1 Project Proposal

1.1 Title

Platform Independent Programs—Steganographic Execution

1.2 Advisor

Dr. Dan Page

1.3 Motivation

Until recently it was believed that having a program that ran on multiple architectures was limited to virtual machines, and a few hand crafted examples of bytecode that would execute on multiple platforms. In 2010, however, a team of researchers from Carnegie Mellon University succeeded in developing a method for creating multi-architecture programs automatically for the x86, ARM and MIPS platforms, as well as between the Darwin, FreeBSD and Linux operating systems.

At the moment there are no publicly available tools to create the platform independent programs, and there has been little work on detecting a PIP from a program for a single architecture. There haven't been any attempts to extend the Carnegie Mellon teams approach to other architectures (such as XMOS's XS1 or the JVM), and there aren't any databases detailing what the gadget headers (which are the key to developing platform independent execution) are between platforms.

There are many applications of platform independent execution. It can be used as a keyed steganographic system for resisting attempts to steal software. Alternatively it can be used to smuggle programs past a warden in the classic *Prisoners' Problem*. It can be used to create generic shellcode that will run on multiple architectures, and it can be used to create viruses that target specific computers on a network similar to the Stuxnet worm. They also offer an alternative to the *Universal Binary* or *FatELF* formats for executables, and can be integrated with the pre-existing *COFF* executable format with no changes.

1.4 Objectives

- Implement the existing PIP-shellcode generation algorithm using a better-than-brute-force method to find the bytecode patterns that define a gadget header.
- Attempt to implement the single instruction PIP generation algorithm, by extending the PIP-shellcode algorithm from the whole program level to smaller subsections of code.
- Extend the PIP generation algorithm to a new architecture, by finding the intersection of semantic-NOPs and jumps between different instruction sets and using HPC resources.
- Investigate methods to detect PIP from non-PIP by analyzing instruction density, searching for possible gadget headers under certain architectures, and using static analysis techniques to analyze code coverage within an executable.

1.5 Plan

- Study the architecture manuals for any relevant architectures. Find all semantic-NOPs and semantic-JUMPs in each architecture and build a templating system to generate gadget headers. Run templating system to generate a database of gadget headers. (A long time)

- Implement the RG-generation algorithm for PIPs at the whole program level, and then the single instruction level. (Hopefully less time)
- Use static analysis to find distinguishing characteristics of PIPs and develop an oracle for distinguishing PIP from non-PIP with a success rate better than guessing. (Depends… Probably a fair amount of time).