A Hardware Virtualization Based Component Sandboxing Architecture

Nuwan Goonasekera, William Caelli and Colin Fidge
Queensland University of Technology, Brisbane, Australia
Email: nuwan.goonasekera@student.qut.edu, {w.caelli, c.fidge}@qut.edu.au

***Abstract*—Modern applications comprise multiple components, such as browser plug-ins, often of unknown provenance and quality. Statistics show that failure of such components accounts for a high percentage of software faults. Enabling isolation of such fine-grained components is therefore necessary to increase the robustness and resilience of security-critical and safety-critical computer systems.**

In this paper, we evaluate whether such fine-grained components can be sandboxed through the use of the hardware virtualization support available in modern Intel and AMD processors. We compare the performance and functionality of such an approach to two previous software based approaches. The results demonstrate that hardware isolation minimizes the difficulties encountered with software based approaches, while also reducing the size of the trusted computing base, thus increasing confidence in the solution's correctness. We also show that our relatively simple implementation has equivalent run-time performance, with overheads of less than 34%, does not require custom tool chains and provides enhanced functionality over software-only approaches, confirming that hardware virtualization technology is a viable mechanism for fine-grained component isolation.

***Index Terms*—component isolation, system call interpositioning, hardware virtualization, component software**

I. Introduction

A "process" is the key software abstraction supported by modern operating systems for protecting and managing separate applications. However, with the rapid spread of component based software, a contemporary application typically extends its functionality by loading components dynamically into its process address space. For example, operating system kernels load device drivers, web browsers load browser plug-ins, and many applications support some form of extension components to provide or augment their basic functionality. Although operating system processes have well-defined isolation boundaries and inter-process communications mechanisms [1], current operating systems provide insufficient mechanisms for isolating components of a particular application from each other [2]. This is clearly demonstrated by the fact that component based software extensions often *decrease* the reliability of the hosting application; a badly-written or misbehaving component can damage the containing host, and other components, either accidentally or deliberately.

The statistics are revealing: over 85% of Windows XP crashes are due to faulty device drivers, and Linux drivers have 2 to 7 times the bug count of the kernel [3]. Such failures are not limited to kernel drivers; software applications also suffer similar problems. Zeigler [4] indicates that over 70% of crashes in the popular browser *Internet Explorer* are due to third party add-ons. In addition, over 50% of CERT-reported security threats are due to buffer overflow vulnerabilities [5]. The Java Virtual Machine (JVM) has similar vulnerabilities, because any misbehaving Java Native Interface (JNI) component has the potential to overwrite critical memory regions of the JVM, bringing down the entire virtual machine [6]. Clearly, as many researchers have emphasized, a critical need is better component isolation so that hosts are isolated from any extension components they incorporate [7-9]. Indeed, microkernel based operating systems take this concept to its ultimate manifestation [10]. Modern trends in browser architectures also emphasize the gravity of this issue; both Microsoft's *Internet Explorer* [4] and Google's *Chrome* [11] browser have changed to multi-process architectures in which program components are isolated into several, disparate operating system processes.

Before discussing component isolation further, the term 'component' must be defined because there is no general consensus on what constitutes a software component. Szyperski [12] defines it as

> *a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Mendelsohn [2] provides an overview of such units of composition, from the earliest kind of reuse in the form of subroutines, to statically linked libraries, followed by dynamically linked libraries, and culminating in component technologies such as Microsoft's ActiveX/COM, and cross-platform portable components such as SUN's JavaBeans.

Since the above definition is somewhat expansive, for our purposes we limit a *component* to be any executable binary unit which is loaded by an application into its own address space, with communication taking place between the component and its host application via a well-defined interface. Primarily, this will be in the form of dynamic link libraries (DLL)/shared object (SO) libraries, which

form the primary means of composability in modern operating systems and applications. Thus, enabling component isolation at the DLL/SO level is a much needed step in creating more robust applications.

Several approaches have been taken to address this problem. Broadly, they can be classified into hardware-supported protection, software based protection and interpretation [13]. In a previous paper, we expanded this categorization into a more fine-grained analysis of the current state of the art while identifying the strengths and weaknesses of different approaches [14].

This paper presents a new solution to the problem based on the hardware virtualization support introduced by both Intel and AMD in their recent microprocessors. We introduce a component isolation architecture which executes each component in a minimal, hardware-supported virtual machine, thus strongly isolating each component from its containing host. In contrast to a heavy-weight, full virtual machine, requiring the emulation of devices, component isolation only requires the virtualization of the processor and memory. Our technique requires only a small Trusted Computing Base (TCB) which reduces complexity and increases confidence in the correctness of the solution. The remainder of this paper provides a description of our approach and its practical implementation, a comparison with previous techniques, and an evaluation of the overall effectiveness of the approach in terms of both run-time performance and functionality.

II. Related Work

In our previous paper, we reviewed the state of the art broadly [14]. Below, we summarise those techniques which are closely related to our current solution.

A key technique in binary code level isolation is Software Fault Isolation (SFI). This method was first described by Wahbe et al. [15] and has been applied in many forms. It allows untrusted code to be placed in the same operating system (OS) process as trusted code and avoids the overhead of Inter-process Communication (IPC) between processes. It uses software based static analysis of the untrusted component's object code to verify that no illegal memory accesses will be made and to inject code for double checking any potentially harmful instructions, effectively sandboxing the original component. The sandboxed code is created so that the high bits of a memory address always fall within the sandboxed region, thus preventing components from accessing memory outside of its bounds [15]. Wahbe et al.'s [15] original idea has been improved and implemented in many forms. SFI, originally demonstrated by Wahbe et al. [15] on a Reduced Instruction Set Computer (RISC) architecture, has also been demonstrated on Complex Instruction Set Computer (CISC) architectures [16]. Techniques such as binary translation [17] are offshoots of the SFI concept.

However, a significant weakness of the SFI approach is that ensuring the correctness of the implementation is a difficult process. As Wahbe et al. [15] point out, modification of the executable binary is complicated and adds significant overhead to the code injection process because, for example, the difference between code and data can be difficult to identify. Therefore, safe execution of arbitrary binary components is difficult using SFI if the program was not compiled using an approved compiler.

The ideas in SFI are directly utilized in the Google Native Client (NaCl), which provides a software framework for safe execution of untrusted binary components [18]. NaCl aims to provide browser-based applications increased computational performance through native binary components which have access to performance-oriented features such as SSE instructions, compiler intrinsics, hand-coded assembler, etc., without compromising on safety [18].

Another component isolation technique is a multi-process application architecture. This model is becoming increasingly popular in web browsers [11, 19]. The basic idea is to isolate individual components in disparate OS processes and use the operating system's IPC mechanisms to communicate between them. In Google's *Chrome* browser, a single browser coordinating process spawns additional processes to perform sub tasks [20]. These additional processes run at a lower privilege level and access is tightly arbitrated by the coordinating browser process. In effect, different components are loaded into different processes and communication takes place using OS-supplied IPC mechanisms. This isolation into separate processes allows the browser to survive component crashes. Microsoft's *Internet Explorer 8* follows a similar model [4]. There is however, an increase in complexity as coordination between several processes is required. Also, Wahbe et al. [15] make a strong case against placing software modules in their own address space, as this requires IPC between them for communication, resulting in unacceptable context-switching overheads [15], so a trade off is made between performance and reliability [20].

Chiueh et al. [21] introduce an intra-address space component isolation scheme by using the paging and segmentation support in the Intel x86 hardware architecture, the most prevalent architecture for desktop machines. This support is used to isolate kernel extensions from the kernel itself, by placing all extensions in a separate segment of lower privilege than the kernel. They demonstrate that hardware solutions can provide high efficiency, although their technique is limited to isolating the application from all components; components themselves are not isolated from each other. Furthermore, this technique is designed for isolating trusted components from accidental attempts to violate their memory boundaries, and is not intended to isolate deliberately malicious components.

A somewhat similar approach is an application level library for isolating components using x86 segmentation hardware [22]. This approach is unique in that the entire library is implemented in user-mode, requiring no changes to the OS kernel. Google's Native Client also utilizes the above hardware segmentation technique for isolating components.

By contrast, our approach makes efficient use of the isolation capabilities provided by the underlying hardware's virtualization support. In comparison to the techniques described above, the advantages of our method can be summarised as follows.

1. It minimizes the size and complexity of the Trusted Computing Base, by moving the bulk of the responsibility for achieving security down to the hardware level, making the implementation more understandable and less susceptible to circumvention.
2. It does not require compiler level modifications, thus enabling already-compiled binary components to be isolated.
3. It avoids incorrectly identifying components as (potentially) unsafe, merely because they contain suspicious-looking instructions, i.e., our analysis does not produce 'false positives'. Instead of attempting to *predict* the component's behaviour statically, as is done in other approaches, our technique checks the program's *actual* run-time behaviour. We thus eliminate the risk of denial of execution for programs which are difficult to prove secure using static techniques.
4. It eliminates the complexity of static analysis and verification and thus eliminates the class of errors which arise from bugs or misses in the verifier.

Of course, the main disadvantage of our approach is that hardware virtualization support is necessary. However, many modern microprocessors from Intel and AMD already provide such support, and it can be expected to become more prevalent in future.

III. Threat Model

Isolation can be analyzed from both a resilience perspective and a safety perspective. While resilience becomes the key reason for isolation in a trusted environment, safety becomes paramount in an untrusted one. This is best exemplified by internet browsers which extend their functionality through plug-in components. In such an environment, both these factors become very important, as the extension code may be of unknown provenance and quality.

Our system is designed to deal with such arbitrary binary components, from untrusted sources, which need to be executed in a constrained environment. Once a component is accepted for execution, it must have controlled access to resources, as determined by the host. Access to memory must be restricted to areas allowed by the host application and attempts to exceed these limits must be caught. The host must be allowed to constrain the component by preventing arbitrary access to the operating system call interface. Such access must be mediated and the host must be allowed to set resource limits on memory usage or disk access. On the other hand, the component must be able to freely avail itself of all safe machine instructions.

Therefore, our system is based on the observation that a process cannot perform any actions harmful to the system as long as its system call interface, which is its window to the outside world, is strictly controlled [23]. We utilize this principle for isolating components within a restricted address space, and provide strict arbitration over all system calls.

IV. Isolation Architecture

This section describes how our isolation container is used by a host application to execute components in a constrained environment. We refer to this isolation container as the *Virtualization Technology Container* (VT Container). It is, in fact, a minimal virtual machine, which uses a portion of the host application's address space for the virtual machine's own memory and exploits hardware virtualization features for safe execution of components.

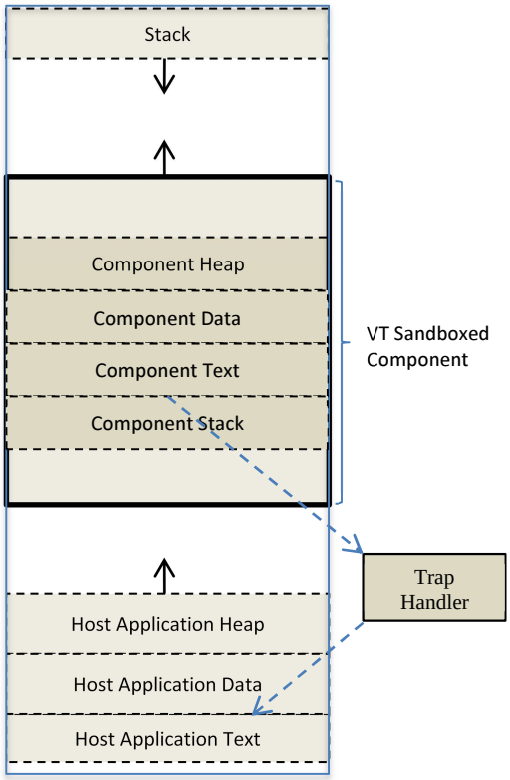

Figure 1: Virtualization Technology Container design

Figure 1 shows how a component is executed within a host application. The host application may initially request that an untrusted component be loaded into its address space. Such requests are handled by our VT Container, which allocates a block of memory for the component within the host's address space, and maps the component into it. The host may then invoke the component during the course of its execution. The component call causes execution to switch over to our VT Container, which will safely execute the component. If the component attempts to execute a potentially unsafe instruction, such as an operating system call, it is trapped by the container and the trap handler decides whether the instruction is allowed to proceed or must be aborted. If no unsafe instructions are encountered, the component's

execution ends normally and control returns transparently to the host application.

A. The VT Container's Relationship to Virtualization Hardware

Our VT Container uses the hardware virtualization support introduced to Intel and AMD processors in 2005 [24-26], to enable safe execution of a component within a minimal, lightweight, virtual machine.

Hardware virtualization is an isolation mechanism which has been used for decades. Although the popularity of Virtual Machine (VM) technology waned somewhat over the years, there has lately been a resurgence of interest in it with the development and marketing of software systems such as VMWare [17, 27], which provide a Virtual Machine Monitor (VMM) for the popular Intel x86 architecture. This has occurred even though the Intel x86 architecture itself had several non-virtualizable instructions which do not meet Popek and Goldberg's virtualization requirements [28]. Many novel techniques have been used to overcome these limitations, such as binary translation [17, 27] and para-virtualization [29-31].

In 2005, Intel and AMD introduced additional machine instructions to their respective architectures to remedy this problem [24-26]. The machine instructions were similar in nature to those of the old IBM System/370 and enabled the interpretive execution of code and additional hardware-managed control blocks. The Intel and AMD extensions are similar [17], which makes it easy to support either instruction set. Uhlig et al. [32] provide an overview of the architecture, with additional details being available elsewhere [24-26]. However, as noted by Adams and Agesen [17], early versions of Intel's and AMD's hardware virtualization did not necessarily result in better performance, due to the lack of support for Memory Management Unit (MMU) virtualization. To remedy this, AMD introduced Nested Page Tables (NPT) [26] and Intel has followed suit by introducing Extended Page Tables (EPT) in their new *Nehalem* processor architecture, both of which add support for IO MMU virtualization [33].

Our VT Container exploits these new instructions for virtualization of processor hardware. The implementation of the VT Container is based on the open source Kernel-based Virtual Machine (KVM) project [34], which provides libkvm, a simplified abstraction of the processor-specific, lower-level machine instructions.

The virtualization hardware allows the VT Container to retain selective control of processor resources, physical memory, interrupt management and I/O. Of special utility is the ability to trap on the execution of sensitive instructions, which allows fine-grained control over those components which attempt to execute potentially dangerous code. The overhead is minimal since the hardware always performs the necessary checks.

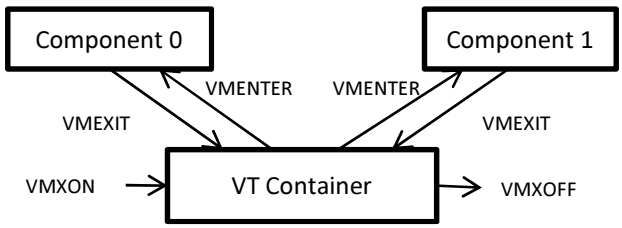

Figure 2: Overview of the VT Container process

Figure 2 is a conceptual diagram of the process on an Intel processor. Our model corresponds closely to the actual hardware implementation. The VT Container plays the same role as a traditional Virtual Machine Monitor (VMM). Each component is comparable to a guest executing on the VMM. The VT Container initializes itself by creating a minimal, light-weight virtual machine and uses the KVM libraries to initialize the necessary processor control structures for virtual execution. These control structures designate memory areas for storing the registers of the host machine as well as the guest machine, which Intel refers to as a Virtual Machine Control Block (VMCB). The areas are initialized by executing the **`VMXON`** machine instruction [24].

There are two main control structures – the host state area and the guest state area. The host state area saves the registers of the host before transitioning to guest mode execution. Similarly, the guest state area contains the processor registers of the guest. When transitioning out of the VM, the guest registers are persisted in the guest state area and the host registers are reloaded from the host state area.

In order to begin execution of a component, we invoke the KVM which in turn invokes the **`VMENTER`** machine instruction, beginning the normal execution of the component. Any exceptional situation will trigger a **`VMEXIT`** event, which is trapped by the VT Container. If the instruction passes validity checks, it can be allowed to continue. Failure to pass the checks triggers a cleanup operation and the host is notified of the failure. If no unexpected errors are encountered, component execution completes normally and the **`VMXOFF`** machine instruction is executed to deallocate the processor's control structures.

B. Implementation

In order to test and compare our VT Container concept, we have developed a demonstrable prototype of our solution, which supports both standard ELF executables as well as the modified native executables supported by Google Native Client. An additional advantage is that we can directly execute standard GCC compiled ELF executables, whereas both the above solutions require custom tool chains. We utilize the ELF loaders provided in both these implementations to create the in-memory layout of the ELF executable. In this section, we describe the loading and execution process of a typical component.

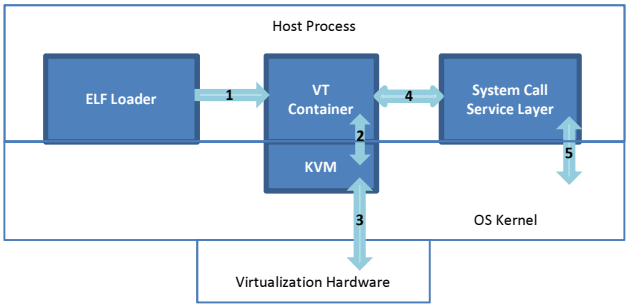

Figure 3: Implementation of the VT Container

Figure 3 shows the VT Container's context at an implementation level. We read and parse in an Executable and Linkable Format (ELF) executable. We then load the executable into our VT Container, which is where the bulk of our implementation lies. The VT Container is responsible for the safe execution of the component and the handling of any potentially dangerous instructions. The VT Container is built on top of KVM (Kernel Virtual Machine), which provides a layer of abstraction over the lower level virtualization instructions, in the form of a device driver.

If a component makes a system call, the VT Container carries it out on the component's behalf. Here too, our prototype saved a significant amount of work by building on top of NaCl's system call layer. The NaCl implementation provides a highly restricted system call layer which we have modified to suit our needs.

The typical process, as per the numbers in Figure 3 is as follows:

1. An ELF executable is loaded into the VT Container.
2. The VT Container uses the KVM device driver to execute the component.
3. The KVM module interfaces with the virtualization hardware and shields the layers above from the specific processor in use (Intel or AMD).
4. System calls by a component are intercepted and passed into the modified system call layer.
5. The system call layer may invoke the operating system to carry out the actual system call and return results to the component.

The following sections give a detailed, step by step description of the tasks carried out by the VT Container, during initialization and execution of a component.

1) Initialization of a component

This section describes the step-wise process for initializing the VT Container and loading an executable into it for execution.

1. When an ELF executable is launched, the module is initially read in and the ELF header parsed. Our implementation supports standard ELF files as well as Google's customized ELF format.
2. The ELF executable is mapped into a contiguous block of memory, which is 256MB in size by default. This 256MB block serves as the initial physical memory for the virtual machine. Thus, there is a 1:1 correspondence between this memory block and the physical memory map seen by the virtual machine. If the ELF executable is in NaCl format, the first 64KB of this memory is padded with nulls, similar to NaCl's default implementation, which helps in the detection of null pointer exceptions. We also use the trampoline code used in NaCl. This trampoline code is used to exit the VT Container in NaCl executables and carry out system calls and is described in detail below. The executable's text and data sections come afterwards. The rest of the memory is uninitialized.
3. Once the executable is mapped in and the memory area initialized, we initialize the Virtual Machine Control Block (VMCB) needed by the processor, specifying the aforementioned memory area as the physical memory block used by the virtual machine. We use KVM's abstraction layer to initialize the VMCB.
4. Once the virtual machine's processor control block is defined, we then initialize the processor registers and switch the machine directly into 32 bit mode. In this way, we avoid having to write a bootstrap loader which would switch the processor from 16-bit real mode to 32-bit protected mode. Protected mode is set by setting the Protection Enable (PE) bit in the **`CR0`** register [Intel 2007b].
5. Before protected mode can be properly used, the machine's Global Descriptor Table (GDT) must be initialized. In order to simplify our implementation, we disable paging hardware altogether and use segmentation hardware only. We use a flat memory model, and the GDT is initialized with a code segment which is the size of the text portion of the memory map.
6. We make the data segment span the entire virtual memory and the stack segment and other segment registers such as FS, GS and ES are also set to use a flat memory model, by spanning the machine's allocated physical RAM. We do not use Local Descriptor Tables and therefore do not need to initialize the relevant structures.

Thus, we directly bootstrap a minimal virtual machine with the processor already in 32-bit mode and assigned a flat memory model, greatly simplifying the programming model for a component. The memory map is shown in Figure 4.

2) Execution within a VT Container

Execution of a component within a VT Container begins as follows.

1. After initialization of the virtual machine, we set the VM's Instruction Pointer to the component's entry point. At this moment, we only support ELF executables which are statically linked and the relative locations zero based, so that we do not need to perform any relocation.
2. An initial stack is set up for the program and the base pointer and frame pointer are initialized to point to the top of the stack. Any command line arguments

used are pushed onto the stack area and the memory is adjusted as required.

3. The virtual machine is then launched via KVM, and execution begins from the program's entry point.

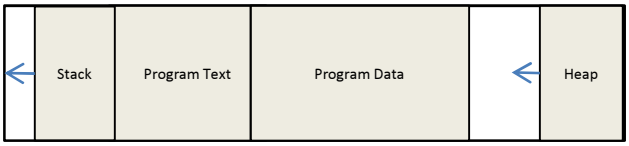

Figure 4: Component memory layout

3) Initialization of the C runtime environment

The first task performed by the running program is to initialize the C runtime environment, which is needed for basic input/output and for accessing system services. NaCl executables use a modified version of the newlib C runtime library, which is statically linked with the component. We support this same version of newlib so that direct binary compatibility with NaCl components can be enabled.

Newlib initializes itself by allocating memory for the Thread Control Block and makes a system call to initialize the corresponding operating system thread. NaCl's default implementation additionally stores a pointer to the Thread Control Block in the GS segment register. In our implementation, we modify the virtual machine's GS segment register instead. This is an example of the kind of modification needed at the system call layer in order to make it compatible with our implementation.

4) Execution of system calls

Figure 5 shows the typical sequence of actions which take place during a system call.

1. A normal ELF executable will initiate a system call by invoking INT 0x80 or by using the fast system call instructions. In the case of Google's Native Client executables, it does this by jumping to a trampoline mechanism where each system call goes through a trusted code routine.
2. We modify this routine to suspend execution of the virtual machine by executing a sequence that triggers a **`VMEXIT`** event, thereby intercepting the system call.
3. Upon interception of the system call, we carry out the actual system call after verifying the parameters. The Dispatcher routine is responsible for figuring out which system call was requested. The component can only execute a subset of the available system calls and are completely controllable, making the execution of arbitrary code secure.
4. Before the system call is executed, the parameters are validated to ensure that the values are within range and that only permitted system resources are accessed.
5. Once the system call is complete, the results are set in the virtual machine's registers and the stack and frame pointers adjusted to store the necessary return values.
6. Finally, execution resumes by changing the virtual machine's instruction pointer to resume execution from the return address stored on the stack.
7. The untrusted code resumes execution.

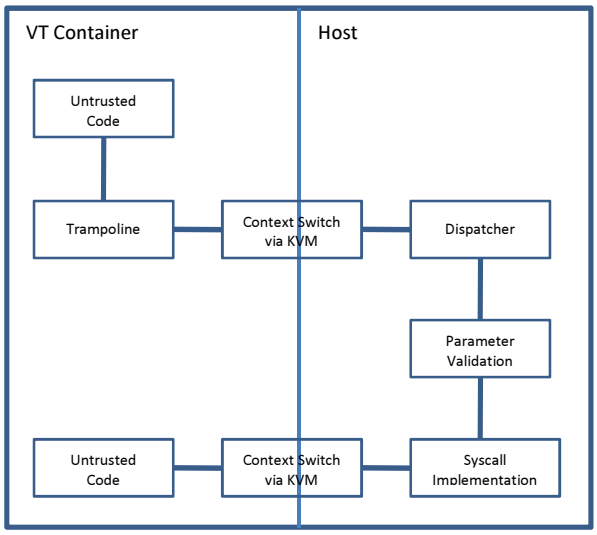

Figure 5: Execution sequence for a system call

5) Handling of unsafe instructions

Potentially unsafe machine instructions are handled by trapping on the execution of sensitive instruction types. The process is described below.

1. The execution of instructions defined as sensitive causes the virtualization hardware to trigger an exit into the VT Container. By default, all privileged ring 0 instructions are trapped. All other instructions are allowed to execute with no constraints within the virtual machine. The VMCB is configured to trap on these sensitive instructions through the KVM layer.
2. If an attempt to execute a sensitive instruction is detected, our trap handlers are invoked. The trap handler then takes steps to terminate the offending component.

We also use this trapping functionality for our implementation of system call handling. However, in that instance, we execute the system call on the component's behalf and return the results via the virtual machine's stack.

6) Threading

We provide an extremely simplified implementation of threads, with one virtual processor per thread.

1. When a component requests the creation of a new thread, we create a new virtual machine, but map in the same memory belonging to the creator's thread. In other words, both virtual machines share the same physical memory.
2. Once the virtual machine is initialized, the virtual processor's instruction pointer is set to the thread's entry point.

3. The GS register must also be set to point to the Thread Control Block of the new thread.
4. The virtual machine execution is then started, thereby having two virtual processors executing the two different threads.

V. Comparative Analysis

This section directly compares our solution to the Google Native Client, which focuses on using static analysis, as well as Vx32, which emphasizes runtime binary translation.

Table 1: Comparison of steps to load and execute a component

VT Container	Google Native Client	Vx32
1. Load component 2. Switch to sandbox 3. Execute code 4. Trap on exception or execute till end	1. Load component 2. Verify component 3. Patch unsafe instructions 4. Switch to sandbox 5. Execute code 6. Trap on exception or execute till end	1. Load component 2. Create sandbox 3. Translate code fragment 4. Execute code fragment 5. Repeat steps 3 and 4 till execution ends or an exception occurs

Table 1 provides an overview of steps needed to load and execute a component. As can be seen, our approach saves significantly on load-time complexity by removing the code verification and patching steps altogether. Our approach thus eliminates an entire class of problems related to static analysis and code verification, as discussed below.

Our implementation saves significantly on execution overheads since no additional instructions need to be inserted. Our initial measurements show that code bloat for Google Native Client is significantly high, because it requires that jumps be aligned to 32-byte boundaries. Neither our implementation nor Vx32 have such an alignment requirement, so can significantly lower the size of the executable code. However, we suffer a heavier penalty for sensitive instructions, as the trap handler may need to perform a context switch back to user space in order to handle the instruction. However, the complexity of our implementation is greatly reduced as no binary code patching needs to be done.

As shown in Table 2 our technique also differs fundamentally at a conceptual level. Google's NaCl relies on pre-execution checking, using static analysis, and rejects a component if it does not meet its defined criteria. It also inserts run-time checks into the code. Our technique simply starts executing the component and aborts if unsafe instructions are encountered. Therefore, our focus is on run-time checking as opposed to both load-time and run-time checking. We argue that our technique is both faster, less complicated and more robust. In particular, our approach will execute components which contain potentially unsafe instructions only in dead code, whereas NaCl will not execute such components at all.

Notably, our implementation is immune to static analysis and verification bugs. In contrast, a security contest conducted by Google to test for loopholes in NaCl revealed several bugs in the code verifier and patching system, which allowed for arbitrary code execution vulnerabilities, enabling a malicious component to escape component isolation [35]. Our implementation is not vulnerable to such errors, since execution is entirely constrained to the virtual machine, making for a more secure implementation. We have tested this by executing similar classes of bugs reported in the Native Client security contest, and showed that the code is unable to break free from the confines of our container. A detailed example is discussed in Section 0.

Our technique also offers the advantage of being easily adaptable to 64-bit code, something that introduces much greater complexity to NaCl and Vx32 and is currently unsupported, because both NaCl and Vx32 make use of segmentation hardware which is no longer available on Intel's 64-bit architecture [18].

However, we do share similar vulnerabilities as NaCl and Vx32 at the system call layer, since any loopholes at this level can be exploited in an identical way. (In both systems the problem can potentially be avoided by carefully validating parameters before execution of system calls.)

Table 2: Comparison of approaches

	VT Container	Google Native Client	Vx32
Approach	Hardware Virtualization	SFI	SFI
Technique	Minimal virtual machine container	Static Analysis	Runtime binary translation
ISA	32bit/64bit	32bit/64bit	32bit
Specific Hardware features used	Intel VT/AMD SVM	Segmentation	Segmentation
Compile time requirements	None	Customized tool chain with code alignment requirements	None

VI. Effectiveness of the Isolation Architecture

In this section we evaluate the effectiveness of our method by demonstrating its use through examples, while comparing and contrasting the results with Google's Native Client and Vx32. These examples were incorporated into test routines and applied against both systems. We show that our solution provides stronger security guarantees than NaCl and Vx32, while eliminating the complexity of the code analysis and verification process and the need for runtime binary translation.

A. Example 1 – Handling Illegal Instructions

Figure 6 shows a C code fragment containing an illegal assembler instruction. It illustrates an attempt to directly access an I/O port through the `out` instruction. Typically,

user level programs are disallowed from accessing I/O ports directly. This kind of problem could arise both from malicious code or a programming error such as an attempt to divide by zero.

```
#include <stdio.h>

void run_test() {
asm ("movl $32, %%eax; \
      out  %%eax, $0xf1"
      :
      :
      :"%eax"
    );
}

int main(int argc, char* argv[]) {
  run_test();
  return 0;
}
```

Figure 6: Example code containing an unsafe instruction

We can selectively forbid sensitive instructions that should not be executed, and our VT Container adopts the policy of disabling such instructions by default. This is done by configuring the VMCB to intercept the specified instruction, in this case, the **`out`** instruction. The virtualization hardware will then automatically trap when an attempt is made to execute the instruction. We can then mediate and terminate the module gracefully or allow it to continue if the instruction is deemed innocuous.

When compared to NaCl, the protection offered is similar. NaCl would refuse to allow execution of the above component since the verification process would detect the presence of the disallowed instruction statically. Vx32 allows execution of the component but, because it dynamically translates the next 'fragment' of code to be executed, it may abort during runtime if it encounters the illegal instruction during its binary translation process, even when the instruction itself is only executed conditionally. We, however, trap only when an illegal instruction is actually reached, if ever. The advantage of this is better illustrated by the example below.

B. *Example 2 – Reducing False Positives*

In this example we modify the previous program slightly to conditionally execute the illegal instruction by only calling the `run_test` method if condition '`argc < 0`' is true. In practice, `argc` will never be less than 0, meaning that this will be dead code in the running program and the potentially harmful instruction can never be executed. NaCl however, would nevertheless generate a false positive and refuse to allow execution of the program and Vx32 would fail at runtime when it encounters that fragment of code. Our method entirely eliminates this class of false positive altogether.

Although the above example is somewhat contrived, it serves to illustrate that NaCl is always forced to err on the safe side, and disallow a range of instructions which are generally innocuous but potentially unsafe, such as all instructions that modify the x86 segment state, including `lds`, far calls, etc. Vx32 also suffers from similar constraints. Our method does not require such caution, as execution is entirely constrained to the virtual machine, and loading segment registers for example, only affects the virtual processor. This produces far fewer false positives.

C. *Example 3 – Addressing Errors*

The program in Figure 7 highlights an extremely common programming mistake. It makes use of an uninitialized pointer which performs a 'wild store' into memory. When an attempt is made to access memory outside of the boundaries defined by the VMCS, the VT hardware can be configured to trap into our specific error handler.

```
#include <stdio.h>

void run_test() {
  int *test, offset = 1024*1024*10;
  test[offset] = 10;
}

int main(int argc, char* argv[]) {
  run_test();
  return 0;
}
```

Figure 7: Example code with an uninitialized pointer

In comparison to NaCl and Vx32, the protection performance is identical, since both NaCl and Vx32 use x86 segmentation hardware to enforce similar constraints.

D. *Example 4 – Addressing Exploits*

Our last example demonstrates a situation where our technique is safer than NaCl. This example is an actual bug detected and submitted during the Native Client security competition, where several flaws in the verifier were identified [35]. Although the bug has been subsequently patched, it serves to illustrate the potential danger of instructions missed during the verification process, and that eliminating the verification process provides far greater security guarantees as well as flexibility.

The exploit took advantage of a miss in the verifier, where opcode prefixes for 2 byte instructions were not constrained. The code fragment in Figure 8 illustrates the key instructions used in the exploit. It works by pushing the value `0x10001` onto the stack, which points to the middle of the first **`mov`** instruction, which now represents the restricted instruction **`int 3`**.

```
cs:0x10000: mov eax, 0xCCCCCCCC
…
…
cs:0x10080: mov    $0x10001,%ebx
cs:0x10085: push   %ebx
cs:0x10086: xor    %eax,%eax
cs:0x10088: test   %eax,%eax
cs:0x1008a: data16 je 0x7f4f
cs:0x1008f: add    %al,(%eax)
```

Figure 8: Example code with an illegal jump

Normally, such an unaligned jump would be disallowed and detected by the verifier. However, the bug exploits the 16-bit data prefix to truncate the jump target, which the verifier miscomputed. As a result, the code jumps into a **`ret`** instruction in the trampoline code region, which results in a return to the address pushed onto the stack, in this case, the illegal **`int 3`** instruction. In our approach the illegal instruction is detected when it attempts to execute.

While this problem was patched in NaCl soon afterwards, it serves to illustrate the difficulty in writing a fool-proof static verifier. As a result, even legal instructions need to be severely restricted in order to prevent potentially harmful exploits. This same class of problems applies to Vx32, as the binary translation process is vulnerable to similar circumvention. In our method, since all execution occurs within the confines of a virtual machine, code execution can be allowed in an unrestrained fashion, as long as proper checking is done when switching between borders. This border crossing happens only during system calls, making our method far simpler and easier to verify correct. Therefore, the above example executes but is unable to bypass the confines of the virtual machine (barring any actual errors in the hardware implementation).

E. Example 5 – General-purpose Applications

In order to evaluate the technique in a more real-world situation, we create a modified version of the bzip2 compression program with various bugs inserted to test isolation effectiveness. This included illegal instructions within dead code, accidental array bounds violations and other suspicious but harmless code. We ran this modified version under both the Native Client, Vx32 and the VT Container. We found that, while all three effectively prevented malicious code from executing, the pre-emptive approach of the Native Client resulted in increased false positives, even though the actual code did nothing harmful. Vx32's binary translation process triggered false positives only when the code fragment was encountered, although it would abort even if the instruction itself was never executed.

VII. Performance of the Isolation Architecture

To ensure that our solution does not introduce unacceptable overheads, we executed some microbenchmarks of illustrative cases as well as some large scale benchmarks, keeping in mind that our current VT Container implementation is merely a proof-of-concept prototype. In all cases our VT Container solution was compared with Google's Native Client, and certain benchmarks were also run against Vx32. Performance was tested in 3 cases—native execution as a linux executable, execution within Google's NaCl Container and execution within our VTContainer.

A. Microbenchmarks

The microbenchmarks were chosen to test performance under highly specific circumstances. These help to establish the upper and lower bounds that can be expected in best and worst case scenarios respectively.

We performed empirical performance measurements on four main workloads:

1. Execution of a simple loop based calculation.
2. Execution of a "null" system call.
3. Execution of I/O instructions (which require an operating system call and therefore, at least one context switch).

The results show that the overheads of our approach in compute-bound scenarios are comparable to those of NaCl with no significant differences in performance (keeping in mind that our prototype implementation utilizes the NaCl system call layer for NaCl compatible executables).

In the second experiment, a simple transition in and out of the VM was performed in a tight loop, added about 20% overhead in comparison to NaCl's performance.

In the third experiment, the system call execution overhead varied, with 10% being typical with an outlier case of 400%. The difference between the typical case and the outlier demonstrates that attendant circumstances of the environment, such as competition with other system processes, internal kernel buffering etc. can be dominating factors in determining overall overheads.

B. Large-scale Benchmarks

In order to empirically measure how these approaches perform under more realistic workloads, we've tested performance for several scenarios.

1. Execution of three compute-bound graphics performance tests provided with Native Client's test suite:
   a. Earth: a ray-tracing workload, projecting a flat image of the earth onto a spinning globe
   b. Voronoi: a brute force Voronoi tessellation
   c. Life: a cellular automata simulation of Conway's Game of Life
2. Quake
3. The SPEC2006 benchmark suite

In all of the above cases, we disabled VSYNC so that the rendering thread would not be put on hold till the display's vertical refresh had completed.

1) Graphics performance tests

The samples were built with nacl-g++ version 4.2.2 with compiler parameters **`-O3 -mfpmath=sse -msse -fomit-frame-pointer`**. The Linux **`time`** command was used to measure the execution time in all 3 cases.

Both Earth and Voronoi were executed with 4 worker threads for 1000 frames, averaged over 3 runs. Life was run as a single thread for 5000 frames. The results are summarised in Table 3.

Table 3: Compute/graphics performance tests. Times are elapsed time in seconds. Lower is better.

Sample	Linux Executable	Native Client		VT Container	
		Execution Time	Overhead	Execution Time	Overhead
Voronoi	34.13	19.18	–43.8%	19.12	–43.98%
Earth	11.38	11.64	2.28%	12.48	9.66%
Life	14.88	17.47	17%	17.88	20.16%

Somewhat surprisingly, we found that both the Native Client and the VT Container significantly out-performed the native executable in the Voronoi test. However, the results are consistent with those reported by Yee et al. [18].

In the other two instances, the results were as expected, with the native Linux executable having the best performance. The Native Client and the VT Container had fairly similar performance in all 3 cases, with the VT Container having a slight edge on the Voronoi example and a loss in the other two tests.

2) Quake

Quake was executed on Suse Linux 11.3 with kernel mode setting switched off at 1024x768 resolution. Quake was built using **–03** optimization. The version used was sdlquake-1.0.9 from www.libsdl.org. The results are shown in Table 4.

Table 4: Quake performance comparison. Numbers are in frames per second. Higher is better.

Run #	Linux Executable	Native Client	VT Container
1	137.1	123.0	122.1
2	136.9	124.0	121.5
3	136.0	124.1	121.3
Average	136.67	123.7	121.63

While performance differences between nulls were minimal and almost negligible, we found that the native Linux executable performed best overall. The difference between the Native Client and the VT Container were extremely small, with the VT Container incurring a slight overhead of about 1.7%.

3) SPEC2006 results

The performance of our approach was tested primarily by executing the SPEC2006 benchmark suite. We compared our approach against native execution of the binary with no modifications, running the binaries using Google's NaCl implementation, the Vx32 implementation and comparing it with our own approach. Only the C integer benchmarks are supported by the Vx32 runtime at the moment.

The tests were run on two machine configurations. Figure 9 shows the results on a Core-i5 540M processor dual core CPU with 4GB of RAM, running on OpenSuse 11.3 with kernel version 2.6.34.07. The executables were compiled with the **–03** gcc flag in all 3 cases. The vertical axis is the ratio of each test's execution time against a reference execution time provided by SPEC. Higher values are better.

As expected, in all cases, native execution of the unmodified binary provided the best results. In all except one case, the NaCl execution time was slightly better than the VTContainer execution. This was not unanticipated, since the context switch overhead takes a toll on execution times. However, in all cases, the performance of the VT container was extremely competitive, with the overhead being less than 1% in all cases, except for the mcf benchmark, which peaked at 4%. In contrast, Vx32's results were slower, with overheads increased up to 4%.

The tests were rerun on a Core-i7 920 quad core processor with 4GB Ram as shown in Figure 10. The configuration was identical to the previous machine, with both kernel versions and executable compilation flags matching.

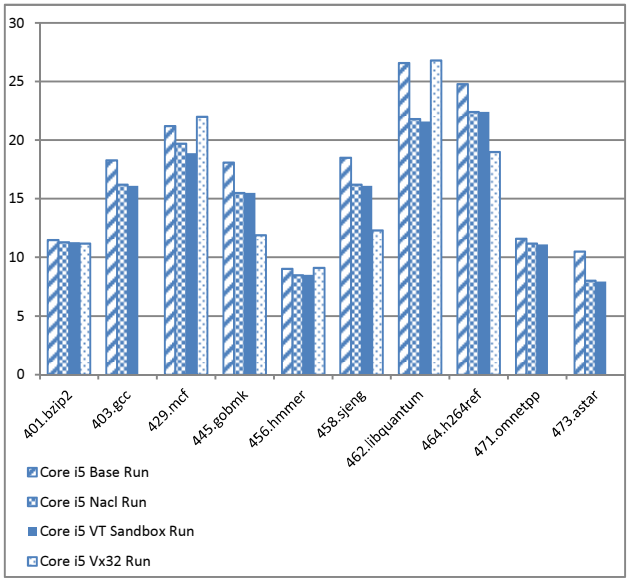

Figure 9: SPEC2006 on Core i5-540M processor

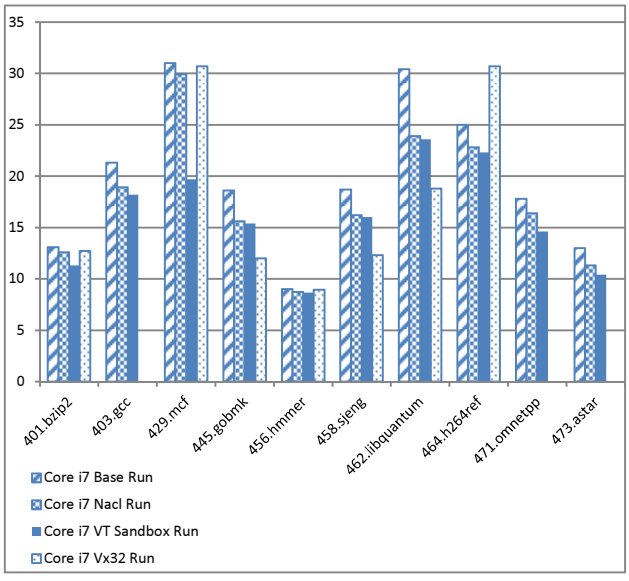

Figure 10: SPEC2006 on Core i7-920 processor

The results were similar, although the differences were more pronounced this time. The overheads ran as high as

10% although the mcf overheads were far more pronounced at 34%. The difference is mainly attributable to cache locality and context switching overheads. However, since this was the only anomolous case, we do not consider it to be representative of average case performance.

Overall, we found that the performance of the NaCl production code, current Vx32 implementation and our initial VT Container prototype were competitive with each other. This is despite the fact that our prototype currently suffers from excessive context switching due to its reliance on the KVM driver. Potentially this overhead could be reduced by moving parts of the code into user space, which would significantly improve the VT Container solution's performance.

VIII. Summary and Conclusions

We have seen that component isolation mechanisms at the operating system level are an increasingly important security need. We have developed a new solution to the problem which uses the virtualization hardware support available in modern processors. By comparison with software-based techniques, as exemplified by the Google Native Client and Vx32, our approach has the following advantages.

1. Elimination of an entire class of problems related to code verification and patching.
2. A significantly smaller Trusted Computing Base and therefore, increased confidence in the safety of the system.
3. Our prototype implementation already provides competitive performance in comparison to NaCl and better performance than Vx32, with the promise of even better performance in an optimized implementation.
4. Since we perform checks at runtime, we minimize false positives which would prevent the execution of valid components.
5. The approach is easily extendable to support 64-bit code.
6. Our approach does not require the use of custom tool chains, and can isolate standard Linux binaries.

The major drawback of our approach is its reliance on hardware specific features, although current trends in microprocessor design suggest that this is not a serious limitation.

We believe that the technique demonstrated in this paper can be further optimized for better performance by reducing unnecessary context switches. In addition, this technique is also applicable to isolating operating system drivers, which have thus far been difficult to handle with other methods due to memory sharing with the OS kernel [3]. While others have explored the isolation of driver isolation using full virtual machines [36], lightweight driver isolation using hardware virtualization support has been described by Tan et al. [37], by using Intel's VT-x extensions. However, their research predated the introduction of Intel EPT and AMD Nested Paging support, and therefore, this also presents opportunities for further performance and isolation improvement, and we intend to explore this avenue in future.

References

[1] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed.: Prentice Hall, 2001.

[2] N. Mendelsohn, "Operating systems for component software environments," in *The Sixth Workshop on Hot Topics in Operating Systems*, 1997, pp. 49-54.

[3] M. M. Swift, *et al.*, "Nooks: an architecture for reliable device drivers," presented at the Proceedings of the 10th ACM SIGOPS European Workshop: Beyond the PC, Saint-Emilion, France, 2002.

[4] A. Zeigler. (2008, 2009, Jan 30). *IE8 and Loosely-Coupled IE (LCIE)* [Online]. Available: http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx

[5] L. Lam and T. Chiueh, "Checking array bound violation using segmentation hardware," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005, pp. 388-397.

[6] Y. Chiba, "Heap Protection for Java Virtual Machines," presented at the Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java, Mannheim, Germany, 2006.

[7] M. M. Swift, *et al.*, "Improving the reliability of commodity operating systems," presented at the Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 2003.

[8] B. Leslie, *et al.*, "User-Level Device Drivers: Achieved Performance " *Journal of Computer Science and Technology*, vol. 20, pp. 654-664, 2005.

[9] J. N. Herder, *et al.*, "Fault isolation for device drivers," in *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 33-42.

[10] A. S. Tanenbaum, *et al.*, "Can we make operating systems reliable and secure?," *Computer*, vol. 39, pp. 44-51, 2006.

[11] A. Barth, *et al.* (2008, 2009 Jan. 30). *The Security Architecture of the Chromium Browser*. Available: http://crypto.stanford.edu/websec/chromium/

[12] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, 2nd ed.: Addison-Wesley, 2002.

[13] C. Small and M. Seltzer, "A comparison of OS extension technologies," in *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, 1996, pp. 41-54.

[14] N. A. Goonasekera, *et al.*, "50 Years of Isolation," in *Proceedings of the 2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, Brisbane, Australia, 2009, pp. 54-60.

[15] R. Wahbe, *et al.*, "Efficient software-based fault isolation," *SIGOPS Operating Systems Review*, vol. 27, pp. 203-216, 1993.
[16] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," presented at the Proceedings of the 15th conference on USENIX Security Symposium, Vancouver, B.C., Canada, 2006.
[17] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 2-13, 2006.
[18] B. Yee, *et al.*, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," *Communications of the ACM*, vol. 53, pp. 91-99, 2010.
[19] C. Reis, *et al.*, "Using Processes to Improve the Reliability of Browser-based Applications," Department of Computer Science and Engineering, University of Washington, Technical Report UW-CSE-2007-12-01, 2007.
[20] The Google Chrome Team. (2008, 2009, Jan 30). *Chromium Developer Documentation: Multi-process Architecture* [Online]. Available: http://dev.chromium.org/developers/design-documents/multi-process-architecture
[21] T. Chiueh, *et al.*, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," presented at the Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, Charleston, South Carolina, United States, 1999.
[22] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in *USENIX Annual Technical Conference*, Boston, MA, 2008, pp. 293–306.
[23] T. Garfinkel, *et al.*, "Flexible OS support and applications for trusted computing," presented at the Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9, Lihue, Hawaii, 2003.
[24] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual* vol. 1: Basic Architecture: Intel Corporation, 2007.
[25] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* vol. 3B: System Programming Guide: Intel Corporation, 2007.
[26] AMD. (2008, 2009 Jan. 30). *AMD-V™ Nested Paging* [Online]. Available: http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf
[27] J. Sugerman, *et al.*, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," presented at the Proceedings of the General Track: 2002 USENIX Annual Technical Conference, 2001.
[28] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," in *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, 2000, p. 10.
[29] P. Barham, *et al.*, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 164-177, 2003.
[30] K. Fraser, *et al.*, "Safe hardware access with the Xen virtual machine monitor," presented at the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure, Boston, MA, 2004.
[31] E. Bugnion, *et al.*, "Disco: running commodity operating systems on scalable multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, pp. 412-447, 1997.
[32] R. Uhlig, *et al.*, "Intel virtualization technology," *Computer*, vol. 38, pp. 48-56, 2005.
[33] Intel. (2008, 25th May, 2011). *Intel® Virtualization Technology* [Online]. Available: http://www.intel.com/technology/virtualization/index.htm
[34] Redhat. (2010, 2010, Jul. 20). *Kernel Based Virtual Machine* [Online]. Available: http://www.linux-kvm.org/page/Main_Page
[35] Google. (2009, 2010 Jul. 20). *Native Client Security Contest* [Online]. Available: http://code.google.com/contests/nativeclient-security/
[36] J. LeVasseur, *et al.*, "Unmodified device driver reuse and improved system dependability via virtual machines," presented at the Proceedings of the 6th Symposium on Operating Systems Design & Implementation, San Francisco, CA, 2004.
[37] L. Tan, *et al.*, "iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support," in *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, 2007, pp. 134-144.

Nuwan Goonasekera: is a PhD Candidate at the Queensland University of Technology. He received his B.Sc. in Information Systems from the Manchester Metropolitan University, UK in 2004. His research interests include software reliability and fault tolerance, security and software engineering. He worked as a senior software engineer at a leading ERP vendor developing enterprise applications.

William Caelli: Professor Caelli is a Senior Research Scientist in the Information Security Institute and an Adjunct Professor in the Faculty of Science and Technology at the Queensland University of Technology. He has had over 47 years research and development, education, consultancy and business involvement in information and communications technology with 37 years of that involved in all aspects of computer and network security, including high trust systems, cryptology and cryptographic systems integration.

Colin Fidge: is a Professor of Computer Science at the Queensland University of Technology where he teaches software engineering. His research interests include modelling and analysis of complex systems, especially those that are deemed security-critical, safety-critical or mission-critical.