End-host VM Techniques Design Document

This document details the design of the end host VM for the Million Node GENI project (proposal 1645, NSF Grant CNS-0834243). We propose to provide an end-host VM by using a programming language virtual machine environment. The VM we provide has the primary goals of being widely deployable and program safety (resource consumption controlled and isolated in the operations it can perform). We feel that a programming language virtual machine environment has the best chance of being widely deployed because 1) it requires no privileged access to the user's machine, 2) it requires no modification to the user's existing software, and 3) it is very lightweight compared to other VM techniques (like OS-VMs). However, even traditional programming language VMs have a great deal of bloat and include a bloated programming interface which makes them too heavyweight and makes safety guarantees difficult to enforce because of the broad interface. The broad interface also complicates safety goals like restricting a program from monopolizing computational resources.

In this work, we provide a lightweight programming language virtual machine with a narrow interface. The narrow interface of VM API calls makes the VM easy to port to new environments which allows us to meet the goal of being widely deployable. The narrow interface also simplifies program safety by having a smaller number of calls that can consume resources. Our programming language virtual machine is called repy (REstricted PYthon). Repy supports a large subset of the python programming language including most python primitives and built-ins. Repy runs on top of the python interpreter and provides security by a few separate mechanisms. First, repy hooks into the python parser and reads the researcher's program's parse tree to verify that the program code is safe to execute. It only allows actions that it can easily verify are safe to be executed by the user's code. Second, repy provides the narrow interface to "potentially unsafe actions" (described in more detail in the companion document). Third, repy monitors the resource use of the program to provide resource isolation. This ensures that resource consumption does not impact the user's system. Each mechanism is now described in detail.

Program Verification

To verify the researcher's program, I used and extended some well known code (svn://www.imitationpickles.org/pysafe/trunk) that hooks into the python compiler module. The "sandboxing" program walks down the parse tree of the researcher's code and ensures that the nodes are safe. For example, it is unsafe for programs to use many dictionaries or functions that begin with "__", the sandbox ensures that the program doesn't have a token that begins with "__" that isn't obviously safe (like __init__).

Many of the actions the researcher's program would like to perform are not inherently safe or unsafe and are difficult to decide. For example, importing a module is safe if the imported code is correctly checked and determined to be safe and a number of subtle name space issues are resolved. However, it's clear that importing C code and many similar operations are unsafe. Our design decision is to err on the side of caution so that we do not compromise the user's computer. We categorically disallow calls that cannot be verified as obviously safe.

The resulting language is powerful enough to write complex applications but is missing support for many of the python standard libraries.   Existing code can be ported to the environment by substituting the unsafe calls with safe ones or removing the functionality that uses the unsafe call.

Executing Unsafe Operations

Repy provides an interface where the program can execute unsafe operations (such as open files, send network traffic, etc.).   Repy verifies that the program is performing a safe operation (like opening a file in the researcher's program's sandbox directory) instead of a malicious action (like opening the user's credit card information).   This is done by fine grained checks on the arguments passed to individual calls.   Once again we err on the side of caution (for example, restricting the names of files the program can open) to prevent the program from escaping the sandbox.

However, not every safe action should be allowed by every researcher's program.   It may be that two researcher's programs would like to run on the same system and each program has been allocated its own port.   The researcher's programs should not be able to use the other program's port or other ports on the system.   To prevent this we have per-program fine-grained restrictions that control the use of calls.   Program A can be restricted to a single port while Program B is restricted to a different port.   Similarly, a program C that does not need to write files to the disk can be prevented from using file operations all together!   We believe that in order to allow different non-cooperative programs to share a common computer, the sandbox must provide protection from other programs.

Resource Limits

Another important type of isolation is resource isolation.   One program that runs on a computer should not be able to significantly impact the execution of other programs or the user's ability to use their computer.   To this end we propose to limit the resource consumption through several mechanisms.   For resources that are renewable (like CPU, network send rate, file write rate, etc.) the program is temporarily "paused" if it tries to over use the resource so that the performance of the system does not suffer.   If the resource is not renewable (like the number of open files, memory use, disk used, etc.) then either the function raises an exception or the program is killed.   This prevents the program from negatively impacting the performance of other programs running on the same system.

Resource use detection is performed through different mechanisms for different resource types.   The narrow API we provide is the only access the user has to disk, network bandwidth, etc. and as a result makes it trivial to do accounting and detection of resource use.   Other resources like CPU and memory are harder to detect directly since the user program does not make API calls to consume these resources.   To restrict the use of these resources, there is a monitor process that examines the program's resource consumption and pauses or kills the program as appropriate.

The Big Picture

While useful in itself, repy is part of a larger ecosystem.   Repy provides a lightweight programming language VM that has restrictions on resource consumption and security isolation.   It does not address how that program is started, which programs are run, and who has permission to run a program.   These issues will be addressed in the design document for the node manager.