, when the core gets back online, the mask remains unchanged; that is, without core 3.

As a results, all of the tasks that were set to run on core 3 lose their initial affinity configurations.

Core 2 behavior. Given the above insight, we can infer that following step 3, the system results with 15 tasks that can run on any core in the system except for core 3. In simple schedulers, these tasks would have spread among the system cores (excluding core 3), and the system would have become balanced again.

However, the Linux scheduler employs the scheduling domains approach, according to which the system must be balanced with respect to its domains in a top-down manner. Since cores 2 and 3 are logical cores which reside on the same physical core, they comprise a scheduling domain (see Figure A.1). As a result, the Linux load balancer strives to load them with the same load that the rest of the groups in their level has; namely, 30 tasks. Overall, we get that core 2 and 3 are loaded with 30 tasks, while 15 of them are unable to run on core 3. Hence, we get that all of these 30 tasks run only on core 2.

Proposed Solution. Given the above insights we propose a fix for the bug, and implement it in version 3.14.1 of the Linux kernel. Our solution basically changes the kernel intervention in affinity mask settings in cases where the affinity mask becomes empty. We add a memory to the intervention process, thus we prevent the changes from being permanent. If necessary, our fixed kernel indeed changes the affinity mask; however, we revert these changes if one of the cores in the original mask gets back online.

#	**Step**	`cpu_allowed_mask`
1	Task is created with affinity to core 3	0 0 0 1 0 0 0 0
2	Core 3 is turned off	1 1 1 0 1 1 1 1
3	Core 3 is turned on	1 1 1 0 1 1 1 1

Table A.1: Core 3 affinity mask changes.

Bibliography

[1] A. Kleen and Y. Huang. *MCE-Inject*, 2011. Available at https://github.com/andikleen/mce-inject.

[2] S. Achleitner, T. La Porta, P. McDaniel, S. Sugrim, S. V. Krishnamurthy, and R. Chadha. Cyber Deception: Virtual Networks to Defend Insider Reconnaissance. In *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats*, MIST '16, pages 57–68, New York, NY, USA, 2016. ACM.

[3] Alexey Kopytov. SysBench - A Modular, Cross-Platform and Multi-Threaded Benchmark Tool, 2016.

[4] AMD®. Machine Check Architecture. In *AMD64 Architecture Programmer's Manual*, volume 2, chapter 9. May 2013.

[5] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for Full System Simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[6] Ashok Raj. CPU Hotplug Support in Linux Kernel. In *Linux Documentation*.

[7] N. D. P. Avirneni and A. Somani. Low overhead soft error mitigation techniques for high-performance and aggressive designs. *IEEE Transactions on Computers*, 61(4):488–501, April 2012.

[8] U. Bandara and G. Wijayrathna. Detection of source code plagiarism using machine learning approach. In *International Journal of Computer Theory and Engineering*, IJCTE 2012 Volume 4, Number 5, 2012.

[9] M. Bauer. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux J.*, 2006(148):13–, Aug. 2006.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *22nd Symposium on Operating Systems Principles*. Association for Computing Machinery, Inc., October 2009.

[11] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[12] E. Biederman and K. Zak. *nsenter - Run Program With Namespaces of Other Processes*. Linux Man Pages, 2017.

[13] D. M. Blei. Probabilistic Topic Models. *Commun. ACM*, 55(4):77–84, Apr. 2012.

[14] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[15] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128, May 2007.

[16] O. Boiman and M. Irani. Similarity by composition. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 177–184, 2006.

[17] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, Nov. 2005.

[18] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[19] B. Bucsay. Chw00t: Breaking Unices' chroot() Solutions, 2015. Available at https://github.com/earthquake/chw00t.

[20] M. Bugliesi, S. Calzavara, R. Focardi, and M. Squarcina. Gran: Model Checking Grsecurity RBAC Policies. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 126–138, June 2012.

[21] L. Bustamante and H. Al-Asaad. Detection of soft errors through checksums in redundant execution systems. In *IEEE AUTOTESTCON, 2015*, pages 134–137, Nov 2015.

[22] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A Technique for Cheap Recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–16, Berkeley, CA, USA, 2004. USENIX Association.

[23] *Linux Programmer's Manual*. 4.10 edition, December 2016.

[24] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[26] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.

[27] Chef. *Chef – Automate Your Infrastructure*, 2017. Available at www.chef.io.

[28] C. Chen and M. Hsiao. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. *IBM Journal of Research and Development*, 28(2):124–134, March 1984.

[29] Y. Chen and B. Malin. Detection of Anomalous Insiders in Collaborative Environments via Relational Analysis of Access Logs. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pages 63–74, New York, NY, USA, 2011. ACM.

[30] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra. Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10, May 2013.

[31] Christer Weingel. The Linux Watchdog API. In *Linux Documentation*.

[32] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[33] Cisco. Cisco™Security MARS, 2017.

[34] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *Micro, IEEE*, 23(4):14–19, July 2003.

[35] N. D. Corporation. TOMOYO: A Security Module for System Analysis and Protection, 2015. Available at http://tomoyo.osdn.jp/.

[36] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, pages 266–280, New York, NY, USA, 2016. ACM.

[37] Y. David, N. Partush, and E. Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 79–94, New York, NY, USA, 2017. ACM.

[38] F. de Aguiar Geissler, F. L. Kastensmidt, and J. E. P. Souza. Soft Error Injection Methodology Based on QEMU Software Platform. In *Test Workshop - LATW, 2014 15th Latin American*, pages 1–5, March 2014.

[39] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[40] B. Döbel and H. Härtig. Who Watches the Watchmen? Protecting Operating System Reliability Mechanisms. In *The Eighth Workshop on Hot Topics in System Dependability*, Berkeley, CA, 2012. USENIX.

[41] S. Dolev and R. Yagel. Towards Self-Stabilizing Operating Systems. *Software Engineering, IEEE Transactions on*, 34(4):564–576, July 2008.

[42] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, 2014. USENIX Association.

[43] I. Egwutuoha, D. Levy, B. Selic, and S. Chen. A Survey of Fault Tolerance Mechanisms and Checkpoint/Restart Implementations for High Performance Computing Systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[44] B. Fechner, A. Garbade, S. Weis, and T. Ungerer. Fault Detection and Tolerance Mechanisms for Future 1000 Core Systems. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 552–554, July 2013.

[45] A. Garbade, S. Weis, S. Schlingmann, B. Fechner, and T. Ungerer. Fault Localization in NoCs Exploiting Periodic Heartbeat Messages in a Many-Core Environment. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 791–795, May 2013.

[46] A. Garbade, S. Weis, S. Schlingmann, B. Fechner, and T. Ungerer. Impact of Message Based Fault Detectors on Applications Messages in a Network on Chip. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 0:470–477, 2013.

[47] S. Gaudin. Ex-UBS Systems Admin Sentenced To 97 Months In Jail, December 2006.

[48] G. Gavai, K. Sricharan, D. Gunning, R. Rolleston, J. Hanley, and M. Singhal. Detecting Insider Threat from Enterprise Social and Online Activity Data. In *Proceedings of the 7th ACM CCS International Workshop on Managing Insider Security Threats*, MIST '15, pages 13–20, New York, NY, USA, 2015. ACM.

[49] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[50] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. M. Lê, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, and M. Valero. TERAFLUX: Harnessing Dataflow in Next Generation Teradevices. *Microprocessors and Microsystems*, 38(8, Part B):976 – 990, 2014.

[51] T. Gleixner, P. E. McKenney, and V. Guittot. Cleaning Up Linux's CPU Hotplug for Real Time and Energy Management. *SIGBED Rev.*, 9(4):49–52, Nov. 2012.

[52] S. Godard. *SYSSTAT Utilities - System Performance Tools for the Linux Operating System*, 2016. Available at http://sebastien.godard.pagesperso-orange.fr/.

[53] Google. gRPC: A High Performance, Open-Source Universal RPC Framework, 2017. Available at https://grpc.io/.

[54] G. Heiser. Many-Core Chips — A Case for Virtual Shared Memory. In *Workshop on Managed Many-Core Systems*, Washington DC, USA, Mar 2009.

[55] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[56] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. In *ACM SIGOPS Operating Systems Review*, 2006.

[57] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[58] Hewlett Packard. ArcSight ESM, 2017.

[59] IBM® X-Force Research. *2016 Cyber Security Intelligence Index*, 2016.

[60] Intel®. OS Machine Check Recovery on Itanium®-Based Systems. Aug. 2008.

[61] Intel®. Intel® Cache Safe Technology. In *The Intel® Itanium® Processor 9300 Series*. 2014.

[62] Intel®. Instruction Set Reference. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2, chapter 4. Dec 2015.

[63] Intel®. Intel TSX Recommendations. In *Intel 64 and IA-32 Architectures Optimization Reference Manual*, chapter 12. Sep 2015.

[64] Intel®. Intel® Transactional Synchronization Extensions. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 1, chapter 15. Dec 2015.

[65] Intel®. Machine-Check Architecture. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 15. Dec 2015.

[66] Intel®. RAPL Interface. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 14. Dec 2015.

[67] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. VM3: Measuring, Modeling and Managing VM Shared Resources. *Comput. Netw.*, 53(17):2873–2887, Dec. 2009.

[68] Jeffrey Katcher. Postmark: a New File System Benchmark. Technical report, October 1997. TR3022, Network Appliance.

[69] R. V. Johnson, J. Lass, and W. M. Petullo. Studying naive users and the insider threat with simpleflow. In *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats*, MIST '16, pages 35–46, New York, NY, USA, 2016. ACM.

[70] Jonathan Corbet. *Scheduling Domains*, 2004. Available at http://lwn.net/Articles/80911/.

[71] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[72] C.-K. Koh, W.-F. Wong, Y. Chen, and H. Li. The Salvage Cache: A Fault-Tolerant Cache Architecture for Next-Generation Memory Technologies. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 268–274, Oct 2009.

[73] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[74] J. D. Kornblum. Implementing bitlocker drive encryption for forensic analysis. *Digit. Investig.*, 5(3-4):75–84, Mar. 2009.

[75] D. Kravets. San Francisco Admin Charged With Hijacking City's Network, July 2008.

[76] A. Lenharth, V. Adve, and S. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 49–60, 12 2008.

[77] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, Sept 2017.

[78] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust-Management Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 114–130, Washington, DC, USA, 2002. IEEE Computer Society.

[79] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *CoRR*, abs/1801.01681, 2018.

[80] LSE. *Linux Scalability Effort Homepage*, 2004. Available at https://lse.sourceforge.net/.

[81] *Linux Containers*, 2017. Available at https://linuxcontainers.org/.

[82] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for Multicore Architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[83] W. Mauerer. Professional Linux Kernel Architecture, 2008.

[84] B. McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[85] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-Copy Update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[86] P. E. Mckenney and S. Boyd-wickizer. RCU Usage in the Linux Kernel: One Decade Later. Technical Report, sep 2012.

[87] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), Mar. 2014.

[88] Microsoft ®. Windows Hot Add CPU.

[89] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[90] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[91] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri. Linux Kernel Hotplug CPU Support. In *Linux Symposium*, 2004.

[92] A. Nakajime. Similarity calculation method for binary executables, 2017.

[93] B. H. Ng and A. Prakash. Expose: Discovering potential binary code re-use. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 492–501, July 2013.

[94] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM.

[95] R. Nikhilesh. FUSE: Add Support for Passthrough Read/Write. February 2016. Available at https://lwn.net/Articles/674286/.

[96] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-Aware Job Scheduling for BlueGene/L Systems. In *IPDPS*, 2004.

[97] *OpenVZ*, 2017. Available at openvz.org.

[98] A. Orebaugh, G. Ramirez, J. Beale, and J. Wright. *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing, 2007.

[99] S. Panneerselvam and M. M. Swift. Chameleon: Operating System Support for Dynamic Processors. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 99–110, New York, NY, USA, 2012. ACM.

[100] D. A. Patterson. An Introduction to Dependability. *login*, pages 61–65, 2002.

[101] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403*, KSEM 2015, pages 547–553, Berlin, Heidelberg, 2015. Springer-Verlag.

[102] L. Poettering, K. Sievers, H. Hoyer, D. Mack, T. Gundersen, and D. Herrmann. systemd-nspawn, November 2016. Available at wiki.archlinux.org/index.php/Systemd-nspawn.

[103] D. Price and A. Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 241–254, Berkeley, CA, USA, 2004. USENIX Association.

[104] Puppet. *Puppet - The shortest path to better software*, 2017. Available at https://puppet.com/.

[105] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch. Methods for Fault Tolerance in Networks-On-Chip. *ACM Comput. Surv.*, 46(1):8:1–8:38, July 2013.

[106] A. Rajgarhia and A. Gehani. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM.

[107] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[108] N. Rath. FUSE: Filesystem in Userspace, 2017. Available at http://fuse.sourceforge.net/.

[109] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving Per-node Efficiency in the Datacenter with New OS Abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 25:1–25:8, New York, NY, USA, 2011. ACM.

[110] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[111] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 21–28, New York, NY, USA, 2010. ACM.

[112] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *SOSP*, 2007.

[113] D. Rossi, N. Timoncini, M. Spica, and C. Metra. Error Correcting Code Analysis for Cache Memory High Reliability and Performance. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[114] A. Roytman, S. Govindan, J. Liu, A. Kansal, and S. Nath. Algorithm Design for Performance Aware VM Consolidation. Technical report, 2013.

[115] M. B. Salem, S. Hershkop, and S. J. Stolfo. *A Survey of Insider Attack Detection Research*, pages 69–90. Springer US, Boston, MA, 2008.

[116] J. H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, July 1974.

[117] N. Santos, R. Rodrigues, and B. Ford. Enhancing the os against security threats in system administration. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 415–435, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[118] R. D. Schlichting and F. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, Aug. 1983.

[119] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, 2009.

[120] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 391–406, New York, NY, USA, 2013. ACM.

[121] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.

[122] J. Song, J. Wittrock, and G. Parmer. Predictable, Efficient System-Level Fault Tolerance in C^3. *2013 IEEE 34th Real-Time Systems Symposium*, 0:21–32, 2013.

[123] Splunk. Splunk™User Behavior Analytics, 2017.

[124] S. Srikantaiah, A. Kansal, and F. Zhao. Energy Aware Consolidation for Cloud Computing. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association.

[125] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186, June 2004.

[126] Srivatsa S. Bhat. *CPU Hotplug: stop_ machine()-Free CPU Hotplug*. Available at http://lwn.net/Articles/533553/.

[127] S. J. Swamidass, C.-A. Azencott, K. Daily, and P. Baldi. A croc stronger than roc: Measuring, visualizing and optimizing early retrieval. *Bioinformatics*, 26(10):1348–1356, 2010.

[128] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, Nov. 2006.

[129] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[130] Symantec. Symantec™Security Information Manager, 2013.

[131] B. Toxen. *Real World Linux Security*. Prentice Hall Professional Technical Reference, 2nd edition, 2002.

[132] B. Van Zant. SSH Certificate Authority, 2017. Available at github.com/cloudtools/ssh-ca.

[133] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan. 2008.

[134] Varonis. "Varonis™Enterprise Security", 2017.

[135] J. Verble. The NSA and Edward Snowden: Surveillance in the 21st Century. *SIGCAS Comput. Soc.*, 44(3):14–20, Oct. 2014.

[136] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[137] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer. Architectural Support for Fault Tolerance in a Teradevice Dataflow System. *International Journal of Parallel Programming*, pages 1–25, 2014.

[138] S. Weis, A. Garbade, and T. Ungerer. Design Exploration of FDUs and Core-Internal Fault-Detection. *Exploiting Dataflow Parallelism in Tera-Device Computing*, 2010.

[139] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

[140] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[141] X. Xu and M.-L. Li. Understanding Soft Error Propagation Using Efficient Vulnerability-Driven Fault Injection. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[142] G. Yalcin, O. Unsal, and A. Cristal. FaulTM: Error Detection and Recovery Using Hardware Transactional Memory. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 220–225, San Jose, CA, USA, 2013. EDA Consortium.

[143] G.-C. Yang. Reliability of Semiconductor RAMs with Soft-Error Scrubbing Techniques. *Computers and Digital Techniques, IEE Proceedings*, 142(5):337–344, Sep 1995.

[144] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel ® Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

[145] W. T. Young, A. Memory, H. G. Goldberg, and T. E. Senator. Detecting Unknown Insider Threat Scenarios. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 277–288, May 2014.

[146] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe. Decoupling Cores, Kernels, and Operating Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 17–31, Broomfield, CO, Oct. 2014. USENIX Association.

[147] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.

[148] zynamics. *BinDiff*, 2011. Available at http://www.zynamics.com/.

Hebrew Section

אנו מציעים שיטה למציאת זהויות בין קודים בינאריים שעברו הידור על ידי מהדרים שונים ועבור פלטפורמות מסוגים שונים. שיטתנו שואלת רעיונות מתחום עיבוד תמונה ועושה שימוש במסווגים מבוססי למידת מכונה כדי לחזות את ההסתברויות להתאמה. בבסיס השיטה אנו מפתחים אלגוריתם שבאמצעותו ניתן לייצג פרוצדורה או קטע קוד בייצוג בינארי על ידי וקטור של מספרים, ובהמשך בהסתמך על ייצוג זה ניתן להפעיל שיטות מתמטיות מהירות כגון רשתות נוירונים כדי לחזות זהויות בין קטעי קוד שונים. השיטה המוצעת מהירה מאוד ומדויקת, כך שהיא מאפשרת לבצע חיפוש עבור קוד פגיע במאגרים גדולים של קטעי קוד בזמן מעשי עם מספר נמוך של התראות שווא.

על מנת להתמודד עם אתגרי האמינות הניצבים בפני מערכות מרובות ליבות, אנו מציגים בחיבור זה אסטרטגיה להסרה פתאומית של ליבה באמצעות מערכת ההפעלה. במערכות מרובות ליבות הנפוצות כיום קיימת מגמה של הקטנת גדלי הטרנזיסטורים והמתחים שבהם. עקב כך, ההסתברויות לתקלות בחומרה גדלות, ובשילוב עם מספר הליבות ההולך וגדל בכל מערכת, הסיכוי לתקלת חומרה באחת מליבות המערכת הולך ונהיה משמעותי. אולם, במערכות המריצות מערכות הפעלה נפוצות כגון לינוקס או ווינדווס, כל תקלה בחומרה של אחת הליבות יכולה להפיל את כלל המערכת ולמנוע את המשך תפקודה, למרות הימצאותן האפשרית של עשרות או מאות ליבות תקינות אחרות. האסטרטגיה שאנו מציגים מעוצבת כך שניתן לשלבה במערכות הפעלה קיימות, ובאמצעותה ניתן להתגבר על תקלות שמקורן בחומרה בתוך שבב העיבוד. המגנגון שאנו מציעים אינו מוסיף תקורה למערכת, ובמקרה של תקלה והתאוששות המערכת ממשיכה לרוץ ולתפקד, מלבד העובדה שהליבה התקולה אינה זמינה. על ידי שימוש ייחודי בטרנזקציות זיכרון בחומרה, המנגנון שאנו מציעים מסוגל לשרוד תקלות שארעו בזמן הרצת ישומי משתמש או ישומי מערכת כגון קוד של גרעין מערכת ההפעלה.

עבור כל הפתרונות המוצעים לעיל, אנו מממשים אב-טיפוס ומבצעים על בסיסו הערכות ומדידות אמפיריות במערכות אמיתיות ובסביבות פיתוח. התוצאות מראות שהגישות המוצגות אפקטיביות ומספקות שיפור משמעותי ביחס למערכות וכלים קיימים.

המחקר נעשה בפקולטה להנדסת חשמל על שם ויטרבי, הטכניון – מכון טכנולוגי לישראל.

תקציר

במחקר זה אנו עוסקים בסוגיות מפתח בתחומי אבטחת מידע ואמינות חומרה במערכות מחשבים, שעלו עקב התגברות סכנות אבטחת המידע ואימוץ גישת היתרון לגודל על ידי התעשיה. אנו דנים בשלושה נושאים: הגנה על ארגונים מפני עובדים בעלי הרשאות מרובות, חיפוש זהויות בין קודים בינאריים, ועמידות בפני תקלות חומרה במערכות מחשבים נפוצות. הפתרונות שאנו מציעים כוללים בין היתר הפשטות חדשות עבור גרעין מערכת ההפעלה, מנגנונים חדשים המסוגלים להשתלב במערכות הפעלה קיימות, שימוש ייחודי ברכיבי חומרה ותכנה, ושיטות הנעזרות ברעיונות מתחום למידת מכונה.

אנו מציגים גישה ייחודית לאבטחת מידע עבור ארגונים המיועדת להגנה מפני מנהלי המידע. מתוקף תפקידם, מנהלי המידע בארגון הם בעלי הרשאות מרובות, אך כפי שהודגם במקרה של אדוארד סנודן, הן יכולות להיות מנוצלות לרעה לגניבת מידע רגיש או מסווג. השיטה שאנו מציגים מסוגלת מחד גיסא להגביל את ראות המערכת של מנהלי המידע תוך כדי ניטור פעילותם, ומאידך גיסא לשמור על ההרשאות המרובות הניתנות להם כדי לבצע את עבודתם, אם כי רק על משאבים הנמצאים בטווח הראות המוגבלת. הפיתרון המוצג עושה שימוש ייחודי במנגנוני מערכת ההפעלה על מנת ליצור סביבה מבוקרת שאנו מכנים "מכולה מחוררת". סביבה זו מסוגלת להגיע לאיזון בין הצרכים המנהליים הדרושים לטיפול בתקלה ובין צרכי אבטחת המידע הארגונית. עבור כל בעית מחשוב, המערכת שלנו חוזה לאילו משאבים נחוצה גישה כדי לטפל בה, בונה מכולה מחוררת עם בידוד מתאים, ופורשת אותה במכונת היעד כך שניתן יהיה לטפל בתקלה בצורה מבוקרת. יתרה מזאת, למקרים שבהם התחזית אינה מדויקת, אנו מספקים אמצעי לעקיפת הבידוד המוכתב על ידי המכולה המחוררת, תוך כדי ניטור ותיעוד הפעולות הנעשות דרכו.

I

המחקר נעשה בהנחיית פרופ' עדית קידר
בפקולטה להנדסת חשמל על שם ויטרבי בטכניון – מכון טכנולוגי לישראל

תודתי נתונה לטכניון – מכון טכנולוגי לישראל,
לקרן מאייר, לאינטל ישראל,
לקרן משפחת לאונרד ודיאן שרמן,
ל-Hasso-Plattner Institute בגרמניה,
למרכז המחקר באבטחת סייבר ע"ש הירושי פוג'יוארה, ולמערך הסייבר הלאומי
על התמיכה הכספית הנדיבה בהשתלמותי.

שיפור האמינות ואבטחת המידע במערכות מחשבים בעזרת מערכת ההפעלה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

נעם שלו

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אב תשע"ח חיפה יולי 2018