NanoXen: Better Systems Through Rigorous Containment and Active Modeling

Chris Matthews[*]
cmatthew@cs.uvic.ca

Justin Cappos[†]
justinc@cs.washington.edu

Yvonne Coady[*]
ycoady@cs.uvic.ca

John H. Hartman[‡]
jhh@cs.arizona.edu

Jonathan P Jacky[†]
jon@u.washington.edu

Rick McGeer[§]
rick.mcgeer@hp.com

ABSTRACT

Modern software design has less writing large programs and more orchestrating the actions of prewritten library elements. These elements, generally known as *components*, are stateful software elements which can operate and interact in unexpected ways. Most errors in large systems result from unanticipated behavior from components, or unexpected interaction between components. In this paper, we argue that two principal innovations permit the rapid construction of far more robust and reliable software systems: rigorous containment, to control interactions, and active modeling with dynamic model checking, to rapidly detect unexpected behavior. We outline a small set of requirements which will produce such a system, NanoXen, of *virtual components*, the component analog to virtual machines.

1. INTRODUCTION

Component models like OSGi [1] and CORBA [19, 27] allow developers to build programs from scratch out of compositions of prepackaged elements. These elements can be further customized and replaced without requiring major changes to the rest of the system. Modern component models provide things like: life cycle management, naming, versioning, and interface lookup services [18, 30]. However, these systems have limitations that make it difficult to build verifiably secure systems upon them. Fundamentally this is due to failure to provide different trust domains within a program. For example, in OSGi if multiple components load the same library, there is no clear isolation between their interactions with it and both clients will share the same instance—including mutable static state. These interactions between components mean that an accurate analysis must account for the interactions of both components concurrently.

We propose to address this challenge with two complementary techniques that are used in concert.

1. A **rigorous containment system** based on the rigorous containment of program actions. We propose to enforce module isolation within a programming language which will rigorously isolate sub-components of a large system.
2. An **active modeling system**, where the behavior and interactions between modules is specified by a model. The models are sufficiently simple that an ensemble of them can be checked for appropriate properties, and sufficiently precise that implementations can be dynamically checked for correspondence with the model.

The importance of combining these techniques cannot be overstated. Without rigorous containment, modeling will be inaccurate or will not scale. Without modeling, isolated components could not interact and thus would be useless.

In the remainder of this position paper, we detail the design of our proposed NanoXen system in terms of components and modeling, and overview specific implementation details concerning the language, interpreter, and device interaction.

2. DESIGN

In this section we describe the specific ways in which virtualization allows us to enforce isolation, and modeling allows us to verify interactions. We motivate much of this work from the perspective of building secure systems, demonstrate why virtualization is necessary, and present a design to leverage modeling for analysis, testing, and runtime checking.

2.1 Virtual Components

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 communication between VMs to be through specific APIs (like the file system), allocates resources to VMs, and provides security isolation between running programs.

Virtualization has provided leverage within the security domain; however, current heavy-weight virtualization is not

[*]Department of Computer Science, University of Victoria
[†]Department of Computer Science and Engineering, University of Washington
[‡]Department of Computer Science, University of Arizona
[§]HP Labs

suitable for use within portions of an application. Some of the advantages can of course be obtained by using very fine-grained processes and replacing intra-process procedure calls with interprocess 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 intraprocess analog to the virtual machine is desirable.

A *virtual component* is the intraprocess analog to the interprocess 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 [19] and OSGi [1].

One might reasonably ask at this point why component virtualization is necessary. We note the following:

1. Executing programs are typically assemblages of components (DLLs, SOs, etc) which 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 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 which run to complete a job, and 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 than a collection of processes in a time-sharing system; however,to date, there is no equivalent of an operating system to control this collection of components. The rise of parallel systems will accentuate this need.

A critical feature in the design of secure, robust, resilient systems is robust, reliable, parallel computation. Clock speeds on processors flattened in the early 2000's, and Moore's Law now describes the doubling of *processor cores* on a die at a constant rate. In such an environment, reliable parallel processing is a requirement. 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 [36, 43]. 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 concurrency semantics are a veritable bug ranch, and a fertile nursery of security holes, including race conditions such as TOCTTOU (time-of-check-to-time-of-use) 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 NanoXen, all nondeterminism is explicit; in particular, there is no implicit synchronization between independent threads of control.

1. Virtual Components are guarded with specific permissions, access is enforced by the system to explicit typed interfaces.
2. Virtual Components communicate events asynchronously. Asynchronity is a natural model for virtual components, as VMs are run on separate cores in many core machines. In the event synchronous communication is needed, it can be built on asynchronous primitives.
3. The interpreter 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 creates a form of synchronous communication between virtual components.

It should be noted that there are many similarities between Virtual Components in the NanoXen system and Virtual Machines in any standard virtualization environment such as Xen. 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. This leads to the essence of this project: 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 linear factor of function call performance.

2.1.1 Isolating Virtual Components

Even well modeled software has bugs. As a result, it is essential to minimize the impact of faults until a fix is available. Fortunately, minimizing the impact of a fault can be provided by dividing code into virtual components and then isolating those components from each other. Apart from a well-defined (and validated) interface that is explicitly defined for communication, virtual components are isolated. Informally, isolating components means they cannot interact spatially, meaning a component can not affect another's memory, or temporally, meaning a component slows another

or causes it to block. Virtualization provides these kinds of isolation well.

To provide isolation between virtual components, the API calls that are used by a virtual container are specified by the virtual component that created it. Namely, the ability to send an asynchronous event to a virtual component requires a capability. This capability may be selectively provided, denied, or replaced by the creating virtual component. In a nutshell, this means that system call interposition is a first class mechanism in this design. This capability allows a virtual component to transparently enforce security policies by substituting functionality with the same semantics. For example, the device owner may wish to pose restrictions on the remote endpoint of any network traffic sent by an application. This can trivially be done on a per-virtual component basis by replacing the network capabilities with capabilities that validate the arguments and utilize the original capability only if the remote endpoint is allowed.

Using virtual components as the unit of componentization in a component model leverages some of the most low level principles of modern system design to stop unintended interaction amongst components. But with virtual components, we can leverage active modeling techniques in new ways.

2.2 Modeling

Many model-checking systems provide a modeling language and an analyzer [45]. However, they often rely on specialized modeling languages different from any production programming language, and do not support testing. As a result the analysis is not easily integrated with development. Conversely, many popular unit test frameworks [28] use the production programming language as the testing language, and (partly thanks to this) have been enthusiastically accepted. However, they require the test engineer to code each test case and the oracle (assertions) that check each test case, so they provide no modeling and little test automation.

There are model-based testing systems that use the implementation programming language and provide a good deal of test automation [55]. The most pertinent of these, Spec Explorer [56], NModel [32], and PyModel [48], are distinguished from most others by special emphasis on providing powerful analysis tools, and also by their use of composition (a generalization of the intersection of finite automata) as a versatile technique for combining models, expressing properties to check, and limiting analysis to scenarios of interest [57]. PyModel is further distinguished by using Python, a dynamic language, instead of a statically-typed language as most others do (Spec Explorer and NModel use C#).

Model-based testing has been used in industry, but only *post-hoc*: test engineers were given informal documentation and an implementation to test, and then reverse-engineered the models [20, 26]. In fact, this led to a significant duplication of effort, since developers and programmers incorporated partial models into their executing code – assert statements, input checks, and so on. This both obscured the model and complicated the code, rendering more difficult testing, verification, and programming. Further, the resulting code bloat had deleterious effects on performance, due to increased instruction cache miss rate.

2.2.1 Modeling, analysis, testing, and runtime checking

While we have argued that it is essential for each component to have precise semantics, we have not yet described how we will achieve this. In order to validate the semantics of the system, we apply several techniques that leverage recent advances in model-based testing [10, 20, 26, 32, 42, 48, 51, 55, 56]. The key idea is to construct a model for the semantic behavior of the system in the system's programming language. We can then use this model in several complementary ways, which together provide very strong assurance. Since the model and system are in the same language, the models and techniques are accessible to developers and testers, not just formal methods experts, and implementation is straightforward. The techniques are *modeling*, which captures important system properties by writing a model program, a kind of executable formal specification; *analysis*, which uses model programs to validate and analyze designs, resulting in machine-checked proofs; *testing* which confirms that the implementation conforms to the model by generating, executing, and checking tests; and *runtime checking*, which confirms that behavior during operation does not violate the intended semantics, by performing runtime checks using the same models.

The following paragraphs explain the techniques in greater detail.

Traces represent samples of behavior. A system's behavior can be completely specified by describing all the traces it is allowed to execute (and all the traces it is forbidden to execute). Traces are central in our modeling, analysis, and testing tools; many activities either generate or use traces.

A trace is a sequence of atomic units of behavior called *actions*, where each action has a name and arguments. For example, in a trace of network activity, the actions are messages; the action names are the message types and the arguments are message contents. In a trace of API activity, the actions are API calls and returns; the names of the API methods are the action names and the API arguments are the action arguments. Each API call and return are separate actions (with different names), in order to account for the possibility that a call might not return, and to represent asynchronous actions. We distinguish *controllable* actions that a tester can invoke (such as API calls) from *observable* actions (such as API returns, exceptions, and events). Collections of traces can represent concurrency by interleaving: actions that occur in different orders in different traces can be considered to occur concurrently.

Model programs are executable specifications; each model program describes a (possibly infinite) collection of traces. Model programs are expressed in a particular style: they are collections of state variables and guarded update rules. For each kind of action (each action name) that can appear in a trace, there are two methods in the model program: The *enabling condition* or *guard* is the precondition, a boolean function on state variables and action arguments which is true in states where the action (with those argument values) is allowed to occur. The *update rule* is a procedure that implicitly establishes the postcondition by updating state variables, possibly using the values of its arguments. Model

programs are usually non-deterministic: several, or many, actions are enabled in each state.

In general, model programs are not finite state machines (FSMs); the action arguments and state variables can include numbers and rich data structures such as sets and maps. We use these infinite *contract model programs* as specifications. If arguments and state variables are limited to a finite number of values, a model program is an FSM. We use these finite *scenario machines* to limit analyses and test generation to scenarios of interest and to describe properties to check.

Model programs can be expressed in almost any programming language. It is usually most convenient if the modeling language is the same as the implementation language. This makes modeling and analysis accessible to developers and testers, not just formal methods experts.

Composition is a versatile technique for combining two or more model programs. It is a generalization of the intersection of finite automata. We use composition to build up contract model programs in modular way, and to combine contract model programs with scenario machines for analysis or test generation.

Analysis uses a process called *exploration* that is similar to model checking. Exploration generates a finite state machine (FSM) from a model program by executing enabled actions, starting at an initial state, drawing action arguments from finite domains, backtracking to explore nondeterministic alternatives, until no more alternatives remain, or some specified stopping condition is reached. Every path through the generated FSM represents a possible trace. Properties are checked by examining the FSM; the analyses are conclusive *within those finite domains.*

We can define *safety conditions* or *invariants*: boolean conditions on state variables that are supposed to be true in all states. In *safety analyses*, we search the generated FSM for *unsafe* states where an invariant is violated. We can define *accepting states*: boolean conditions on state variables that are supposed to be true at the end of every trace. In *liveness analyses*, we search for *dead states* that have no path to an accepting state.

We determine whether a model program *accepts* a trace by computing their composition: exploring the model program and the trace in parallel, synchronizing on shared actions and interleaving unshared actions. If exploration reaches the end of the trace, the model program accepts it. This can be generalized to any scenario machine, so we can check any property than can be expressed as an FSM (any temporal logic formula).

Validation is analysis that determines whether a model program exhibits the intended behaviors by checking whether it accepts traces that are known to be allowed or forbidden. Such traces can be constructed *a priori* or collected "in the wild".

Offline testing proceeds in two stages. In the first stage, the framework's *offline test generator* produces a trace from a model program, in a process similar to exploration but with no backtracking. In the second stage, the framework's *tester* or *test runner* causes the implementation to execute each controllable action in the trace (for example, it calls the API), and checks whether the implementation performs each observable action in the trace (it checks the return values). This second stage does not require a model because all the needed information is in the traces. Offline testing is effective where traces are expected to be reproducible (deterministic).

On-the-fly testing is needed in situations where reproducibility is not expected, due to nondeterminism (in the network environment, for example). To perform on-the-fly testing, the test runner does not use a pre-computed trace, instead it uses the model program to generate the trace as the test run executes. The test runner executes the model program during the test run in order to choose controllable actions to execute in the implementation, and also executes the model program to check the results (observable actions) from the implementation. The test runner captures data from observable actions and uses that data in subsequent controllable actions.

Runtime checking confirms that behavior during operation does not violate the intended semantics. It is similar to on-the-fly testing, except it uses an interposition layer in the live system rather than a standalone test runner (it also considers all actions to be observable.) In effect it is similar to checking assertions coded into the implementation, but here the separate model program and interposition layer contain the assertions and do the checking.

Governance uses the modeling layer to dynamically check invariants and temporal properties of both the behavior of virtual components and the environment within which those components operate. This is a natural extension of runtime checking. In this procedure, detection of an anomalous or malformed incoming message results in a dropped message or exception; anomalous behavior of a component results in an exception or other corrective action. This has two central advantages. First, of course, the system becomes self-monitoring and adaptive, and resilient to failures. More subtly, code is simplified. Even a cursory examination of the code of most programs reveals that much of the code is devoted to anomalous behavior, poorly formed inputs, and exception handling. Much of this code is devoted to implicit model enforcement: a check for a null pointer on input, for example, is essentially a dynamic model check. Explicitly factoring this code into a dynamic execution model both simplifies the code of the component, and makes the model of usage of the component far more transparent to the caller's user. To our knowledge, this is the first proposal to make this logical extension of runtime checking.

2.3 Design Diagram

We propose a layered architecture where each layer has precise semantics. The semantics are validated by modeling the API's behavior at each layer. A diagram showing our design can be found in Figure 1. We divide both the computational and device interaction portions into separate layers that have precise semantics. We iteratively build up to a full featured implementation through a series of incremen-

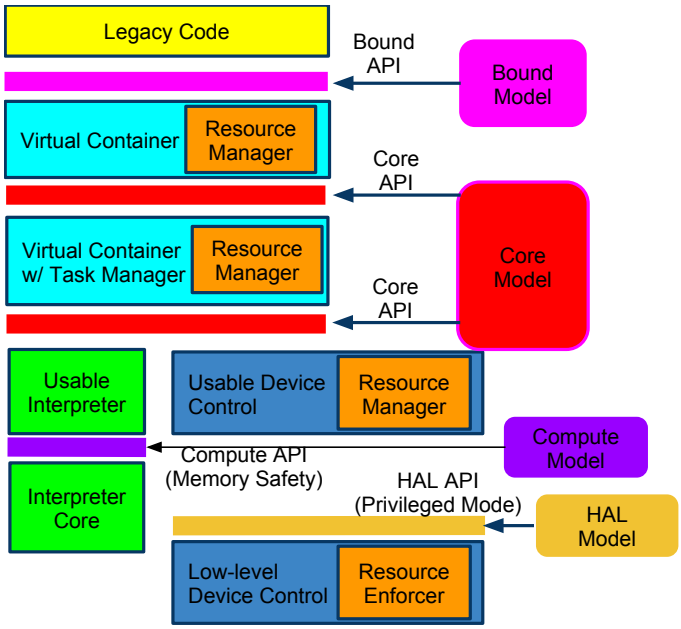

Figure 1: Layers in the NanoXen System

tally more expressive layers. Once we arrive at the core API, all of the virtual components in the system are built using this API (the core API is analogous to the system call API or standard library API in modern OSes).

When instantiating a new virtual component, the parent virtual component maps in functionality that performs the set of API calls for its new child virtual component. The API that is provided can be the parent's core API (which behaves like spawn or fork/exec in modern OSes) or may be modified functions that allow the parent to interpose on the child's API calls. In addition to simply running custom code in response to API calls, the parent may also separately track and control the resources allocated to the child virtual component. To execute legacy code, a virtual component may be bound to a legacy program instance. This allows the legacy code to use a single system call to map a memory request page into the parent virtual component. The legacy program uses this page to communicate with the parent virtual component which will perform API calls on its behalf. All of the APIs and virtual components in the system are tested and semantically validated according to model-based tests.

3. IMPLEMENTATION DETAILS

Building and modeling the entire system with one language, requires a language which is appropriate for all levels of the programming, from the low level run time, to the applications which are running it it. The SeSL language and our device model is our attempt to satisfy that need, and translate well modeled components into a well running system.

3.1 SeSL Language and Interpreter

Existing programming languages have strengths and weaknesses with regard to writing secure systems code. The important features we want from the language are readability of code, strong program semantics, verifiability, and ability to integrate with low-level systems code. We believe the existing language that best fits these goals is Python, and so chose to base the initial language syntax loosely on the Python programming language. The main reason why Python itself is inadequate is mostly due to difficult to understand program semantics, particularly with respect to namespaces and mathematical operations. Python also has a significant amount of introspection built into the language which makes security restrictions difficult. Based on our experiences with building secure Python subsets, we will construct a language with the features of Python that are useful for writing secure systems code, but lacking the semantic complexities and introspection that exist in the full language.

The interpreter is a high-risk area for bugs. Instead of writing the entire interpreter as a single program in a non-memory safe language, the interpreter is constructed iteratively. First in a non-memory safe language we build the *interpreter core*, a container that understands a minimal subset of the language. This subset is enough to allow us to build an interpreter that understands the full language in memory safe code — removing the possibility of buffer overruns, etc. in this code. The language provided by the interpreter core only supports a single namespace, does not support objects or functions, and can compute using basic comparison and arithmetic operators. We use this subset as a building block. This language is kept primitive in order to reduce the amount of memory safe code in the system and to simplify the semantics of the interface.

On top of the API provided by the interpreter core we construct a *usable interpreter* that supports objects, functions, namespaces, exceptions, and other common language functionality. This set of language functionality provides the computational processing that is essential for clean programming practices. Using these abstractions, we construct the rich built-ins that modern languages typically have to make programming easier. These include helper routines that interact with basic types to provide type conversion, string processing, list processing, and similar functions. This set of functions is the bare minimum computational environment that exists in the core API.

One important feature of the usable interpreter is that it supports, encapsulates and validates validates object-capability design patterns. The model of the interpreter validates that capabilities (i.e. function and object references) do not leak across namespace boundaries unless they are explicitly mapped. This mechanism is leveraged to construct the isolation boundaries between virtual components.

3.2 Device Interaction

Operating system semantics differ widely from system to system, even for purportedly portable APIs like POSIX [31]. The result is that programs written for one operating system mostly work on other systems, in particular if the programmer limits themselves to a most commonly used portions of the API. This is also a major problem for programming languages that wish to be portable, such as Java [33]. This has led to developers recoining the Java slogan as "write once,

debug everywhere" because of portability issues [11, 12, 22]. To illustrate the difficulty of the problem, over half of the code in the Seattle testbed's programming language virtual machine solves OS-specific problems [10]. This is despite the fact that all of our supported OSes provide the POSIX API. It is essential that the programmer understand an API's semantics in order to implement vulnerability-free programs.

The device interaction system comprises the portions of the core API that perform I/O and similar tasks. While device interaction occurs much less frequently than the interpreter, the semantics of device interaction tend to be more complex. In most operating systems, device interaction functionality primarily resides in the OS kernel as well as parts of the standard libraries. As with the interpreter, we divide this component into sub-components to mitigate the risk of certain types of bugs.

The *Low-level Device Control* layer acts as a Micro-Hypervisor and interacts directly with the raw devices on the system, much like the boot loader and device drivers in traditional OS hypervisor. The role of this layer is to provide a minimal, semantically consistent interface for controlling devices. All non-memory safe code that is needed for controlling devices exists at this layer. This also contains the entirety of the system code that must execute with escalated processor privileges. This layer includes functionality that is necessary for enforcing control of system resources. That is, the mechanism (but not the policy) that can assign and revoke control of resources to a specific legacy program or virtual component.

The *Usable Device Control* layer is constructed on top of this, written in SeSL and running on the usable interpreter, the usable device control layer calls the low-level device control mechanisms. The usable device control layer is responsible for providing higher-level abstractions for resource access. For instance, the low-level device control layer supports reading and writing disk blocks, while the usable device control layer provides a file abstraction as part of the core API. In addition, the resource manager provides the lowest-level resource allocation policies. This is constructed in a manner that allows secure reasoning about resource guarantees along with separate restrictions for subcomponents of application processes.

4. DISCUSSION

The efficacy of NanoXen rests on two fundamental assumptions:

- The continuing development of software as relatively large executables contained in a single address space, composed of stateful components.
- The ability of software developers to construct intelligible, lightweight, accurate, dynamic models of a component's behavior, and maintain the accuracy of those models as the component changes over time.

Both of these assumptions are of course open to question. An alternate architecture for large software systems is as a collection of collaborating processes communicating through some form of a services-oriented architecture implementation. This relies on the development of very lightweight interprocess communication, and possibly very lightweight VMs. As for the second assumption, programmers historically have embraced tools which make programming easier and software more reliable: memory management, exceptions, and object-oriented programming, to name three examples. Indeed, exceptions can be thought of as a very lightweight form of dynamic model, and typing a weak form of static model. Further, static modeling tools such as UML have found adoption, even with weak ties to implementation.

Over the next two years, we will complete the development of a prototype NanoXen system in SeSL, and demonstrate its efficacy by building a simple multicomponent system in NanoXen. This comprises an implementation study and a design study. In the implementation study, we will determine whether virtual components are efficiently, scalably implementable. In particular, we must demonstrate that the intercomponent communication overhead is tractable and that the pervasive, dynamic model checking adds little overhead.

In the design study, we will attempt to demonstrate that virtual components actually make a programmer's life easier. In particular, we will attempt to show that programs are more reliable, less buggy and that incorporating models makes implementations smaller and simpler.

5. RELATED WORK

Weaknesses in modern "secure" programming language VMs such as Java are well known [7,13,16,35,38,44,49,50,54]. To deal with this, capabilities [37] have been proposed as a way to secure programming languages [40,41]. Object capability languages have proven a principled technique for achieving the long studied problem of dividing an application into security contained components [14, 29, 52].

Besides the customized use of Python in the Seattle testbed [8, 9], Joe-E [40] is the only other object-capability language that uses a subset of a widely used programming language (Java). Current work on Joe-E is entirely reliant on the Java interpreter and hundreds of thousands of lines of standard library code, a major limitation they acknowledge [40]. In this work, we have a layered design where the maximum amount of functionality possible is pushed out into virtual components. This should help to minimize our Trusted Computing Base (TCB). Work by Stiegler and Miller [53] on a capability subset of OCaml demonstrated that object-capabilities do not have to impact the language's expressivity or performance.

Our design draws inspiration from a significant amount of historical work on layering [17,34] and object-capability based operating systems [37]. More recently, failure isolation techniques have been applied to many different domains including operating system separation of processes [3], virtual machine separation of operating systems [5, 24], constraining the functionality of a process [39], or running mutually distrustful programs within a single process [4, 15, 58]. In this work, we focus holistically on the problem and co-design both isolation and separation of code to promote security concurrently.

As a way of adding security to processes in legacy operating systems, some researchers have also proposed to interpose on a process, usually at the system call layer [2, 6, 21, 25, 46, 47]. However, in legacy systems system call interposition mechanisms are prone to subtle errors [23, 59] which has led most practical systems to abandon them. This problem is due mostly to the difficulty of getting the semantics for such an interface correct — a problem that we can avoid through model-based testing.

6. CONCLUSION

In this paper we explore the construction of large programs using a virtual component system. Our virtual component design uses an active modeling system to evaluate the behavior of rigorously isolated program components. We believe the combination of small, self-contained components and rigorous modeling will allow us to quickly and reliably detect errors due to unanticipated component behavior.

While our work is still in early stages, we are working to apply this to a complete software stack from the hardware to the user to test its effectiveness. By employing models in between system components, we hope to retain efficiency, while increasing system security and robustness. We believe that this work will lead to the development of trustworthy and secure computational devices.

7. REFERENCES

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

[2] A. Acharya and M. Raje. Mapbox: using parameterized behavior classes to confine untrusted applications. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2000. USENIX Association.

[3] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, page 10. ACM, 2006.

[4] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in Java. In *OSDI'00*, pages 23–23, Berkeley, CA, USA, 2000. USENIX Association.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP'03*, pages 164–177, New York, NY, USA, 2003. ACM.

[6] A. Berman, V. Bourassa, and E. Selberg. Tron: Process-specific file protection for the unix operating system. In *In Proceedings of the USENIX 1995 Technical Conference*, pages 165–175, 1995.

[7] A security vulnerability in the Java Runtime Environment (JRE) related to deserializing calendar objects may allow privileges to be escalated. `http://sunsolve.sun.com/search/document.do?assetkey=1-26-244991-1`. Accessed April 8, 2010.

[8] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: a platform for educational cloud computing. *SIGCSE Bull.*, 41(1):111–115, 2009.

[9] 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 *To appear in the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, 2010.

[10] J. Cappos and J. Jacky. Model-based testing without a model: assessing portability in the Seattle testbed. Submitted to SSV10, System Software Verification 2010.

[11] P. Chanezon. Write Once, Run Anywhere: the devil is in the details, 2006. `http://wordpress.chanezon.com/?p=7`.

[12] B. Charny. Write once, run anywhere not working for phones, 2005. `http://mcall.com.com/Write-once,-run-anywhere-not-working-for-phones/2100-1037_3-5788766.html`.

[13] Cve-2003-0896. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0896`, Accessed April 8, 2010.

[14] G. Czajkowski. Application isolation in the Java Virtual Machine. In *OOPSLA'00*, pages 354–366, New York, NY, USA, 2000. ACM.

[15] G. Czajkowski and L. Daynés. Multitasking without comprimise: a virtual machine evolution. In *OOPSLA'01*, pages 125–138, New York, NY, USA, 2001. ACM.

[16] D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy.*, pages 190–200, 1996.

[17] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[18] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Ltd, 2000.

[19] 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.

[20] J. P. Ernits, R. Roo, J. Jacky, and M. Veanes. Model-based testing of web applications using NModel. In M. Núñez, P. Baker, and M. G. Merayo, editors, *TestCom/FATES*, volume 5826 of *Lecture Notes in Computer Science*, pages 211–216. Springer, 2009.

[21] T. Fraser, L. Badger, and M. Feldman. Hardening cots software with generic software wrappers. *Foundations of Intrusion Tolerant Systems*, 0:399–413, 2003.

[22] J. Fruhlinger. LWUIT: Write once, run anywhere (mobile) (hopefully), 2008. `http://www.javaworld.com/community/node/1113`.

[23] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS'03*. Citeseer, 2003.

[24] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP'03*, pages 193–206, New York, NY, USA, 2003. ACM.

[25] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *SSYM'96:*

Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography, Berkeley, CA, USA, 1996. USENIX Association.

[26] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F. L. Wurden. Model-based quality assurance of Windows protocol documentation. In *ICST*, pages 502–506. IEEE Computer Society, 2008.

[27] O. M. Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006.

[28] P. Hamill. *Unit Test Frameworks*. O'Reilly, 2004.

[29] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *USENIX ATC'98*, pages 22–22, Berkeley, CA, USA, 1998. USENIX Association.

[30] G. Heineman and W. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Publishing Co. Inc., 2001.

[31] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[32] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

[33] Learn about Java technology. `http://www.java.com/en/about/`, Accessed April 8, 2010.

[34] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.

[35] S. Koivu. Calendar bug. `http://slightlyrandombrokenthoughts.blogspot.com/2008/12/calendar-bug.html`. Accessed April 8, 2010.

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

[37] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[38] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *CCS'09*, pages 442–452, New York, NY, USA, 2009. ACM.

[39] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX ATC'01*, pages 29–40, 2001.

[40] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium*. Internet Society, 2010.

[41] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[42] NModel software. `http://nmodel.codeplex.com/`.

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

[44] N. Paul and D. Evans. Comparing Java and .NET security: Lessons learned and missed. *Computers and Security*, pages 338–350. Volume 25, Issue 5, July 2006.

[45] D. Peled. *Software Reliability Methods*. Springer, 2001.

[46] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, page 10. Washington, DC, 2003.

[47] PTrace. `http://en.wikipedia.org/wiki/Ptrace`. Accessed April 2, 2010.

[48] PyModel: Model-based testing in Python. `http://staff.washington.edu/jon/pymodel/www/`.

[49] Fujitsu Java Runtime Environment reflection API vulnerability. `http://jvndb.jvn.jp/en/contents/2005/JVNDB-2005-000705.html`, Accessed April 8, 2010.

[50] Sun Java Runtime Environment reflection API privilege elevation vulnerabilities. `http://www.kb.cert.org/vuls/id/974188`, Accessed April 8, 2010.

[51] D. Rosenzweig, D. Runje, and W. Schulte. Model-based testing of cryptographic protocols. In R. D. Nicola and D. Sangiorgi, editors, *TGC*, volume 3705 of *Lecture Notes in Computer Science*, pages 33–60. Springer, 2005.

[52] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys'06*, pages 161–174, New York, NY, USA, 2006. ACM.

[53] M. Stiegler and M. Miller. How Emily tamed the Caml. Technical Report HPL-2006-116, Advanced Architecture Program. HP Laboratories Palo Alto, 2006.

[54] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *Proceedings of the USENIX Security Symposium*, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[55] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[56] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.

[57] M. Veanes, C. Campbell, and W. Schulte. Composition of model programs. In J. Derrick and J. Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2007.

[58] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *SOSP'94*, page 216. ACM, 1994.

[59] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT'07*, pages 1–8, Berkeley, CA, USA, 2007. USENIX Association.