Secure Operating Systems

Christopher A. Wood
Department of Computer Science
Rochester Institute of Technology
caw4567@rit.edu

***Abstract*—Operating system development is a very diverse task, usually drawing from various applications of Computer Science, Software Engineering, Computer Engineering, and Information Technology. As such, the complexity of most usable systems is significantly higher than traditional software projects. This fact presents a problem when implementing security features and policies for existing systems. Often times it is already too late in the development process to make any substantial headway, and any gain from newly added features will likely not be good enough to deter determined attackers. Therefore, security needs to be involved in the development process and system design from the very beginning in order to be effective.**

This work explores operating system security concepts that should be at the foundation of any usable system. Specifically, it covers program and operating system security concepts that are present in modern systems. This background information is necessary for an analysis of state-of-the-art designs that incorporate security from the ground up. It also includes a survey of popular commercial and research systems in order to cover the different tradeoffs that exist between implementation techniques and security gains.

While it is true that there is no such thing as perfect security for operating systems, depending on the context in which the system will be used it is possible to find the right balance between implementation efficiency, complexity, and security that satisfies its users. After all, the overall goal of operating system security is to provide an environment for applications to run such that all user data and private information is kept secret.

***Index Terms*—Program Security, Operating System Security, Malware, Software Development Process**

I. Introduction

This project is the next step in a two year effort focused on malware research. My previous work was centered on keylogger malware development and exploitation vectors [1]. Instead of focusing on how to prevent and detect malware from infecting current operating systems, the idea is take a step back and think about the design of the system in general. As such, the focus of this project is on operating system design concepts and implementations that are employed to keep both the system and its users safe from malware and unwanted program side effects.

The motivation for this research came from operating system vulnerabilities and exploits. Today's systems are plagued with program errors and flaws that are easy to exploit by attackers looking to gain a foothold in the system and compromise the user's private data. Some of these systems rely on implementation obfuscation to deter such attackers, but the complexity of these systems can be easily uncovered given enough time and effort on the attacker's behalf. As such, it is very important that the system is designed from the ground up with security in mind.

There has been substantial research in operating system design for security, and there are many different flavors of these designs available for use. An analysis of these different implementation shows that each operating system is unique in how it handles security, and the only way to learn about these systems is to analyze them one by one.

Therefore, the work of this project and paper is as follows. Firstly, program errors and flaws and software engineering practices used to prevent such errors are explored as the influence for all operating system security designs. Second, common operating system security concepts are discussed to give a foundation for the case studies analyzed. Thirdly, different operating system implementations are examined from a security perspective to ascertain how they handle the program errors and flaws discussed in the paper. Lastly, I have proposed some design concepts of my own that will serve as the starting point for a custom, security-oriented system.

II. Program Security

Security practitioners, software developers, and end users all have entirely different views on what security means given the context in which they interact with programs. As such, it is important to understand program security from these different perspectives. The following list identifies issues surrounding secure programs that appeal to all of these different users, and they all must be considered when discussing the general notion of program security.

1) Programming errors and software flaws with security implications
2) Malicious code
3) Software Engineering practices for security

A. Programming Errors and Software Flaws with Security Implications

Errors in programs that cause them to behave in unintended or unexpected ways are called program flaws. They are typically caused by program vulnerabilities, which in turn are the result of unintentional programmer errors or malicious functionality in the program. Software developers and security practitioners are the ones who are primarily concerned with this aspect of program security. Many security vulnerabilities can be traced back to single programmer errors in the source code. This means that such vulnerabilities, and any successful exploits that are implemented upon such weaknesses, can be avoided before the software is even ready for use.

Whether or not these programmer errors arise from a lack of proper software engineering practices or poor programming

practice on behalf of individual developers is a separate issue. It is important to note how such errors can arise and the impacts they can have on software products in any field, especially those which deal with highly sensitive data.

The most common programming errors are memory corruption, incomplete mediation, time-of-check to time-of-use, and information leakage [2]. Memory corruption, from a development perspective, is a very vague term and can be applied to a number of different errors. Perhaps the most notable memory corruption error is a memory leak. In traditional, non-managed languages like C the developer is responsible for managing their own memory resources that are allocated on the heap. If the programmer neglects freeing unused memory from the heap then it is possible for the amount of available memory to be exhausted. Although this is not directly a security problem, a lack of available memory can cause a fatal error and ultimately a program failure, thus compromising whatever data the program was working with and the system it was running on.

Incomplete mediation is when the programmer fails to check every possible branch of code that grants to access to sensitive objects or data within an application or system. There may be cases not covered by the suite of unit and integration tests that lead to such unauthorized access to sensitive objects. Given the increasing complexity of modern software projects and operating systems (as of March 14, 2011 the Linux kernel has over 14 million lines of code) and the schedule and financial limitations placed upon developers, it is often difficult if not impossible to cover all possible cases. Therefore, the likelihood of this error occurring will naturally increase as the size and complexity of software increases in the future.

The problem that typically arises from incomplete mediation is unauthorized access to sensitive objects. Incomplete mediation is especially harmful when considering system software as it can lead to underprivileged users gaining escalated rights as super user, accessing hardware devices and kernel data that contain sensitive information to other users and processes, and even system corruption and failure. Typically, however, malicious users will exploit unchecked mediation paths simply to escalate their privileges to root, from which they will execute other malicious software to do the rest of the work.

The next programmer error, and perhaps the most famous for being exploited, is a lack of input validation. These errors are infamous for being the attack surface on which buffer overflow attacks are layered. Other attacks such as heap overflows and stack smashing are also possible due to this error.

A buffer overflow is a theoretically simple attack that is used to inject code into a running process to be executed by placing more data in a buffer than it can hold. The idea was first popularized by the Morris Worm in 1988, which relied on lack of input validation in the fingerd application to inject code to spread. From the attacker's standpoint, there are two steps involved in implementing a successful buffer overflow attack. The first of which is to actually locate the buffer (which is usually a simple array of data). The second is to design the exploit to target that specific buffer. Fortunately for white hat hackers and security practitioners, buffer overflows require intimate knowledge of the system architecture and a great deal of practice to implement. This means that attacks are not portable across applications and systems [3].

Consider the following C program.

Listing 1: Vulnerable Program

```
#include <stdio.h>
#include <string.h>

void doit(void)
{
    char buf[8];
    gets(buf);
    printf("%s\n", buf);
}

int main(void)
{
    printf("Before...\n");
    doit();
    printf("After...\n");
    return 0;
}
```

The main function has two calls to printf() that display two strings and the doit() function displays the contents of the buffer as entered by the user. Suppose the user entered the string "SAFE" when prompted inside the doit() function. This would return the expected output:

Listing 2: Expected Output

```
Before...
SAFE
After...
```

Now consider what happens when the user enters the string "WARNING, BUFFER OVERLOW EN ROUTE". The output will be something similar to the following:

Listing 3: Unexpected Output

```
Before...
WARNING, BUFFER OVERFLOW EN ROUTE
Segmentation fault
```

The application crashes because too much data was placed in the buffer. Under the hood, the stack was essentially "smashed", meaning that other data on the stack was overwritten by the extra data placed in the buffer, thus causing unwanted (or wanted, depending on who the user is) side effects. Since there is no range checking in the function gets() (thus qualifying it as an unsafe function to be avoided in any application) any strings larger than 8 characters will continue to be placed on the stack and overwrite whatever data is in its place. To illustrate this, examine figure (1) in which the equivalent operation to open a shell is strategically placed in an unchecked buffer to cause it to overrun and be executed.

Fortunately, the example program above simply crashed with a segmentation fault error after the buffer overflowed.

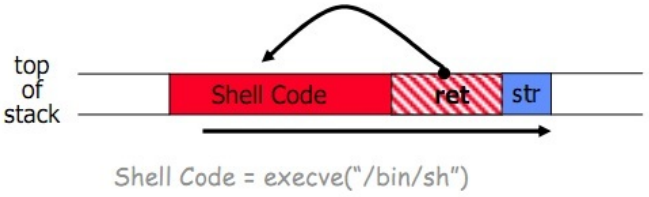

Figure 1: A visual representation of the runtime stack, showing how malicious code is strategically placed on the stack to overwrite the return address and point to executable shell code inserted by the attacker. Such code injections typically result from buffer overflow attacks.

However, if the malicious attacker were to be able to determine the location and contents of the stack at run-time, they would be able to place malicious data inside the buffer that resembles executable instructions (as shown in the figure above). These instructions are typically branches or jumps that will move the program counter to an unusual location in memory to start executing malicious code, thus successfully completing the code injection phase of the attack.

Some other unsafe functions aside from gets() include strcpy(), strcat(), scanf(), and printf(). Let's consider the strcpy() function. The function takes two arguments: a pointer to the destination character array and a pointer to the source character array. The basic implementation of this function is as follows:

Listing 4: strcpy implementation

```
char *strcpy(char *dest, const char *src)
{
    while (NULL != *src)
    {
        *dest++ = *src++;
    }
}
```

As you can see, there is no bound checking to verify that the destination array can fit all of the data contained in the source array. This function is safe for use only when the destination array is at least the same size as the source array. Otherwise, this will cause a buffer overflow and potentially lead to unwanted behavior or a segmentation fault altogether if the destination array is accessed.

A safe version of this function is strlcpy(), and its signature is as follows:

Listing 5: Safe strcpy prototype

```
size_t strlcpy(char *dst, const char *src,
    size_t size);
```

The implementation of this function is different in three regards. Firstly, it makes sure to never write outside the bounds of the destination array because the argument size specifies the size of this array. Secondly, it will properly null-terminate the source string, reducing the chance the function traversing through memory until it finds a null character or size bytes have been copied. Lastly, it returns the number of bytes copied from the source to the destination array. Clearly, these adjustments make the function safe to use for development. Developers should strive to use such safe variants of the common functions provided by stdio.h and string.h libraries to avoid program errors and potential exploits.

In today's software world, it is an implicit responsibility of software developers who write code in unmanaged languages to use safe functions and libraries. In most cases the developer can employ the use of static code analysis tools to check for potential buffer overflows. However, given the many different implementation strategies for buffer overflows and their dependence on specific platforms, these tools are not able to catch every possible program flaw. This is not as big of a problem for developers who use managed languages such as Java and C#. However, buffer overflows are still possible in these languages.

Another common software bug that is cause for concern is a time-of-check vs. time-of-use error. This is a special kind of race condition in which there is a prolonged lapse between the checking of a condition and the use of the results of that check. This is a common problem with the implementation of file systems. For example, consider the following UNIX program snippet written in C.

Listing 6: TOC vs. TOU UNIX code

```
if (access("file", W_OK) != NULL)
{
    exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

In this program, access is a function used to check whether the user who executed the program has access to the file. Upon examination, one can see that this program suffers from a critical time-of-check vs time-of-use bug. Consider what would happen if, between the calls to access and open, the same user creates a symbolic link between "file" and another sensitive file "secret" they don't normally have access to. After the call to access the user is given permission to modify the file, but the "secret" file is opened instead due to this new link. This can compromise the contents of the "secret" file and even expose them to the malicious user, which is clearly a security problem.

Despite the simplicity of this exploit from the program bug, these conditions are very difficult to avoid and eliminate. In fact, the problem of avoiding and eliminating these conditions was determined to be non-deterministic [4]. Since this result, most UNIX systems have taken precautions to avoid these bugs. Specifically, they rely on file handles, which are private mappings to a file, instead of file names, which can be modified with symbolic links. Since these handles cannot be modified by other processes and users, the possibility of this attack being implemented is greatly reduced.

B. Malicious Code

Malware is a piece of software that behaves in some way that can cause harm to the infected person's machine,

information, and even their identity. Each form of malware can loosely be defined as one of the following:

1) Virus
2) Worm
3) Trojan horse
4) Trapdoor
5) Logic bomb
6) Rootkit

Most malware is referred to as a virus since that was the first popularized form among the general public. However, viruses are only a subset of all malware, and there is a unique difference between them and the rest of the types of malware.

A virus is a piece of malware that attaches itself to another program and propagates through the host machine and across networks by copying itself on to other programs it can find. They can be copied or appended to, surrounded around, and even integrated into other program code. They can also reside in many different places within the operation system, including individual files, boot sectors, and even stay in memory (with the caveat that it is lost when the system is restarted).

The other forms of malware (worms, Trojan horses, trapdoors, logic bombs) are similar in intent to viruses but operate quite differently. Trojan horses contains unexpected, additional functionality that is used to infect the system. Trapdoors allow unauthorized access to program functionality. Worms propagate copies of itself through the infected machine's network. Lastly, logic bombs trigger (usually malicious) actions when certain conditions occur on the user's behalf.

Rootkits were left out of this list because they are very interesting forms of malware that can target virtually any embedded, personal, and mobile computing platform. Also, from a user's standpoint, they pose to cause the most damage and information loss. At a high level, a rootkit is a set of small tools and programs that are used to gain undetected and long-term access to a system without the user's consent. They typically serve to open up back-doors into systems and gain access to highly sensitive information that only the operating system and underlying hardware should access. Once a well-crafted rootkit has infected a machine they are notoriously difficult to detect and remove and can potentially cause serious permanent damage to the system. This is because common anti-virus software only monitors the user-mode of operating systems. There are programs available that will look for the presence of common rootkit installation techniques (such as hooking the system call tables), but any clever attacker can circumvent these and remain undetected.

There are many different flavors of rootkits that can target virtually any layer of a system, including the user, kernel, and even hypervisor (virtualization) levels. Considering the trade-offs attackers must make between implementation difficulty and the actual usefulness of the rootkit, the most common form of rootkit is one that targets the kernel-level of an operating system. Such rootkits are typically deployed within the kernel as a fraudulent device driver or kernel module that masquerade as legitimate components. Since these components are deemed trusted by the system, they are given full access to all memory segments (including both user- and kernel-mode memory) and devices. This means the rootkit can modify sensitive kernel data structures and device driver operation for malicious purposes, whether it is to hide its presence on the system or intercept and monitor data transfer on hardware devices.

Keyloggers often take the form of kernel-mode rootkits. This specific form of malware is installed on the system (either by the user's permission or through subversion of the kernel security mechanisms) and used to retrieve information directly from the physical keyboard device connected to the system. This is often done by monitoring the keyboard data port for new information or intercepting I/O requests as they pass between the operating system and keyboard. There are several possible ways to intercept keystrokes once the rootkit has access within the kernel, which is why such forms of malware are the most serious threat.

C. Software Engineering Practices for Security

Secure programs don't come by chance. They are the direct result of knowledgeable programmers and sound software engineering practices and processes. Secure programming practices for individual programmers consist of a variety of implementation techniques that circumvent and prevent the errors discussed in the previous sections from occurring.

Some of these implementation techniques include [5]:

1) Error/exception handling in code
2) Avoiding unsafe functions (fgets, sprintf, strcpy, strcat) and using safe ones (strncpy, strncat)
3) Validating all input from the user or external sources to ensure fully deterministic program behavior
4) Properly managing memory and object references so as to avoid any memory problems (e.g. leaks)
5) Heed compiler warnings (especially those that have stack smashing protection mechanisms) and use static and dynamic analysis tools to uncover security flaws
6) Keep the design of the system as simple as possible - security does not come through complexity
7) Deny access to sensitive objects by default
8) Filter unnecessary information from outgoing data

In addition to these techniques, it is good practice for programmers to follow standardized quality assurance and development processes for security. Perhaps the most notable security-oriented development lifecycle, which includes quality assurance, is the Microsoft Security Development Lifecycle (SDL) process. The phases and activities of this lifecycle are shown in figure (2).

From a software engineering perspective, the structure of this lifecycle is very similar to the standard waterfall model. As you can see, each phase of this lifecycle can be mapped to a phase in the waterfall model. This allows the security practices outlined in each phase to be built on top of the existing phases of the waterfall model, so security becomes an integral part of the entire development process.

The SDL is based on the three very important concepts for developers: 1) education, 2) continuous process improvement, and 3) accountability [6]. The idea is to strive to keep developers aware of security issues and have them evolve with

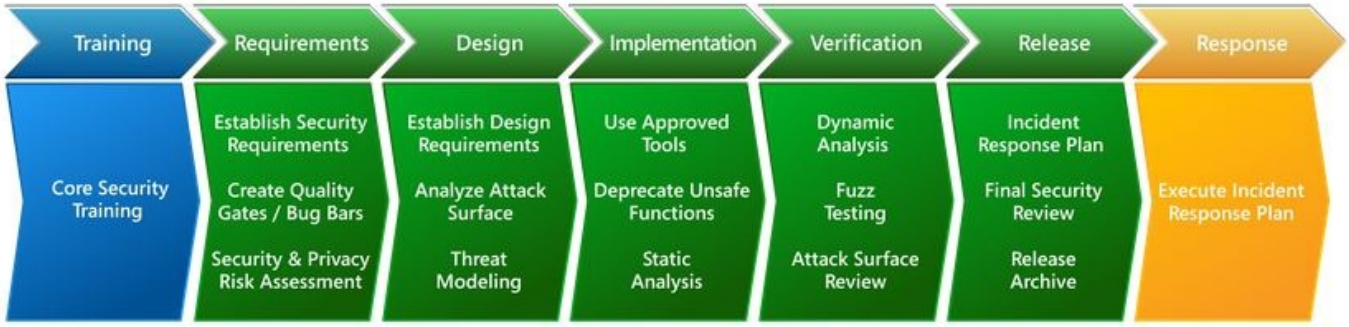

Figure 2: The various phases and activities of the Microsoft Security Development Lifecycle, arranged in a linear (waterfall) fashion.

the process itself, all the while keeping them accountable for their actions and any bugs they may introduce into the project.

There are four levels of maturity for the practices and capabilities in these areas. They are 1) basic, 2) standardized, 3) advanced, and 4) dynamic. The level of maturity for a specific instance of the SDL depends entirely upon the competency of the developers and their adherence to the activities outlined for each phase of the lifecycle. Clearly, teams that follow the SDL want to work their way towards the dynamic level of maturity, in which the team and process can quickly and efficiently adapt to emerging security issues [6].

Using the SDL does not remove the risk of security problems with software altogether; it is not a silver bullet. However, using it as a model for the development process will help reduce the risks and impact of security flaws with software.

III. Operating System Security Security

A. Protection Motivation

Operating system security is built upon the concepts of trust and protection. Both of these elements must be satisfied in order for a system to be deemed secure. For example, if protection mechanisms are bypassed, the trusted portion of the system can be compromised. Similarly, if components in the system are untrusted, then protection of individual objects within the system cannot always be guaranteed.

Protection is the second prong of operating system security. Objects that are used by the system, such as files, memory, and hardware devices, must be protected from malicious and unauthorized usage to prevent unwanted behavior. The basis of any protection scheme is separation, and in most operating systems there are varying degrees of separation [7]:

1) **Physical separation** - different processes use different physical objects or devices
2) **Temporal separation** - processes that have different security requirements are executed at different times
3) **Logical separation** - processes are restricted to their own address space and domain by the operating system, and the user is unaware of the existence of other processes
4) **Cryptographic separation** - processes conceal their state, code, and data using cryptographic primitives

However, in any realistic computing environment, processes must interact and share data in one way or another. Therefore, complete separation is only half of the solution for protection. Careful design considerations must be made in order to allow both object protection and sharing of data. There are, however, several ways operating systems can support separation in order to achieve these two goals, and they are outlined as follows [7]:

1) **Do not protect** - No protection needed when sensitive procedures are run at separate times.
2) **Isolate** - Each process is unaware of the presence of other processes, and each process has its own address space, files, and other objects. The operating system provides a sandbox around each process such that its private objects are kept secret.
3) **Share all or share nothing** - Process information is declared to be public or private.
4) **Share via access limitation** - Both user and process requests to operating system objects are checked against the limitations imposed on such users/processes for the object in question.
5) **Share by capabilities** - An extension of access limitation in that usage and sharing rights can be created dynamically.
6) **Limit the use of an object** - Limit both the access to objects and the usage of such objects after access has been granted.

The use of these different protection policies depends on the environment in which the operating system is deployed, whether the system is multi-user or single-user, and multi-process. It is important to note that these protection policies are ordered in increasing order of difficulty to implement and the granularity of protection they provide. Therefore, more thorough protection schemes require more work on behalf of the operating system designers, and such schemes must be outlined at the beginning of development, not implemented at the end.

B. Memory Protection

Memory is the most frequently accessed and exploited operating system resource in multi-programming systems. Ensuring that process code and data is unaffected from other

processes has become a task of both hardware and software components. Most modern operating systems delegate memory protection to hardware mechanisms as there is essentially no overhead for such protection. The two most commonly used protection mechanisms are segmentation and paging. Segmentation is the simple of idea of separating a program into logical pieces. Each of these pieces has a logical unifying relationship among all its code and data. Segmentation is great for producing the effect of a variable number of unbounded base registers used for code memory protection. This means that a program can be divided into any number of segments, each with its own enforceable access rights [8].

From an implementation standpoint, each segment has a unique name that is used to address its internal code and data. Specifically, the pair ¡name, offset¿ is used to reference the element that is located offset from the start of the name segment. Each of these segments can be arranged in memory in a non-contiguous fashion, meaning that they can be moved from one location to another so long as the name and offset fields are updated with the new location information. However, from the developer's viewpoint, each segment is arranged in a contiguous fashion.

This also means that the user does not need to know (and actually cannot know) the true memory address of a given segment without access to the processes' Segment Translation Table (STT), which is used to translate ¡name, offset¿ pairs to actual memory locations. Possession of this pair is all that's needed to access segments within memory, which means that physical memory addresses remain hidden from the user.

There are a couple benefits that come from hiding the layout of physical memory. Firstly, the operating system can relocate segments at runtime without the user having to worry about these new locations. This helps handle external fragmentation that occurs as more and more processes are allocated memory for code and data. Secondly, a segment can be removed from main memory if it isn't being used. This is typically utilized to store segment data to secondary storage devices for later access. Lastly, and probably most importantly, every memory address passes through the operating system. This provides the system with the opportunity to check each for valid access rights and authorization (essentially serving as a memory reference monitor), which is very appealing from a security standpoint. This also implies that a process can only access a segment if it is within its STT. Furthermore, each segment within a process can be assigned classes with different levels of protection granted by the operating system [8]. Some additional protection benefits from segmentation include multi-user access with enforced access rights and the assurance that a user cannot generate an address to a segment which they are not permitted to access.

Despite these many benefits there is one major aspect of segmentation to consider when during before use. That is, the size of segments must be maintained by each STT and checked against the offset value in the segment pair. The reason for this is that any useful segmentation scheme must allow for dynamically-sized segments that change with the code and data for each process. If the size of these segments is not checked against the offset value, then a use may be able to craft segment pairs that refer to segment data they are unauthorized to access. Clearly, checking each offset value is tedious and expensive, but it must be done for practical implementations [7].

Additionally, segmentation schemes can suffer in performance from the encoding of name values inside CPU instructions and the possibility of fragmentation. Given these consequences of its implementation, segmentation schemes must balance protection with efficiency in real-world applications.

Paging is the other most popular memory protection scheme. It is similar to segmentation in that program code and data is separated into blocks of main physical memory. They are also similar to segmentation schemes in that each address in a paging scheme is a tuple consisting of a page number and offset. The addresses are translated in a similar fashion through a Page Translation Table (PTT).

The biggest difference for paging schemes is that each page is a fixed size, where each page is referred to as a page frame. Perhaps the biggest implication of this fixed size is that the programmer does not need to be aware of the specific page locations for code and data within a program. However, this also means that any change to the program's contents (whether it's code or data) will result in a change in the page frame contents.

One common problem with paging is that a page frame can only have one set of read and write permissions. Therefore, if two programs share a page frame for code and data there is no way to differentiate between the separate content and no way to enforce separate permissions for that frame.

Based on the characteristics of each of these memory protection schemes, one can see that paging clearly offers improved performance and segmentation offers improved logical protection for memory addressing. The characteristics of these two schemes have led to hybrid designs that utilize both, referred to as paged segmentation. An image describing how this scheme works is shown in figure (3).

C. General Object Protection

Although protecting memory is arguably the most important goal of protection for an operating system, there are other objects in the system that must be protected as well. Some of these objects include:

1) Files and data sets on secondary storage
2) Executing programs
3) Directories of files
4) Hardware devices
5) Operating system tables (e.g. system call tables)
6) Passwords and user authentication data

There is no real standardized way of protecting these objects as they can differ between operating system distributions. However, a common set of complementary design goals and practices has been accepted by the scientific community to help guide the development of these protection schemes. Firstly, the system should check every access to these objects against the set of permissions it maintains. Secondly, the system should practice the principle of least privilege, in which an object only has access or permission to access only those

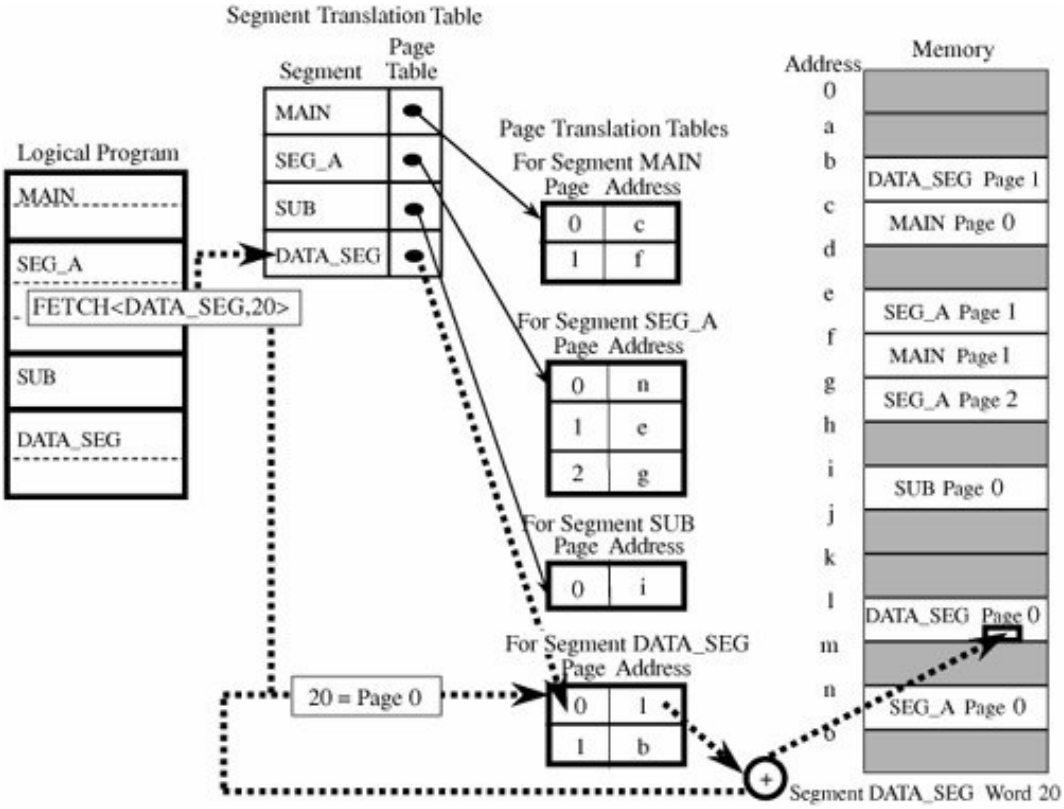

Figure 3: This image depicts the paged segmentation memory protection scheme. As you can see, the STT is used to map logical segments to specific page entries, which are then mapped to physical addresses using a unique PTT depending on the segment entry.

objects it needs to operate correctly. Lastly, the system should strive to verify appropriate usage of these objects.

All modern operating systems make tradeoffs between these three goals when designing protection schemes. This is largely due to the difficulty of protecting against all of the possible threats that target these objects. A very simple example of this idea revolves around keylogger malware. Consider the keyboard and corresponding keystrokes of data that need protection from keylogger malware. Without checking the usage of all keystroke data as it flows throughout the system it is very difficult to determine whether or not the physical device drivers or I/O threads are using them appropriately. Therefore, protection schemes for these types of objects generally rely on permissions and limited access to such data.

These generic protection schemes are usually implemented using a variety of different mechanisms, including access control lists and matrices and capabilities. Access control lists are conceptually simple structures that store objects, the list of the subjects that should have access to that object, and what that access is (e.g. read, write, execute permissions). Quite similarly, an access control matrix is a way of associating access rights for objects with subjects using a two-dimensional matrix (the objects make up the row and the subjects make up the columns). One might choose an access control matrix over a list if there are very specific access rights for each object in the system (which means the matrix would be dense as opposed to sparse).

Capabilities are another form protection that lift the burden of protection from the operating system and place it on the user. Formally, capabilities are unforgeable tokens that give the possessors certain rights to an object. In theory, subjects should be able to create new objects and specify the operations allowed on them. For example, a user can create a file and mark it as read-only, which means that only this user and those who have a higher set of privileges (typically administrators) are capable of modifying these access rights.

IV. Operating System Survey

A. *Singularity*

Singularity is a small, modular, and managed operating system that has been under development by Microsoft Research since the early 2000s. It is centered on five basic concepts: a type safe abstract instruction set as the system binary interface, contract-based communication channels, strong process isolation architecture, managed and verifiable system components, and a ubiquitous metadata infrastructure describing code and data [9].

Architecturally, Singularity adheres to a microkernel design to limit the amount of code running in the kernel. However, its trusted computing base (TCB) is comprised of both the kernel and the managed runtime, as shown in figure (4).

One of the goals of Singularity was to ensure that the components of the TCB could are small and simple enough to be formally verified. Clearly, the complexity of the TCB is

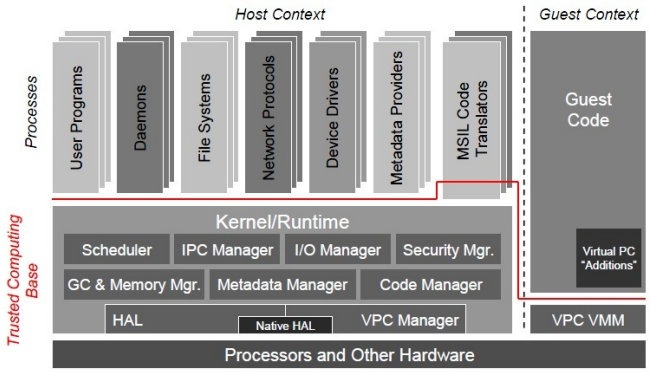

Figure 4: The Singularity architecture. Note that the red line indicates the separation between the trusted and untrusted portions of the system.

relatively small since file systems, network protocols, device drivers, etc. are located outside the bounds of this component. This also means that no third-party code can run within the TCB. This separation between trusted and untrusted code and data in Singularity is one of its core security features.

1) System Binary Interface: Singularity's system binary interface (more commonly referred to as their application binary interface, or ABI), is used to allow processes to communicate with the kernel. It was designed with the following goals in mind:

1) Enable separate garbage collection domains for each process and the kernel
2) Enable separate runtimes for each process and the kernel
3) Enable independent versioning of processes from the kernel
4) Enable strong process isolation

The ABI is a set of static functions exposed by the kernel that are used to create and manipulate threads within the calling process, create and use thread synchronization objects, activate child processes, create and manipulate message content, send and receive messages via channels, securely determine the identity of another process accessed through a channel, allocate and free pages for GC memory, grow and shrink thread stack segments, access process parameters, and terminate the calling process when it completes [10].

Perhaps the most unique element of the design of this interface is that a link between a process and the kernel cannot be intercepted or hooked without explicit approval of the process' author. Modern operating systems such as Windows NT and Linux allow user programs to hook into system call chains to intercept data as it is forwarded down the call stack and sent back up the call stack from the kernel. The ability to hook such system calls has been exploited countless times by malware developers when trying to gain access to the kernel mode of the system or attain super user privileges. Without the ability to hook into this call stack without explicit permission from the calling process, these hooking attack vectors are rendered useless and the private data sent to and from the process to the kernel is kept secret.

Another important part of the ABI design is that no function call can modify the state of other processes. The scope of any ABI function is limited to the calling process, the communication channel used to invoke the function, and the kernel. As such, the integrity of other processes' code and data is maintained and kept safe from malicious processes trying to modify other processes in the system and implementation bugs that would normally bring down a system.

In addition to exposing this set of functions, the ABI also provides processes with core services (e.g. debug and device servers) that can be used when developing programs. These primitive operations provide the following services:

1) Thread construction and destruction operations
2) Channel construction and destruction operations
3) Endpoint bind and unbind operations
4) Operations on thread synchronization primitives
5) Message send and receive operations
6) Operations to determine the security principle associated with the opposite endpoint of a channel

2) Contract-Based Communication Channels: In traditional multi-process operating systems there exists a variety of inter-process communication (IPC) mechanisms used to establish lines of communication between different process and services within the system. In systems where process cooperation and communication is important for information sharing, computation speedup, modularity, and convenience, the two fundamental models of IPC that are commonly used are shared memory and message passing [8].

Shared memory is an IPC scheme that allows multiple processes to access the same block of memory so as to avoid redundancy of data between such processes and increase IPC efficiency. The typical usage scenario is where one process will create an area in memory and then hand out permissions to access such memory to multiple processes that need to access its data. This increases the speed at which processes communicate as there is no significant implementation difference or overhead when accessing shared memory areas versus local (and thus private) memory. Dynamic link libraries are typically stored in shared memory when they are loaded so that their exposed functions can be invoked by multiple processes without copying the library into the address space of each process.

On the other hand, message passing is an IPC scheme that allows processes to send and receive messages, which are typically data or segments of code, to other processes. There are many different implementation forms of such schemes, such as remote procedural calls, signals, and data packets, but all of them address the following basic principles:

1) Whether messages are transferred reliably
2) Whether messages are guaranteed to be delivered in the order in which they were sent
3) Whether messages are passed one-to-one, many-to-one, or one-to-many
4) Whether message passing is synchronous or asynchronous (e.g. blocking or non-blocking)

These design choices depend on the specific type of application, platform, and protocol requirements.

Singularity makes use of a message passing model of IPC with contract-based communication channels [11]. Furthermore, there is no use of shared memory throughout the entire system, thus prohibiting dynamic code loading by processes. The reason for this comes from the design decision to use strict FIFO communication interfaces for process communication and single-process ownership of data. When data is sent between two processes across a channel, the ownership of that data is transferred from the sender to the receiver. Since only one process can manage a piece of data at a time the risk of such data being compromised by other processes is greatly reduced.

Furthermore, processes cannot simply communicate with any process they choose at run-time. Communication channels are governed by contracts that are statically defined, meaning they cannot be changed after installation. The main properties enforced by channel contracts are 1) senders send messages that are expected by receivers, 2) receivers are capable and willing to handle all messages allowed by the contract, and 3) senders and receivers that have been verified separately against a given contract cannot deadlock with communicating over a channel governed by such contract. This level of specification for IPC goes far beyond traditional schemes. One can clearly see how it assists in formalizing and verifying program behavior, which is a very appealing security attribute for operating systems.

Channels are created dynamically at run-time based on the contracts available between two processes. Each channel has exactly two endpoints that are associated with exactly two processes. From the processes perspective, only the endpoints can be seen and used to send data; not the channel object itself. Internally, every channel contains two FIFO, unidirectional queues that are used to send and receive messages between the processes. What is unique about channel endpoints is that, just like other data objects, they can be passed to other processes across other channels. It is also important to note that when a process is terminated every channel it is associated with is shut down as well. This prevents loss of data by removing the possibility of sending messages to processes that no longer exist.

3) Process Isolation: Perhaps the most significant design feature of Singularity is its introduction of Software Isolated Processes (SIPs) [12]. Unlike traditional operating systems that provide process isolation and address space isolation through custom hardware schemes like paging and segmentation, Singularity SIPs provide isolation by type and memory safety features of managed programming languages.

There are several benefits for this approach. Firstly, it removes the errors associated with using unmanaged languages for software development. Using unsafe languages like C and C++ to develop operating system and application software is a very error-prone task and introduces unwanted security risks. Languages like C and C++ rely on the knowledge and expertise of the programmer to know what data they are manipulating in memory and how to properly manage their own memory resources. For example, device drivers are mainly written in C so that the programmer can directly access device I/O ports and status registers necessary for I/O operations. However, this freedom comes at the cost of safety and security, since applications in C are compiled directly into machine language representations that are executed directly on the host processor. Any mistake in memory address references, memory leaks introduced by poor memory management, and null pointer references made on behalf of the programmer can wreak havoc on other parts of the operating system. Another benefit to this approach is that it greatly reduces the possibility of buffer overflow and stack smashing attacks.

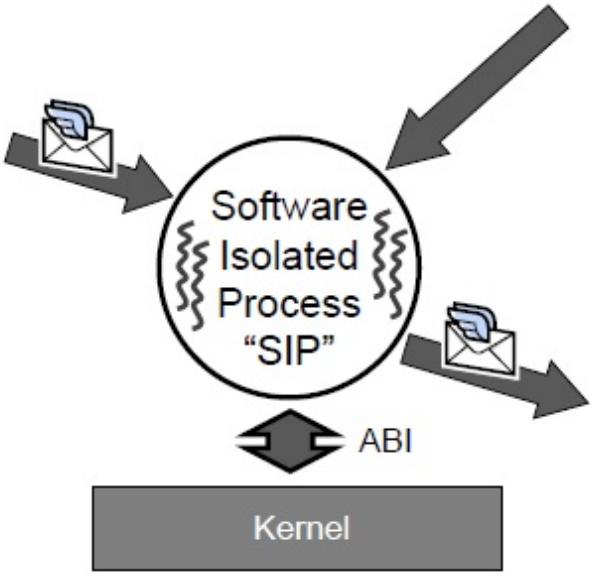

Figure 5: A visual representation of a software isolated process. The only input and output for the process is through messages sent by other processes. Also, the kernel functionality is exposed through the ABI that is made available to all processes.

Most modern operating systems are written in a combination of C and C++, both of which are considered to be unsafe languages. From an operating system development perspective, C and C++ are appropriate languages because they give the programmer direct access to memory resources they need for development. For example, device drivers written in C can easily access I/O ports and status registers using pointers. Unfortunately, however, this freedom comes at the cost of safety and security. Since C and C++ code is compiled directly into machine language, any bugs in the code can wreak havoc on the rest of the operating system if not handled properly. This makes them susceptible to buffer overflow attacks.

Finally, since most managed languages are object-oriented by default, the design and development of software processes is greatly simplified. Modern software engineering practices such as program design using class diagrams that identify the relationships between components of the program can be employed. Furthermore, it is almost always easier to translate security policies and models to object-oriented software. Security policies typically place restrictions on information flow and component interactions, and the notion of relationships and interactions is a natural part of the object-oriented paradigm.

Internally, a SIP is an object space composed of objects that can be accessed only through object references, which is a protection mechanism ensured by the Singularity run-time. This means that every process can run within the same address space due to the protection that exists between processes objects. This removes the overhead of context switches between address spaces for different processes during the scheduling process. Now, a context switch can be done in a single system call.

4) Managed and Verifiable System Components: The majority of the operating system kernel and all user programs, file system drivers, and device drivers are written in Sing#, an extension of C#. Certain elements of the system that make up the trusted computing base (TCB), such as the kernel runtime and garbage collector, are written in an unsafe version of Sing#. This allows those components to perform unsafe operations that typically involve pointer usage. The most notable implication of using Sing# for the majority of the implementation is that the majority of the operating system can be mathematically tested and verified using formal methods. Most operating systems are simply too complex to be verified, meaning that their unchecked behavior is susceptible to attack from malicious code.

B. Caernarvon

Caernarvon is a smart card operating system that was designed and built from the ground up with security in mind. It targets the Evaluation Assurance Level (EAL) 7, which is the highest level of the Common Criteria standard for evaluating information security components. To meet this standard, Caernarvon was designed and implemented with a formally specified, mandatory security policy providing multi-level security (MLS) suitable for any mission critical application. Additionally, it includes a strong cryptographic library that has been certified under the Common Criteria at EAL5+ for use with other systems [13].

Caernarvon was designed with the following security properties:

1) Prevention of unauthorized information disclosure
2) Privacy-protecting authentication protocol
3) Hardware protection mechanisms to enforce security
4) Assurance of system state across power failures
5) Cryptographic library certified at EAL5+ (2048-bit RSA, DSA, 3DES, SHA-1, and pseudorandom number generators)

The prevention of unauthorized information disclosure is accomplished using both mandatory access control and discretionary access controls. The difference between these types of access control lies in how access rights are granted. Mandatory access control is a policy in which the system manager or security officer sets permissions on system objects. On the other hand, discretionary access control is a policy in which the user or owner of an object sets the access rights of such object.

The reason that two forms of access control are implemented lies in the impact of malware on system information. For example, malware can be implemented to make copies of protected information that is then passed along to unauthorized users to view. Confining protected information using discretionary access control has been proved to be an undecidable problem equivalent to the Halting problem [14]. Therefore, mandatory access control was put in place to prevent known malware from leaking protected information by placing strict access rights on specific objects of interest.

All of remaining components of the operating system are depicted in the system architecture image shown in figure (6).

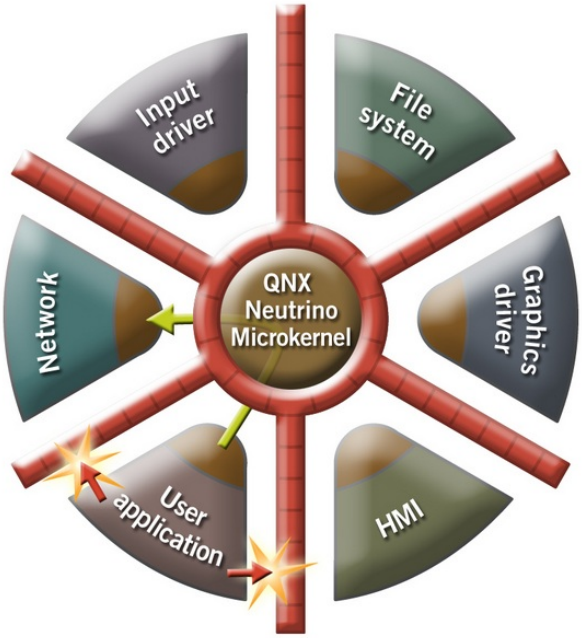

Figure 7: A high-level depiction of the QNX operating system. The microkernel is the basis for the entire system, as it provides a means of communicating with each component and a reference monitor boundary around such components.

C. QNX Neutrino

The QNX Neutrino RTOS Secure Kernel is a mission critical, real-time operating system designed to meet the needs of aerospace, defense, and security systems that have strict safety and security requirements for their applications. It was the first full-featured RTOS to be certified under the common criteria standard (meets standards of Evaluation Assurance Level 4+), include symmetric multi-processing support for multi-core processors, and include QNX's unique partitioning technology in the certification [15].

The security foundation in Neutrino comes from its design, not from its implementation. As in most modern operating systems that are designed with security in mind, it has a microkernel based architecture to provide the system with a fault tolerant way of dealing with poorly designed and implemented applications or device drivers. The basic structure of Neutrino is shown in figure (7).

As you can see, there exists a strong separation of concerns that is enforced by boundaries between the components of the system. These boundaries provide a layer of security not

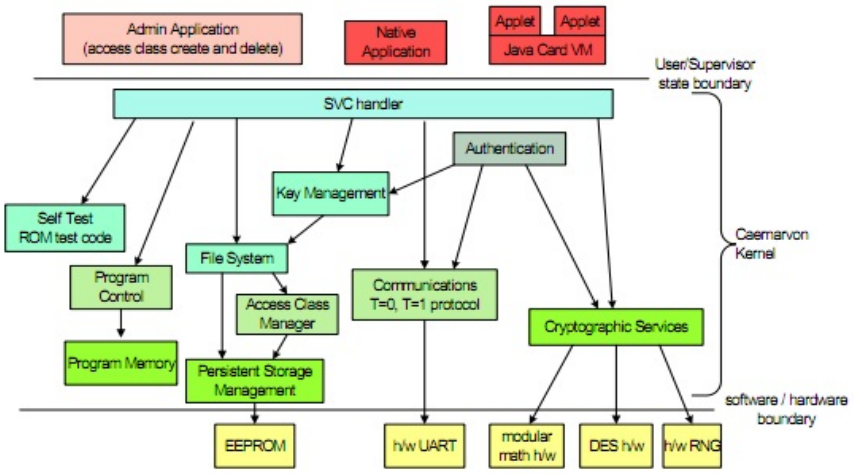

Figure 6: The system architecture for the Caernarvon operating system, which shows the logical segments of the hardware and software portions of the system and their relationships.

only around the kernel itself, but around those applications that run in user-mode as well. They are also responsible for the enforcement of process isolation between the system components.

Within these boundaries lies the reference monitor, which serves to ensure that every resource is accessed not only by the appropriate software process, but also by the right process operating against the correct data in the correct context. In addition, the reference monitor is also tamper resistant, it is always invoked, and it is small and simple enough to be easily verifiable. These are the properties of the reference monitor concept proposed by James Anderson in his 1972 survey paper of computer security entitled Computer Security Technology Planning Study [16].

Clearly, if malicious users can bypass the reference monitor to access confidential data or corrupt another portion of the system, then the usefulness of the reference monitor degrades. If the integrity of the reference monitor is affected by another component or process then the entire system can no longer be trusted to behave correctly. In Neutrino, the reference monitor is part of the operating system kernel, which is marked as read-only on boot. This means that it cannot be tampered with or modified at run-time. Integrity checks are also performed to ensure that the kernel image is undamaged and uncorrupted on boot. This means that, if the integrity checks pass, the kernel is loaded and the reference monitor is started to form a part of the trusted computing base (TCB) of the system [15].

The reference monitor is also always invoked in order to ensure that every resource access request by a process is verified thoroughly according to the system security policy. Since the reference monitor is implemented as part of the kernel, every access request goes through this uniform verification mechanism. Depending on the security policy in place that the reference monitor uses to verify and validate accesses, the resource is either granted to the calling process or it is denied. Any invalid requests are saved in the system auditing information log, as shown in fig (7).

Finally, the reference monitor, in addition to the kernel itself, must operate correctly at all times in order for the system to be trusted. Therefore, the design of the kernel and its internal reference monitor is simple enough to be verifiable by modern formal method tools and techniques. By keeping the design and implementation as small as possible, the chance of unseen security holes existing decreases substantially. This is yet another reason why the microkernel approach for the operating system was chosen. Since only the bare essential services of an operating system, such as virtual memory management, IPC, and scheduling, are implemented inside the kernel in this architecture, the complexity of the kernel design is greatly minimized, which ultimately improves the formal verification and validation process. Additionally, since there is less code that actually makes up the kernel, there is less chance of programmer errors propagating throughout the system and causing unwanted security vulnerabilities.

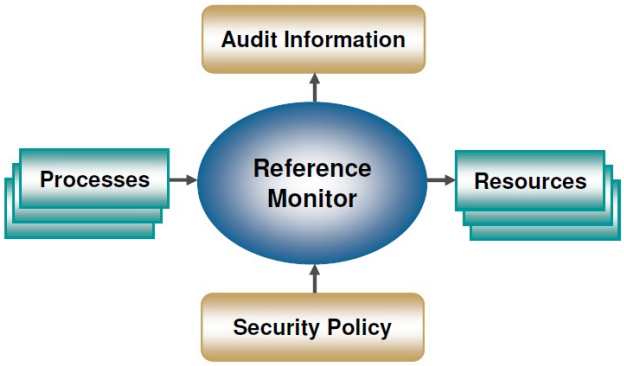

Figure 8: The general operation for a reference monitor. Based on the specified security policy, the reference monitor will either grant or deny a process access to system resources. Also, audit information is generated based on the activity of the monitor.

1) Principles of Security Design in QNX: In 1974 Jerome Salter and Michael Schroeder used Anderson's three principles

Eight principles of a security design	
Economy of mechanism	Reduce complexity to eliminate unexpected side effects or behavior.
Fail-safe defaults	Deny a subject access to an object, unless the subject has been given explicit permission.
Complete mediation	Check every access to an object to ensure that the access is allowed.
Open design	Don't rely on security through obscurity.
Separation of privilege	Grant access to an object only if the subject satisfies multiple conditions.
Least privilege	Provide a process or user with the least set of privileges possible to complete a given job.
Least common mechanism	Prevent resources from being shared implicitly.
Psychological acceptability	Ensure the system is understandable by users.

Figure 9: The eight security principles that guided the design of QNX.

of a security design to create eight of their own, as shown in the table above. The following enumeration provides a more thorough description of each principles and their role in QNX.

1) **Economy of mechanism** - The goal of this principle, as stated in the table, is to reduce the complexity as an attempt to reduce unexpected side effects or behavior. When it comes to operating system development, this idea emphasizes a microkernel architecture because the complexity of the most trusted component of the operating system is kept to a minimum. Microkernels are small enough that their implementation and behavior can be verified using formal method techniques so that it only does exactly what it was programmed to do.
2) **Fail-safe defaults** - According to this principle, resource access must be explicitly given and implicitly denied at run-time. Following this principles, the operating system must assign each new object in the system known good security attributes. Furthermore, it must only grant objects additional security attributes or clearance when they make valid requests for such items.
3) **Complete mediation** - Complete mediation in operating systems is often a computationally intensive task that involves a great deal of overhead. However, it is necessary in order to ensure that every access request is consistently and explicitly evaluated. QNX ensures mediation through strict adherence to the POSIX API and interlock hardware (typically through the memory management unit).
4) **Open design** - An open design makes it easier for reviewers to evaluate the system from a security perspective. As mentioned earlier, QNX adheres to the POSIX guidelines and other standard APIs to ensure that the behavior
5) **Separation of privilege** - Privileges for objects and users are a fundamental part of operating system security. Keeping privileges separated by multiple authentication criteria (e.g. using two locks to open a door) ensures that the system is more resilient to accidental and intentional request forgeries.
6) **Least privilege** - Least privilege is the concept that a process is granted only the security privileges and resources that it needs to function correctly, nothing more. This minimizes the risk of damage caused to the system by a process either accidentally or intentionally.
7) **Least common mechanism** - This principle implies that resources cannot be shared implicitly between two components of the system. QNX satisfies this principle by strictly managing the possession of data being used by the system and its components. If a process relinquishes data or control over a resource, then QNX will quickly regain control of it and clear any information that is still lying behind.
8) **Psychological acceptance** - The system should operate in a deterministic fashion in accordance with the provided documentation and APIs. This is typically done by following well-known and accepted standards (e.g. POSIX APIs).

There have been changes to these security principles since their release in 1974. The addition of accountability, priorities, self-tests, fault tolerance, and adaptive partitioning have become a critical part of this list and are also directly applied to the QNX Neutrino system.

Any system that manipulates sensitive information must have some form of accountability of mechanism. QNX satisfies this requirement by time-stamping all events that happen within the system. These time-stamps cannot be modified, which prevents attacks that try to subvert auditing systems.

Additionally, priorities must be assigned to process threads and system operations. Higher priority items must be able to execute without interference from lower priority items. QNX uses prioritization during the assignment of system resources. If such resources become limited or scarce, higher priority items will be moved to the head of the resource allocation queue so as to prevent starvation. In any secure system it should be impossible for lower priority items to cause higher priority items to starve.

Self-tests are an obvious addition to the secure design principles list. There must be some way to test the integrity of stored code and data so as to stop attacks as early as possible. QNX performs self tests on its trusted computing base and program code and data in order to detect corruption. This prevents further damage to system by not executing or invoking code or using data that has been corrupted by malicious intent or accidental failure caused by hardware, firmware, or software.

Fault tolerance plays a large role in secure operating systems. Any secure operating system should always operate in a known safe state. In the event of a failure that compromises the state of the system, it should also be able to roll back to

a safe state to avoid any data loss or system damages. QNX relies on the results of self-tests to determine if the system is in a known safe state. Also, if a process in QNX crashes, it can easily relinquish the resources used by the process and restart it from a known good state.

As an added level of fault tolerance through design, QNX employs a high availability framework that allows developers to use the services of such framework to implement fault tolerance instead of reinventing the wheel in their own way. The QNX high availability framework provides the following [15]:

1) Services for automatically restarting failed processes without restarting the system
2) Embedded watchdog timer to monitor processes for failures or deadlocks
3) An API that allows the programmer to determine the watchdog timeout and error conditions and, in the event of a failure, specify which actions the watchdog will execute to handle the failure
4) A guardian process that stands ready to take over the watchdog process if necessary

The last notable design concept in QNX is adaptive partitioning. A common technique for designing security into an operating system is to clearly partition system components that are susceptible to attacks and program errors. This is especially important for systems that are open to access from an external network or hardware devices that may contain malware. According to this principle, system resources such as CPU time are partitioned to ensure that critical system components have access to these resources at all times. Traditional partition schemes use fixed limits on system resource usage to ensure that each partition gets access to the resources they require. The problem with this approach is that any partition that is not using these resources actively during their allotted time will spin in an idle state, thus taking away these resources from other partitions.

QNX has been pioneering a form of adaptive, dynamic partitioning that takes advantage of unused system resources [15]. The basic idea is that if a partition doesn't use all of its allocated resources, then the partitioning scheduler reallocates those resources to other partitions that could make use of them. The fixed limits are still in place to avoid starvation of other partitions, but the adaptive partitioning solves the problem of unused resources.

From a security perspective, adaptive partitioning allows the system designer to separate vulnerable components from system critical components in different partitions in an efficient manner. This helps mitigate malicious program behavior or code in one partition from affecting the contents of another partition.

D. Chromium OS

Chromium OS has been designed from the ground up with security in mind with the realization that it is an ongoing, iterative process throughout the entire lifetime of the operating system. It is a unique operating system in that it is entirely web-based. The chromium browser is the user's portal to the Internet and all web applications contained therein. Given the inherent security risks of Internet usage and web applications, the Chromium designers took care to build an operating system and browser that is both modular and secure in design by making the system secure by default with multiple levels of defense.

In order to mitigate the probability of the system being compromised by a malicious user or program, Chromium uses a variety of OS hardening, data protection, and verification procedures. OS hardening is the idea of minimizing the systems exposure to threats by fully configuring the operating system and removing unnecessary applications [17]. The OS hardening techniques used in Chromium serve to minimize the number of attack vectors into the system, reduce the likelihood of a successful attack, and reduces the usefulness of user-level attacks if they are successful. Each of these techniques are independent from each other to provide multiple levels of security. This helps reduce the possibility of a failure in one to cascade into another.

Out of all of these OS hardening techniques, process sandboxing is the most significant as a means of isolating processes from each other and the rest of the system. Since the user only interacts with Chromium through the browser, and every tab in the browser is its own process, one can see why isolation of individual tabs is crucial for security. Malicious code that is executed in a single tab's process might cause such process to crash, but it will not be able to harm any other parts of the system since it is maintained within a safe sandbox. This also keeps malicious code from executing outside of the user space and potentially corrupting the integrity of the operating system. In a sense, the sandbox can be viewed as a reference monitor that keeps checks the validity of every request to system components and resources.

1) Chromium Security: Process sandboxing is a not a new way of implementing process isolation (in fact, it's the same idea used in iOS), but the implementation specific to Chromium does include some interesting features that make it secure. Specifically, sandboxing in Chromium is implemented following four key design decisions [17]:

1) Mandatory access control for processes
2) Control group filtering and resource abuse constraints
3) Chrooting and process namespacing
4) Media device interposition

Mandatory access control for processes is an obvious security practice by which the operating system limits or constrains processes from performing certain operations on other system objects (including other processes). The policy set in place for access control was delegated to the operating system designers and is not modifiable by the users.

Control group device filtering is another useful security practice by which devices connected and internal to the system are separated by the privilege at which processes and users can access them. For instance, it might require a higher privilege to access the web-cam on a Chromium device than it does to access the hard drive (as all processes generally reside in RAM). The use of these devices is also limited so as to avoid starvation among other processes and overuse by a single process.

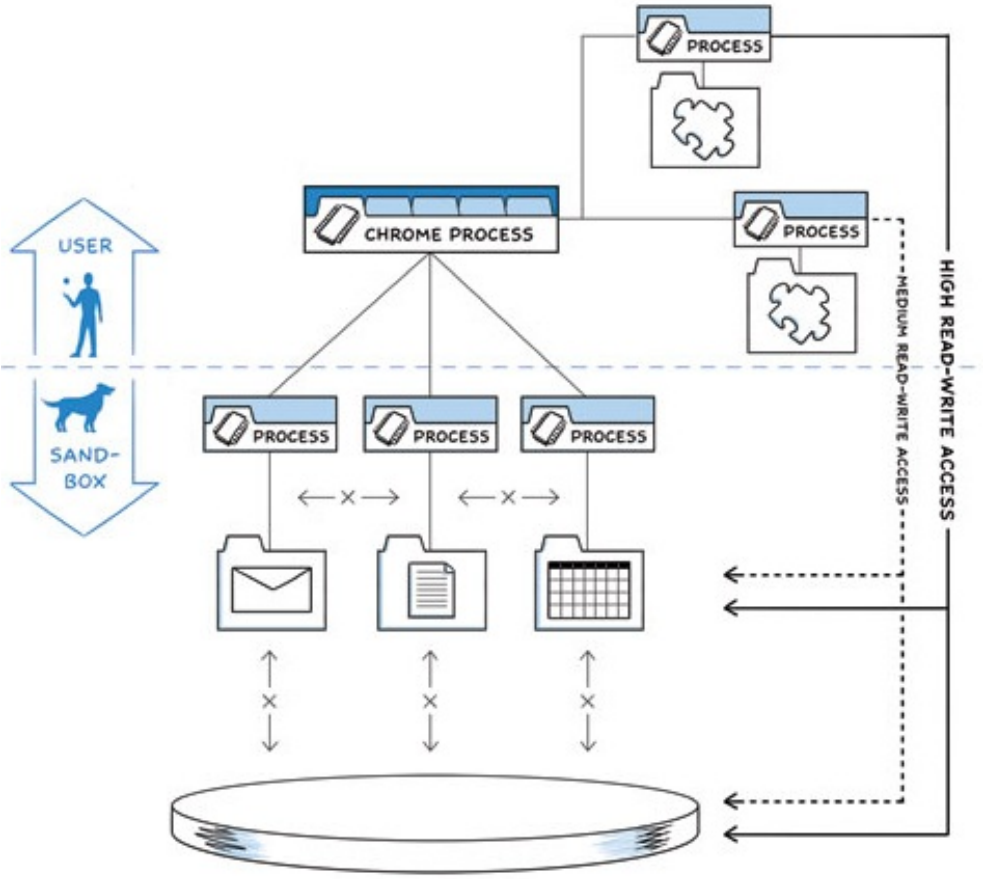

Figure 10: The Chromium process sandboxing scheme from the perspective of the user and the system. Note the varying degrees of read/write access that exist between the processes.

Finally, media device interposition, as the description implies, is the practice by why the operating system places protection barriers around media devices connected to the system. This reduces the possibility of malicious content on such devices causing damage to the rest of the system.

Sandboxing is managed entirely by the Chromium kernel, which is obviously part of the TCB. Since the sandboxing mechanism has been verified and is trusted, it is guaranteed to keep processes at a safe state throughout the lifetime of the operating system.

Some other OS hardening technique implemented by Chromium include file system restrictions and kernel hardening and configuration pairing. At the file system level, there are several key features implemented that serve to prevent the execution of code that could damage or corrupt the operating system and thwart attempts to escalate user privileges. Some of these include using a read-only partition for the root directory and prohibiting the presence of executable (privileged or not) and device nodes within a users home directory.

Kernel hardening and configuration pairing simply refers to the idea that security policies and mechanisms that govern the kernel are directly paired with a specific configuration of such kernel. If the system is updated securely (a procedure discussed in the next section), then the new configuration is paired with the new kernel image.

Another security strength of Chromium is its secure update procedure. Malicious attackers will often try to circumvent the update procedure of an operating system by injecting malicious code into forged update packages. From the user's perspective, these updates will look benign. However, once installed, the system's integrity is compromised without the user's knowledge.

Chromium's secure update procedure is a mechanism that thwarts such attacks. The techniques used to implement this feature are very simple and heavily based off of authentication and verification procedures provided by cryptographic primitives. For example, the system will only allow users to install updates that digitally signed by a legitimate source and downloaded over SSL. The digital signature alone would be enough to thwart update forgery attacks. However, using the SSL protocol to transfer update files is just an added layer of security that keeps the user safe. In addition to this feature, the update security policy also dictates that update versions can only forwards, never backwards, and the integrity of such updates is always verified on boot-up.

Verified boot is a form of self-test that uses a two-fold verification procedure at the firmware and kernel level to ensure code and data loaded from the operating system is correct and trusted (essentially, verifying that the TCB has not been compromised) [17]. It is a means of getting cryptographic assurance that the TCB of the system has not been compromised either intentionally or accidentally.

The firmware-based verification consists of the following steps:

1) Read-only firmware checks writable firmware with a permanently stored key
2) Writable firmware then checks other non-volatile memory, including the bootloader and kernel

Similarly, kernel-based verification consists of authentication and integrity checks on system files and metadata on the root file system. Blocks of data in the file system are verified using cryptographic hashes stored after the root file system on the system partition.

One useful characteristic of separating kernel and firmware verification routines is that there are no dependencies between them, so firmware-based verification is compatible with any trusted kernel. If any of these verification steps fail then the user is given the choice of continuing as normal or recovering to a safe state, which is done by loading a safe configuration from read-only, non-volatile memory.

E. Android

Android has become an extremely popular and ubiquitous computing platform for embedded and mobile devices. It is an open source software stack with an operating system, middleware, and key applications. It is built upon the Linux kernel for key features such as security, memory management, process management, network stack, and the driver model.

The Android framework is outlined in figure (11) [18]. Notice how the stack is organized in such a way that user-level applications lie at the top, and the services and devices that they use lie below. This organization scheme ties in nicely with the privilege system that governs application and user permissions in the system.

Android security is delivered in a multi-tiered design with its foundation in the Linux kernel. Higher layers in the software stack contain the other components of the security architecture, including application signatures, user identification and file access, and permissions.

Android is a privilege-separated operating system, meaning that each application runs with a unique identity (user ID and group ID). In addition, parts of the system are also separated into distinct identities. Therefore, identification-based partitioning provided by the Linux kernel isolates applications from each other and from the system critical components. This behavior is similar in nature to sandboxing, a technique utilized by other operating systems (such as Chromium) [18].

One implication of the application sandboxing design pattern is that applications must explicitly share resources and data with each other. This is done by application developers through the use of XML-based permissions (shown below), which extend the capabilities of an application beyond what is typically allowed by the sandbox.

Listing 7: Android Permissions

```
<manifest xmlns:android=
    "http://schemas.android.com/apk/res/
    android"
    package="com.android.app.myapp">
    <uses-permission android:name=
        "android.permission.RECEIVE_SMS" />
    ...
</manifest>
```

These permissions are statically declared when the application is installed, which makes users subject to social engineering techniques that try to trick them into installing malicious applications. However, to help mitigate this threat, all applications must be signed using a private key held by the developers. This certificate is used to verify the identity of the developer. Using these certificates the operating system can determine the access rights based on the application signatures.

One caveat of application sandboxing is that the kernel is only responsible for keeping processes separate; it is not a security boundary. The Dalvik virtual machine (a custom register-based Java virtual machine optimized for mobile devices) that drives the Android runtime will execute any byte code instructions it is given. Furthermore, any application can run native code (implemented using the Android Native Development Kit) [19]. This means that the strict use of both permissions and application sandboxing is the key to security in Android; not one or the other.

F. Security-Enhanced Linux

Security-Enhanced Linux, or SELinux, is a feature of the Linux kernel that provides mechanisms for supporting DoD-style mandatory access controls through the use of Linux Security Modules (LSMs) inside the Linux kernel. It is not a Linux distribution, but rather a set of features utilized by *nix-based kernels. The SELinux architecture is an attempt to enforce software security policies outlined in the Trusted Computer System Evaluation Criteria (often referred to as the Orange Book).

SELinux provides the following security policy features:

1) Clean separation of policy from enforcement
2) Well-defined policy interfaces
3) Support for policy changes
4) Individual labels and controls for kernel objects and services
5) Separate measures for protecting system integrity and data confidentiality
6) Controls over process initialization and inheritance, file systems, directories, files, sockets, messages, network interfaces, and use of system object capabilities

The reader is referred to [20] for a more thorough description of SELinux.

G. Cloud Operating Systems

With the advent of cloud computing in recent years, operating systems have started to separate from the user's machines

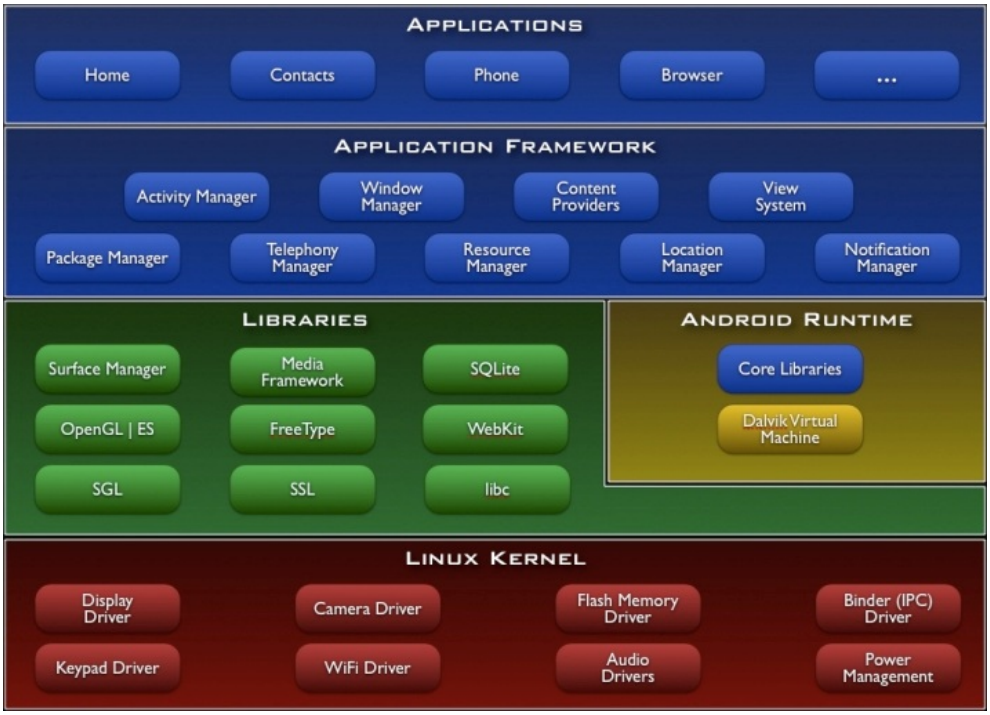

Figure 11: The Android software stack.

and migrate into the cloud to provide a seamless, scalable, and on-demand environment for application deployment. In this computing paradigm the user is no longer responsible for managing their own hardware or underlying system software, as both of these items are the located within the cloud. Furthermore, applications are no longer installed and run on the user's local machine. In this way, the cloud provides applications and software as a service through which the users securely access using a light-weight client (typically a web browser).

The framework on top of which these applications are deployed is the cloud operating system. It provides the computational and storage resources for any application that is located within the cloud. Additional functions and services are available depending on the flavor of operating system, but these two are the basis for most distributions. The goal of these operating systems is to remove the overhead and tediousness of configuring hardware and system software and focus the effort of the developers on what matters most - the applications.

Cloud operating systems are specifically designed to manage and abstract large collections of computational power and storage (e.g. CPUs, storage, networks) into dynamic operating environments that are used on demand. In contrast to traditional operating systems that are responsible for managing the resources of a single machine (or multiple machines connected through a network), cloud operating systems are responsible for managing the complexity and resources of a data center. In addition, they are platforms that provide developer-accessible services for creating applications and storing data.

Security is an important issue that cloud operating systems must consider very carefully. Since user applications and data are run in the cloud and can be accessed from any terminal with a network connection, there is an obvious need to ensure that only authenticated users can access this information (identity management). This is typically done through the use of cryptographic primitives (e.g. PKI schemes, digital signatures and certificates, password- and key-oriented protection) to enforce user authentication.

Protection mechanisms similar to those provided by traditional operating systems are also implemented to segregate the data that belongs to different users. These protection schemes rely on authentication to ensure users can only access their own data and on the storage management mechanisms to physically separate data within the cloud computing fabric. If user data requires additional security (e.g. more than protection from other user data) it can be encrypted at the different stages within the cloud for added privacy. This is a standard practice that all critical applications (e.g. online bank services) employ to keep their data safe.

The use of communication protocols such as SSL are also employed as an additional level of security. The idea is to design security in independent layers so breaches in one layer will not compromise or impact lower layers. Additional security mechanisms such as application isolation, data availability, access control for storage services, and even network traffic filtering are provided depending on the specific cloud service provider and operating system.

At a fundamental level, most security policies are enforced with public key cryptographic primitives. Since the cloud data centers store almost every portion of an application (code, data, metadata, etc.), user authentication and data privacy are two security goals that must be fulfilled in any cloud computing environment. More advanced cryptosystems that would provide data security and privacy, such as fully homomorphic en-

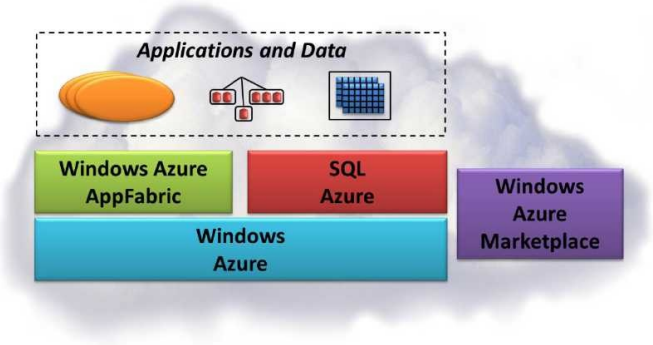

Figure 12: The Windows Azure platform, consisting of the Windows Azure system, AppFabric, SQL Azure, and the Azure Marketplace.

cryption, have been a topic of research for many years. While such systems would greatly simplify the security policies and implementation schemes for cloud computing environments by offloading data privacy to the users, this system is really only a theoretically nice idea at this point in time. Currently, there does not exist a practical fully homomorphic encryption scheme for use in cloud operating systems.

1) Windows Azure: Azure is a state-of-the-art cloud operating system platform that is composed of the our parts shown in figure (12) [21].

1) **Windows Azure** - the Windows environment for running applications and storing data on computers connected to Microsoft datacenters.
2) **SQL Azure** - relational data services in the cloud based on Microsoft's SQL Server.
3) **Windows Azure AppFabric** - cloud-based infrastructure services for applications running in the cloud or on premises.
4) **Windows Azure Marketplace** - online service where users can purchase cloud-based data and applications.

This analysis will focus solely on the Windows Azure platform and its security properties. The reader is referred to [21] for a more thorough description of the other components.

2) Windows Azure Platform: The purpose of the Azure platform is simple: run Windows applications and store data in the cloud. Internally, however, it is quite sophisticated. There are five parts of Azure that work together to provide the framework for application deployment and data storage: 1) computation service, 2) storage service, 3) fabric controller, 4) content delivery network (CDN), and 5) connect service. The compute part is responsible for actually running Windows applications on an instance of Windows Server. This means that developers can write their applications using a variety of different languages, including C#, C++, Visual Basic, and Java. Additionally, applications can take advantage of the .NET framework that is provided by Windows Server.

The storage component is responsible for allowing the application developers to store large objects (referred to as blobs) and access them using a standard querying language. It is important to note that this is not a traditional relational database, so if developers are in need of such storage they can always use the SQL Azure component, which is discussed later. Another unique property of this storage component is that it can be accessed by both Azure and Windows applications using a RESTful (Representational State Transfer) protocol approach.

The fabric controller is a very important part of the Azure platform. Again, one of the goals of the operating system is to manage the complexity of large data centers that are composed of copious amounts of computers connected together through a network. The fabric controller serves to knit these individual nodes together and provide them as a single computational and storage unit on top of which the computation and storage services are built.

Finally, the CDN is simply responsible for caching application data to data centers that are closest to its users in order to provide the fastest possible access time.

3) Windows Azure Security: At a bare minimum, Windows Azure must provide confidentiality, integrity, and availability of user data [22]. However, it must also provide built-in accountability mechanisms that can be used by customers to track the usage and administration of applications and infrastructure components by themselves and Microsoft (since all of the information is maintained on Microsoft datacenters). Confidentiality is implemented using three different policies: identity and access management that ensure only properly authenticated entities are granted access to objects, isolation to minimize interaction with data, and encryption to protect control channels and provide further protection for user data.

Azure makes use of several different key-based authentication schemes in order to enforce authentication of its users. Such keys are comprised of Windows LiveIDs and storage account keys that are granted on a per-customer basis. The user is also allowed to enforce their own authentication schemes within their applications to provide yet another layer of protection.

Certificates and user private keys play a very important role in the security framework of Azure. For example, users interact with the cloud service through the Service Management API (SMAPI) via the REST protocol over SSL [22]. These channels are authenticated with certificates and private keys generated by the users, so by maintaining the privacy of the secret keys there is a very low probability that unauthorized users will access their data. These certificates and private keys are installed in the cloud via a mechanism that is separate from the code that uses them. Both of these items are encrypted and transferred over SSL to the cloud service to be stored in the computing fabric, where the certificate and public key are also stored.

Application code and data isolation is implemented in layers in Azure at the operation system (hypervisor, root OS, guest VMs), fabric controller, and network level (incoming and outgoing packet filtering) [22]. The trusted computing base for Azure is comprised of the root OS and hypervisor. All of the user applications are deployed within guest VMs that are layered on top of the TCB. Thus, the TCB is responsible for enforcing VM isolation and separation. These separation policies draw on the decades of operating system research in the past and on Microsoft's Hyper-V product.

At the fabric controller level, communication between the fabric controller and fabric agents (those services running on virtual machines that serve users and guests) is unidirectional. Specifically, the fabric agent provides an SSL-protected service that is accessed by the fabric controller to pass requests to the agent. The fabric agent cannot initiate communication with the fabric controller or any other privileged internal node running on another piece of fabric. After the fabric controller receives a response from the agent it must be parsed as though it is malicious in content or intent. In a way, the fabric controller acts as a reference monitor for every response generated by the fabric agents, which in turn helps mitigate any potential threats to the fabric controller portion of Azure.

Finally, at the network level, packet filtering is utilized to minimize the probability that untrusted virtual machines and users generate spoofed traffic. User access to virtual machines where their applications are deployed is very limited by this filtering, which takes place at the edge load balancers and at the root OS [22]. Features such as remote debugging, remote terminal services, and remote access to VM file shares are currently not permitted by default due to this filtering scheme, but it is something that Microsoft is considering for future releases. Also, user access to storage nodes that run on Azure is very limited this filtering scheme so as to enforce strict control policies for legitimate access to data stored in the cloud.

Data privacy is enforced with the Advanced Encryption Standard (AES), hash functions such as MD5 and SHA-2 (although these are likely to be replaced with the winner of the SHA-3 competition), and pseudorandom number generators to seed the other cryptographic primitives. Furthermore, basic key management schemes, some of which probably rely on Diffie Hellman key management and distribution, are employed to allow authenticated users and the cloud service to manipulate keys within the storage service.

V. Design Proposals

This section outlines my design proposals for an experimental operating system. These ideas stem from my analysis of program and operating system security, as well as the various case studies I have explored throughout the course of my research. For the sake of brevity, these design features are simply listed below. The next stage of this project will be to experiment with these concepts in a working system.

1) Strong, verified separation of concerns between application and system software
2) Secure communication channels between OS objects (processes, device drivers, kernel, etc.)
3) Create a sandbox around each process and enforce communication using message passing queues
4) Unique identity of all processes and components in the system
5) Static enforcement of message passing scheme between processes
6) Completely verifiable design and implementation using formal method tools and techniques
7) Encrypted boot and kernel read-only memory (enforced by hardware) for kernel integrity and establishment of TCB
8) Abstraction of system hardware configuration from OS
9) Fault tolerance between components of the system (think QNX-type separation with the microkernel at the base of it all)
10) Limited dynamic code loading and shared libraries
11) Interface between processes and kernel with an efficient and secure API
12) Information sent across channels and enforcement of that policy
13) Managed language, compiler, and runtime (think C#/CLR and Java/JVM) as the basis for all applications
14) Permissions-based capabilities for access to hardware devices (ACL) based on cryptographic certification of identity

VI. Acknowledgments

I would like to thank Dr. Rajendra K. Raj of the Department of Computer Science at RIT for providing the guidance, assistance, and motivation for this project. I would also like to thank the RIT Honors Program for providing the necessary funds to support this research.

References

[1] R. K. R. Christopher A. Wood, "Keyloggers in cybersecurity education," in *Proceedings of the 2010 International Conference on Security & Management*, 2010.
[2] D. Lie. [Online]. Available: http://www.eecg.toronto.edu/ lie/Courses/ECE1776-2006/Lectures/Lecture2.pdf
[3] The Open Web Application Security Project. [Online]. Available: https://www.owasp.org/index.php/Buffer_overflow_attack
[4] A. J. H. Drew Dean, "Fixing races for fun and profit: How to use access," in *Proceedings of the USENIX Security Symposium*.
[5] R. Seacord. (2011) Top 10 secure coding practices. [Online]. Available: https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices
[6] Microsoft security development lifecycle. Microsoft. [Online]. Available: http://www.microsoft.com/security/sdl/default.aspx
[7] S. L. Charles P. Pfleeger, *Security in Computing*. Prentice Hall Professional Technical Reference, 2002.
[8] G. G. Abraham Silberschatz, Peter Baer Galvin, *Operating System Concepts*. Wiley Publishing, 2008.
[9] *Singularity Design Note 0 - Design Motivation*, Microsoft Research.
[10] *Singularity Design Note 9 - Security Model*, Microsoft Research.
[11] *Singularity Design Note 5 - Channel Contracts*, Microsoft Research.
[12] *Singularity Design Note 4 - Process Model*, Microsoft Research.
[13] E. R. P. S. K. M. S. M. W. Paul A. Karger, David C. Toll, "Design of a secure smart card operating system for pervasive applications," *IBM Research Division*, 2008.
[14] J. D. U. M. A. Harrison, W. L. Ruzzo, "Protecting in operating systems," *Communications of the ACM*, vol. 19, pp. 461–471, 1976.
[15] B. G. Paul Leroux, "Secure by design: Using a microkernel rtos to build secure, fault-tolerant systems," *QNX Software Systems*.
[16] J. P. Anderson, "Computer security technology planning study," 1972.
[17] Security overview. The Chromium Projects. [Online]. Available: http://www.chromium.org/chromium-os/chromiumos-design-docs/security-overview
[18] What is android? Android Developers. [Online]. Available: http://developer.android.com/guide/basics/what-is-android.html
[19] Security and permissions. Android Developers. [Online]. Available: http://developer.android.com/guide/topics/security/security.html
[20] Security-enhanced linux. National Security Agency. [Online]. Available: http://www.nsa.gov/research/selinux/

[21] D. Chappell, “Introducing the windows azure platform,” October 2010, microsoft Corporation.
[22] R. V. Charlie Kaufman, “Windows azure security overview,” August 2010, microsoft Corporation.