

Platform Independent Programs

Joseph Hallett

May 9, 2012

Contents

1	Executive Summary	5
2	Introduction	7
	Constructing PIPs	7
	Aim Of The Project	8
	Applications Of PIPs	9
	Other Approaches To Program Obfuscation	11
	What Is The Challenge?	12
3	Technical Basis	15
	Computer Architecture Background	15
	Semantic NOPs	16
	Generating PIPs	18
	An Alternative Approach	20
	Malware Detection Methods	21
	Tool-Chains Used	22
4	Components	23
5	Execution	25
	Semantic NOPs	25
	The JVM	26
	PIPs	28
	Why So Few For MIPS?	28

Detecting PIPs	29
Hiding PIP Code	31
Writing Programs with PIPs	31
6 Conclusion	35

Chapter 1

Executive Summary

This is where the executive summary will go... but there is not one yet.

Chapter 2

Introduction

Constructing PIPs

In 2010 a team of researchers developed a generalized method for creating Platform Independent Programs (PIPs)[\[11\]](#). A PIP is a special sort of program which can be run on multiple different computer architectures without modification. Unlike shell scripts or programs written for a portable interpreter; a PIP does not require another program to run or compile it; rather it runs as a native program on multiple architectures with potentially different behaviour on each.

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:

$$m_1(b) \neq \perp \wedge m_2(b) \neq \perp.$$

To construct a PIP one must analyse the instruction sets of each architecture and find instructions which compile to identical patterns of bytecode. The approach taken by the authors in [\[11\]](#) was to find small PIPs with a very specific form: do nothing then jump. By ensuring each architecture jumped to a different point and that each architecture did not accidentally run into a region another 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.

Consider the following example (taken from [\[11\]](#)). The disassembly for the x86 architecture is shown above, and for the MIPS platform below.

$$\begin{array}{ccccccc} & \text{NOP} & \text{JMP} & & & & \\ & \underbrace{} & \underbrace{} & & & & \\ \underbrace{90} & \underbrace{eb202a} & \underbrace{90eb203a} & \underbrace{24770104} & & & \\ \text{NOP} & & \text{NOP} & \text{B} & & & \end{array}$$

The string is valid on both platforms and has similar behaviour on both despite jumping to different locations. In fact this is a valid PIP for the x86, MIPS and ARM architectures. If we disassemble the pattern with the Radare2 reverse engineering framework[36] we can see that it disassembles to:

Architecture	Disassembly
x86	nop; jmp 0x100000023; sub dl, [eax+0x243a20eb]; ja 0x10000000c; ???
ARM	bcs 0x10083ae48; bcc 0x10083ae4c; streq r7, [r1], #-1828
MIPS	slti zero,s1,-5232; xori zero,s1,0xeb90; b 0x10001dc9c

For the x86 architecture it is a nop instruction then a jump instruction. The rest won't be executed (though some of it is valid x86 code) as the unconditional jump will have moved the program counter along. For the ARM architecture it starts with two conditional jumps. The first tests if the processor's carry flag is set, and the next checks if the carry flag is not set. One of these two instructions will be executed so one of the two jumps will be taken. For MIPS architecture the first two instructions write the result of the operation back to register zero. On the MIPS architecture any writes to register zero are discarded and the `slti` and `xori` can not cause any errors to occur. This means the first two instructions are equivalent to a nop instruction. The third instruction is a MIPS branch instruction, so the sequence for a MIPS computer is equivalent to do nothing and jump.

Since for each of the x86, ARM, and MIPS architectures the bytecode is equivalent to do nothing and jump; the instruction is a valid PIP.

They go on to give 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.

Aim Of The Project

For this thesis I have implemented a section of the PIP finding algorithm: the section for finding the *gadget headers*; the PIPs that link the specific code sections together. To generate the PIPs a list of *semantic NOPs*¹ and potential branch instructions has been found for each architecture in the original paper

¹A 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*.

and to extend the work of the original paper I have also analyzed a new platform: XMOS 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[5][35], but there still appears to be no exhaustive list exists documenting them. Part of the work required is to create one.

Applications Of PIPs

PIPs can be used for a variety of applications. One potential application suggested by Brumley et. al.[11] 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*². The program which they wish to communicate becomes their *steg-text*, and they construct a *cover-text* by writing some other program which does not 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 architecture³. The warden runs the program and sees it is not something he would want to restrict and delivers it. In fact unless he is aware that it has been 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.

A more real world example of this is to consider the relationship between computers (usually using an x86 processor) and modern smartphones (which often use ARM processors) running *apps*. The computer often stores backups of the applications used on the smartphone. Apple's iTunes program, for example, stores all the mobile applications associated with a user in its library folder as zip-compressed archives. Suppose an attacker were to construct a program which was a legitimate application on the ARM platform but when run on x86 behaved as a form of malware. The computer might have some form of anti-malware software, but unless it knows to scan the mobile applications as potential x86

²which the authors[11] call *execution-based steganography*.

³if they were using ELF they would not even need to do that—it is part of the header in the file[29].

viruses rather than the ARM ones they identify themselves as (and which aren't ARM malware) then the anti-malware program might miss the dangerous code.

Another application is *exfiltration protection*. Exfiltration is a military term meaning the removal of a resource from enemy control. In the context of PIPs this probably involves taking programs from protected PCs; kind of like DRM. 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 is running on. They create a PIP out of these two programs and now if the program is stolen and the exfiltrator is not aware of the PIP nature (or exactly what modifications were made to the architecture) they cannot execute the program they removed.

Microcode offers another interesting 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[18][37]. Now suppose that as well as the microcode update they also modify an existing program, Brumley et. al suggest ls, 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. point out[11] that if this was done by Intel and the PIP was a preexisting and digitally signed application: 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 could also be used to create platform independent shellcode to take advantage of buffer overflows on software ported between different architectures and operating systems. As well as developing PIPs to create architecture independent programs, Brumley et. al.[11] extended the basic technique to create operating system programs. For operating system independent programs they exploited overlaps in calling conventions and interrupts to develop PIPs which could be valid programs on multiple systems. Brumley et. al. give an example remote bind-shell shellcode for multiple architectures at the end of their paper [11].

Another application for PIPs is to create actual platform independent programs. The idea here is to compile a program for multiple architectures and create a PIP out of them. You would 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.

Other Approaches To Program Obfuscation

The problem is that although PIPs could be used to write architecture independent 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*^[22] format. Another problem is that for some operating systems this just would not work: Linux normally uses the ELF format^[29] which has a flag in the header which specifies exactly what architecture the binary was compiled for. If it does not match the architecture of the machine it is being run on, then the loader refuses to run it⁴.

Collberg et. al. [17] describe different methods for hiding the structure of a program. They give many different transforms but three are of particular interest: adding dead code, adding redundant transforms, and outlining⁵ sections of code.

Adding dead code i

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 is not a particularly *resilient*⁶ transformation (a program could replace or remove them) they are potent⁷ especially if they are 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 you're using just one PIP at the start of the program then it is not 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 [14][13], even describe obfuscation techniques where they unobfuscate the addition of semantic NOPs using a novel (and patented [12]) tool called *Hammock*. Hammock appears to be interesting because rather than using a catalogue of pre-known semantic NOPs it finds them by analysing whether sequences of code have any net effect on the state of the machine. They find it to be a very slow transform to unobfuscate (implying adding NOPs is

⁴Of course there is nothing to stop you flipping the flag to some other value with `elfedit` utility from the GNU Binutils.

⁵Outlining 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.

⁶Resilience 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.

⁷Potency measures how confusing for a human the transform is. For example self-modifying code is a very potent transform, but renaming the jump targets is not.

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 [14][34][9] have suggested using semantic NOPs as an obfuscating technique. Wartell et. al.[41] suggest using them as part of a heuristic for differentiating between code and data for disassembled programs. The GNU Assembler has a short list of efficient, low power semantic NOP⁸ instructions it uses to pad instruction sequences to cache-lines[21].

What Is The Challenge?

The original PIP paper[11] contains an anecdote where the effort required to create platform independent 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 are not trying to fully generate platform independent programs; rather we are 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 are 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 instructions 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* easy—the instruction sets are small and everything in the instruction set is accessible through a standard assembler.

The MIPS architecture[32] 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

⁸A comment above the function[21] notes that most of the instructions used as part of the semantic NOP sequences by the assembler aren’t in fact assemblable with the assembler.

make it an ideal platform for writing semantic NOPs. Several million single instruction semantic NOPs can be found with minimal effort. The Intel x86 architecture[23] 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 not be assembled by the GNU toolchain[21]. 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. Once we have them in an indexable format we need to search for the patterns that overlap and find all the PIP headers. There are significant problems associated with finding these PIP headers. For platforms like AARCH32[2] and MIPS[32] instructions are all compiled to be of fixed length (four bytes). In this case we could find short PIPs by comparing the lists of NOP instructions for one architecture and jump instructions for the other. We could extend them to arbitrary lengths by finding the NOPs which do nothing on both architectures and padding to the length required.

In practice however this approach doesn't work. Variable length instruction sets, such as x86⁹, mean you need to combine instructions together to get them to the length you require. If there were only a few identifiable patterns of semantic NOPs and jump instructions then this approach might be feasible but the numbers become huge. For example on many architectures there is an unconditional jump instruction. If this instruction takes a thirty-two bit address to jump to then there are 2^{32} , over four billion, possible forms of this instruction to check. And this is just one instruction. On x86 it even has more than one compiled form. Conditional jumps exasperate the problem further. For a conditional jump you need two (or more!) jump instructions, so that means 2^{64} possible variants to check which is huge.

The MIPS architecture demonstrates well another issue. With the MIPS architecture you have a register called zero which discards any value written to it. This offers great opportunities for finding semantic NOPs, but it also presents further problems with the size of patterns you can find. The ADDI instruction, for example, is used to add a sixteen bit number to a register and then write back to another one. MIPS registers are represented in an instruction using five bits and it doesn't matter which of them we use (so long as we write back to zero). A sixteen bit immediate, plus a five bit register means twenty-one bits we don't care about in this instruction, and over two million permutations of this single instruction. Even if we use tricks to reduce the problem size we still have problems. Even restricting the search to a small subset of the possible patterns the amount of memory required to store them is large—hundreds of gigabytes. If we want to be able to detect when there are PIPs in a file we need to be able

⁹For example the instruction `nop` compiles to `[0x90]`, but the `movsldup xmm0,xmm1` instruction becomes `[0xF3, 0x0F, 0x12, 0xC1]`.

to search these files; again computationally expensive.

Detecting PIPs is another difficult problem. There is currently no data as to how often these patterns occur in regular files. Since the instruction sequences used in PIPs are valid for multiple architectures a PIP instruction sequences could turn up in a program without being part of some malicious behaviour. Data for how often PIPs turn up in normal code is needed before any statistical model for detecting them can be made.

This data does not currently exist.

Chapter 3

Technical Basis

To construct 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 compare the potential PIPs for each architecture to see if any of them exist in both architectures. These are the PIPs we are interested in.

Computer Architecture Background

To construct PIPs instructions must be assembled and the bytecode examined. Processors fetch instructions in a binary format, a string of 1s and 0s, which we call bytecode. Each architecture has its own specific form of bytecode; that is to say if 1010100101 means add two registers on one architecture then there are no guarantees that this pattern means the same thing on another—or even that the other has registers to add together.

Every instruction a processor wishes to offer has to be mapped to a sequence of bytecode¹. Some architectures, such as ARM[2] MIPS[32] and X86[23], are register based (they expect data to be used in calculations to be stored in special pieces of memory called registers) and the instructions take arguments saying exactly which registers to use. Others, such as the JVM[28], are stack based (they expect arguments to operations to be stored in a data structure called a stack where the top one or two items are the operands to most instructions).

¹This is true in general but slightly oversimplified. The x86 and ARM instruction sets both feature special instructions which can change how sequential pieces of bytecode are decoded. For example the ARM architecture[2] can switch to decoding the THUMB or JVM instruction sets by using the BX instruction (which has the bytecode of e12fff10). X86[23] offers similar a similar mechanism for turning on or off feature sets which can alter how instructions are decoded.

Different architectures offer different sorts of instructions. The x86 achitecture offers a large number of instructions which can do many different things such as AES encryption and arithmetic[31]. The ARM architecture[2], however, is much smaller—it doesn’t have a division instruction. The XS1 architecture[30] has several instructions for concurrency and communicating over ports which are not present on other architectures. To make matters more complex the length of instructions also varies on an architecture by architecture basis: MIPS and ARM instructions are always four bytes long but the x86 and JVM instruction sets use a variable length instruction size.

When a processor wishes to execute a program (formed of bytecode) it *fetches* the instruction (or instructions if the processor is superscalar) to be run, *decodes* what the instruction is to do before *executing* it and *writing back* the result. For this project we are targetting the decode stage. We are trying to find bytecode that decodes to legitimate instructions for multiple processors, and then using this bytecode to make arbitrary programs.

Semantic NOPs

Formally a semantic NOP is an instruction that has no net effect on the state of the processor other than moving the program counter to the next instruction. A semantic NOP is functionally equivalent to the NOP opcode (which often is a synonym for a low-power semantic NOP). Specifically if the outcome of the machine executing an instruction is functionally equivalent to the machine executing the NOP instruction, independent of the state of the machine², then it is a semantic NOP.

For example on the ARM architecture[2] the NOP instruction is assembled into the bytecode `e1a00000`. The instruction `mov r0, r0` is also assembled to `e1a00000`. Because they have the same bytecode we can see that these two instructions are actually one and the same. The designers of the architecture chose to replace the NOP instruction with a functionally equivalent one in the bytecode format. This is done (amongst other reasons) to compress the instruction set. Here the `mov r0, r0` instruction is the NOP instruction as well as being functionally equivalent to it, but the designers of the architecture could have chosen differently. The instruction `mov r1, r1` is functionally equivalent to the NOP command as well but the bytecode to represent it is `e1a01001` which is different to the NOP instruction. This means that `mov r1, r1` is a semantic NOP—it has the same behaviour as the NOP instruction and a different bytecode.

The instruction `movgt pc, r2` however would not be a semantic NOP instruction. Here the instruction will behave as a NOP instruction unless the greater than

²e.g. the instruction would always behave as a NOP instruction even if a conditional execution flag was set differently.

flags are set when it will move the contents of register two into the program counter. This isn't a semantic NOP because though it will behave like a NOP instruction some of the time it might not do. If a programmer had added this instruction knowing that the greater than flags would never be set when this instruction was executed then it would be very similar to a semantic NOP but more often called *dead code*[\[17\]](#).

There is another sort of semantic NOP we have used: a multi-instruction semantic NOP. This is a sequence of instructions that may alter the state of the machine, but will reverse any change they make by the end of the sequence; they are a redundant transform. For example the sequence `ADD r0,r0,#1`, `SUB r0,r0,#1` is a redundant transform as any change to the state of the machine (specifically register zero) is undone by the second instruction³. We call sequences like this semantic NOPs as well though more care must be taken when using these as if the machine were to handle an interrupt whilst executing one the state of the machine might be altered unpredictably.

Searching for the semantic NOPs is book work. You take the architecture manual and search through it; making a notes of the mnemonic, arguments and whether any exceptions could be raised or flags overwritten. For simple instruction sets (like ARM or MIPS) this can be done in a couple of hours; but for complex instruction sets this can be an arduous process⁴.

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[\[32\]](#) there are no exceptions 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:

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

³On the X86 architecture[\[23\]](#), however, this wouldn't be a semantic NOP as arithmetic operations alter status flags as well as the registers they operate on[@refxasmnet:vu].

⁴Resources such as the *X86 Opcode and Instruction Reference* [\[31\]](#) are invaluable for discovering what each instruction actually does in a clear format.

If we were going to enumerate every possible combination of operands for this single instruction we would get around two-million⁵ 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*⁶ 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* symbols⁷. Again multibyte jumps are possible on some architectures (such as ARM) 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 has not 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 is not long enough. Pseudo code for this process is given below in a python-esque language.

```
def generate_possible_nop_jump_patterns(length, nop_list, jump_list):
    for jump in jump_list:
        pattern = [jump]
        for PIP in pad_with_nops(pattern, length, nop_list):
            PIP = pad_with_dont_cares(PIP, length)
            print PIP
```

⁵There are twenty-one free bits in the instruction, so there are 2^{21} possible enumerations; which is 2,097,152 in real numbers.

⁶Actually 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... 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.

⁷A *don't care* symbol represents a bit or byte of the instruction set whose value we don't care about. For example the x86 short jump instruction could be represented in byte-code as eb.. where . is the don't care symbol. We care that it is a jump instruction, which the eb encodes; but we might not care where it jumps to so we can represent the destination with *don't cares*.

```

def pad_with_nops(pattern, length, nop_list):
    if length(pattern) < length:
        yield pattern
    for 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 if required
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[11] 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[7]. 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.⁸

One approach to identify these semantic NOP based sequences is to use signatures⁹, but this requires the semantic NOPs to have been previously identified. An alternative scheme, used by the SAFE tool[15], 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 become unfeasible quickly and there are no guarantees that they will find any 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 is 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.

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 did not miss any details, such as side effects or missing instructions. For finding the PIPs I believe this was the better method in this case.

⁸Actually this does not 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 necessarily have no net effect on the program execution but because it will never be run it does not 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 is not useful (or rather implementing the always-taken-jump before the dead code is what we are 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.

⁹Have a big list of *signatures*; see if any match the bit of code you're looking at.

Malware Detection Methods

Given that we can use PIPs to create programs with steganography it would be helpful to be able to distinguish PIP from non-PIP. Malware detection gives a set of methods to accomplish this.

Using the detection notion defined in [35] we define a malware detector as the following. Given the set of every possible program \mathbb{P} there is a subset \mathbb{M} that have malicious behaviour.

$$\mathbb{M} \subset \mathbb{P}$$

For some of the programs in \mathbb{M} we have a signature s which is formed in some way from the program. A detector D is a function that given a $p \in \mathbb{P}$, and a signature s says true if the signature was formed from that program and false otherwise. The

We can evaluate the detector and a set of signatures by seeing how well it can distinguish malware from non-malware. The false-negative rate is the percentage of all the malwares which the detector fails to report as being malware. The false-positive rate is the percentage of all non-malware detector incorrectly reports as being malware. Depending on the where the malware detector is being used the false positive and negative rate can be tweaked to requirements by altering the set of signatures and how they are generated. The approach of detecting malware by its appearance is popular; however in general detecting whether a program is malware by appearance is an undecidable problem [16][38].

There are several different approaches to generating malware signatures. One approach is to use a section of the malware itself to form a signature. To do this they give a regular expression that specifies a sequence of byte or instruction sequences that are considered malicious. If the malicious sequences of code are seen in a program the malware detector reports it as malware. To avoid these techniques new forms of *polymorphic* and *metamorphic* malware have been developed which use self modifying code and encryption to hide these signatures from signature based detectors[13].

Other approaches to getting and evading signatures have been developed. One approach to avoiding signature matching is to randomize the order of some expressions[7][13]. To counter these obfuscations approaches which use control flow patterns as signatures have been developed[6]. The idea is that whilst some sub-expressions may be rearranged the algorithms themselves cannot be so dramatically changed. These approaches have been relatively succesful at detecting polymorphic malware [24][10]. Another approach is to use model checking to try and identify sections of code that copy the instruction sequences to different locations or which use suspicious parts of the Windows API[26]. To counter these improved protections more techniques have been developed. One approach to detecting malware is to simulate it and see if it does anything suspicious. To counter this malware authors have taken to putting in NP-HARD problems into

their code so that any simulator also has to solve a problem such as 3-SAT[33]—this slows detection.

Various toolchains have been developed to aid detecting malware. The Binary Analysis Platform[8] is one such platform which works by reduction to an intermediary language. It offers support for as well as malware detection; automatic exploit generation[3], signature generation, and formal verification. Other platforms, such as CodeSurfer[4], are built on top of IDA Pro[20]. CodeSurfer works with IDA to provide more representations of a program; the idea is that these extra-representations allow an analyst to reason about what a piece of malware does.

Tool-Chains Used

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 dissassembler and assembler to explore the JVM for creating PIPs.
- I used Ruby and Haskell to write various tools to create the PIPs.
- I referred to the architecture manuals for ARM, MIPS[\[32\]](#), X86 and XS1 extensively throughout the project but also made use of the ARM and Thumb Instruction Set Quick Reference Card[\[27\]](#) and X86 Opcode reference[\[31\]](#).

Chapter 5

Execution

Semantic NOPs

Architecture	Semantic NOPs Identified
ARM	187,879
MIPS	18,958,336
X86	1,266
XS1	792

Table 5.1: Semantic NOP sequences identified per architecture.

Around nineteen million semantic NOP sequences for the ARM, MIPS, X86 and XS1 architectures were identified and stored in a database of the form:

SEMANTIC-NOPS (architecture, instruction prefix, instruction suffix, bytecode prefix, bytecode suffix)

By using prefix and suffixes we can separate certain multi-instruction semantic NOPs from the rest. Some multi-instruction semantic NOPs can be written with more semantic NOPs within them and by using this prefix and suffix form we can distinguish the bit which needs to go first from the bit which must come at the end.

For example consider these entries from the database:

Architecture	Instruction Prefix	Instruction Suffix	Bytecode Prefix	Bytecode Suffix
x86	PUSH %rax	POP %rax	50	58
x86	NOP		90	

The bytecode 90 is a semantic NOP—the nop instruction. Equally the sequence push %rax; pop %rax is a semantic NOP sequence with bytecode 5058. For the push-pull sequence we can place any code in between the push and the pull. If that sequence is a semantic NOP too then the sequence as a whole is a semantic NOP also. So 509058 is a semantic NOP; as is 50909058; or ever 5050905858.

Looking at the numbers found in Table 5.1 the MIPS architecture is by far the easiest to find semantic NOPs for. The MIPS register zero (which discards all writes to it) enables any instruction to be easily converted to a semantic NOP just by writing back to register zero. Furthermore there are four different instructions in the MIPS architecture which take a sixteen bit immediate value as an operand and can be used without triggering an exception[32]: addiu, andi, ori and slti. These can all be used to generate semantic NOPs; but more importantly give us sixteen free bits for when we are trying to find the PIPs.

ARM is the next easiest (though there are a hundredth of what can be found for MIPS). The ARM7 architecture supports conditional execution which helps for finding semantic NOPs. Conditional execution is implemented by having four bits encode a conditional flag and one bit used to indicate that the system flags should be updated[2]. If the flag is matched then the command is executed else the command becomes a NOP. We have less registers than MIPS and while we have three instructions which can be used with immediate values (add, sub and eor) they only use an eight-bit value (as well as input to a barrel shifter).

X86 has significantly less semantic NOPs than ARM or MIPS. One reason for this is a lack of instructions that don't alter the state of the processor in some way: all the arithmetic instructions update flags inside the processor. There are no instructions we can use with an immediate value to write a semantic NOP. The XS1 architecture has a similar number of semantic NOPS to X86 for similar reasons. There are less registers than X86 and only a limited number of instructions that take an immediate value that can be used for writing semantic NOPs.

The JVM

The JVM is an interesting architecture but very different from all the others I looked at. The JVM is a virtual stack based architecture[28]. Stack based architectures don't use registers like the X86 ARM or MIPS architectures, but rather expect most of their instructions operands to be on a stack in memory. Some JVM instructions do take arguments passed as part of the bytecode instruction; such as the goto and goto_w instructions which take the two or four

byte address to go to as an argument. Most do not however and most JVM instructions are only one byte long. Within functions the JVM imposes some strict rules about the size of the stack and constants available. If the size of the stack exceeds the limit imposed then an exception is triggered.

This leads to some problems with trying to find semantic NOPs for the JVM. Most JVM NOPs that can be found are multi-instruction. There is a nop instruction (00), but in general to write a semantic NOP for the JVM you need to push and pull values on and off the stack. There are JVM instructions for rearranging the stack which can be used to create semantic NOPs—the swap instruction (5f) could be issued twice but even this only works if you know the type of the top two elements of the stack and can be sure they are the same type. Unless you know something of the program you’re adding these kinds of semantic NOPs too you can very easily end up triggering an exception from misuse of the stack.

Another problem with the JVM is from the complexity from chaining together the instructions. If you ignore the problems associated with limited stack space and assume an unlimited amount of stack then you still have to cope with the problems of enumerating. Specifically you need to find a sequence of instructions such that the stack is unchanged overall; but since most of the instructions take from the stack and add back to it you can use most of them so long as you pop (perhaps using the pop (57) or pop_2 (58) instructions) any additional values back off at the end and make sure there are enough useless values on the stack initially so as not to alter any pre-existing ones. Any dead-code program will work, which unfortunately means that there are a lot of them and they can be very long. An interesting side point here is that there are quite a lot of tools out there to detect Java dead code sequences: such as DCD[40] and UCD[39]. The JDK hotspot compiler can optimize deadcode sequences away[19]. It would be an interesting problem to see how rare dead code sequences are in regular compiled code (i.e. programs from Java code rather than handwritten JVM bytecode). Dead code elimination is a common optimization, and I would suspect the answer is not often.

PIPs

<i>Architecture</i>	ARM LE	ARM BE	MIPS LE	MIPS BE	X86
ARM LE		6.6×10^4	0	2.6×10^5	0
ARM BE	6.6×10^4		2.6×10^5	0	7.0×10^4
MIPS LE	0	2.6×10^5		1.0×10^6	0
MIPS BE	2.6×10^5	0	1.0×10^6		2.8×10^5
X86	0	7.0×10^4	0	2.8×10^5	

Table 5.2: Four byte PIPs found between architectures.

<i>Architecture</i>	ARM LE	ARM BE	MIPS LE	MIPS BE	X86
ARM LE		3.1×10^{14}	2.8×10^{14}	1.2×10^{15}	1.1×10^{12}
ARM BE	3.1×10^{14}		1.2×10^{15}	2.8×10^{14}	6.2×10^{14}
MIPS LE	2.8×10^{14}	1.2×10^{15}		4.5×10^{15}	4.2×10^6
MIPS BE	1.2×10^{15}	2.8×10^{14}	4.5×10^{15}		2.4×10^{15}
X86	1.1×10^{12}	6.2×10^{14}	4.2×10^6	2.4×10^{15}	

Table 5.3: Eight byte PIPs found between architectures. On top of these results 2.0×10^{10} were found which were valid for the ARM LE MIPS BE and X86 architectures.

Tables 5.2 and 5.3 show the number of PIPs found of length four and eight bytes respectively. For four byte headers we found a similar number to Brumley et. al. [11] for the ARM and X86 architectures (tens of thousands), however we found significantly more for the MIPS and any other architecture than Brumley (hundreds for Brumley et. al. versus tens to hundreds of thousands for us). Brumley et. al. don't give numbers for how many eight byte headers they can find however their numbers for twelve byte headers are around a thousand to ten-thousand times bigger than the number found for eight byte headers. This seems fairly reasonable considering the number of possible different bytecode sequences for a twelve byte sequence is 2^{96} rather than only 2^{64} for an eight byte one.

Why So Few For MIPS?

I am unsure why Brumley et. al. found so few four byte PIP headers for the MIPS architecture. For the twelve byte sequences the number of PIP headers

they found between the MIPS and any other architecture is significant but for four byte sequences their number found is very low. For example they only found six PIP headers between the MIPS little endian and big endian architectures. This suggests they didn't use the MIPS jump instruction to find any of their sequences.

The MIPS jump instruction has the following format[32]:

000010 address

It's easy to find a four byte PIP header for the little and big endian variants of the MIPS architecture by using this instruction. If we switch the endianness of the instruction and then remove all the `s` that overlap with a fixed bit we find that a jump instruction for both MIPS endiannesses has the form in binary of:

000010 000010 . .

Where \cdot indicates either a 1 or a 0. If we convert this sequence to hexadecimal we get the set of four byte PIP headers for different MIPS endianness variants that I identified.

$$\left\{ \begin{array}{l} 08 \dots 08, 08 \dots 09, 08 \dots 0a, 08 \dots 0b, 09 \dots 08, 09 \dots 09, 09 \dots 0a, 09 \dots 0b \\ 0a \dots 08, 0a \dots 09, 0a \dots 0a, 0a \dots 0b, 0b \dots 08, 0b \dots 09, 0b \dots 0a, 0b \dots 0b \end{array} \right\}$$

Detecting PIPs

Program	ARM BE (%)	MIPS BE (%)	MIPS LE (%)	X86 (%)
DrawSomethingFree	203 (0.0%)	834 (0.1%)	215 (0.0%)	1 (0.0%)
Dropbox	79 (0.0%)	375 (0.1%)	89 (0.0%)	1 (0.0%)
Mother 3	3426 (0.1%)	3181 (0.1%)	2227 (0.1%)	2 (0.0%)
Pages	696 (0.0%)	2659 (0.1%)	651 (0.0%)	7 (0.0%)
SwordAndSworcery	136 (0.0%)	545 (0.1%)	129 (0.0%)	0
Vim	32 (0.0%)	160 (0.1%)	33 (0.0%)	0
<i>Random</i>	25 (0.0%)	94 (0.1%)	17 (0.0%)	0

Table 5.4: Number of times eight byte PIP headers occur in ARM programs and the percentage of the total program which they occupy. All of the programs listed *apart from Mother 3 and Random* are taken from iPhone applications for Apple’s iOS operating system. *Mother 3* is a program for Nintendo’s GameBoy Advance. *Random* is a long string of random bytes.

Program	ARM BE (%)	ARM LE (%)	MIPS BE (%)	MIPS LE (%)
hello	0	0	0	0
cat	0	0	0	0
clang	223 (0.0%)	0	6299 (0.2%)	0
echo	0	0	4 (0.1%)	0
linux-2.6	264 (0.1%)	0	918 (0.2%)	0
ls	2 (0.0%)	0	19 (0.2%)	0
mach_kernel	237 (0.0%)	0	5266 (0.3%)	0
nasm	7 (0.0%)	0	72 (0.2%)	0
pandoc	582 (0.1%)	0	2147 (0.1%)	0
<i>Random</i>	39 (0.0%)	1 (0.0%)	205 (0.2%)	0

Table 5.5: Number of times eight byte PIP headers occur in X86 programs and the percentage of the total program that they occupy. The programs *cat*, *echo* and *ls* and small UNIX utilities. *Hello* is the hello-world program written in C. *Clang* is a C compiler; *nasm* is an assembler and *pandoc* is a Haskell based markdown compiler. *Linux-2.6* is the linux kernel and *mach_kernel* is a version of the mach kernel by Carnegie Mellon University found in Apple’s MacOS Lion.

To test the steganographic properties of PIPs I looked at how often they occur in various programs for the ARM and X86 architectures. I chose to look at X86 and ARM as they are two of the most commonly found architectures today. A processor with the X86 instruction set architecture is inside most consumer PCs and servers. The ARM architecture¹, however, is found everywhere. It is the dominant processor inside mobile phones with both Google’s Android and Apple’s iOS platforms running on this architecture. ARM chips are often found in embedded systems and have even been found in massively parallel supercomputers[25].

For the ARM architecture I focussed on *app-like* programs. I looked at a variety of apps from games to text editors as well as a sequence of random bytes and a Gameboy advance game. The Gameboy game is interesting as it also contains sound and graphics files built into it that the iOS applications do not. The results are shown in Table 5.4. The results seem to show that PIP headers very rarely turn up in ARM code; less than 0.1% typically. Some PIP headers turn up for the ARM little endian and MIPS architectures, but next to none ever turn up for X86 PIP headers in ARM little endian programs. It would be surprising if more than ten turned up in any program. Another interesting point

¹Specifically the ARM 7 32-bit architecture known as AArch32 not the new shiny 64-bit one.

In Brumley et. al.'s paper[11] they suggest that whole platform independent programs could be created by splitting the program into several *gadgets* each with a PIP header and a block of code to be executed for each platform the program author wishes to target. Brumley et. al. go on to suggest that a program could be split up into gadgets one instruction in length, however since each gadget would feature a PIP header this would likely destroy any steganographic properties the author want in their program. Because PIP headers are rare; a program with execution based steganography could be distinguished from a plaintext by counting the number of PIP headers that could be found and deciding whether that number is statistically significant. When an author is trying to hide X86 behaviour this is a much bigger problem as the number of PIP headers that could be expected to turn up naturally in a program is very low.

polymorphic pip code
microcode updates

Listing 5.1: An example of a shellcode PIP for X86 and MIPS which attempts to spawn a shell and elevate permissions. Shellcode for each architecture was taken from [40].

[illegible]

[illegible]


```

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 2806ffff 3c0f2f2f 35ef6269 afaffff4 3c0e6e2f 35ce7368
afaefff8 afa0fffc 27a4fff4 2805ffff 24030fab 0101010c

```


Chapter 6

Conclusion

Bibliography

- [1] GCC Administrator. “Gcov - Using the GNU Compiler Collection (GCC)”. In: ().
- [2] *ARM architecture reference manual*.
- [3] Thanassis Avgerinos et al. *AEG: Automatic Exploit Generation*.
- [4] G Balakrishnan, R Gruian, and T Reps. “CodeSurfer/x86—A Platform for Analyzing x86 Executables”. In: *Compiler Construction* (2005).
- [5] D Bilar. “Fingerprinting malicious code through statistical opcode analysis”. In: *ICGeS’07: Proceedings of the 3rd International ...* (2007).
- [6] G Bonfante, M Kaczmarek, and JY Marion. “Control flow graphs as malware signatures”. In: (2007).
- [7] JM Borello. “Code obfuscation techniques for metamorphic viruses”. In: *Journal in Computer Virology* (2008).
- [8] David Brumley et al. *djb research: binary analysis*.
- [9] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. “Code Normalization for Self-Mutating Malware”. In: *IEEE Security and Privacy Magazine* (2007).
- [10] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. *Detecting self-mutilating malware using control-flow graph matching*.
- [11] SK Cha, B Pak, and D. Brumley. “Platform Independent Programs”. In: *... of the 17th ACM conference on ...* (2010).
- [12] M Christodorescu and S Jha. “System for Malware Normalization and Detection”. In: *WO Patent WO/2009/ ...* (2009).
- [13] M Christodorescu, S Jha, and SA Seshia. “Semantics-aware malware detection”. In: *... and Privacy* (2005).
- [14] M Christodorescu et al. “Malware normalization”. In: (2005).
- [15] Mihai Christodorescu and Somesh Jha. “Static Analysis of Executables to Detect Malicious Patterns”. In: (2006).
- [16] F. Cohen. “Computer viruses: theory and experiments”. In: *Computers & security* (1987).

- [17] C Collberg and C Thomborson. “A taxonomy of obfuscating transformations”. In: (1997).
- [18] Ang Cui and Jonathan Voris. *28C3: Print Me If You Dare*.
- [19] Brian Goetz. *Java theory and practice: Dynamic compilation and performance measurement*.
- [20] HexRays. *Executive Summary: IDA Pro – at the cornerstone of IT security*.
- [21] *i386.c*.
- [22] Icculus. *FatELF*.
- [23] *Intel architecture software developer’s manual*.
- [24] Boojoong Kang et al. “Fast malware family detection method using control flow graphs”. In: *RACS ’11: Proceedings of the 2011 ACM Symposium on Research in Applied Computation*. 2011.
- [25] MM Khan et al. “SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor”. In: *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on* (2008).
- [26] Johannes Kinder et al. *Detecting Malicious Code by Model Checking*. Ed. by David Hutchison et al. Springer Berlin Heidelberg, 2005.
- [27] ARM Limited. *ARM® and Thumb®-2 Instruction Set Quick Reference Card*.
- [28] Tim Lindholm et al. *Java Virtual Machine Specification, The*. Oracle, 2012.
- [29] man.cx. *Man (5) elf*.
- [30] David May. *The XMOS XS1 Architecture*.
- [31] MazeGen. *X86 Opcode and Instruction Reference*.
- [32] MIPS Technologies Inc. “MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set”. In: (2011).
- [33] A Moser, C Kruegel, and E Kirda. “Limits of Static Analysis for Malware Detection”. In: *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. 2007.
- [34] R Owens. “Non-normalizable functions: A new method to generate metamorphic malware”. In: ... *CONFERENCE* (2011).
- [35] Mila Dalla Preda et al. *A semantics-based approach to malware detection*. ACM, 2007.
- [36] *radare*.
- [37] Scythale. *Hacking deeper in the system*.

- [38] R K Shyamasundar, Harshit Shah, and N V Narendra Kumar. “Malware: from modelling to practical detection”. In: *ICDCIT'10: Proceedings of the 6th international conference on Distributed Computing and Internet Technology*. 2010.
- [39] Jörg Spieler. *UCDetector*.
- [40] Emeric Vermat. *Dcd — Java.net*.
- [41] R. Wartell et al. “Differentiating code from data in x86 binaries”. In: *Machine Learning and Knowledge Discovery in Databases* (2011).