Platform Independent Programs

Joseph Hallett

April 24, 2012

Contents

Chapter 1

Abstract

This is where the abstract will go... but there isn't one yet.

Chapter 2

Introduction

In 2010 a team of researchers developed a generalized method for creating platform independent programs (PIPs)[5]. 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

[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.

$$\overbrace{\underbrace{90}_{\text{NOP}}\ \underbrace{eb20}_{\text{JMP}}}^{\text{NOP}}\ 2a\ \underbrace{90eb203a}_{\text{NOP}}\ \underbrace{24770104}_{\text{B}}$$

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.

for the x86, ARM, and MIPS platforms, as well as the Windows, Mac, and Linux operating systems.

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][20], but there still appears to be no exhaustive list exists documenting them. Part of the work required is to create one.

Why is this interesting?

PIPs can be used for a variety of applications. One potential application suggested by Brumley et. al.[5] 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

[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*.

[4] which the authors[5] 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[16].

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[11][21]. 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[5] 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 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

[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[5] they suggest `ls`.

of preexisting systems for doing this such as Apple's *Universal Binary* or the *FatELF*[13] format. Another problem is that for some operating systems this just wouldn't work: Linux normally uses the ELF format[16] 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*[10] 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 [8][7], even describe obfuscation techniques where they unobfuscate the adding of semantic NOPs using a novel (and patented [6]) 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 [8][19][4] have suggested

[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.

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[23]. The GNU Assembler has a short list of efficient, low power semantic-nop[12] instructions it uses to pad instruction sequences to cachelines[12].

What is the Challenge?

The original PIP paper[5] 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 instructtions 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[18] 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[14] 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[12]. It is considerably harder to find semantic-nops for the x86 architecture.

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

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[22] and MIPS[18] 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[14], 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 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.
- Produce a database of Platform Independant Program headers.

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

Chapter 3

Technical Basis

To find the PIPs three tasks need to be accomplished: find the instructions that can be used to form semantic-nops; chain them together with jump instructions to create the potential PIPs for a given architecture; finally you need to compare the potential PIPs for each architecture to see if any of them exist in both architectures. These are the PIPs we're interested in.

Semantic-NOPS

Searching for the semantic-nops is book work. You take the architecture manual and search through it; making a notes of the mnemonic and whether any exceptions could be raised or flags overwritten. For simple instruction sets like (AArch32 or MIPS) this is easy; but for complex ones this can be a long process[1].

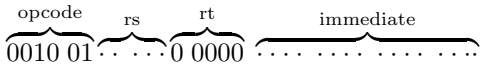
Once you have found the instructions you want to use for a semantic-nop you have to deal with the problem of scale. There are a lot of them. Here you have two concerns: for a list of semantic-nops you want clarity of instruction and to be easily able to identify assembled and disassembled forms. An easy way to do this is to store the bytecode with the assembly instructions used to generate it. A problem with this approach, however, is that the lists can become large. An alternative to this is to introduce *don't care* bytes into the compiled forms.

Consider this example: a simple semantic-nop for the MIPS architecture is addiu zero,t0,0. It has the (big-endian) compiled form of 25000000. Another semantic-nop is addiu zero,t0,1 which compiles to 25000001. But for this instruction so long as you write back to the zero register the instruction is always a semantic-nop. Looking at the architecture manual[18] there are no exceptions

[1] Resources such as the *X86 Opcode and Instruction Reference* [@refxasmnet:vu] are invaluable for discovering what each instruction actually does in a clear format.

that can be raised by it so its safe to use as a semantic-nop. The manual lists describes the instruction addiu rt,rs,immediate as:

$$GPR[rt] \leftarrow GPR[rs] + immediate$$



If we were going to enumerate every possible combination of operands for this single instruction we'd get around two-million[2] possible semantic-nops just from this single instruction. Whilst for a database this is desirable to accurately describe every possible semantic-nop, for generating PIPs this becomes a problem. To generate the PIPs we need to combine them with other semantic-nops and jump instructions. By working with the representation with *don't cares*[3] we can dramatically cut the number of permutations of instructions. This is important when the numbers of semantic-nops becomes huge, and the instructions are shorter (i.e. with X86) as the running time to find them can become excessive.

Generating PIPs

Once we have the list of semantic-nops we then need a list of possible jump instructions. We generate this list the same way we find the semantic-nops: enumerating the possible operations and then generalising by introducing don't care symbols. Again multibyte jumps are possible on some architectures (such as AARCH32) by exploiting conditional execution.

Once you have the two sets—semantic-nops and jumps—for the architecture you can proceed as follows: pick a length of the PIP pattern you want to find, add a jump instruction on the end of the pip. Subtract the length of the jump instruction from the pattern length and then for every possible semantic-nop add it to the start of the pip. If it hasn't made the pip too long, output it as a possible pip before trying to add another semantic-nop onto the pattern. Finally pad any output pips to the required length with don't cares if it isn't long enough. Pseudo code for this process is given bellow in a python-esque language.

```
generate_possible_nop_jump_patterns(length, nop_list, jump_list):
    for each jump in jump_list:
        pattern = [jump]
        for each pip in pad_with_nops(pattern, length, nop_list):
```

[2]There are twenty-one free bits in the instruction, so there are 2^{21} possible enumerations; which is 2,097,152 in real numbers.

[3]Actually you end up using the hexadecimal notation because it works better with disassembler tools. For this instruction ends up being stored as `2[4567][2468ace]0`$\cdots$ which has twenty-eight possible enumerations not including don't cares. This turns out to give a good balance between wanting a short list of instructions and a readable format.

```
            pip = pad_with_dont_cares(pip, length)
            print pip

pad_with_nops(pattern, length, nop_list):
    if length(pattern) < length:
        yield pattern
        for each nop in nop_list:
            pattern = nop : pattern
            for each pip in pad_with_nops(pattern, length, nop_list):
                yield pip
```

Once we have done this for multiple architectures we can try and find pips which are valid for two or more architectures. To do this we find the pips in each architecture which have equivalent forms and produce a new pip from them which forces some of the don't-cares to actualy values if one of the pips demands it.

```
-- Equality with don't cares
(~=~) :: Nibble -> Nibble -> Bool
'.' ~=~ _   = True
x   ~=~ '.' = True
x   ~=~ y   = x == y

-- Do two pips match?
matches :: PIP -> PIP -> Bool
x `matches` y = and $ zipWith (~=~) x y

-- Resolve two pips to remove don't cares
resolve :: PIP -> PIP -> PIP
resolve = zipWith resolve'
        where
                resolve' x y
                        | x == y    = x
                        | x == '.'  = y
                        | y == '.'  = x
                        | otherwise = error "Resolving unresolvable characters"

{- Given two sets of pips, produce a third containing
   the valid pips for both architectures -}
findPIPs :: [PIP] -> [PIP] -> [PIP]
findPIPs pips1 pips2 =
    [ resolve x y
    | x <- pips1
    , y <- pips2
    , x `matches` y
    ]
```

These pips are the ones we particularly interested in. We can repeat the process again to find pips for multiple architectures if we like by using the generated set of pips as one of the input sets.

An Alternative Approach

The approach taken here to find the pips is similar to the one taken by Cha et al[5] to find their gadget headers (though they do not give a specific algorithm for this section). However alternative approaches are also possible. A good area to provide an alternative method for is the semantic nop section.

As described earlier; adding dead or garbage code is an established obfuscation technique, and it is in current use in several metamorphic codes such as Evol, ZMist, Regswap and MetaPHOR[3]. Identifying dead code sequences is a technique already used by several antivirus tools as part of their toolchains. These can be leveraged to finding semantic-nops by getting them to output the offending sequences.[4]

One approach to identify these semantic-nop based sequences is to use signatures[5], but this requires the semantic-nops to have been previously identified. An alternative scheme, used by the SAFE tool[9], is to find the semantic-nops is to try and analyze sections of code and see if there would be any net change in the state of a machine if they were to run them. This is similar to simulation but has the added challenge of being able to tell if *any* input would cause a change of state rather than just the single input that is simulated.

To do this you need to keep track of all the variables and what transformations are applied to them over the course of the program. Calculations like this get big and slow quickly and there are no guarantees that they will find andy semantic-nop regions. The other problem with this strategy for finding semantic-nops is that the regions of code can be very large. For the purposes of this project it's preferable that any patterns found be short; so actually just using the architecture manual and hand picking, as it were, one or two instruction sequences to form the semantic-nops is preferable. Also for the sequences of only one or two instructions it is quite easy to find every possible semantic-nop pattern.

[4] Actually this doesn't work quite as described. A really simple and often used trick to implement dead code insertion is to introduce unreachable code that looks as if it could be reached (i.e. by placing a conditional jump that is always taken just before it). This unreachable code might not necesarilly have no net effect on the program execution but because it will never be run it doesn't matter anyway. From the point of view of a metamorphic virus this is an attractive technique because of the greater freedom of content inside the dead-code segment; and so many more variants of the malware. For pips this technique isn't useful (or rather implementing the always-taken-jump before the dead code is what we're trying to do rather than the writing dead-code). Coverage tools such as *gcov* can be used to find unreachable code such as this.[1] For finding semantic-nops more advanced tricks need to be used.

[5] Have a big list of *signatures*; see if any match the bit of code you're looking at.

Because of the increased complexity and running time of this method I did it by hand instead. This had the advantage of being simple, easy and ensured I didn't miss any details, such as side effects or missing instructions. For finding the pips I believe this was the better method in this case.

Chapter 4

Components

- I used the GNU compiler toolchains for ARM, MIPS, X86 to assemble lists of semantic-nops.
- I used the XMOS toolchain to assemble lists of semantic-nops for the XS1 architecture.
- I used the Jasmine assembler to explore writing semantic-nops for the JVM.
- I used the Radare 2 framework to write semantic-nops and jumps with don't care bytes for ARM, MIPS and X86 as well as to verify the pips at the end. I also used its JVM dissasembler and assembler to explore the JVM for creating pips.
- I used Ruby and Haskell to write various tools to create the pips.
- I refered to the architecture manuals for ARM, MIPS[18], X86 and XS1 extensively throughout the project but also made use of the ARM and Thumb Instruction Set Quick Reference Card[15] and X86 Opcode reference[17].

Chapter 5

Execution

Conclusion

[1] GCC Administrator. "Gcov - Using the GNU Compiler Collection (GCC)". In: ().

[2] D Bilar. "Fingerprinting malicious code through statistical opcode analysis". In: *ICGeS'07: Proceedings of the 3rd International ...* (2007).

[3] JM Borello. "Code obfuscation techniques for metamorphic viruses". In: *Journal in Computer Virology* (2008).

[4] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. "Code Normalization for Self-Mutating Malware". In: *IEEE Security and Privacy Magazine* 5.2 (Mar. 2007), pp. 46–54.

[5] SK Cha, B Pak, and D. Brumley. "Platform Independent Programs". In: *... of the 17th ACM conference on ...* (2010).

[6] M Christodorescu and S Jha. "System for Malware Normalization and Detection". In: *WO Patent WO/2009/ ...* (2009).

[7] M Christodorescu, S Jha, and SA Seshia. "Semantics-aware malware detection". In: *... and Privacy* (2005).

[8] M Christodorescu et al. "Malware normalization". In: (2005).

[9] Mihai Christodorescu and Somesh Jha. "Static Analysis of Executables to Detect Malicious Patterns". In: (2006).

[10] C Collberg and C Thomborson. "A taxonomy of obfuscating transformations". In: (1997).

[11] Ang Cui and Jonathan Voris. *28C3: Print Me If You Dare*. URL: http://events.ccc.de/congress/2011/Fahrplan/events/4780.en.html.

[12] *i386.c*. URL: http://www.opensource.apple.com/source/cctools/cctools-758/as/i386.c.

[13] Icculus. *FatELF*. URL: http://icculus.org/fatelf/.

[14] Intel Corporation. *Intel architecture software developer's manual*. 1997.

[15] ARM Limited. *ARM® and Thumb®-2 Instruction Set Quick Reference Card*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf.

[16] man.cx. *Man (5) elf*. URL: http://man.cx/elf(5).

[17] MazeGen. *X86 Opcode and Instruction Reference*. URL: http://ref.x86asm.net/.

[18] MIPS Technologies Inc. "MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set". In: (Mar. 2011), pp. 1–300.

[19] R Owens. "Non-normalizable functions: A new method to generate metamorphic malware". In: ... *CONFERENCE* (2011).

[20] Mila Dalla Preda et al. *A semantics-based approach to malware detection*. Vol. 42. New York, New York, USA: ACM, Jan. 2007.

[21] Scythale. *Hacking deeper in the system*. URL: http://www.phrack.org/issues.html?issue=64&id=12#article.

[22] D Seal. *ARM architecture reference manual*. 2000.

[23] R. Wartell et al. "Differentiating code from data in x86 binaries". In: *Machine Learning and Knowledge Discovery in Databases* (2011), pp. 522–536.