Eliminating the Long-Running Process

Separating Code and State

by

Patrick Colp

B.Sc., The University of British Columbia, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

January 2010

Abstract

Many critical services are necessarily long-running. However, this creates a large temporal surface that is an alluring target for attackers, both in terms of the increased opportunity to find an exploit and the length of time a service is owned once exploited. While in some instances it may be possible to perform periodic restarts to reduce the window of exploitation and return a service to its fresh, unexploited operational status, this carries with it a high cost. The more often it is restarted, the larger the unavailability due to reinitialisation of the service. Furthermore, it must recover its persistent state, which is not always possible.

In order to protect these services, we propose a form of virtual machine disaggregation which partitions a service into two parts: code (logic) and state (data). Each lives in its own virtual machine, with communication performed over a narrow, well defined interface on which policy can be externally enforced to ensure correctness. This separation enables a service to be continually restarted by rolling back only the code virtual machine to a snapshotted known good state, which can be measured and attested. This prevents exploits from persisting while still maintaining good performance.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Dr. William Aiello and Dr. Andrew Warfield for their careful guidance, without which I would have been completely lost, as well as Dr. Norm Hutchinson for his invaluable advice as second reader. I would also like to thank the members of the Networks, Systems, and Security lab, in particular Geoffrey Lefebvre and Brendan Cully, for all their help with my questions. Finally, I would like to thank the administrative staff for handling the miscellaneous details associated with being a graduate student so that I could focus on my research.

Dedication

For my parents, for their patience and support.

Chapter 1

Introduction

1.1 Problem

Many critical services are necessarily long-running. These can range from general applications, such as databases and web servers, to device drivers, to system specific services, like XenStore.[1] In some cases, these services are intrinsically tied to the operation of the system and brought up during boot-time or initialisation and live until the machine is shut down.

Whatever the nature of the long-running service, they all share something in common: a large temporal surface. These services can live for days, months, or even years without being restarted. This provides an attacker with a long time to attempt to exploit a service and, once the attack succeeds, a long time to own it, as he will have access until the next time it is restarted. In addition, after the exploit has been discovered, it will be relatively quick and easy for the attacker to regain control of the service after the restart (again, owning it for a long time).

Other research has improved the security of long-running processes, but their objective was not the reduction of the temporal surface. Many approaches have implemented privilege separation, dividing out the small subset of the program which performs privileged operations from the rest, which runs at a reduced privileged level [5, 14, 16, 18]. However, while these approaches may help reduce the privilege escalation attack surface, they do not address problems associated with the long-running nature of the services. Other

[1]XenStore is a service for the Xen Virtual Machine Monitor.

approaches have suggested using checkpointing and rollback as a **reactive** defence against attack, however this can require hardware modifications [20].

Our work aims to overcome these limitations by proposing a novel secure system architecture designed to address the inherent security concerns of long-running services. In our architecture, the *code* (logic) and *state* (data) of a program are separated. This enables us to constantly refresh the code portion of the service, using **proactive** execution rollback, to reduce the temporal surface. Furthermore, this separation makes reasoning about the interaction between the two parts explicit, enabling information control flow policy to be enforced on this interaction.

We show the viability of our approach by presenting a prototype implementation of a real-world service. We target XenStore, a service which not only must exist for the entire duration of the operation of the system, but which is intrinsically tied to it. This demonstrates that even long-running services which are mission-critical and brought up during the initialisation of the system as a whole can benefit from and be applied to our architecture.

1.2 Organisation

A review and discussion of related research is explored in Chapter 2, including work in security and other areas, such as high availability. Chapter 3 presents an overview of our architecture. The specific details of the prototype implementation are described in Chapter 4 along with our initial results. Chapter 5 discusses areas of future research and Chapter 6 provides a final overview and conclusion.

Chapter 2

Related Work

2.1 Privilege Separation

Privilege separation is a technique which divides a program into two parts. The privileged operations are all placed in one part and the remainder of the application is placed in the other. Thus, the amount of code which needs to be trusted and verified is reduced to just that of the privileged portion. This is an implementation of the *principle of least privilege*, which states that each component in a system should only have as much privilege as it requires to perform its function.

2.1.1 Libraries

Kilpatrick (2003) [14] presents a library designed to make privilege separation easy. A program specifies the subset of privileged operations that it needs in order to function. It then makes calls into the library to have these operations performed on its behalf by a privileged agent.

Murray and Hand (2008) [16] extend this concept into the virtualised environment. Instead of just separating the privileged operations into a library, this library is then disaggregated into its own virtual machine. The mechanism is extended further to allow for binary compatibility with current operating systems and applications.

These techniques explicitly separate out privileged operations. It is reasonable to consider that some of the long-lived state associated with these operations is also separated out. This prevents a compromised service from performing any privileged action it likes. However, it does not protect against attacks on the service itself. So, while this may prevent an exploited service from gaining privileged access to the machine running the service, it does not prevent an exploited service from providing erroneous results to clients.

2.1.2 Program Partitioning

Provos *et al.* (2003) [18] and Brumley and Song (2004) [5], similar to the library approaches mentioned above, separate out a service's privileged operations. However, in these approaches, they use the idea of a monitor and slaves. A monitor contains privileged code and little else. Its task is to launch unprivileged slave processes which perform the rest of the operations. In both these works OpenSSH is used as their example program. The monitor contains only the code needed to authenticate a user, which then launches a slave process which runs at the authenticated user's privilege level. Provos *et al.* do this separation manually, while Brumley and Song use programmer provided annotations to perform the separation automatically.

This form of privilege separation suffers from the same drawback as the library approaches above. While it protects the privileged part of the service, it does not prevent an attacker from hijacking the user (slave) session.

2.1.3 Wedge

Wedge (Bittau *et al.* (2008)) [3] is a system designed to help separate a program into least-privilege components. It consists of a run-time component which aids a developer to find potential compartment boundaries and a series of OS primitives which force the privileges to be made explicit. Compartments are created by calling a variant of *fork* which only maps those areas of memory that the security policy allows. Privileged operations are performed using callgates, which are bits of code with a higher privilege level. This approach addresses the principle of least-privilege, protecting high privilege operations from low privilege callers.

2.2 Information Flow Control

In order to keep high security inputs protected, Smith and Thober (2006) [21] suggest refactoring programs into high and low security modules. The high security modules can be accessed by the low security modules through public method calls. This keeps the high security data protected from direct tampering by the low security modules. This allows programmers to more easily add information flow controls to the program. However, the low and high security components both operate within the same address space. Thus, if an attacker is able to break outside the bounds of a low security module, for example through a buffer overflow exploit, then the high security data is at risk.

2.3 Restarts

Restartable systems is a well researched area. Many systems have been created, from restarting the entire system to restarting components of an application, as a way of recovering from errors or promoting freshness. However, most of the work to date on this has been for high availability and not for security.

2.3.1 Recovery Box

Baker and Sullivan (1992) [1] discuss using non-volatile memory to store certain information which can be used to reduce the time it takes to recover from a failure. The stored information is long-lived program state, such as files open on client machines. The focus of this work is to reduce recovery time from a system failure to in turn reduce the down time of provided services. The only validation or control of information is checksumming of the data to ensure that it has not been corrupted (and if it has, to perform a normal, full restart of the system).

2.3.2 Recursive Restartability

Candea, Fox, *et al.* (2001, 2002) [6, 8] extend the restart technique by proposing that restarts be performed at multiple levels. Restarting only the failed or failing components reduces the overall time for recovery. If restarting at the smallest granularity does not solve the problem, then a restart is performed at a higher granularity. However, this work requires that the restartable components are stateless or contain only soft-state. This makes it impractical for general usage. The focus of this work is also for high availability, accepting that bugs in software are inevitable.

2.3.3 Crash-Only Software

Candea and Fox (2003) [9] propose crash-only software as a way to deal with restartability and long-lived state. They suggest that software should be designed to crash. This ensures that a program makes no assumptions about resources being released cleanly and is thus able to easily recover from a crash. This in turn makes performing restarts easier, as there is no need to deal with long-lived state since a program will recover itself. However, the time granularity at which restarts can be performed is limited, since the time to recover becomes an increasingly dominant cost as the time between restarts is reduced.

2.3.4 Microreboots

A microreboot, as described by Candea, Fox, *et al.* (2004) [10], is the individual restarting of application components. These restarts are performed at a fine granularity in order to reduce down-time due to application failure. In the example application, a web application for a modified version of

JBoss, session state must be moved outside the application so that it can survive the microreboots. Two different ways of separating out this session state are proposed. The first is to move it inside JBoss. This allows the state to only survive microreboots. The second method moves the session state into a distributed store system. Stored this way, the state can survive microreboots, JVM restarts, and even complete node restarts. However, as in their previous work, the focus is on high-availability and speed of recovery from application failure. They do not consider the security of the data nor impose any information flow controls. Instead, they were considering only ways to remove the state from their application so that when it fails and is restarted, the session state is not lost.

2.4 Other

Chandra, Lam, *et al.* (2005) [17] propose The Collective, a system to centrally manage virtual appliances. It uses virtual machines, which are downloaded from a centralised location onto a user's machine. Each time the machine is restarted, a fresh copy of the virtual machine is fetched (using differences between the current image and the updated image). Thus, the machine is always restored to a known-good state. User data (*e.g.*, a home directory) is stored separately and persists beyond a reboot. This helps prevent viruses from persisting. Furthermore, updates can be managed more easily. This work focuses on system-level freshness and security, rather than on individual programs. This technique would also not be applicable to long-running processes because the granularity is too coarse.

Chapter 3

Architecture

The goal of the work described in this thesis is to address the security concerns inherent in long-running processes. In this chapter, we provide an overview of our proposed architecture. At a high level, a long-running process is separated into two parts. Each part is placed in its own virtual machine to provide strong isolation between the components. A narrow communication channel is placed between the two parts, which can be easily monitored and subjected to policy enforcement.

3.1 Threat Model

In our work, we assume a relatively simple threat model. We consider systems that provide a long-running external service which is the target of attack. Examples of the type of service in this model include web-servers, device drivers, or even virtual machines.

More specifically, we target long-running exploits in long-running services themselves. These exploits could be used in a variety of ways: to inject attacks into web-pages by including malicious code which will be run on the victim's web browser; to snoop on network traffic, reads/writes from/to disk, or private information contained within a database; or to shut down a competitor's running virtual machines. We are less interested in attacks on a service for other purposes, such as privilege escalation exploits.

Our design cannot completely eliminate all exploits in our threat model. Instead, it provides an architecture which makes it easier to reason about execution and information flows. This in turn makes it easier to protect against these exploits.

3.2 Overview

The fundamental problem with long-running processes is their very nature. As long as they remain long-running, their temporal surface is large. Thus, the solution is to not have long-running processes. Unfortunately, this is a seemingly contradictory statement. Some processes are necessarily long-running, but in order to secure them, they cannot be so. In order to solve this conundrum, we must consider what makes a long-running process long-running. Typically, this type of service has a processing loop which handles a subset of the overall computation per iteration, such as one single operation. However, associated with these computations is state. This state is what, ultimately, is long-lived.

With the current programming architecture and paradigm, the state and the logic of a program are entangled. We propose an architecture which makes these two parts distinct. The actual code of the program, its logical computation, is separated from the (long-lived) state upon which it acts. The interaction between the two is done through an explicit, narrow communication channel. We further propose to place policy enforcement on this channel, separating it from the code and state. This separation enables the program to essentially be run once, repeatedly.

Constantly restarting the process ensures the code being run is always fresh. Any exploit designed to affect the operation of the program can only persist for as long as one restart. We propose the smallest granularity of restartability. As an example, consider Apache 1.3. The original code, with unrelated areas removed for brevity, is shown in Figure 3.1. The while loop processes a single client request. This loop can be removed, so that instead the program starts, processes one client request, then terminates (see Figure 3.2). The only change to the structure of the program is that instead of a while loop,

```
int main(int argc, char *argv[]) {
  ...
  while ((r = ap_read_request(conn)) != NULL) {

    if (r->status == HTTP_OK)
      ap_process_request(r);

    if (!conn->keepalive || conn->aborted)
      break;

    ap_destroy_pool(r->pool);
  }
 ...
}
```

Figure 3.1: Unmodified Apache 1.3 code

there is an if statement. When this new program is run repeatedly, then the overall result will be the same: client requests will be processed one at a time, in a loop, until the program is completely terminated.

This freshness allows us to reason more easily about the code. While it is easy to measure and attest software as it is loaded from disk, remeasuring and reattesting it while it is running is a very difficult, if not impossible, problem. However, by constantly reverting to a known good fresh copy of the code, we can measure and attest it at each invocation, assuring that the code has not been tampered with. Further more, the verifiability of the program is made easier. Previously, the entire state-space of the program needed to be explored, which could lead to the state space explosion problem. Our proposed architecture reduces the state that needs to be verified to that which can be reached within a single operation. Since each iteration only handles one request at a time, each individual request type can be explored independently.

The rest of this chapter will describe the architecture in more detail.

```
int main(int argc, char *argv[]) {
  ...
  if ((r = ap_read_request(conn)) != NULL) {

    if (r->status == HTTP_OK)
      ap_process_request(r);

    if (!conn->keepalive || conn->aborted)
      break;

    ap_destroy_pool(r->pool);
  }
 ...
}
```

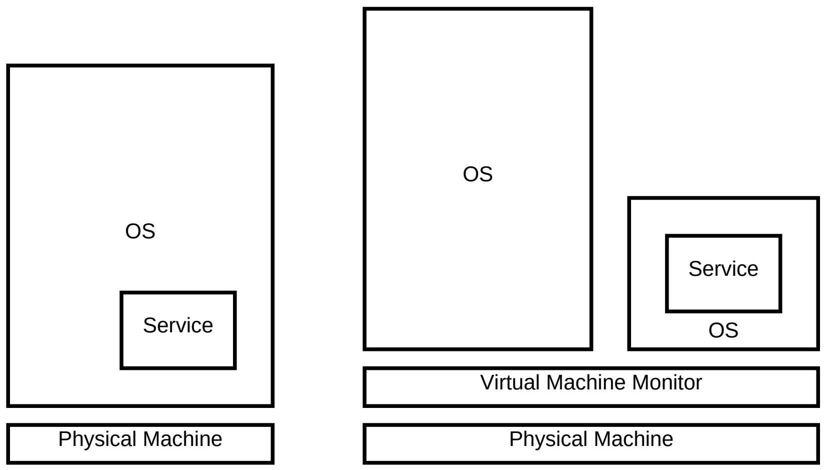
Figure 3.2: Run-once Apache 1.3 code

3.3 Separation

The portion of a program which makes it intrinsically long-lived is its state. If all a program does is statelessly perform operations, then it is trivially restartable since there is no difference between the first run and the millionth. However, most long-running programs also have long-lived state. In order to be able to constantly refresh the program, this state must be stored.

One option is to store it on disk and reload it each time the program is restarted. However, this carries with it two problems: there is a high overhead writing to and reading from disk with such frequency and the file is relatively easily exploitable.

Instead, we use virtual machines to perform the separation. Disaggregating the service even into a single virtual machine is already an advantage from a security perspective (see Figure 3.3). Virtual machines provide a strong isolation container. Aside from a minimal operating system, the service is the only thing running in the virtual machine. An attacker that is able to exploit the service and escape into the rest of the system is restrained to the



(a) Original (b) Disaggregated service

Figure 3.3: The original and disaggregated versions of the architecture

virtual machine in which the service is running. To cause damage to the rest of the system, the virtual machine must also be exploited. This layering of isolation levels provides *Defence in Depth*.

We take this one step further and separate the code, which is the portion of the program which performs the actual logical execution, and the long-lived state into two different virtual machines (see Figure 3.4). The virtual machine which contains the code can now be restarted, with the long-lived state preserved in the other virtual machine.

A potential vector of attack is to infect data being placed into the state store. This data could then be used later which would cause the exploit to have an effect. To address this, we use a narrow channel of communication between the virtual machine with the code and the one with the state. This communication is done using a simple wire format which is easy to inspect and reason about. Policy can then be enforced upon this communication (see Figure 3.5) to help keep it secure.

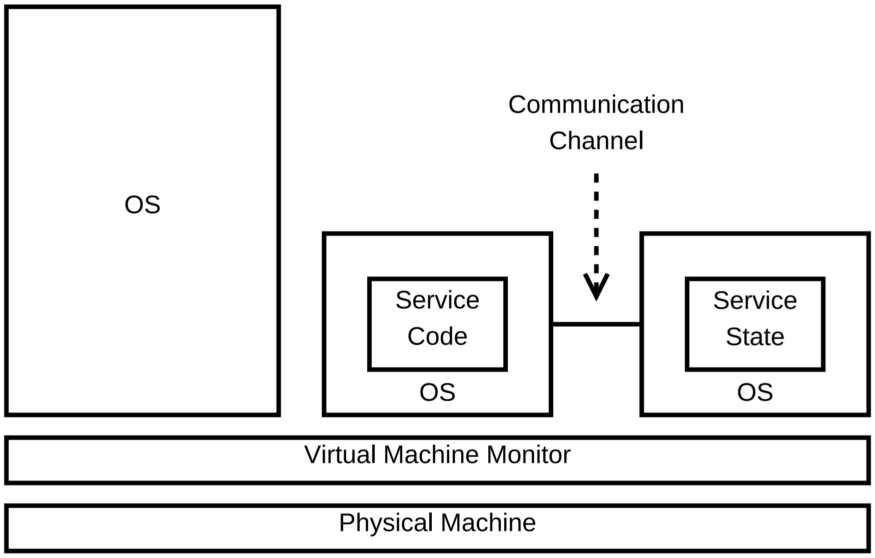


Figure 3.4: Fully separated service into code and state virtual machines

3.3.1 Code Virtual Machine

The code virtual machine retains the core of the original service. Long-lived state must be explicitly separated out and replaced with requests to the state virtual machine. These requests can cause new memory to be allocated, freed, or accessed. When accessing data, it must be copied over the wire to the code virtual machine, which can then use its local copy. Any changes which are made (and that should be kept) need to be reflected back onto the data stored in the state virtual machine.

Since the intention of the program is to now execute only once, some of the control flow will change. In the example above in Figures 3.1 and 3.2, the while loop was replaced with an if statement. There may be other changes that can or need to take place in order for the service to operate properly.

Currently, all the analysis and transformation must be done manually. It is up to the developer to locate which pieces of state must be separated out. For small projects, this can be a relatively simple task, but for larger projects, it is likely to be tedious. The control flow modifications must also

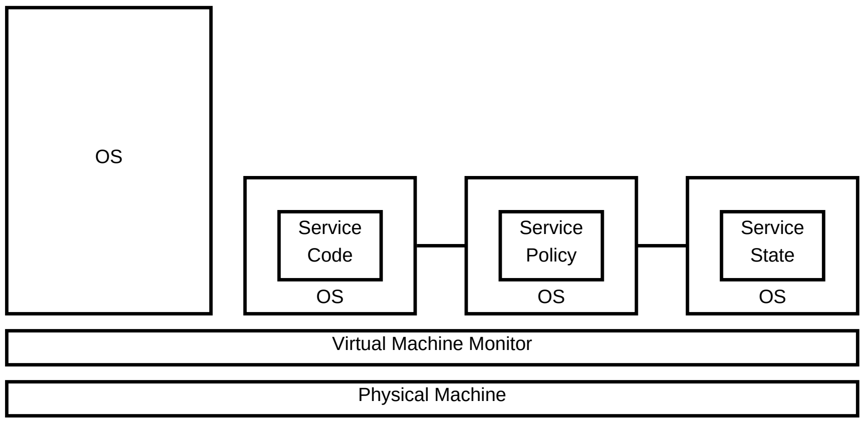


Figure 3.5: Fully separated service with policy enforcement

be determined by the developer. All of this requires a new way of thinking about the normal execution flow of a program.

3.3.2 State Virtual Machine

Our architecture is designed to eliminate long-running processes. However, long-lived state is still long-lived state. Therefore it must reside somewhere which must necessarily be long-lived. The state store is designed to be as simple as possible. This makes the code easy to reason about and verify.

The sole task of the state store is to store state. It performs no checks on the data being stored or retrieved. Instead, this is left up to the policy enforcement virtual machines or, in some cases, the code virtual machine.

3.3.3 Communication Channel

Communication between the code virtual machine and the state virtual machine happens over an explicit and narrow communication channel. This makes it easy to monitor the communication and enforce policy upon it. The communication protocol is designed to be as simple as possible.

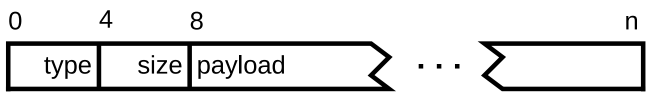


Figure 3.6: Communication packet wire format

The general message format is shown in Figure 3.6. It consists of three parts: a message type (4 bytes), a payload size (4 bytes), and a payload (`size` bytes). The total packet size is the size of the header (8 bytes) and the size of the payload (`size` bytes), for a total of $n = 8 + \texttt{size}$. The maximum length of a message payload is $2^{32} - 1$ (or 4,294,967,295), which is the maximum size of an unsigned 32-bit (4 byte) integer.

Only the code virtual machine communicates with the state store. It may be desirable to have multiple code virtual machines access the same state store (see Section 5.3), however this would require a way to authenticate which virtual machines may communicate with which state stores. This type of policy could be enforced by the virtual machine monitor itself.

3.3.4 Policy Enforcement

Separating the code and state and constantly restarting the code virtual machine prevents long-lived exploits from persisting there. A key vector of attack is to corrupt the long-lived state such that it is possible to cause the short-lived code portion to retrieve the corrupted state and continually get infected or exploited. However, the clean separation makes the information flow in the service explicit.

In the original version of a service, any policy decisions enforced on the data is most likely tangled up with the code itself. For example, limiting the size of a URL to fetch. With this new architecture, it is possible and desirable to remove as much of this policy as possible and place it into policy enforcement virtual machines. This makes both the core program code and the specific policy being enforced easier to reason about.

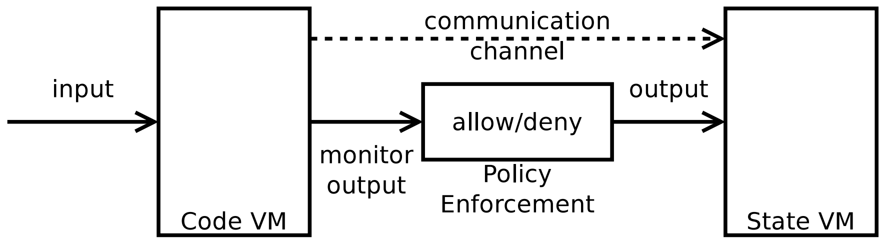


Figure 3.7: Allow/deny policy enforcement on output from the code VM

Policy enforcement modules can be placed upon the communication channel between the two virtual machines. These modules can then be used to perform information flow control. By using virtual machines as the containers for policy modules, it is possible to apply this architecture to the policy modules themselves. With stateless policy, this is trivial. The policy enforcement virtual machine can be restarted with the code virtual machine, restoring both to known good states. If the policy requires some state, depending on the longevity of this state, the module may run for several iterations of the code virtual machine before being restarted or its state could be stored either in the current state store or in a dedicated state store. The policy enforcement module can then be restarted after each operation, as with the code virtual machine.

The policy being enforced can be simple rules, such as an allow/deny rule (*e.g.*, a regular expression which matches all good HTTP requests) as in Figure 3.7, or more complex rules, such as denying access to all or part of a data structure based on the client attempting the access.

It is even possible to impose externally enforced semantic reasoning on the inputs and outputs of the service (see Figure 3.8). This could be used, for example, to verify that a request being made of the state store correctly correlates to the web page being requested by a client. If the request data is inconsistent with what is expected, it could indicate that the service has been exploited and the request denied, returning an error (*e.g.*, error 404) to the client. This provides a powerful mechanism which allows for intricate

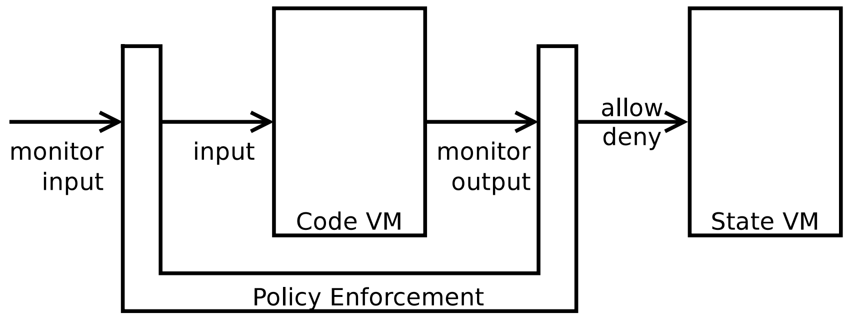


Figure 3.8: Checking for consistency between the input and output

policy to be hoisted outside the execution of the service itself.

Policy which exists in the original program can now be extracted and placed into policy enforcement modules. This could include anything from the maximum size of a request packet to controlling which users are authorised to access which data. This reduces the total size of the code and makes it easier to reason about. Furthermore, each bit of policy also becomes easier to reason about and the policy being enforced as a whole becomes disentangled and explicit.

3.4 Restarts

Restarting a service provides freshness. When restarting the service, the original version, that has not been tampered with, is loaded. Thus, if the service had been exploited, it will no longer be. The copy being loaded can be measured and attested to ensure it is the original copy. With the separation described in Section 3.3, it is now possible to restart a service without losing any long-lived state. This greatly simplifies the process, as there is no need to explicitly save and reload it.

The more frequently a service is restarted, the smaller the temporal surface, but the more significant the overhead of restarting it becomes. In order to minimise the impact an exploit can have, the smallest possible restart frequency is desired. This is typically each request, operation, or task, depending on the service. This reduces the time an attacker has to exploit a system. More importantly, the lifetime of an exploit is that of a given execution run.

The more often a service is restarted, however, the more significant the cost of restarting it becomes. Thus, the smaller the restart overhead, the better. In our architecture, we discuss three mechanisms to restart a process. The first approach is to only restart the service itself. This can easily be accomplished with a script which runs in a loop. Essentially, the processing loop of the service is moved outside the program. The second approach is to perform a full reboot of the entire code virtual machine. This provides freshness not only for the service itself, but also for the entire operating environment. The final approach is to use virtual machine snapshotting to take a snapshot of the initialised virtual machine and service and to then roll back to that point each time. This carries with it the same advantages of performing a full reboot, but does so much more efficiently.

Restarting a process to maintain freshness is already a well researched area for high availability and fault-tolerance for applications [6–10] and device drivers [12, 13, 15, 22, 23]. However, from a security perspective, this approach is limited, which will be discussed in more detail in Section 3.4.1.

These limitations are addressed by fully restarting the virtual machine containing the code portion of the program. This entails destroying the current machine, allocating resources for a new machine, then loading the service's code image into that machine and running it. This can be considered as rebooting the virtual machine. However, this carries with it a fairly high overhead. The total cost of the amount of time to bring up the operating system and the service is incurred with each restart.

The third approach, designed to reduce this overhead, is to take a snapshot of the code virtual machine just after it has initialised the operating system and the service to the point where it is ready to start servicing requests. This eliminates the cost of destroying, creating, and booting a fresh virtual machine and initialising the service on every restart. Freshness is maintained, as it is possible to perform measurement and attestation on the snapshot image. Each time the service is rolled back, it can be re-measured and re-attested to be the original, known good version. This is very difficult to do for long-running processes otherwise.[2]

The rest of this section will describe these three techniques in more detail.

3.4.1 Process Restart

The simplest approach for freshness of the service is to continually restart the process itself. The service needs to be modified as shown in the example in Figure 3.2 so that the event loop is removed and the program made to be "run-once". The event loop is essentially moved outside the service itself. This can be accomplished using a simple script which continually launches the service after each time it exits. This approach provides speed, but less security than performing a full reboot.

It might be possible to assure the freshness of the service, but now the operating environment in which it exists is long-running. If the measurement and attestation software is installed in the code virtual machine, it is possible that over time it could produce incorrect results. Even if moved outside of the code virtual machine, it is possible for the operating system to become corrupted, which could prevent the service from operating normally.

If we expand the threat model to include exploits which also break out of the confines of the service itself, then this approach becomes no more secure than allowing the service to run continually. An exploit could continually reinfect the service or simply install itself alongside it, for example to send spam e-mails. This also presents a large temporal surface for an attacker to attempt an exploit to gain access to the virtual machine monitor, which carries with it a far higher cost if exploited.

[2]As far as we know, this is currently an unsolved problem and, under normal conditions, it may be impossible to solve.

3.4.2 Reboots

In order to address the short-comings of only restarting the desired service, the entire virtual machine in which the service operates can be restart instead. This guarantees that the entire operating environment is constantly executing from a known good state, not just the service. The same "run-once" version of the service can be used as with the process restarting technique.

A major advantage of this approach is that the operating system is no longer a long-lived entity. This pushes the elimination of long-lived execution one layer lower. Instead of just measuring the service, we can now measure the entire virtual machine, which provides much strong assurances. This relies on the virtual machine monitor being long-lived, but without hardware support this is necessarily the case. Furthermore, virtual machine monitors have a far smaller code base than operating systems, which make them easier to reason about and less prone to error. They also have no device drivers, which account for a large percentage or operating system faults.

The major drawback of restarting the entire virtual machine is the high overhead associated with it. Each time the service is restarted, the entire operating system must be loaded and booted. Depending on the system being loaded, this could take several seconds. If each operation can be completed in a fraction of a second, the overhead imposed is several orders of magnitude.

In the extended threat model, as discussed in Section 3.4.1, this approach still provides protection. An attacker must now be able to break out of both the service and the virtual machine, providing Defence in Depth. This exploit must also occur in a very narrow time frame in order to exploit the system. In addition, a spamming exploit cannot be installed alongside the service, as the entire machine will be shutdown and restarted, removing the malware.

3.4.3 Snapshots

We have a desire for both security and speed. Process restarting (Section 3.4.1) provides speed, but not security. While entire virtual machine restarting (Section 3.4.2) provides security, but not speed. To address this issue, we implement virtual machine snapshotting, a technique whereby the state of an entire virtual machine is captured at a given instant in time. Snapshotting involves saving everything from memory to disk to CPU state.

Given a snapshot, it is possible to resume execution from the saved point in time. This can be used, for example, to suspend a virtual machine and resume it later, much like the concept of *hibernating* a running system. However, for our purposes, we use the snapshot as a template from which to execute.

After the virtual machine has fully booted and the service launched and initialised, a snapshot can be taken. The service is then allowed to run forward, as in the previous cases. However, once this execution is complete, instead of either restarting the service or the entire virtual machine, the virtual machine is rolled back to the snapshot. Thus, each request proceeds from the snapshot point, which is in a known good state.

The snapshot image taken can be measured and attested. Each time the machine is rolled back, the snapshot can be remeasured to ensure that it has not been tampered with. Reverting a snapshot is a relatively quick operation, thus providing both security and speed.

The service must be slightly altered from the “run-once” version described previously. Where the main event processing while loop used to be in the original version, and was removed in the “run-once”” version, a snapshot/rollback ‘loop’ is added instead (see Figure 3.9). This loop has the same general effect as the while loop, but now the code being executed is the exact same each time. This may seem more similar to the original version of the program than to the “run-once” version. However, the rest of the program must look like the “run-once” version, with all the long-lived

```
int main(int argc, char *argv[]) {
  ...
<SNAPSHOT>
  if ((r = ap_read_request(conn)) != NULL) {

    if (r->status == HTTP_OK)
      ap_process_request(r);

    if (!conn->keepalive || conn->aborted)
      break;

    ap_destroy_pool(r->pool);
<ROLLBACK>
  }
 ...
}
```

Figure 3.9: Modified Apache 1.3 code for snapshot and rollback

state stored and accessed from the state store. Furthermore, the program structure will need to be altered slightly from the original to make it more amenable to being snapshotted and rolled back.

To perform virtual machine rollback, there are three major approaches. The first is to simply copy the snapshot image over top of the active memory, irrespective of whether the memory has been modified or not. The second approach is to track the memory being modified and replace only that memory. The third involves preserving the original memory and remapping any that is modified to use newly allocated memory. The new memory can be discarded and the original mappings restored.

For the first approach, when it is time to perform the rollback, the entire contents of the snapshotted memory image are copied over. This is a relatively simple approach, but inefficient. Any memory that was not modified during execution will be copied over unnecessarily. Depending on the size of the memory image and the amount of memory being modified, this could increase the amount to copy by many orders of magnitude.

In the second approach, the memory which is written to is tracked. Then, when rolling the virtual machine back, only the memory which has been dirtied is replaced with the original memory stored in the snapshot. This adds some overhead to the execution of the virtual machine, as it must trap into the virtual machine monitor on each new memory page write. However, if there is not a lot of memory being modified, this extra overhead will not be significant.

The final approach requires support for *copy-on-write* of virtual machine memory. This is similar to the second approach, but instead of tracking the memory that is dirtied and reverting it, when memory is about to be modified, it is replaced with new memory. The new memory is mapped into the virtual machine which replaces the old mapping. Execution then continues to run as normal. When rolling back the image, the new memory can simply be discarded and the memory mappings restored to their original state. A big advantage of this approach is that there is no need to maintain a separate copy of the snapshotted memory image. If the virtual machine running the service has a lot of memory, then keeping a separate copy could be prohibitively expensive. It may even have to be copied to disk if there is not enough available system memory, which would incur a huge overhead.

3.5 Verifiability

Verification of a program is desirable, but can be a difficult and time consuming task. From a security perspective, there are two important types of verification. First, verifying the code for correctness to determine that it will execute the way it was designed to. Secondly, verifying that the code being run has not been altered, so that it will continue to execute correctly.

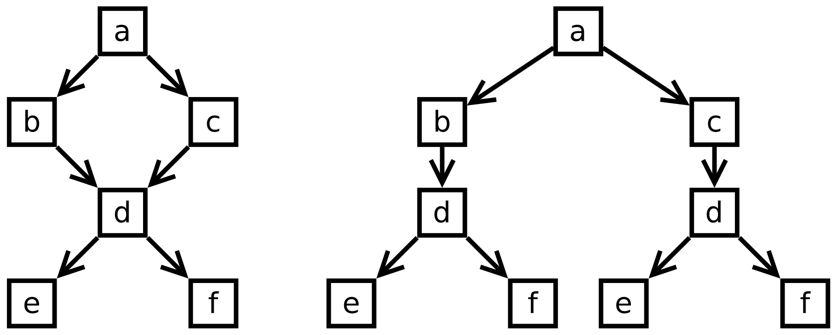
Formal verification of a program checks the actual code of a program against a model of it to determine if the code matches the model. This can prove that, given an input, the program will produce the correct output. This check is done statically, during the development phase.

If a program has been exploited, however, then it can be altered and this correctness no longer applies. Measurement and attestation involve dynamically analysing a program binary (*e.g.*, performing a hash of it) and asserting that the code being executed has not been altered in any way (*e.g.*, by comparing the current hash of the program with the known-good value).

This section discusses how these two properties apply to our architecture.

3.5.1 Formal Verification

With large and/or long running-processes, there are many states the program can be in. Furthermore, there can be side-effects between these states which complicates the analysis. These side-effects must be considered for each state transition. This is known as the *state-space explosion problem* and with current techniques can often render a program unverifiable.



(a) Control flow model (b) Verification flow model

Figure 3.10: Verification treats common execution paths separately

Figure 3.10 shows the difference between how the program's control flow model is structured (Figure 3.10(a)) and how the verifier must treat the control flow of the program (Figure 3.10(b)), as common control flow paths must be expanded out individually due to side-effects. Notice that in Figure 3.10(a) there are only six states, but in Figure 3.10(b) there are nine. This is a trivial example and as the depth (and width) of the control flow graph increases, the states in the verification model increase exponentially.

Code Virtual Machine

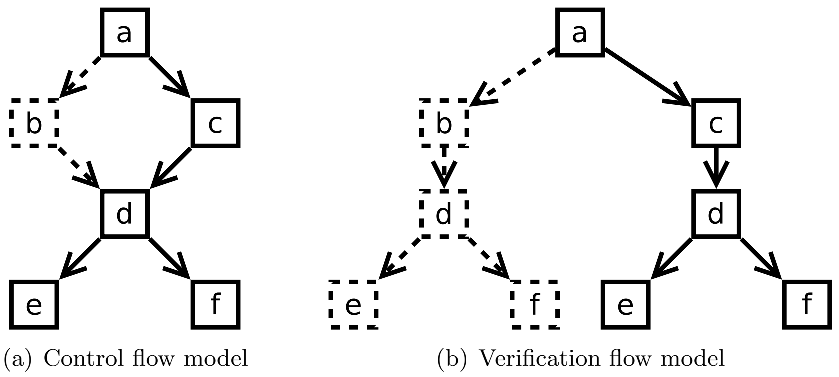


Figure 3.11: Verifying operations independently prunes the flow graph

Constantly restarting a program greatly simplifies its control flow model. Since each request can be independently verified from an initial state, the control flow graph can be pruned to only include the relevant states, massively reducing the state-space. This is illustrated in Figure 3.11. The dashed lines represent states that have been pruned from the graph. Notice that in this example, removing a single state from the control flow graph (Figure 3.11(a)) reduces the number of states in the verification graph by almost half (Figure 3.11(b)).

Verification is simplified further by using snapshots. The program can be verified to the point of the snapshot independently of the rest of the code. Once this has been completed, the snapshot image is known to be correct. Each operation can then be verified from the point of the snapshot to the point of the rollback.

The interface between the code virtual machine and the state store must also be verified. However, this is a relatively simple interface with a simple wire format (described in Section 3.3.3). Each half of the interface can be verified independently. That is, verify that a message will be sent correctly and separately verify that a message will be received correctly. Furthermore, each message type can be checked independently of the others.

State Virtual Machine

The state store is more difficult to verify as it remains long-lived (and essentially has an infinite loop). However, it is designed to have very little, simple code. This may make the code very easy to reason about, making it less prone to bugs. Moreover, the state store is a generic, reusable component. Thus, there is more impetus to ensure that its code is as bug free as possible.

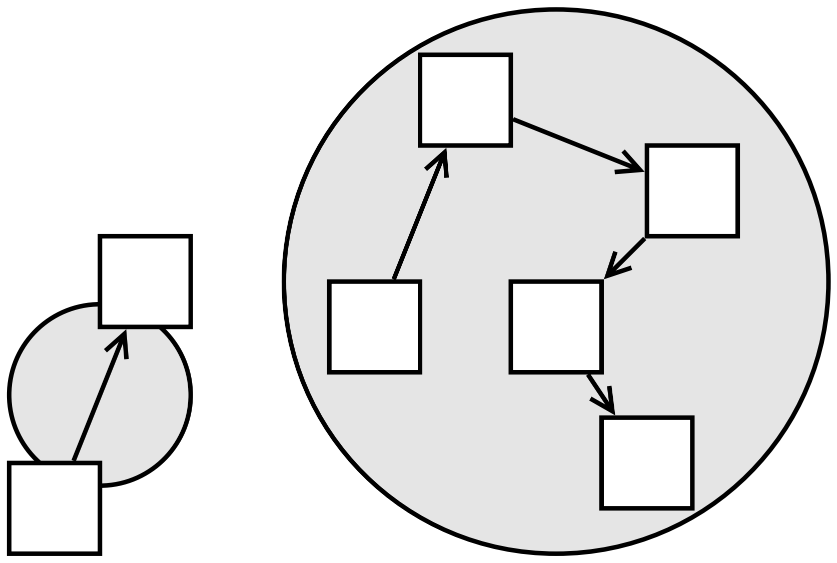
Policy

Imposing policy externally helps to simplify and reduce the service program's code (further easing its verification) as well as making the policy itself explicit. Since the policy enforcement modules are independent, they can be verified separately. Each module is also relatively simple and, in many cases, stateless. Thus, each bit of policy can be placed in its own virtual machine, which can be also be restarted (snapshotted and rolled back). Therefore, only one run through each policy enforcement module's code is required.

3.5.2 Measurement and Attestation

From an information assurance perspective, it is desirable to be able to measure and attest running programs for correctness. When loading a program, this is a relatively easy task. The binary, or even the entire virtual machine, being loaded can be measured. This measurement is then compared against the known-good version. If it matches, then one can be sure that the program has not been tampered with. However, the longer a program runs, the more state it acquires and the more difficult it becomes to measure it.

In order to obtain accurate results, the program needs to have its execution suspended so that the state does not alter during measurement. If it does, this could result in a *time-of-check-to-time-of-use* exploit. This problem is currently considered to be an unsolvable problem with long-running process.

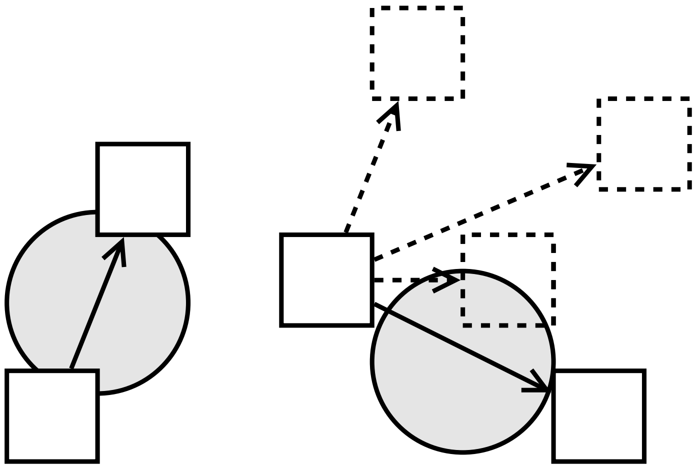

(a) After one operation

(b) After several operations

Figure 3.12: The state-space grows with execution

Figure 3.12 illustrates this problem. The boxes indicate different states (*e.g.*, after an operation), the arrows indicate the flow of execution, and the grey circles represents the state-space. Initially, the state-space is small (Figure 3.12(a)). However, over time, as the program executes, the state-space grows (Figure 3.12(b)). The state-space will continue to grow as the program executes, making it more and more difficult to accurately perform measurements.

Using execution rollback, the snapshot image can be remeasured periodically. Figure 3.13 shows the state-spaces for the code virtual machine of a snapshotted process. The dashed lines and boxes indicate execution paths and states the process went through in the past. However, since after each execution all the state is reverted, each path originates from the initial posi-


(a) After one operation

(b) After several operations

Figure 3.13: The "run-once" state-space does not grow with execution

tion. This prevents the state-space from growing, enabling remeasurement. Depending on the overhead or the desired level of assurance, it could even be remeasured on each rollback. This ensures that the snapshot has not been tampered with and that each time the service executes, it is running correct, unmodified code.

3.6 Limitations

The presented architecture, while providing many benefits, also has some limitations and drawbacks. Our architecture is unsuitable for certain types of services. To ensure security, the policy modules are heavily relied upon to be correct. Currently, there are some classes of attack which will succeed despite the increased security. Addressing these limitations is a goal of our future work (see Chapter 5).

3.6.1 Caching

For a program which uses a cache for performance, rolling back will either destroy the cache or storing the cache in the state store virtual machine will lose the performance benefits. This could be solved by using a separated, high-performance cache virtual machine or by marking the memory containing the cache so that it is not rolled back. However, this introduces a potential vector of attack. It may be possible to corrupt the cache, allowing an exploit to persist.

Another approach is to aggregate several iterations per restart loop. The cache could then be kept locally, as the iterations in the restart loop would benefit from the cache without being subjected to any overhead caused by externalising it.

3.6.2 Concurrent Access

The state store only supports sequential accesses. This could pose a problem for a multi-threaded service. If multiple threads attempt to access the store at the same time, it could be difficult to sort out which responses go to which thread. A solution to this is discussed in Section 5.3 by using a *fork* mechanism, but for whole virtual machines instead of just processes.

3.6.3 Policy

An incorrect or buggy policy module can make it possible to exploit the service. Our architecture does not automatically guarantee security, instead it makes the information flow and policy explicit. This enables them to be reasoned about and more easily verified, which reduces, but does not eliminate, the chance of errors.

3.6.4 Attacks

Constantly reverting a service to a known good state reduces the time that an exploit has to succeed. However, if an attack is able to break out of a service before it restarts, then it could replace the service with a copy that has been tampered with and will not restart as it should. Furthermore, it is possible for the attacker to now exploit the operating system to try to break into the virtual machine monitor. Currently, we do not protect against such attacks. However, this is a direction for future work, discussed in Section 5.4. Essentially, the virtual machine can be forced to restart if it has not done so within a bounded limit, returning it to its known good state.

Chapter 4

Prototype

We have designed and built a prototype implementation of our architecture. We targeted XenStore, an inter-virtual machine communication tool for the Xen virtual machine monitor [2]. We used an OCaml implementation of XenStore which we developed earlier to address some of the shortcomings of the original C version. The state store virtual machine has also been developed in OCaml. No external policy was implemented in this prototype, but it is a top-priority for our on-going work on this project.

This chapter will first give an overview of Xen and XenStore, then describe the prototype implementation in detail. Finally, we evaluate the prototype.

4.1 Xen Overview

Xen is an open-source virtual machine monitor. A virtual machine monitor is a relatively small piece of software which runs underneath operating systems and controls access to and allocation of resources. It can be viewed as an operating system for operating systems. A virtual machine abstraction is exported which can run a guest operating system. Several virtual machines (and thus guest operating systems) can be run concurrently.

In Xen, the first virtual machine to boot, called Domain-0, is a **privileged** virtual machine. This means that it can perform privileged operations, such as creating, destroying, and mapping the memory of other virtual machines or rebooting the physical machine. It also contains device drivers which are exported to the other virtual machines as virtual devices.

4.2 XenStore Overview

XenStore is an inter-virtual machine communication and information storage system. It is primarily used for communicating configuration and status information between virtual machines. It is a key-value store, which makes it an ideal candidate for porting to our architecture.

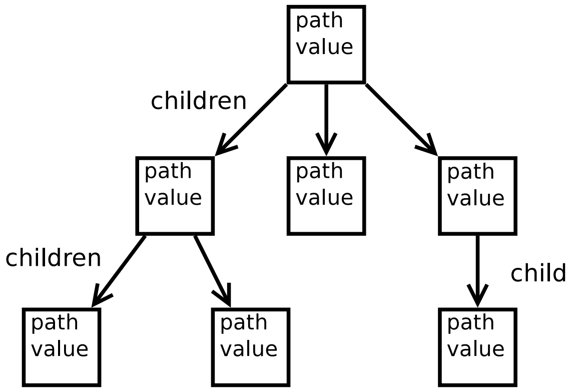


Figure 4.1: Example XenStore tree

XenStore provides a hierarchical, tree-based storage system (see Figure 4.1). The basic component of this system is the node. Each node has a path and may have at most one value assigned to it, and may have none or many children. According to the XenStore specification, a node should have **either** a value **or** children, but not both. However, `xend` (pronounced xen-d), a suite of tools for managing virtual machines in Xen, violates the specification. This is due to an implementation error in the original C version of XenStore. There is no restriction on the value stored at a node except for its length. However, the storage of binary data in XenStore is discouraged, and is generally not done in any current usages.

4.2.1 C XenStore

The original version of XenStore is written in C. Unfortunately, it has many drawbacks and security flaws. The store is implemented as a file using

Trivial Database. On each transaction, this file is copied in its entirety. When a transaction commits, its file replaces the main store file. This introduces overhead, as the program must constantly go to disk. Moreover, this provides a vector of attack. The file can be directly altered, subverting all of XenStore's security mechanisms.

Domain-0 tools communicate with XenStore through a socket interface while other virtual machines must use XenBus, a simple communication bus. This complicates the code base and an attacker capable of connecting to XenStore through the socket interface will automatically be granted privileged access.

CPU Consumption Attack

An attack on XenStore has been demonstrated to be capable of completely exhausting Domain-0's CPU availability. Since XenStore runs as a process in Domain-0, if it is excessively accessed by a guest virtual machine, it will consume all the CPU. This has an impact on the performance of the entire system since all services run in Domain-0.

While it may be possible to curtail this attack by throttling guest virtual machine access to XenStore or by reducing the priority level of XenStore, each of these solutions has their own drawbacks. When throttling guests, the responsiveness of XenStore for each virtual machine will be affected. If there are only a few virtual machines running on the system, then maximal performance will not be achieved. This can be improved by allocating virtual machines a proportional share of XenStore rather than fixed quanta. Reducing the priority level of XenStore within Domain-0 will cause an overall decrease in XenStore responsiveness.

Denial-of-Service Attack

It is possible for a guest virtual machine to attack XenStore such that XenStore itself becomes unavailable. The attack repeatedly starts a transaction, writes a small amount of data, then commits the transaction. Due to the nature in which concurrent transactions are handled in XenStore, all other transactions are prevented from committing. Concurrent transactions, whether they interfere with each other or not, cannot all commit. Instead, the first transaction to commit will succeed and all other on-going transactions will fail when they try to commit. Since the attack uses a very short transaction, it will constantly be the only one allowed to commit.

Critical management tasks such as starting, terminating, and migrating virtual machines are thus rendered impossible to perform. These tasks require longer, more complex transactions which are prevented from ever successfully committing. Improving the way in which transactions are handled would prevent this attack from denying access to XenStore.

4.2.2 OCaml XenStore

To address some of the drawbacks of the original version of XenStore, we reimplemented it in OCaml. OCaml provides efficient support for tree structures. The file-based backing store was completely removed. This is a performance gain as well as removing it as an attack vector.

Denial-of-Service Attack

The main focus of our initial OCaml XenStore implementation was to address the denial-of-service attack. To solve this problem, we implemented an ACID (atomic, consistent, isolated, durable) transactional system. This new transaction system allows for the committing of concurrent but non-interfering transactions. Thus, when subjected to the denial-of-service attack described in Section 4.2.1, the transactions of other virtual machines, including Domain-0, are unaffected. However, the persistent access to XenStore will still result in a successful CPU resource consumption attack, as described in Section 4.2.1.

4.2.3 Disaggregated XenStore

The OCaml XenStore described above (Section 4.2.2) is an improvement on the original C version (Section 4.2.1). However, it is still vulnerable to some of the same attacks. In order to increase the security of XenStore, and the system as a whole, we *disaggregated* XenStore into its own virtual machine. By doing so, some simplifications can be made. For example, the tools in Domain-0 were modified to use the XenBus interface instead of sockets. This allowed for the complete removal of socket code from XenStore. Reducing the interface of XenStore to only XenBus allows for easy modelling of the external interface.

Since the file-backed store was removed in the OCaml version of XenStore and now the socket interface has been removed, XenStore requires neither a disk nor a network device. Thus, the virtual machine in which the disaggregated XenStore runs does not require these components either. This reduces the complexity of the operating system and removes potential vectors of attack.

CPU Consumption Attack

There exists an attack on XenStore which can consume Domain-0's CPU (discussed in Section 4.2.1). By running XenStore in its own virtual machine, its CPU usage is independent of Domain-0's CPU usage. Thus, the attack will no longer affect Domain-0.

It may still be possible to consume all of the disaggregated XenStore's virtual machine's CPU. However, the physical CPU usage can be controlled and throttled by Xen. To further prevent the attack from consuming XenStore's CPU, XenStore can be modified to stall communication with a virtual machine which consumes more than a certain threshold of CPU cycles. It is possible to extend this modification even further to monitor each guest virtual machine's usage of XenStore and deduct accordingly from the guest virtual machine's schedulable CPU time. Thus, if a guest virtual machine bombards XenStore with requests, its own CPU time will be used up and Xen will disallow it from executing further, preventing any significant disruption to XenStore.

4.2.4 Split-Virtual Machine XenStore

Our disaggregated XenStore offers many security advantages over the original C version. It is now able to withstand both the CPU resource consumption attack (Section 4.2.1) and the denial-of-service attack (Section 4.2.1). However, it is still a long-running process and therefore susceptible to long-lived exploits. Furthermore, as a long-running process, it is difficult to verify its code and measure its running state and attest it to be correct.

4.3 Code Virtual Machine

Our prototype is a proof-of-concept implementation. Therefore, performance was not our main concern. The separation of state from code was divided into three major milestones: separate out state, perform full reboots, and implement snapshot and rollback.

4.3.1 State

The first milestone was concerned only with separating out state. This enabled us to test the interface between the two virtual machines and ensure the serialisation/deserialisation of data was working correctly. To simplify this process, all state was separated out. Any data retrieved was only kept long enough to perform whatever operation required it. This is a very inefficient implementation, but would ensure that when attempting to perform full reboots, errors would be due to mechanistic fault and not due to a misconception of the length of life of some state.

4.3.2 Reboots

Rebooting the code virtual machine required a method of determining when execution had completed. Normally, `xend` and XenStore are used to detect a terminated virtual machine. However, since XenStore is the service being restarted (and XenStore is required to be operational *before* `xend` can run), this approach was not possible. Instead, extra support needed to be added to the XenStore program itself.

Xen provides *event channels*, which are much like hardware interrupts, to enable inter-virtual machine event notification. An event channel was created between XenStore and a tool in Domain-0. XenStore notifies the tool once it has completed its execution (one request/response). The tool then calls into Xen to destroy the code virtual machine and rebuild it.

Doing full reboots adds a tremendous amount of overhead. This was to be expected and motivated the snapshot and rollback mechanism. However, it proved that the concept was possible.

4.3.3 Snapshots

The final milestone was to reduce the major cause of overhead with the architecture: full reboots. Xen has support for saving a running virtual machine to disk and restoring it at a later point in time. However, this mechanism requires the co-operation of the guest operating system. XenStore has been implemented on top of Mini-OS, a minimal operating system designed for use with Xen. Mini-OS has no support for save/restore. Furthermore, constantly saving to and restoring from disk would be expensive.

The Domain-0 tool previously created to reboot the XenStore virtual machine was modified to instead perform snapshot and rollback. The tool contains a memory buffer which is used to store the snapshot image of the XenStore virtual machine. When Xenstore has been initialised, it notifies the tool over the event channel. The tool then takes a snapshot of the entire

```
let process_add store payload =
  try (
    let (key, value) = Utils.split_first payload in
    Hashtbl.add store key value;
    Kvmessage.ok
  )
  with _ -> Kvmessage.err
```

Figure 4.2: Add operation for state store

virtual machine, including CPU state. Once XenStore has completed its execution, it once again notifies the tool, which, instead of rebooting the virtual machine, performs a rollback.

In order to ensure correctness, the entire memory contents from the snapshot image is copied back over top of the virtual machine's current memory and the CPU state is restored. The virtual machine then runs forward again, from the point in time that the snapshot was taken. This snapshot and rollback is equivalent to performing full reboots, but with much less overhead.

4.4 State Virtual Machine

The state store virtual machine is small and simple. It has been developed in OCaml so that it can be more readily verified. To store the data, the built-in hash table module is used. Leveraging existing modules enables the code to remain simple and to benefit from well tested and efficient data structures.

Figure 4.2 gives the code for the `add` operation. As can be seen, there are only seven lines of code. The `Utils.split_first` function splits a string into two parts on the first occurrence of the null character ('\0'). The function returns an `ok` message if everything is successful. If an error occurs, an exception is thrown. This is caught and an `err` message is returned.

4.5 Communication Channel

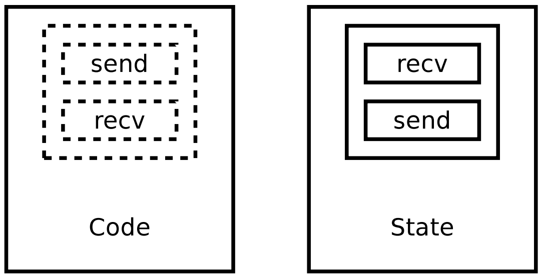


Figure 4.3: Communication channel as mapped buffers

The design of the communication channel is similar to that of XenBus. There are two buffers, send and receive. The buffers are actually contained in the state store virtual machine. The code virtual machine maps the page of memory containing them into its address space. The state store and code treat the buffers as opposites. The send buffer in the code virtual machine is the receive buffer in the state virtual machine and vice versa. Figure 4.3 illustrates this configuration. The dashed lines represent memory that has been mapped. Notice that the send and recv buffers are inverted.

If a packet is larger than the length of the buffer, which is 1,024 bytes, then it is broken up into pieces and reassembled at the other end. If, while sending, the packet would go past the end of the buffer, then as much as will fit is sent and the rest is wrapped around to the beginning of the buffer (*i.e.*, this is a circular buffer).

There are seven messages types, as outlined in Table 4.1. The top four message types (`add`, `get`, `put`, `del`) are used by the code virtual machine to make requests of the state store. The bottom three message types (`ok`, `ret`, `err`) are response messages. It is important to note that the space seen between `key` and `value` is a null character ('\0') in the actual payload. The payload for an `ok` message is empty (and thus the `size` field is 0).

If the request completes successfully, then an `ok` message is returned, except for `get` which will return a `ret` message containing the associated value for

Type	Bits	Payload	Description
`add`	000	key value	Add a new key-value pair to the store. If the key already exists, return an error.
`get`	001	key	Retrieve the stored value associated with the given key.
`put`	010	key value	Replace the value associated with the given key. If no association exists, create it.
`del`	011	key	Delete a key-value pair. An error is returned if the association does not exist.
`ok`	100		Return than an operation completed successfully.
`ret`	101	value	Return a value, which is contained in the payload (from a `get` request)
`err`	110	errno	Return that an error occurred.

Table 4.1: Message types

the given key as the payload. If an error occurred, then an `err` message is returned with the appropriate `errno` set in the payload field (see Table 4.2).

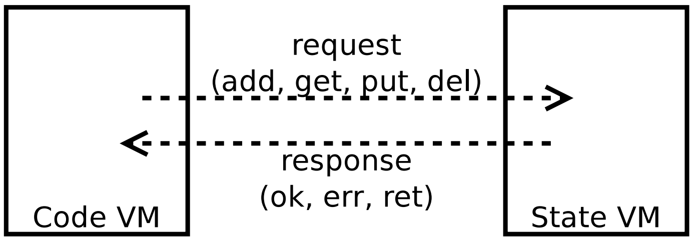


Figure 4.4: Request/response flow between code and state VMs

Request messages always flow from the code virtual machine to the state virtual machines and response messages from the state virtual machine to the code virtual machine (see Figure 4.4). It is a violation of protocol for either type of message to flow in the opposite direction or for any other message type to be sent. Each request receives a response and messages are delivered in order. Typically a full request/response interchange is completed before the next request is sent, however this is not a requirement.

Request	Error	`errno`	Description
`add`	`EEXIST`	17	The key is already associated with a value.
	`ENOMEM`	12	The store is out of memory.
	`EINVAL`	22	The input was invalid.
`get`	`ENOENT`	2	There is no mapping for the specified key.
	`EINVAL`	22	The input was invalid.
`put`	`ENOMEM`	12	The store is out of memory.
	`EINVAL`	22	The input was invalid.
`del`	`ENOENT`	2	There is no mapping for the specified key.

Table 4.2: Error response types for each request type

4.6 Evaluation

Version	Time
Reboot (Mini-OS)	40ms
Rollback	4ms

Table 4.3: Times for mechanisms

The prototype implementation has two variants: full reboots and snapshot/rollback. Table 4.3 gives the time required by each raw mechanism. The results are the average of ten trials. The rollback technique used in these trials is the naïve implementation which copies the entire memory contents of the snapshot. Thus, the **least** efficient rollback mechanism is an order of magnitude faster than performing a full reboot. Note that the operating system being rebooted is Mini-OS, which is very small and simple. Performing a reboot of even a stripped down Linux would be of the order of seconds. Also note that the amount of memory for the virtual machine is 4MB.

We have done some preliminary evaluation of our prototype. We first evaluate the length of time it takes for `xend` to start. For stress-testing, `xenstore-ls` is used. `xenstore-ls` is a program which lists all the nodes in XenStore, displaying them in hierarchical order. To accomplish this, it first reads the root node, then it requests its children (see Figure 4.1). For each child, this process repeats recursively using depth-first traversal of the tree. Each operation is done independently (no transactions). Finally, we evaluate the creation and booting of a virtual machine to test a "real-world" use.

Version	xend start	xenstore-ls	ls/entry	xm create
Original	16.5s	0.04s	1.0ms	1.28s
Reboots	39.4s	10.80s	93.9ms	20.17s
Snapshots	22.7s	1.37s	11.5ms	4.41s
Reboots (1s)	*∼600.0s*	*∼230.0s*	*∼2050ms*	*∼460.0s*

Table 4.4: Times for tasks

The results are shown in Table 4.4. Each result is the average of three trials. Three versions of XenStore are evaluated: the original C version, the split-virtual machine implementation with full reboots, and split-virtual machines using snapshotting. A fourth, hypothetical version is also provided illustrating what the costs would be using full reboots if a reboot took one second.

The "ls/entry" column is an average of the time taken for each operation during `xenstore-ls`. Since XenStore stores information about virtual machines existing on the system, when split virtual machines are being used there is more information to be stored (since there are more virtual machines running on the system). Thus, this column corrects for time differences as a result of the difference in the amount of information stored (and retrieved).

The overhead of using snapshots is an order of magnitude greater than the original version. The overhead of using reboots is an order of magnitude greater than that (and two orders of magnitude greater than the original). Using a reboot time of just one second produces an overhead of another order of magnitude. However, even with a one second reboot time, rolling back would still take the same amount of time.

Starting a virtual machine incurs a three and a half times slowdown using snapshots, and a twenty times slowdown using reboots (and almost a five hundred times slowdown with a one second reboot time). This overhead is less than for just performing one `xenstore-ls` operation because there are other factors in the creation of a virtual machine that just accessing XenStore.

These initial results are promising, especially considering that the current prototype implementation is far from optimised. We believe it should be possible to approach near-native speed. This is discussed in greater detail in Section 5.2.

Chapter 5

Future Work

Our prototype implementation shows that the proposed architecture is viable. However, there are currently some limitations to this system and many exciting directions this research could take in the future.

5.1 Policy

An important part of this architecture is the ability to impose policy on the communication. However, we currently do not have a concrete mechanism or an explicit example of policy enforcement. This is an important direction for our future work. The external nature of the enforcement enables intricate policy decisions to be made and the enforcement of the policies themselves to be secure from direct exploitation (see Section 3.3.4).

It should be possible to create a policy virtual machine with two communication channels, as described in Section 4.5. The code virtual machine would connect to one channel of the policy module instead of to the state store's channel. The state store would connect to the other channel. The policy module could then intercept and analyse the communications between the code and state virtual machines.

This section aims to present an overview of example policies that the capability check system in XenStore might enforce. The following sections will provide a detailed description of each policy proposed below.

The first proposed policy provides the same security checks as the existing, hard-coded checks in XenStore. This policy is known as the legacy policy. This will enable safe removal of the hard-coded security checks from XenStore in favour of the policy module, completing the separation of XenStore.

The second policy proposed is the exclusionary policy. This policy disallows all communication between virtual machines using XenStore. Communication between virtual machines is allowed only if it is explicitly permitted. This policy is similar to the Simple Type Enforcement policy in the sHype [19] security subsystem in Xen.

Lastly, and most interestingly, a protocol enforcement policy for XenStore is proposed. This policy enforces the correct usage of XenStore by all virtual machines. This is similar to a stateful firewall ruleset in that it allows certain operations only if they follow a well defined protocol. This policy could, for example, enforce correct usage of the split-level block driver and the split-level network driver.

We propose a human-readable interface for creating and modifying policies in XenStore. This would be similar to policy generation in SELinux, but with a more user-friendly format. Because we would like the policy enforcement to be stateful, we may wish to look at policy module configuration using firewall-like semantics. This could be used to enforce rate-limiting and well known driver protocols and to dynamically group virtual machines.

It is worth noting that the intention of this design is that policies be composable. In most deployments a coarse-grained universal policy will be applied to the entire XenStore namespace, along the lines of an exclusionary policy that severely restricts the ability of virtual machines to communicate with one another. Then, where they do communicate through specific sub-trees in the store, more fine-grained policy modules may be used to enforce particular communication semantics, for instance the device protocol for a specific split-driver implementation.

5.1.1 Legacy Policy

As a first step we propose the implementation of the legacy policy. This policy will implement all current security checks in XenStore, allowing us to safely remove the existing permission enforcement code from the XenStore code base in favour of policy modules. The existing permission enforcement code must be removed in order to fulfil our goal of policy/code/state separation.

5.1.2 Exclusionary Policy

The Exclusionary Policy works by forbidding all communication between virtual machines using XenStore, unless explicitly specified. This policy is similar in nature to the simple type enforcement (STE) policy in the sHype security subsystem of Xen. In this policy, virtual machines are labelled by groups and communication is only permitted within groups. A virtual machine may be a member of multiple groups, in which case it is able to communicate with the members of each of those groups. In terms of XenStore behaviour, a virtual machine will not be able to read or manipulate any node owned by a virtual machine outside of its groups, regardless of the permissions of that node. Virtual machines are assigned groups when they are booted, and they cannot join or leave groups over their lifetimes.

This policy could easily be extended to provide a Chinese Wall policy [4], whereby conflicting groups are not allowed to use XenStore concurrently. In order to support this policy, certain data structures in XenStore need to be modified. Additional security information needs to be stored in nodes and connections.

5.1.3 Protocol Enforcement Policy

All drivers that use XenStore use it in a specific, well-defined way. With the protocol enforcement policy, we can leverage this fact to ensure that only well-behaving virtual machines may use XenStore. This policy tracks the state of all connections to XenStore and ensures that they are behaving in an appropriate manner. It can be thought of as being similar to a stateful packet filter.

This policy is aware of the protocols used by all device drivers when accessing XenStore. Device drivers that communicate using XenStore follow a state-based protocol. This policy is given a formal encoding of all protocols used in XenStore, and assigns states to virtual machines accessing the database. Much like a finite state machine, the protocol enforcement policy decides whether an operation should succeed or fail based on the virtual machines's current state and the operation requested (the virtual machine's input). For example, in a virtual machine that has a frontend block driver, the driver would only be able to write its ring buffer and event channel details into the store when it is in the initialising state.

This policy could be used to ensure that only authorised domains are able to export backend drivers, and ensure that only authorised frontend drivers are able to connect to the respective backend drivers.

5.1.4 Filter Policy

The ideas presented in the protocol enforcement policy can be generalised to provide an extensible, stateful mechanism to configure XenStore security policies. This is similar to a stateful firewall in that permissions, labels, groups, and states can be specified in an abstract, human readable way. Rate limits can be enforced for misbehaving connections and malicious or unauthorised connections can be dropped or refused outright.

The syntax used by this policy is general enough such that all previous policies can be expressed as specific instances of the filter policy.

5.2 Performance

The initial prototype was not designed with performance in mind. Realising that doing a full boot sequence on each restart would introduce significant overhead, a snapshotting technique was developed. This increased performance by an order of magnitude. We believe that there are many other areas which can be targeted to further increase performance. It should be possible to achieve near-native speed, with the only major overhead being the communication channel between the code and state virtual machines.

5.2.1 Communication

The current prototype accesses most state, long-lived or not, from the state store. Furthermore, in many cases it refetches the same data multiple times during one execution. This was done to help debug the separation. However, after some analysis it became clear that short-lived state (*e.g.*, local variables) did not need to be stored at all. It is also possible in many cases to retrieve long-lived state only once per execution and, further, to fetch this state lazily so that only the data that is actually required is requested and retrieved.

The communication mechanism is based on XenBus, which is used primarily by XenStore. XenBus was designed to communicate configuration data between virtual machines, not to transport large amounts of data. It was chosen as the basis for our communication mechanism because it is relatively simple with well-tested code. However, a mechanism designed for large data transfers will likely decrease the overhead associated with communication.

5.2.2 Snapshots

Further analysis of the snapshot mechanism is required to determine if there is a performance advantage, and if so how much, to using copy-on-write on the code virtual machine's memory. With copy-on-write, only the pages which have been modified are reverted. For example, in a typical run of the prototype only three pages of memory are dirtied. This reduces the total amount of memory that needs to be copied from 4MB to 12KB (pages are 4KB).

5.3 Virtual Machine Fork

Our current architecture is not well suited for services with concurrency. For example, a multi-threaded service would not be easily adapted to it if multiple threads need to access the state store at the same time. Implementing a version of `fork`, but for virtual machines, could solve this problem. Each thread could then be launched in its own forked version of the code and state virtual machines. However, this would require a mechanism to merge the forked state stores back together.

5.4 Attack Prevention

An attacker capable of breaking out of the service into the operating system can currently prevent the service from restarting. Furthermore, the service could be replaced with an infected version which does not restart. However, there are several ways to prevent this type of attack. The total execution time for the code virtual machine can be limited. For example, if no request takes longer than 50ms, then if the service has been running for longer than that amount of time without restarting, it can be forced to restart from the outside.

A more complicated approach is to use static analysis of the program to create an abstract model of execution bounds. For example, the total number of branches or the number of instructions of the longest (correct) path through the code. These attributes can then be monitored and the virtual machine forced to restart if any of the conditions are violated.

Chapter 6

Conclusion

As attackers become more sophisticated and attacks more available, it is clear that we need a better way of protecting systems. Long-running processes are an alluring target as they provide both a large temporal surface to attack and a long time to own the service once it is successfully exploited. Restarting a service restores it to a fresh state. However, once an exploit has been discovered it is relatively easy for the attacker to regain control of the service.

We have presented a new architecture designed to help secure this class of service. By separating the code and state of a program into two virtual machines it is possible to constantly restart the code portion, bringing it back to a fresh, unexploited state. Moreover, it is easier to reason about the interaction between the two parts and policy can be externally enforced upon this interaction. Simplifying the code, and thus the program model, allows for easier verification of correctness. This architecture enables dynamically measuring and attesting that the software has not been tampered with.

Bibliography

[1] Mary Baker and Mark Sullivan. The recovery box: Using fast recovery to provide high availability in the unix environment. In *In Proceedings USENIX Summer Conference*, pages 31–43, 1992.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[3] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: splitting applications into reduced-privilege compartments. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[4] D.F.C. Brewer and M.J. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214, May 1989.

[5] David Brumley and Dawn Song. Privtrans: automatically partitioning programs for privilege separation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[6] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings of the 2002 International Conference on Dependable Systems and*

Networks, pages 605–614. IEEE Computer Society Washington, DC, USA, 2002.

[7] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr: an autonomous self-recovering application server. In *Autonomic Computing Workshop, 2003*, pages 168–177, June 2003.

[8] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, 2001.

[9] George Candea and Armando Fox. Crash-only software. 2003.

[10] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[11] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[12] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[13] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 41–50, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Douglas Kilpatrick. Privman: A library for partitioning applications. In

USENIX Annual Technical Conference, FREENIX Track, pages 273–284, 2003.

[15] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[16] Derek G. Murray and Steven Hand. Privilege separation made easy: trusting small libraries not big processes. In *EUROSEC '08: Proceedings of the 1st European Workshop on System Security*, pages 40–46, New York, NY, USA, 2008. ACM.

[17] Ramesh Chandra Nickolai, Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The collective: A cache-based system management architecture. In *In Proc. 2nd Symposium on Networked Systems Design & Implementation (NSDI)*, pages 259–272, 2005.

[18] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[19] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 276–285, Washington, DC, USA, 2005. IEEE Computer Society.

[20] Jesse Sathre, Alex Baumgarten, and Joseph Zambreno. Architectural support for automated software attack detection, recovery, and prevention. In *CSE '09: Proceedings of the 2009 International Conference on*

Computational Science and Engineering, pages 254–261, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Scott F. Smith and Mark Thober. Refactoring programs to secure information flows. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 75–84, New York, NY, USA, 2006. ACM.

[22] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, 2006.

[23] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222, New York, NY, USA, 2003. ACM.

[24] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.