Lind: Challenges turning virtual composition into reality

Chris Matthews
Department of Computer Science
University of Victoria
Victoria, Canada
cmatthew@cs.uvic.ca

Justin Cappos
NYU-Poly
Brooklyn, New York
jcappos@poly.edu

Rick McGeer
HP Labs
Palo Alto, California
rick.mcgeer@hp.com

Stephen Neville
Department of Electrical Engineering
University of Victoria
Victoria, Canada
sneville@ece.uvic.ca

Yvonne Coady
Department of Computer Science
University of Victoria
Victoria, Canada
ycoady@cs.uvic.ca

ABSTRACT

Security is a constant sore spot in application development. Applications now need structural support for better isolation and security on a domain specific basis to stave off the multitude of modern security vulnerabilities. Currently, application developers have been relying upon cumbersome workarounds to address these issues. We propose the design and initial implementation details for Lind, a highly flexible composition infrastructure that can be well-integrated with modern application development processes and extends traditional mechanisms like virtualization and software fault isolation in a way that can be tailored according to an application's need. Lind does this by providing the structures and services needed to build a virtual component model. Since compositions of virtual components are different than current software systems, building and using virtual component models provides a new set of software engineering challenges in composition and system construction. As a possible solution to many modern security problems, it is important to understand how virtual component models can be evaluated, to further both the user's understanding of them, and future research in this area. This paper proposes a design and implementation strategy for components that run in isolation. Then the paper presents an evaluation of the efficacy of this approach in terms of performance, isolation, security and composition provides insight into the possible advantages and disadvantages of a virtual component model.

1. INTRODUCTION

At a conceptual level security problems are just a type of software bug. One technique that has proven effective at limiting the scope of both bugs and malware in the operating systems area is *system virtualization.* Virtualization restricts inter-VM communication to specific APIs (like the network), allocates resources to VMs, and provides security and temporal isolation between running programs. Virtualization has provided leverage within the security domain; however, current heavy-weight virtualization is not suitable for use within an application. Some of these isolation advantages can be realized using very fine-grained processes and replacing intra-process procedure calls with inter-process service interactions. However, because of the performance and programmer burdens of doing so, we expect that programmers will continue to write relatively large, unfactored applications. For this reason, an analog to the virtual machine that is suitable for use within a process is desirable.

A *virtual component* or *virtual container* is the intra-process analog to the inter-process virtual machine. Conceptually, this adds guarantees of protection and isolation to the popular *software component architecture* concept, which has such concrete realizations as Java Beans[5] and OSGi[10]. One might reasonably ask why component virtualization is necessary. We note the following:

1. Executing programs are typically assemblages of components (DLLs, SOs, etc) that are written by different programmers, at different times, for different circumstances. Despite this, when they are combined into a program, they all have the same authority.
2. Consumption of resources by a program component is unrestricted, and access to shared state is only partially controlled.
3. A component can and will block the remainder of the program from executing while it is executing; there is no provision for the independent monitoring and control of a sub-process sized component.
4. A collection of components need not be locked to one physical machine, they could be moved dynamically, or be distributed across several machines.

In sum, the intuitive model of a single program is a collection of tightly-coupled routines that execute until the job is complete, then terminate. However, in today's world of persistent services, a program is more likely to run indefinitely. Further, (1) above suggests that such programs are assemblages of mutually opaque, and essentially untrusting, components. This means that a program now looks less like a batch job more like a collection of processes in a timesharing system; however, there is currently no equivalent of an operating system to control this collection of components.

What is needed to construct this is isolation that is simultaneously low overhead and strong. One potential solution to this problem is to isolate separate pieces of code in their own system virtual machines. While this provides strong isolation, it is far too heavyweight for practical use. Similarly, one could extend isolation techniques like software fault isolation (SFI) or object capability systems to have lightweight isolation within a process. However, this would not provide resource isolation or correctly separate out privileges. Our design allows isolation that is simultaneously strong and low overhead.

A critical feature in the functioning of secure, robust, resilient systems is robust, reliable, parallel computation. Unfortunately, reliable parallel processing within a single address space has proven to be a challenging problem. The most common abstraction in use today is threads: multiple independent control threads and call stacks in a single process. While these are quite lightweight, they have proven to be a debugging and security challenge [11, 7].

The essence of the problem is that the behavior of a multithreaded program is no longer deterministic and solely dependent on the program text. Rather, it is dependent on the behavior of the program text *and* the implicit thread scheduler, whose behavior is generally completely unspecified.

These semantics are a veritable bug ranch, and a fertile nursery of security holes, including race conditions such as time of check to time of use (TOCTTOU) bugs. Analysis of such errors indicates that the fundamental problem underlying multiple independent threads of control is *false synchrony*: an implicit guarantee that the state of a remote thread is determined, when in fact it is indeterminate. For example, in a TOCTTOU bug, the fundamental assumption is that the checked variable has not changed value between the time it is checked and the time it is used. In Lind, all nondeterminism is explicit; in particular, there is no implicit synchronization between independent threads of control.

Of course, it is always possible for a sufficiently careful programmer to check the state of threads, lock only shared variables that are required, adopt a locking scheme that avoids deadlocks (a system of hierarchical locks, for example, avoids deadlock). In practice, this is not often enough done. An analogy to typed languages is appropriate here. A sufficiently careful programmer can avoid type errors in an dynamically language, by adopting a discipline that essentially amounts to a program-specific type system. In practice, programmers don't. In a model which imposes isolation boundaries at a component level, inter-component communication becomes a key aspect of the system. Virtual Components provide a structured inter-component communication mechanism with the following properties:

1. Virtual Components are guarded with specific permissions. Access is enforced by the system to explicit typed interfaces.
2. Virtual Components reliably communicate events asynchronously. Asynchronity is a natural model for virtual components, as VMs are run on separate cores in multi-core machines. In the event synchronous communication is needed, it can be built on asynchronous primitives.
3. The system enforces memory isolation between Virtual Components. There is no way for the virtual component itself to break this isolation.
4. Virtual Components share no variables; shared variables would create a form of synchronous communication between virtual components.

This model is intended to encourage the developer to use component level parallelism to produce the system's parallelism.

It will be noted that there are many similarities between Virtual Components in the Lind system and Virtual Machines in any standard virtualization environment such as Xen[1]. In particular, communication between virtual components is explicit; virtual components are independently scheduled and logically asynchronous; there is no possibility of state interdependence between virtual components. An analysis of the problems of multi-threaded programming indicates that the fundamental problem is one of false synchrony: one component presumes knowledge of another's internal state, in general by updating some shared variable. In Lind, the state of a component is not exposed to external components, except by explicit calls, with firm limits on expressed warranties. In particular, shared state is only updated by accessor messages, and these give explicit feedback as to their success, failure, and stateful warranty. This leads to the essence of our model: virtual machines at the granularity of a thread in a programming language. These components must therefore be lightweight. In particular, creation of a virtual component is similar in concept and implementation to object instantiation in a Java-like language. Inter-component communication involves only a few extra function calls and so should be within a small incremental factor of function call performance.

1.1 Inter-Component Programming Model

Though components themselves are not a new abstraction in software development, we need to carefully consider how to construct this programming model in the context of core system infrastructure which previously has been more monolithic and hardwired. The key feature of a component system is the ability to customize and replace prepackaged components without requiring changes to the rest of the system. Psychologically however, this can leave system developers feeling a lack of control. Ultimately, adoption will hinge on our ability to mitigate this reaction through a programming model and tool support that fits workflow practices of modern system developers. Specifically, we need to consider life cycle management, naming, versioning, and lookup services, and include mechanisms for customizing, packaging and deploying components. Here we focus on some of the key issues we must consider in terms of the root of communication costs and complexity of composition strategies.

1.1.1 Communication

Inter-component communication will play an important role in how scalable the system is. If communication overhead is too high, it does not encourage the developers to decompose their systems; but conversely, components are a granularity of parallelism in the system, so a well decomposed system naturally becomes parallelisable. The trade-off the developers face will be to balance latency between components, versus the possible gains multiple components give. In terms of communication latencies, we plan to carefully assess costs we will incur with respect to the management of the Translation Lookaside Buffer (TLB) and cache coherency protocols. Both TLB and CC protocols if not treated correctly can slow a system by orders of magnitude. Early simulation studies revealed the ability to mitigate costs of Translation Lookaside Buffer (TLB) misses through better configuration strategies [4]. More recent work on performance isolation for VMs running on multicore architectures includes mechanisms for tagging the Translation Lookaside Buffer (TLB) entries, partitioning this shared resource to improve performance. Specifically, in [13] a combination of process and VM specific tagging proved promising in terms of performance isolation and Quality of Service (QoS) guarantees for VMs. Another consideration in the context of multicore architectures is the cost of cache misses (hundreds of cycles) when a core uses data that other cores have written [2]. Though the details can vary depending on the cache coherency protocol, this does not just involve reads, but writes as well. For example, when writing a value to a core's local cache, the write cannot be completed until all the other copies are invalidated. Since well informed developers will know best how to build their systems, we will provide them with analysis tools to help them understand how the structure of their system affects its performance in terms of concrete numbers like number of TLB misses, and how cache coherency traffic is impacted. We will offer both synchronous and asynchronous inter–component communication primitives, though we will pick the most reasonable default, which will likely be asynchronous communication. We will provide programming language level mechanisms to help use both modalities effectively.

1.1.2 Composition

In many current models, system composers are faced with attempting to leverage intricate hardwired dependency infrastructures in terms of scripts for configuration. Generally, this problem is known as strong coupling—a property that modern software development practices attempt to minimize. One common way to mitigate some of these problems is by way of dynamic component models. The life cycle events of the OSGi Framework [9] are aimed at producing loose coupling [6]. Specifically, the Life Cycle Layer in the OSGi Framework allows components to listen for installations, updates and uninstallations of other components, either synchronously or asynchronously. In the synchronous case, the updater can evaluate the merit of the update from a security point of view.

OSGi provides a solid reference model for our work in terms of the ways in which it helps components establish communication, regulates what parts of a component's interface are exposed, which versions of components are running in the system, and manages dependencies of the system. The basic container of an OSGi component is a jar file. The jar file contains a manifest which the OSGi class loader reads to find out more about the module including: its name, its version, the modules and version on which it depends, the interfaces which it contributes to the system, and the packages and classes which it uses within the system. A component can also define a service, which is an interface that can be dynamically attached to. One of the keys to OSGi's success is the tool support provided by modern IDEs like Eclipse[9]. Eclipse provides wizards and property sheets to help developers understand the interface between components, and how the eventual system will be constructed. Without this support developers would be left to construct the elaborate XML files for each bundle needs by hand. A task that is possible but undesirable. Recent work has shown that OSGi's life cycle model works well over a network[12]. In that work, they show that they can map the failure of the links or components, to an OSGi uninstallation, which other components can monitor and deal with.

Using OSGi as guidance for a new component model, we wrap sandboxes with component metadata to form *virtual components*. Virtual components are sandboxes with component model metadata which describes them fully in terms of how they interact with the system and other components, which other components and versions they depend on, what state they are in, etc. This metadata is key to structuring and controlling the system.

2. DESIGN OF LIND

Lind is an attempt to provide a new secure lightweight cloud computing environment in the form of a new library operating system which is a concrete implementation of the virtual component idea. The goal of the project is to create a lightweight cloud runtime environment for anything which can be executed on x86 processors, using Native Client[14] (NaCl) and RePy[3]. NaCl is a project from Google which aims to create safe websites written in X86 instead of html and JavaScript, by running the X86 website in a SFI sandbox. RePy is a restricted subset of the Python language, used to implement the Seattle distributed system testbed. The Lind project implements a useful subset of the POSIX API within NaCl to run through RePy. RePy has a more advanced policy mechanism with regards to resource consumption than NaCl, RePy also has rate limiting for file and network I/O, white and black lists for network connections ports etc, memory usage monitoring, and CPU consumption monitoring. The motivation of this NaCl RePy hybrid is to expand NaCl's access to the system to commonly used functionality like sockets and simple file I/O and other NaCl sandboxes, while still providing the necessary spatial and performance isolation[3] as well as portability to make running untrusted applications anywhere possible. We think, when RePy is coupled NaCl's safe execution, this makes a safe and powerful environment for some class applications which have CPU intensive compute requirements, but simple system access needs.

As Lind is an extension to NaCl, Lind is designed to minimize its footprint within NaCl's TCB (trusted code base).

To do that, most of the Lind code runs within NaCl RePy. We have added support to allow RePy programs to launch NaCl instances via a new RePy system call. The `safe_execute` system call allows the application to specify a file from within its working directory to execute. `safe_execute` uses a process similar to NaCl's built-in launcher (`sel_launcher`) to fork a new loader process with the program and arguments. When NaCl starts, it establishes a shared memory connection to communicate with, these channels are opened and then handed over to the RePy program. One the RePy program is running, the programmer is able to query if the NaCl instance is still running, get its communication channels and, if needed, kill the NaCl subprocess.

Using the `safe_execute` mechanism, we built a library OS. The RePy program runs an RPC server which allows for it to service calls from the NaCl instance. With in the NaCl instance, we run a modified glibc where each system call which accesses outsides of the sandbox (file I/O, networking, process control) is redirected to the RePy server. The initial version of Lind will focus on networking and file I/O. For example when a file is opened, the `fopen` system call, is marshaled and sent via RPC to the RePy server. The RePy server opens the file on behalf of the NaCl program (with all the restrictions RePy programs have on their file operations). RePy's restrictions include limiting allowed filenames, blocking directory access, and rate limiting file read and write.

One nice property of the Lind design is that we can pick which system calls we will support with Lind. System calls broadly fall into a few categories. First, system calls which we leave alone. Those are calls like `brk` (memory allocation). Second, some calls are emulated, for example `open` (open a file). Third, some calls will be faked, for example `stat` (check file information). The calls which must be faked calls are those which there will be no direct analog for in RePy. For instance file system permission related calls will be faked because RePy programs do not have access to the global file system, only their own local file system so maintaining permissions in not needed.

One part of Lind is the library OS built with RePy. Though in the early stages of development, we intend to make a component based system, where components satisfy different subsystems requests. The intent is to allow custom setups including transparent in memory file systems, or over network file systems. Finally, we intend to support process migration, so that will be designed into the OS base.

A modified version of NaCl's glibc forwards the target system calls from NaCl to RePy for processing. To do this, code which previously performed a system call now results in the call arguments being marshaled and passed into RePy via RPC. Our current implementation uses an RPC mechanism built on NaCl's IMC communication channels. For example, an `open()` system call is changed into an RPC which passes the path and other information to RePy. The RePy POSIX code will emulate or provide functionality for directories, permissions, and persistent storage. Since RePy does not provide access to directories, directories will be emulated by metadata that is stored in separate files on disk (and cached in memory) and accessed from there. We have started to choose the set of calls to initially support by running common programs and gathering their system call traces. Note that some system calls will not be provided because they are unsafe or cannot be efficiently executed in a portable manner across the diverse OSes that are supported by RePy.

Though simple, building the RPC facility still presents some challenges because programming at the lowest levels of glibc is a complex and arduous task. We have already tackled several challenges related to setting up a fast build environment, effective testing, marshaling atypically sized calls, and a host of similar issues.

3. IMPLEMENTATION STATUS

While we currently have a very early stage prototype that does an RPC between glibc and RePy for three calls, much more work remains. Some preliminary exploration of real programs using strace, a program which monitors which system calls another program makes, has shown that we can safely ignore a large number of system calls. System calls can be ignored for several reasons, but mainly because they have no meaning in our new system. We will further understand which portions of calls can be effectively faked and what needs to be emulated. Our eventual goal is to have a POSIX emulation API which works with nearly all code right after compilation. If it will not work, our implementation will provide the developer a clear error message explaining why the software has failed to operate. Of course, we intend to deploy our software so we will follow standard software QA practices when developing our code.

4. EVALUATING A VIRTUAL COMPONENT MODEL

How do we know we have won? Building a virtual component system involves picking from a large set of trade-offs between performance, isolation and security and composition. The problem with evaluating new systems like this is that all the dimensions are interdependent. There is an inherent fixed cost per communication; however, the more modules the system has, the "better" the other properties appear.

4.1 Characterising and Evaluating Performance

To truly understand the performance of a system like this we have to be able to accurately characterize the costs of the virtual component model. This falls into two categories, first, the cost to components for running in the system (over native execution), second, the inter-component communication cost, a cost which changes based on the structure of the system.

To evaluate the system cost, we rely on the numbers of the underlying system. In the case of Lind, that is the numbers for the overhead of running in NaCl. The overheads associated with running SFI modified native code are well documented in [14], which shows performance overheads (and speedups) in the 0%-10% range, depending on the instruction composition of the program, and the number of system calls it makes.

To evaluate the crossing overhead, we can use custom micro and macro benchmarks like those described in [8] which evaluated the same overheads in popular virtualization. Besides raw timing numbers, running benchmarks like the aforementioned gives us a controllable environment from which TLB and cache coherency traffic can be measured. This should allow us to inductively reason about the characteristics of Lind at scale.

4.2 Characterising and Evaluating Isolation

Sandboxing techniques form the basis for the desirable properties which come from a virtual component model. The techniques themselves have limitations; but furthermore, there is a need to circumvent the sandbox to create a meaningful system, as components must communicate with each other.

To evaluate isolation, we need to verify that the previous isolation systems properties are not lost when we start adding communication to the system. Further, we need to show that by adding communication that we opened no new side channels.

4.3 Characterising and Evaluating Security

We claim that systems utilizing Lind are immune to a broad class of attacks, including privilege isolation attacks and serialization attacks, including but not limited to many types of resource exhaustion attacks, Cross-Site Request Forgery, Cross-Site Scripting, SQL injection, and Cross-Zone Scripting. Moreover, most demonstrations of immunity from privilege escalation and abuse of serialization (essentially, passing misleading types in text data structures) can be tested and demonstrated in non-pathological situations. Malware is, by definition, exploitation of a bug for malicious purposes; if a system is free of a class of bugs exploited by a specific class of malware, it is necessarily immune to those classes of malware. Our initial security work will focus on demonstrating freedom from privilege escalation and serialization errors.

Security comes from a few properties of the system. Small components with good interfaces and no side channels make it hard for one component to intentionally or unintentionally stop another component from fulfilling its task. The component system also provides an opportunity for the system to help enforce the policy of the system.

One way to evaluate the security of the proposed system is to reason about possible attack vectors on it. Another way is to see how it impacts current common security problems, such as those listed by SANS[1] or the OWASP[2].

4.4 Evaluating Composition

The programming environment impacts directly on the ease with which virtual components can be constructed. The goal of the Lind environment is to extend standard tools which programmers are already familiar with, making composition a first class citizen relative to current means of composing heavier weight virtual appliances. By better integrating composition with current agile development practices and tool support, we can provide explicit customizability in ways that currently require intricate configuration management. For example tools similar to those in Eclipse's PDE[9], could dynamically check dependencies, or even construct communication models to describe and later enforce component communication behaviour. Since components share no state, it would be possible for these tools to prepackage entire component graphs, or even setup dynamic update plans. All in a format very similar to how developers build OSGi based application in Eclipse right now.

Current compositions of virtualized subsystems are typically manually configured through less explicit means, making composition more of a side effect than an explicit element of the system. In the most common case composition is hard coded as the system is built with no intent to ever have it changed again, and configuration takes the form of manually editing text configuration files.

5. CONCLUSION

In this paper we presented a new composition technique called virtual components, and our initial implementation of the idea in the form of a prototype system called Lind. Lind is a component model which runs x86 code on a POSIX API within each component, and provides a lightweight communication mechanism. Challenges associated with the evaluation of the efficacy of this approach show direct tradeoffs in terms of performance, isolation, security and composition. Our hope is that by leveraging existing workflow practices and integrating with known tools, the programming experience with compositions of virtual components will be more flexible relative to manual approaches currently in use by heavier weight systems. We see this as a means of potentially supporting more customization according to application specific needs for component isolation and security.

6. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP'03*, pages 1–14, 2003.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[3] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 212–223, New York, NY, USA, 2010. ACM.

[4] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation-based study of tlb performance. In *In Proc. 19th Annual Intl. Symposium on Computer Architecture*, pages 114–123, 1991.

[5] W. Emmerich and N. Kaveh. Component technologies: Java beans, com, corba, rmi, ejb and the corba component model. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002.

[1] `http://cwe.mitre.org/top25/`

[2] `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

[6] P. Kriens. The osgi extender model. `http://www.osgi.org/blog/2007/02/osgi-extender-model.html`.

[7] E. A. Lee. The problem with threads. Technical Report UCB/EECS 2006-1, EECS Department, University of California, Berkeley, January 2006.

[8] C. Matthews, Y. Coady, and S. Neville. Quantifying artifacts of virtualization: A framework for mirco-benchmarks. In *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops*, WAINA '09, pages 1079–1084, Washington, DC, USA, 2009. IEEE Computer Society.

[9] J. McAffer, P. VanderLei, and S. Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison-Wesley Professional, 2010.

[10] OSGi Alliance. OSGi - The Dynamic Module System for Java. Website, 2008. `http://www.osgi.org`.

[11] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation at the Usenix Technical Conference, 1996.

[12] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-osgi: distributed applications through software modularization. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, MIDDLEWARE2007, pages 1–20, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] G. Venkatasubramanian, R. J. Figueiredo, R. Illikkal, and D. Newell. A simulation analysis of shared tlbs with tag based partitioning in multicore virtualized environments. In *Proceedings of the Workshop on Managed Multi-Core Systems (MMCS '09)*, 2009.

[14] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2009.