JOSEPH HALLETT

PLATFORM INDEPENDANT PROGRAMS

Contents

Introduction

In 2010 a team of researchers developed a generalized method for creating platform independent programs (PIPs)[4]. A PIP is a special sort of program which can be run on multiple different computer architectures without modification[1]. Unlike shell scripts or programs written for a portable interpreter a PIP doesn't require another program to run or compile it; rather it runs as a native program on multiple architectures with potentially different behaviour on each.

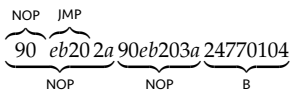
To find a PIP you would have to analyze the architecture manuals for each architecture you wanted and find the instructions in each which compiled to identical patterns of bytecode and use them to construct your PIP. The approach taken by the authors was to find small PIPs with a very specific form: do nothing then jump[2]. By ensuring each architecture jumped to a different point and that each architecture didn't accidentally run into a region anouther architecture jumped into they could construct PIPs for any arbitrary program by splitting them up into blocks of instructions specific to each architecture and connecting them with the small PIPs.

They go on to give in the paper a generalized algorithm for constructing these PIPs, and say that they have a working implementation of it for creating PIPs for the x86, ARM, and MIPS platforms, as well as the Windows, Mac, and Linux operating systems.

[1] For a more formal definition a PIP is a string of bytecode b such that for different machines m_1 and m_2, b is a valid program if:

$$m_1(b) \neq \bot \wedge m_2(b) \neq \bot$$

[2] Consider the following example (taken from the original paper). The dissassembly for the x86 architecture is shown above, and for the MIPS platform bellow.



The string is valid on both platforms and has similar behaviour on both despite jumping to different locations. It is a valid PIP for the x86 and MIPS platforms.

Aim of the Project

For this thesis I have implemented a small section of the PIP generation algorithm—finding the *gadget headers*; the PIPs that link the specific code sections together. To generate the PIPs a list of *semantic NOPS*[3] and potential branch instructions has been found for each architecture in the original paper and to extend the work of the original paper I have analyzed two new platforms: the Java Virtual Machine, and XMOS's XS1.

Unfortunately there does not seem to be a public database of these instructions available for any architectures. Semantic NOPs have been used in areas other than creating PIPs, for example malware classification[2, **(author?)** [15]], but there still appears to be no exhaustive list exists documenting them. Part of the work required is to create one.

[3] A semantic NOP is an instruction which has no effect, but which might not necessarily be the *NOP* assembly instruction. For example the ARM instruction: MOV r4, r4 Causes the contents of register four to be moved into register four and as such is equivalent to an actual NOP instruction. Equally the sequence of instructions: PUSH r3 POP r3 If equivalent to two NOP instructions when taken as a whole and so is a *multi-instruction semantic NOP*.

Why is this interesting?

PIPs can be used for a variety of applications. One potential application suggested by Brumley et. al.[4] is for sneaking programs past dynamic

execution checkers. Suppose two prisoners *Alice* & *Bob*, wish to send a malicious program between themselves. To send the message they have to send it through a *Warden* who checks first that there communications don't contain anything malicious and only delivers the message if its believed to be harmless.

To sneak the program Alice and Bob use a form of *keyed-steganography*[4]. The program which they wish to communicate becomes their *steg-text*, and they construct a *cover-text* by writing some other program which doesn't contain anything malicious. They either pre-arrange a shared secret which is the platform that their programs should really be run on: this forms the key. With the cover-text and steg-text created they create their message by generating them into a PIP where on one platform (x86 say) it appears to run the innocuous program and on the secret key platform (ARM for the sake of argument) it runs the program they really wish to communicate. Alice hands the warden the program and tells him that it is for the innocuous architecture[5]. The warden runs the program and sees it isn't something he would want to restrict and delivers it. In fact unless he is aware that it has been in constructed in this way he may not even check any other architectures as for most platforms it will appear to be garbage just like any normal executable.

Another application is *exfiltration protection*[6]. Here the idea is that to protect its software from theft a secret agency could make a modification to an existing platform (the JVM or another virtual machine would be a good choice here) and compile their program for this modified platform. They then create another program for the unmodified platform which does something else; maybe it phones home, maybe it destroys itself from the computer it's running on. They create a PIP out of these two programs and now if the program is stolen and the exfiltrator isn't aware of the PIP nature (or exactly what modifications were made to the architecture) they're not going to be able to run the program they removed.

Microcode offers another neat way to use PIPs. Suppose an attacker manages to compromise a system in such a way that they can alter the microcode of the processor, such as the recent HP printer attack amongst others[9, **(author?)** [16]]. Now suppose that as well as the microcode update they also modify an existing program[7] so that on the compromised system it gives a backdoor or acts maliciously, but on another (say one which is trying to forensically work out what is wrong with the printer) it acts normally. Brumley et. al. go on to point out[4] that if this was done by Intel and the PIP was a preexisting and digitally signed application then it is a particularly scary prospect. Merely signing the program would be insufficient protect a user it would not check if the machine it was executing on had been modified.

PIPs can also be included in shellcode and viruses. For shellcode the idea is that you can write it once and use it anywhere. For viruses the idea is that if you could get the virus on a disk that is mounted on multiple architectures (say an NTFS share or USB key) then you can attack any platform you're plugged into.

Another application for PIPs is to create actual platform independant programs. The idea here is compile a program for multiple architectures

[4] which the authors[4] call *execution-based steganography*.

[5] if they were using ELF they wouldn't even need to do that—it's part of the header in the file[1].

[6] Exfiltration is military term meaning the removal of a resource from enemy control. In the context of PIPs were talking about taking programs from protected PCs; kind of like DRM.

[7] In the PIP paper[4] they suggest `ls`.

and create a PIP out of them. You'd get a program that behaved the same but ran on multiple architectures. This could be useful, for example, if you have a network of computers (some Linux x86 based, some ARM based) and you want to run a server hosting all the programs to share between them you don't have to maintain multiple versions.

The problem is that although PIPs could be used to write architecture independant programs there are more elegant solutions available than relying on the intersection of instruction sets between architectures. There are a couple of preexisting systems for doing this such as Apple's *Universal Binary* or the *FatELF*[11] format. Another problem is that for some operating systems this just wouldn't work: Linux normally uses the ELF format[1] which has a flag in the header which specifies exactly what architecture the binary was compiled for. If it doesn't match the architecture of the machine it's being run on the loader refuses to run it[8].

Collberg et. al. in their paper *A Taxononmy of Obfuscating Transformations*[8] describe different methods for hiding the structure of a program. They give many different transforms but three are of particualar interest: adding dead code, adding redundant transforms, and outlining[9] sections of code. These three are of interest because they describe what a PIP is doing, namely adding redundant NOPs and transforms which don't alter the state of the program before jumping to the actual code.

Whilst adding the NOP instructions isn't a particularly *resilient*[10] transformation (a program could replace or remove them) they are potent[11] especially if they're combined with multi-instruction semantic NOPs where the state of the program does change only to be reversed later. The jumps added by the PIPs act to outline blocks of code. If your using just one PIP at the start of the program then it isn't that obfuscating but in a situation where you're outlining every single instruction with a PIP like structure and possibly embedding different behaviour if it is run on a different architecture (such as Java or Thumb mode on an ARM chip) this has the potential to be massively obfuscating.

Interestingly papers, such as [6, **(author?)** [5]], even describe obfuscation techniques where they unobfuscate the adding of semantic NOPs using a novel (and patented [7]) tool called *Hammock*. Hammock appears to be interesting because rather than using a catalogue of pre-known semantic-nops it finds them by analyzing whether sequences of code have any net effect on the state of the machine. They find it to be a very slow transform to de-obfuscate (implying adding NOPs is a potent obfuscation technique) but the removal is quick once they have been found.

Semantic-nops are another interesting aspect of the PIP problem. Semantic-nops are important for PIPs as they give you multiple ways of doing nothing—so there is a greater chance of finding an overlap between different architectures but they turn up in other places too. Many people [6, **(author?)** [14], **(author?)** [3],] have suggested using semantic-nops as an obfuscating technique. Wartell et al suggest using them as part of a heuristic for diffenentiating between code and data for disassembled programs[17]. The GNU Assembler has a short list of efficient, low power semantic-nop[12] instructions it uses to pad instruction sequences to cache-lines[10].

[8] Of course there is nothing to stop you flipping the flag to some other value with elfedit utility from the GNU Binutils.

[9] Outlining is the opposite of inlining. For inlining we take a function call and replace it with the functions code inserted into the main program verbatim. For outlining we take a block of code inside a function and make a function call out of it. We might do inlining to skim a couple of jump instructions from our program at the expense of a longer program; but outlining (especially of sections only run once) just adds to the spaghetti nature of the code.

[10] Resilience is a measure of how easy it is to de-obfuscate a transform. It is usually measured in terms of the time and space the deobfuscator has to run.

[11] Potency measures how confusing for a human the transform is. For example self-modifying code is a very potent transform, but renaming the jump targets isn't.

[12] A comment above the function[10] notes that most of the instructions used as part of the semantic-nop sequences by the assembler arn't infact assemblable with the assembler.

What is the Challenge?

The original PIP paper[4] contains an anecdote where the effort required to create platform independant programs is described as requiring:

> "a large, flat space to spread out the architecture reference manuals, and an ample supply of caffeine. Do not underrate the second part."

Brumley et al go on to note that:

> even the most caffeinated approaches have only been met with limited success;

For this thesis we're not trying to fully generate platform independant programs; rather we're just trying to find the headers that enable them. To do this we need two things: a list of semantic-nop and jump instructions for each architecture we're interested in, and a method for combining them to form the headers.

Finding the semantic-nops and jump instructions in theory is quite easy. You can go through the architecture manual making notes of the all the instructtructions which you're interested in (checking that they don't alter the state of the processor in any surprising way) before assembling them to get the bytecode. For some architectures it is as easy—the instruction sets are small and everything in the instruction set is accessible through a standard assembler.

The MIPS architecture[13] is a good example of a platform which it is easy to find semantic-nops. A short RISC instruction set, a limited number of status-altering instructions and a register that discards any value written to it make it and ideal platform for writing semantic-nops. Several million single instruction semantic-nops can be found with minimal effort.

The Intel x86 architecture[12] is completely different however. There are a large number of instructions here including multiple forms of the same instructions which the assembler is free to pick between. All arithmetic instructions alter status flags. Worse still there are some assembly instructions that can't be assembled by the GNU toolchain[10]. It is considerably harder to find semantic-nops for the x86 architecture.

Once we know the form of the instructions we want to assemble we need to compile and disassemble them to get the bytecode, and store them in a database. This isn't hard. Once we have them in an indexable format we need to search for the patterns that overlap and find all the PIP headers. This is a harder problem. For platforms like AARCH32[@AARCH32] and MIPS[13] instructions are all compiled to be of fixed length (four bytes). In this case finding the PIPs is easy: grep the list of jumps for one architecture with the list of semantic-nops for the other. Intel's x86[12], again, makes things more complex. The x86 platform has variable length instructions[13], however, and this makes the problem slightly more complex. We need strings of them to match up with just one MIPS instruction. We need a better method to find them.

In Brumley et al's paper they use take a brute force approach and use regular expressions to generate all possible strings of semantic-nops and then search those for the PIPs. This approach works well for them but they

[13] For example the instruction *nop* compiles to *[0x90]*, but the *movsldup xmm0,xmm1* instruction becomes *[0xF3, 0x0F, 0x12, 0xC1]*.

are limited to searching for four, eight and twelve bytes PIP sequences. I intend to use constraint programming techniques to allow for a more flexible approach.

Summary

For this project I aim to:

- Study the architectures for a variety of platforms (including x86, AARCH32, XS1, and the JVM) with a view to finding semantic-nops.
- Create a database of semantic-nop instructions that is publicly available.
- Use constraint-programming techniques to create an algorithm for finding PIP headers for various architectures.
- Produce a database of Platform Independant Program headers.

Bibliography

[1] AT&T. Elf header. In *SYSTEM V APPLICATION BINARY INTERFACE*. The Santa Cruz Operation, Inc., 1997.

[2] Daniel Bilar. Fingerprinting malicious code through statistical opcode analysis. *International Journal of Electronic Security and Digital Forensics*, 2007.

[3] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code Normalization for Self-Mutating Malware. *IEEE Security and Privacy*, 5(2):46–54, March 2007.

[4] Sang Kil Cha, Brian Pak, David Brumley, and Richard Jay Lipton. Platform-independent programs. *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[5] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, Helmut Veith, and Technische Universität München. Malware normalization. Technical report, IEEE Symposium on Security and Privacy, 2005.

[7] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, Helmut Veith, and Technische Universität München. System for malware normalization and detection, 2010. US Patent No. 20100011441A1.

[8] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997.

[9] Ang Cui and Jonathan Voris. Print me if you dare. In *28th Chaos Communication Congress Behind Enemy Lines*, 2011.

[10] Eliot Dresselhaus. GNU GAS tc-i386.c. i386_align_code().

[11] Icculus. FatELF: Universal binaries for linux, 2009.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developers Manual*.

[13] Inc. MIPS Technologies. *MIPS32® 1074KTM CPU Family Software User's Manual*.

[14] Rodney Owens and Weichao Wang. Non-normalizable functions: A new method to generate metamorphic malware. In *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, pages 1279–1284, 2011.

[15] M.D. Preda, M. Christodorescu, S. Jha, and S Debray. A semantics-based approach to malware detection. *CONFERENCE RECORD OF THE ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*, 2007.

[16] Scythale. Hacking deeper in the system. *Phrack 64*, 2007.

[17] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. *Lecture Notes in Computer Science*, volume 6913. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.