Platform-Independent Programs

Sang Kil Cha, Brian Pak, David Brumley
Carnegie Mellon University
Pittsburgh, PA, USA
{sangkilc,brianairb,dbrumley}@cmu.edu

Richard J. Lipton
Georgia Institute of Technology
Atlanta, GA, USA
rjl@cc.gatech.edu

ABSTRACT

Given a single program (i.e., bit string), one may assume that the program's behaviors can be determined by first identifying the native runtime architecture and then executing the program on that architecture. In this paper, we challenge the notion that programs run on a single architecture by developing techniques that automatically create a single program string that a) runs on different architectures, and b) potentially has different behaviors depending upon which architecture it runs on. At a high level, a primary security implication is that any program analysis done on a program must only be considered valid for the assumed architecture. Our techniques also introduce a new type of steganography that hides execution behaviors. In order to demonstrate our techniques, we implement a system for generating platform-independent programs for x86, ARM, and MIPS. We use our system to generate real platform-independent programs.

Categories and Subject Descriptors

D.4.6 [**Operating systems**]: Security and Protection

General Terms

Security

Keywords

Malware, Platform-Independent Program, Steganography

1. INTRODUCTION

The world is powered by a variety of computer platforms. Everyday platforms include laptops and desktops running on x86, iPods and cell phones running ARM, and broadband routers and DVD players running MIPS. A typical and often implicit security assumption is that a program is only semantically meaningful on one platform, e.g., an ARM program is typically not a valid x86 program, and vice-versa. This assumption may seem justified since different architectures typically have radically different instruction sets, and potentially even different program encodings. Further, practical evidence of programs that run on multiple architectures has been sparse, with only a few *hand-coded* examples in existence [8, 16]. A poignant description of the difficulty of writing such programs was given by Drew Dean, who described the effort 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." [8] Even the most caffeinated approaches have been met with limited success; they have always been hand-generated and only handled very simple, straight-line code.

In this paper, we challenge the notion that generating a single program string that runs on multiple architectures is inherently difficult. We do so by developing techniques and an infrastructure for automatically generating a *platform-independent program* (PIP) from an existing program or programs. By platform, we mean a hardware or emulated architecture. By program, we mean a bit string that is decoded to a valid set of instructions with operands for a platform. A PIP is a program that runs on two or more platforms without change. In particular, we formulate and address the *PIP generation challenge*. In this challenge, we are given a program b_1 compiled for platform m_1, b_2 for m_2, and so on. A solution to the challenge is a single string b_{pip} such that executing b_{pip} on m_1 is equivalent to b_1, executing b_{pip} on m_2 is equivalent to b_2, and so on. We develop techniques that address the PIP generation challenge by finding a Turing-complete set of platform-independent operations for the set of platforms, allowing any program to be generated in a platform-independent manner.

For concreteness, we demonstrate automatic PIP generation for the x86, ARM, and MIPS platforms. Our techniques allow us to automatically generate a single binary string that i) is a valid program on all three architectures, and ii) can have completely different desired runtime behaviors depending upon which architecture it is ran. We choose x86, ARM, and MIPS because they collectively are used on virtually all commonly encountered networked devices, exhibiting a wide range of typical platform instruction and operand encoding issues. We also extend our techniques to developing OS-independent programs and demonstrating a single platform-independent shellcode that works against Linux, FreeBSD, and Mac OS X. We have made our implementation (minus specific exploits and malware) available at `http://security.ece.cmu.edu`.

There are several security-critical implications of our techniques and implementation, as illustrated in the following usage scenarios:

Steganography. Suppose two users wish to smuggle a program b_{secret} past a dynamic analysis checker for x86. In a steganographic setting, the program execution is the secret the users wish to communicate, and the dynamic analysis checker is the warden. The two users choose a platform m_i different than x86 and a benign program b_{safe} that would pass the dynamic analysis check. Using our techniques, the users can create a single program string b_{pip} such that executing b_{pip} on m_i is equivalent to b_{secret}, yet executing b_{pip} on m_{x86} is equivalent to b_{safe}. More concretely, suppose b_{secret} is malware for x86. Then b_{pip} is a program that is safe on ARM, yet malware on x86. We call this *execution-based steganography* because a) the sender and receiver only need to share the secret (the specific platform specification m_i) to uncover secret functionality and b) the execution behavior will look normal to parties without the secret.

Rogue Updates. Suppose a hacker compromises a system and installs a microcode update. The hacker also installs a PIP-compiled version of a safe program, such as '/bin/ls'. When the safe program is run with the microcode update, it acts like a rootkit. When the safe program is run on a system without the update, e.g., a forensic analysis machine, the program acts benign. We note that the scenario may be more serious when the adversary is powerful, e.g., Intel could design a CPU update that turns safe, signed programs into malware. Note that in such systems the "safe" program may have been installed long ago and is unchanged; thus, security measures, such as digitally signing the code, are insufficient since they only verify the code itself has not been tampered with, not the execution environment.

Exfiltration Protection. Suppose that a secret government agency wishes to protect programs against exfiltration. They create a new instruction set m (potentially as a modification of an existing architecture like x86). They then compile down a program to produce b such that executing b on m outputs the desired behavior, but running it on x86 deletes the program. Such capabilities could help protect against exfiltration to parties unaware b is a platform independent program.

Viruses. Our techniques allow an attacker to write a single virus (e.g., for x86), which is then fed to our algorithm to produce a single platform-independent virus (e.g., for x86 and ARM). When the platform-independent virus is executed by a user attached to a network file system, such as AFS and NFS, the platform-independent virus could then infect all executables, regardless of the architecture. Other machines attached to the network file system can then be infected again by users executing those files.

Shellcode. Control hijack attacks include shellcode that carries out the attackers' intentions, e.g., opening a shell on a network port. Shellcode is also used by some malware to propagate themselves [13]. Our techniques allow an attacker to automatically generate a single string that is valid shellcode for multiple platforms. We show in our experiments that we can take existing x86 shellcode and automatically generate a single binary string that also runs on ARM with equivalent functionality.

New Architectures. A company switches from architecture A to B (e.g., Apple switched from PowerPC to x86), and wishes to distribute a single program that works on both platforms. The current approach is to modify the executable format to include meta-data that either runs the appropriate program for A or B (called "fat binaries"). We show that such meta-data is unnecessary: the program itself can identify at runtime which architecture it is running on and act appropriately.

In general, our techniques mean that any semantic security analysis of a program should be explicitly qualified by the assumed analysis architecture(s). For example, dynamic analysis of x86 programs only tells us about the x86 functionality, and cannot say anything about what would happen if we execute the bit string on another platform. We also show that there are a large number of platform-independent byte sequences, which means syntax-based analysis is also likely to be insufficient to accurately detect platform-independent programs. More specifically, this paper makes the following contributions:

- We develop automatic methods for identifying platform-independent gadgets and assembling gadgets into programs. In particular, gadgets are atomic instructions, though each gadget may have different functionality per platforms. By using gadgets, we build a *Turing-complete language* for creating PIPs. Note the small amount of previous work for PIP generation was completely manual, and only straight-line (e.g., no branching).
- We propose a new type of data-hiding technique, called execution-based steganography.
- We also show that detecting PIP is not straight-forward by developing polymorphic PIP mutation and generation algorithms.
- We provide empirical measurements of the overlap between x86, ARM, and MIPS instruction sets. Our results show a surprising amount of overlap, e.g., about 12% of all 4-byte strings are valid (thought likely different) x86, MIPS, and ARM instructions.

Scope. We generate bit strings that are valid programs when executed on multiple platforms. At a high level, this means that even though one can figure a program string is valid for platform x, our results show one cannot conclude behaviors on the platform x are the only behaviors of the program.

However, there are additional factors that may prevent a platform-independent program from running in a specific machine. For example, the ELF executable file format has a field that specifies the intended CPU for the program. Such checks may afford some protection, e.g., Linux versions we tested will not run programs that did not have the bit set to the underlying architecture. However, the COFF format [11] does not include such a flag. Thus, one small ramification and contribution is that architecture checks are important to security, even though they were likely not intended as a security measure. Nonetheless, the above motivating scenarios show that automatically generating platform-independent programs is a relevant and important security consideration.

2. PROBLEM STATEMENT

2.1 Notation

A key aspect in our work is encoding programs so that they run on multiple platforms. In particular, we need a vocabulary for specifying particular platforms, programs for those platforms, and specific offsets within programs. We adopt the following. A program is a string b taken from some alphabet Σ, i.e., $b \in \Sigma^*$. We use subscripts to distinguish particular programs, and superscripts to refer to particular offsets in the program. For example, b_1^5 refers to the program b_1 starting at a byte offset of 5. In our implementation, we focus on executable programs where $\Sigma = \{0, 1\}$ and specific programs are binary strings.

A platform is specified by a machine m for executing programs. We focus on instruction set architecture (ISA) machines, although a machine could also be an interpreter or emulator. We again use subscripts to distinguish particular machines. In particular, we let $\{m_{x86}, m_{mips}, m_{arm}\}$ denote x86, MIPS, and ARM, respectively.

The execution of program b_i on machine m_j is denoted as $m_j(b_i)$. We denote when a program b_i is compiled to run on a specific machine m_j by the tuple (b_i, m_j). We denote by $\perp$ that b is not properly formatted for machine m. For example, $m_{x86}(\texttt{90902890}_{16}) = \perp$ because $\texttt{90902890}_{16}$ is not a valid x86 string. [1]

A central component of our work is creating a single program that produces semantically identical results when ran on two or more platforms. We denote by "=" (the equal sign) when two executions are semantically identical, i.e., $m_{x86}(b_1) = m_{arm}(b_2)$ means that executing b_1 on machine m_{x86} results in the same output as executing b_2 on m_{arm}. Equality could be determined by showing an isomorphism between the final state spaces of two executions, a literal comparison between screen outputs, and so forth. In our implementation, equality is determined by an isomorphism on the state after each atomic step of execution. We stress that, in our approach, we are generating platform-independent programs that have semantically identical outputs by construction.

2.2 Problem Definition

A platform-independent program is a program that executes on multiple machines. Thus, the only requirement for a platform-independent program b is that it is a valid program encoding and not $\perp$. More formally

Definition 2.1. ***(Platform-Independent Program)*** *A string b is a* platform-independent program (PIP) *for machines m_1 and m_2 when*

$$m_1(b) \neq \perp \Leftrightarrow m_2(b) \neq \perp$$

Platform independence can be extended to lists of machines in the obvious way.

In this paper, we are interested in general techniques for generating PIPs. For example, when Apple migrated from PPC to x86, they may have wanted to produce a single program b_{pip} that behaved the same as separately compiled programs b_{x86} and b_{ppc}. However, one may also want the PIP's behavior to be *different* for each platform, e.g., a malware author may want a single program that is equivalent to "hello world" on x86 and malware on ARM.

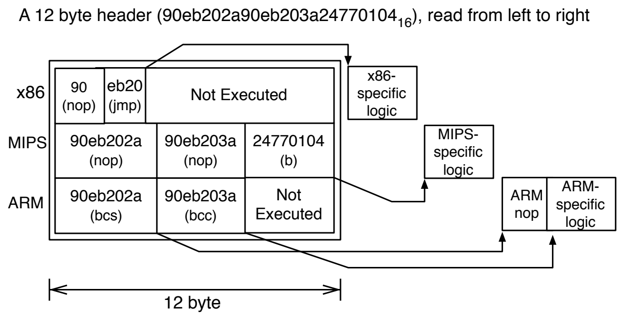

Figure 1: Self-identifying program concept.

In order to allow for a wide variety of scenarios, in our problem statement we assume we are given a program b_i for each architecture m_i. Each b_i acts as a specification of the desired behavior for the generated PIP for platform m_i. The goal is to output a single program b_{pip} that has the same behavior on each platform m_i as b_i, e.g., $m_i(b_i) = m_i(b_{pip})$. More formally, we define the PIP generation challenge as

Definition 2.2. ***(PIP generation challenge)*** *Given a list of n programs, machine pairs (b_i, m_j), the* PIP generation challenge *is to automatically generate a single program b_{pip} such that*

$$\forall (b_i, m_j) : m_j(b_i) = m_j(b_{pip})$$

The PIP generation challenge takes in a list of programs and outputs a single PIP that i) meets the definition of a PIP, and ii) meets the desired functionality specification (as given by b_i) for each architecture. Note that the PIP generation challenge allows for both cases where the final program b_{pip} has the same functionality on all architectures, as well as different functionalities, depending on the architectures.

3. APPROACH

3.1 Gadgets

The central intuition behind our approach is that there are program strings which are valid for multiple platforms, but their behaviors depend upon which platform executes these strings. We call these program strings *gadgets* [2], which are chunks of program logic. We solve the PIP generation challenge by constructing the desired behavior using gadgets. One of the main challenges we address is finding enough gadgets so that we have Turing-complete functionality for each platform. For example, one gadget might be `xor` on x86, while `add` on ARM. We need to make sure we have enough gadgets to cover all needed operations for each platform. By creating a Turing-complete set of gadgets, we ensure we can address the PIP challenge in the broadest set of scenarios possible.

Gadgets themselves consist of two parts: a platform independent *gadget header* that identifies the underlying platform, and a platform-specific *gadget body*. Figure 2 shows the overall structure of a gadget. A gadget header h is a platform-independent jump statement whose jump target

[1] $\texttt{90902890}_{16}$ decodes to `nop; nop; .byte 0x28 ; nop` on x86, which is not a valid instruction sequence.

[2] Our gadgets are chunks of logic and are not to be confused with gadgets in return-oriented programming [17].

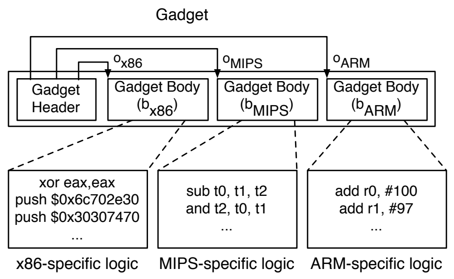

Figure 2: Gadget structure: single gadget.

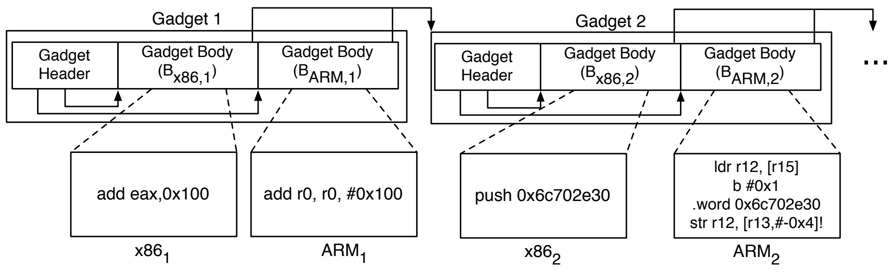

Figure 3: Gadget structure: multiple gadgets (instruction-by-instruction).

offset depends upon the executing platform. More specifically, h begins with zero or more semantic nops, followed by an architecture-identifying jump for all architectures. We discuss gadget headers whose prefix is not a nop in § 8. Figure 1 depicts a gadget header for x86, ARM, and MIPS. The header is the 12-byte string $90eb202a90eb2023a24770104_{16}$ which decodes to

- A nop followed by the jump `jmp 0x22` to the relative jump target offset o_{x86} on x86.
- A nop followed by the jump `b 0x1dc94` to offset o_{mips} on MIPS.
- Two conditional jumps `bcs` and `bcc`, one of which will always be taken, to offset o_{arm} on ARM.

The gadget header transfers control to a platform-specific gadget body. Each gadget body will be at a different offset from the gadget header. For example, in Figure 1, the gadget header transfers control to b_{x86} at offset o_{x86}, b_{mips} at offset o_{mips}, and b_{arm} at offset o_{arm}.

3.2 Generation Algorithm

In the PIP generation challenge, we are given as input a program specifying the desired behavior for each architecture. At a high level, our approach matches each disassembled instruction for a program with a gadget. The gadgets are assembled into the final PIP. Note that we describe methods for solving the challenge at the instruction level in order to be the most general, e.g., one could implement a PI compiler that produces binary code PIP from a given source. It is straight-forward to modify our approach to handle coarser granularities, such as the per-program level.

For example, suppose we are given two programs b_{x86} and b_{arm} of the same length, we disassemble each to get a pair of lists of instructions $[b1_{x86}, b2_{x86}, ...]$ and $[b1_{arm}, b2_{arm}, ...]$. We then find a gadget b_{pip} such that executing it on x86 is the same as executing $b1_{x86}$, and on ARM is the same as executing $b1_{arm}$. We then assemble the gadgets into a complete program implementing the correct control flow. For example, if an instruction in ARM branches to address a_1, we need to make sure the gadget for the instruction branches to the corresponding gadget for address $a1$ when executed on ARM.

More formally, our overall approach to solving the PIP generation challenge consists of four algorithms: HEADER-INIT, DISASSEMBLE, GADGET-GEN, and MERGE. We use B to denote the type of programs, M for machines, H for headers, and G for gadgets.

Header-Init: (l, M list) → H list. HEADER-INIT returns a list of gadget headers up to maximal length l bytes for the given list of machines. We assume l is an overall system parameter. In our implementation, HEADER-INIT is a pre-computation step which is ran once. One challenge we address in our system is efficiently finding such headers.

Disassemble: (B,M) list → ((B,M) list) list. DISASSEMBLE first disassembles each input program $[(b_1, m_1), (b_2, m_2), ...]$ into a list of instructions $[([b_1^{a_1}, b_1^{a_2}, ...\ b_1^{a_n}], m_1), ([b_2^{a_1}, b_2^{a_2}, ...\ b_2^{a_n}], m_2), ...]$. (Without loss of generality, assume that the number of instructions in each list is identical because we can pad shorter lists with junk instructions). Then DISASSEMBLE pairs instructions at each address, and outputs $[[(b_1^{a_1}, m_1), (b_2^{a_1}, m2), ...], [(b_1^{a_2}, m_1), (b_2^{a_2}, m_2), ...], ...]$.

Gadget-Gen: (H list, (B,M) list) → (G, (B,M) list). GADGET-GEN takes a list of programs, machine pairs (b_1, m_1), (b_2, m_2), ..., and the set of headers for the platforms under consideration, and generates an appropriate gadget g such that

$$\forall i. m_i(b_i) = m_i(g)$$

GADGET-GEN returns the tuple $(g, (b, m)$ list) where g is the gadget and the $((b, m)$ list) is the list passed in.

Merge: ((G, (B, M) list) list) → B. MERGE assembles gadgets into the final PIP as $b = g_1||g_2||...||g_n$ where $||$ denotes concatenation, as shown in Figure 3. When there is more than one tuple given, MERGE handles control flow between gadgets, e.g., when the gadgets themselves are constructed from single instructions in a larger program. MERGE "fixes" the control flow in the PIP b to match control flow between the list of instructions, or, more generally, programs:

- MERGE rewrites conditional jump targets. Whenever an input program $m_i(b_i)$ results in a conditional jump to address a, then MERGE needs to rewrite the jump of the corresponding gadget body g_i target to be the new gadget containing a.
- MERGE ensures appropriate sequential control flow by adding a jump at the end of each gadget (for each machine type) to the next gadget.

One of the significant challenges we address in our design and implementation is ensuring that we handle all direct jump types in the program. (We do not currently handle computed jump targets or self-modifying code but discuss possible directions in § 8.)

Given a list of programs (b_i, m_i), the overall algorithm for solving the PIP challenges is:

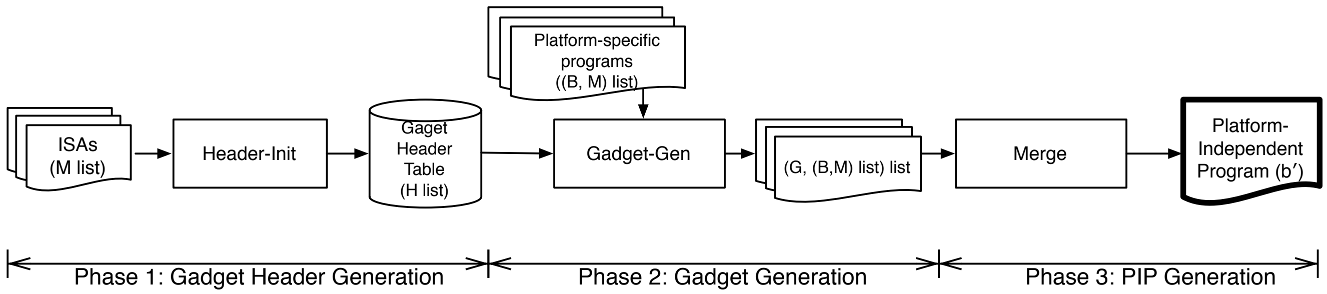

Figure 4: Overview of platform-independent program generation components and algorithm steps.

```
1. let PIP-Gen input:($b_i$, $m_i$) list  =
2.   let machines = List.map snd input in
3.   let headers = HEADER-INIT (max, machines) in
4.     MERGE([GADGET-GEN headers input])
```

Given two programs (b_1, m_1) and (b_2, m_2), the PIP generation challenge is solved by first finding headers for m_1 and m_2 (line 3), creating a gadget for b_1 and b_2, and assembling the gadget into the final PIP. Figure 4 describes the high-level algorithm of the PIP generation.

In the above definitions, PIP-GEN passes MERGE a whole program at a time. It is trivial to modify the algorithms to perform PIP generation at a finer-grained level, such as the per-instruction or per-block level. For example, we can generate gadgets at the per-instruction level by, first, running a disassembler on the input, then running GADGET-GEN and MERGE:

```
1. let PIP-Gen input:($b_i$, $m_i$) list =
2.   let machines = List.map snd input in
3.   let headers = HEADER-INIT (max, machines) in
4.   let insts = DISASSEMBLE input in
5.   MERGE (List.map (GADGET-GEN headers) insts)
```

PI-Translate. In our implementation, we employ an optional algorithm, called *PI translation* (§ 4.4). PI translation takes as input a program (b_1, m_1) and a list of desired target architectures $(m_2, m_3, ...)$. The translation procedure outputs a semantically equivalent program b_i for each target architecture, and then runs the regular PIP generation solution. The PI translation algorithm has the practical benefit that we can leverage one binary as a specification for the behavior on multiple platforms for PIP generation.

4. RG DESIGN

In this section we describe the design of RG, our architecture for addressing the PIP challenge. [3] Figure 5 shows the overall design of RG. We first look at how RG efficiently implements the HEADER-INIT, GADGET-GEN, MERGE, DISASSEMBLE algorithms. We also describe RG's translation capabilities, as well as additional problem-specific enhancements and considerations.

4.1 Header-Init: Finding Gadget Headers

At a high level, RG's HEADER-INIT algorithm consists of two steps. First, for each architecture, it finds all sequences of instructions up to length l of the form $(nop)^*(jump)(.)^*$. We call these potential headers. Second, it computes the set

[3]RG is named after Rube Goldberg machines, which accomplish tasks with a (sometimes convoluted) number of connected gadgets.

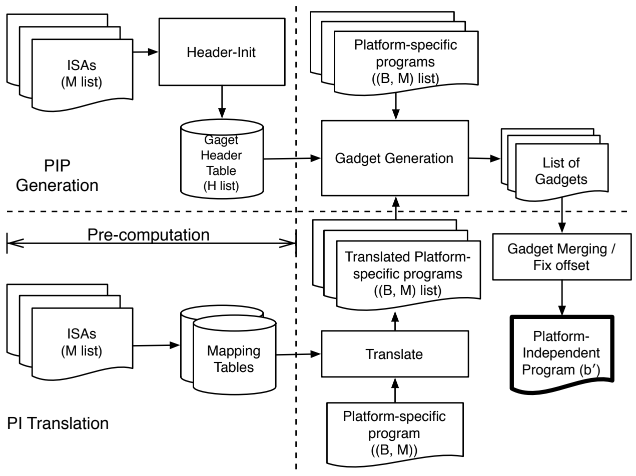

Figure 5: RG Architecture.

of gadget headers in common between all architectures by computing the conjunction of all potential header strings. RG uses a template-based algorithm to speed up finding potential headers and enumerating the final header set.

Gadget headers are found once during precomputation in RG, then stored in a database for subsequent steps. Gadget headers are represented internally by RG as tuples $(b, (m_i, o)$ `list`), where b is the binary gadget header string, and each (m_i, o) pair is the size of jump offset for machine m_i, so we can place a machine-specific gadget body at the corresponding address in the GADGET-GEN phase. For instance, $(\text{eb0200ea}_{16}, [(m_{x86}, \texttt{0x4}), (m_{arm}, \texttt{0xbb4})])$ represents a header string eb0200ea_{16}. When it is executed on x86, it transfers control to the current instruction pointer plus 4 byte from the header, and on ARM transfers control to the current instruction pointer plus 2,996 byte.

Below we first describe the types of nop and jump statements we look for. We then describe our algorithm for finding potential headers.

4.1.1 *nop Instructions*

Every gadget header begins with zero or more instructions that are semantically a nop. In our setting, a nop is a sequence of one or more instructions that may advance the program counter, but does not otherwise change the state of the machine. RG uses four heuristics for finding nops:

- Instructions that move the same value from a register to the same register, e.g., a consecutive push and pop instruction for the same register (`push eax; pop` eax), exchange instructions (xchg), and move instructions whose operand are the same register (mov eax, eax).

- A sequence of jumps (branches) to the next instruction, e.g., `i:  jmp i+1`, where the instruction at address i will simply increment the program counter. Note we can use conditional branches to the next instruction since it does not matter whether or not the branch is taken.
- Identity arithmetic operations such as `addi $t0,$t0,0` on MIPS. Note we avoid such operations on x86 because the instruction may have side-effects on status registers.
- Miscellaneous platform-specific heuristics. For example, MIPS has a special register r0, which always maintains the zero value. Any instruction that assigns a value to r0 is a semantic nop.

4.1.2 *Jump Instructions*

RG uses syntactic and semantic jump instructions. Syntactic jumps are straight-forward to find. RG finds semantic jumps by looking for mutually exclusive branch instructions where the conjunction of branch guards is always true [23]. For example, the bcs and bcc instruction in Figure 1 are mutually exclusive. In some cases, we need to pad the jump targets during gadget generation with nops, as done in Figure 1.

4.1.3 *Header Generation Algorithm*

One naive approach for finding potential headers is a brute-force search through all possible instructions up to some fixed length. For example, if $l = 4$ bytes, the brute force approach enumerates each 32-bit number from 0 to 2^{32}, decodes it, and checks if it is a potential header. Initially, we took this approach and found all headers up to 32-bits long using several computers over several days. We have now optimized the search for potential headers by defining *header templates*, described in the following steps:

Step 1: Making a list of header templates. These templates are regular expressions over bit strings. We generate the possible gadget header templates of fixed length for each machine based upon the list of nop and jump instructions and the pre-defined header length. The gadget header is the concatenation of nop instruction, jump instruction, and any characters: $(nop)^*(jump)(.)^*$. Since we find fixed-length gadget headers, the last dot character of the regular expression is to pad extra bytes after a jump instruction. This allows us to find headers of arbitrary length if needed.

Step 2: Computing the intersection of templates. We compute the intersection of all the possible templates of given machines. Note since the templates are regular expressions, intersection is well defined. For example, given a header template [4] (90eb....) for x86, and (......ea) for ARM, the intersection is (90eb....) $\wedge$ (......ea) = (90eb..ea). The advantage of using templates is that we do not need to explicitly enumerate all the possible instructions to find the possible gadgets.

[4]We must represent all the instructions using a regular expression where the alphabet is binary numbers. However, we take hexadecimal notation in this paper for space efficiency.

Step 3: Enumerating gadget headers. Once we have the complete list of intersection templates from all the combinations of given machine templates, we then enumerate all the possible gadget headers from the intersection templates. We store the headers into a database with the branch-offset for each machine. We can precompute the gadget header databases and simply fetch the gadget headers from the database without additional computation in the future.

4.2 Disassemble and Gadget-Gen

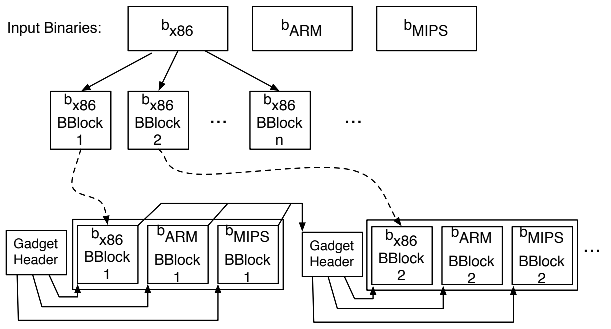

Figure 6: Constructing gadgets and linking them into a PIP.

RG disassembles programs into a sequence of instructions, builds a control flow graph, then performs gadget generation on the CFG basic blocks, as shown in Figure 6.

GADGET-GEN takes a list of programs $[(b_1, m_1), (b_2, m_2), ...]$ for different architectures and outputs a single gadget. Let $|b_i|$ denote the length of b_i. Then a suitable gadget header may have a jump to b_1 just after the header, to b_2 at offset $|h| + |b_1|$, to b_3 at offset $|h| + |b_1| + |b_2|$, and so on. When $|b_i|$ is large, it becomes hard to find an appropriate gadget header, e.g., in our experiment, when $|b_i|$ was megabytes in size it became difficult to find a header with a large enough jump target offset. Basic blocks, instead of whole programs, make it much more likely RG can find an appropriate header. Optionally, RG can operate at the per-instruction level (by changing the disassembler to return instructions instead of basic blocks). The per-instruction level requires more gadgets, which in turn creates somewhat larger and slower PIPs. Additionally, we discuss another possible way of constructing PIP in § 8, which does not require RG to perform the DISASSEMBLE and GADGET-GEN steps even though $|b_i|$ is larger than the jump offset.

4.3 Merge

MERGE takes a list of gadgets, and constructs the final PIP by ensuring each instruction jumps to the next appropriate gadget, and that each input program terminates appropriately by inserting an appropriate exit call at the end of the last gadgets. Abstractly, MERGE also takes each input program as a sequence of instructions. MERGE uses this sequence to fix up conditional jumps. In RG, we take the CFG generated by DISASSEMBLE since we already generated control flow information to find basic blocks. Figure 6 depicts linking up gadgets.

For each gadget, RG does the following:

- Fixes direct jump targets for machine m_i. RG changes the offset of direct jump instruction, so it points to the

ARM	r0	r1	r2	r3	r4	r5	r6	r7
x86	ebx	ecx	edx	t	t	edi	esi	eax

ARM	r8	r9	r10	r11	r12	r13	r14	r15
x86	t	t	t	ebp	t	esp	t	eip

Table 1: x86-ARM register mapping table.

x86	**ARM**
ADD r32, Imm	ADD r_i, r_i, #Imm
AND $r32_i$, $r32_j$	AND r_i, r_i, r_j
CMP $r32_i$, $r32_j$	CMP r_i, r_j
INT 0x80	SVC 0x0
JMP r32	MOV r15, r_i
PUSH Imm	LDR r_t, [r15] .word 0xea000000 .word Imm STR r_t,[r13,#-0x4]!

Table 2: Partial x86-to-ARM instruction mapping table.

target gadget in the control flow graph for the original m_i program.

- Fixes conditional branch targets for machine m_i. This involves two steps: one for the branch target that would be executed if the branch predicate is true, and the other for the fall-through semantics when the branch predicate is false. RG handles the former by changing the offset of the branch instruction to point to the appropriate next gadget in the CFG, and the latter by appending a direct jump to the next gadget in sequential execution.

4.4 PI Translation

RG works at the binary level, thus cannot simply recompile b_1. Binary-level solutions are attractive in security because it allows us to translate malware for machine m_1 into PIPs for multiple machines. The following is RG's general algorithm for solving the PI translation challenge:

1. Precompute register and instruction mapping tables from m_1 to the desired architectures.
2. Translate each instruction in the input binary (b_1, m_1) to the desired machines $(b_1, m_1), (b_{1'}, m_2), (b_{1''}, m_3), \ldots$.
3. Run the PIP generation algorithm on the output of step 2.

RG currently implements the mapping from x86 to ARM. We focus on static interpretation from x86 to ARM because previous work has not focused on these architectures, e.g., UQBT [7] and DAISY [9], yet, in our problem setting, ARM is an important domain. Currently, RG does not handle typical problems for static binary translation, such as indirect jump address problem, but we discuss possible directions in § 8.

RG takes care of the ABI differences between operating systems on different architectures. In Linux on x86, a system call is accomplished by first placing the relevant system call number in the `eax` register and the arguments in `ebx, ecx, edx` registers, and then issuing an `int 0x80` instruction. However, in Linux on ARM, a system call is accomplished by placing the system call number in the `r7` register, and arguments are passed by `r0, r1, r2` registers. Thus, we match each register accordingly when we generate the register mapping table. A partial description of RG's register mapping is shown in Table 1, where "t" means that the corresponding register on ARM can be used as temporary register.

The instruction mapping table is generated for the atomic operations, such as `add` and `mov` on x86. Table 2 shows typical instructions as mapped by RG. The table contains the semantically equivalent operation mappings between two machines. For example, `mov` $r32_d$, $r32_s$ in x86 is mapped into `mov` $r32_d$, $r32_d$, $r32_s$, where the subscript d means destination and s means source. The `push immediate` in particular becomes the sequence of ARM instructions, as shown in the table.

4.5 Polymorphism

One of the significant application domains for PIPs is generating multi-platform malware. In such domains, syntax-based signatures are typically the most widely deployed defense mechanisms. We have adapted RG to generate polymorphic PIPs, in order to evade syntax-based signature defenses.

RG is the first platform to generate PIPs, where there is control flow between gadgets. As a result, RG can be augmented to generate polymorphic variants in a number of typical ways. Consider the following examples:

- Changing jump offsets for gadgets, i.e., we can move gadget body code b_i to different offsets by using different headers with appropriate jump target offsets.
- Padding gadgets and programs with semantic nops.
- Reordering how blocks are laid out as a sequence of bytes. Compilers lay out basic blocks on disk via a compiler trace generation algorithm [3]. [5] However, there is no single canonical layout, as any permutation of basic blocks is valid. This approach requires that we fix up jump targets in a way similar to what is already performed by `Merge`.
- Flipping branch conditionals. We can simply change the branch type and flip the offset, e.g., change `jo` to `jno` on x86 and fix up the appropriate jump targets. Alternatively, we can negate the branch conditional (as above) and negate the branch conditional.
- Replacing instructions with semantically equivalent ones. RG already analyzes the type of instructions via translation. We can build a similar table for translating instructions to semantically equivalent variants, e.g., by replacing all x86 multiplications by 2 with left shifts by 1.

We also note that, by definition, behavior-based detection is not straight-forward, as a PIP may have benign behaviors under one platform, but malicious behaviors when executed on another.

4.6 Platform-Independent Code Injection

A second significant domain for PIPs is shellcode. In real scenarios, we must consider two important properties: the size of the shellcode, and removing NULL bytes.

The size of shellcode string is important because many attacks require that both fit within a limited amount of space. Currently, RG uses only a single gadget when creating PI shellcodes, in order to minimize the overhead due to gadget headers. Suppose we are given shellcodes (b_1, m_1) for a vulnerable server on m_1, and (b_2, m_2) for the sample vulnerable server running on m_2 (potentially produced via

[5]The concept of trace here should not be confused with dynamic execution traces. In this context, trace generation is purely a static analysis.

translation). RG will produce a single PI shellcode $h||b_1||b_2$, instead of disassembling b_1 and b_2 and introducing headers for each instruction or block. In our experiments, this is sufficient to produce real PI shellcode. Nonetheless, it is possible that the resulting multi-gadget PI shellcode is larger than the size of the writable memory, and our shellcode may not work while the minimum-sized one would. We leave as future work optimizations to address this issue.

A second issue is that shellcodes typically cannot have NULL bytes, but PIP generation may introduce them. RG handles this by performing an additional processing step that replaces any NULL bytes introduced by gadget generation (e.g., in the gadget header) with semantic equivalents that do not contain NULL bytes. For example, the ARM encoding for instructions using `r0` will introduce NULL bytes. In order to generate shellcode free of NULL bytes, we encode the shellcode and prepend a corresponding decoder at the beginning of the shellcode to eliminate the NULL bytes, as seen in [10].

5. IMPLEMENTATION

RG is currently implemented in about 5k lines of a mixture of C++ and Ruby. RG uses the GNU Binutils opcode library to decode the binary string into the assembly language. RG consists of three command-line programs. The gadget finder program finds all the possible 4-byte, 8-byte, and 12-byte gadget headers [6] and constructs the table in a MySQL database. The PI generator program merges two binary programs that run on different architectures into a single ELF binary. The PI translator program takes an x86 binary and outputs a PI program for x86 and ARM.

6. EVALUATION

We evaluated our techniques and RG for PIP generation on three different platforms: x86 (IA32), ARM (specifically, ARM7TDMI), and MIPS (specifically, MIPS32). We focused on these platforms because, currently, they appear to be the most popular in typical security-relevant scenarios. We performed PI translation on x86 and ARM.

Our evaluation highlights three significant points:

1. Section 6.2 shows that while there are a large number of headers (e.g., up to 66,092 4-byte headers for ARM and x86), they are relatively rare to find (e.g., 66,092 is about 0.0015% of the total possible 32-bit sequences). Our template-based strategy was necessary to efficiently explore such a large state space. Headers must be at least 12-bytes for tri-platform gadgets, but 4 byte headers exist for bi-platform gadgets. One interesting point is we found valid headers for architectures that use different endianness, e.g., MIPS little endian and ARM big endian.
2. Section 6.3 shows that RG can find a Turing-complete set of gadgets and create realistic PIP by creating PIPs at the instruction level for a CPU-intensive program demonstrating conditional and looping control flow (Prime Checker), the standard "hello world" program, and a number of popular shellcode used in practice.

[6]We choose the size of gadget header as multiple of four, because the size of a single instruction of MIPS and ARM is four.

Architectures	# of valid instructions
x86 + ARM	815,891,149
x86 + MIPS	908,451,552
ARM + MIPS	1,918,735,696
x86 + ARM + MIPS	528,989,737

Table 3: Count of 32-bit numbers successfully decoded as an instruction on multiple platforms.

3. Section 6.4 demonstrates that RG can be used to create steganographic malware that is safe on ARM, but a virus on x86.

Machines. PIP generation and translation steps were performed on an Intel Pentium D CPU 2.80GHz with 4GB of RAM. We tested the resulting PIPs on a Nokia's N800 and Apple's iPhone for ARM, SGI's O2 for MIPS, and the above Intel machine for x86. All the machines were running on the Linux (Debian) or Linux equivalent OS, e.g. Maemo in N800 is based on the Debian Linux. In order to test performance, we compared the running time of all PIPs on the x86 hardware above using QEMU version 0.9.1.

6.1 Instruction Validity

We first calculated the approximate instruction density for each architecture. We enumerated all instructions up to 32-bits in length, and calculated the instruction density as the number n of valid instructions divided by 2^{32}. The instruction density was 90.12% for ARM, 68.46% for MIPS, and 32.69% for x86.

We then calculated the number of 32-bit instructions that are valid for 2 or more architectures. Table 3 shows our results. Overall, 12.31% of all 32-bit numbers decode to a valid instruction sequence on all 3 architectures. From this, we draw two conclusions. First, the 12% overlap in instructions indicates it should be relatively easy to find platform-independent code. Second, as a consequence of the larger overlap, detecting platform-independent code likely requires more than simply looking for known fixed sequences.

6.2 Gadget Headers

Nops. Recall from § 4.1 that a gadget header consists of zero or more nops followed by a machine-identifying jump. An atomic nop is the smallest basic unit of instructions that decodes semantically as a nop. Since multiple atomic nops can be strung together, roughly speaking, a limiting factor in the diversity of nop strings is the number of atomic nops. We found 326 atomic nops for x86, 241 for ARM, and 14,709,948 for MIPS. MIPS has a large number of nops because arithmetic instructions, unlike x86 and ARM, do not set processor status flags. Thus, on MIPS, any arithmetic operation that does not change the value, e.g., a value plus zero, can be used as a nop in our setting.

Number of Gadget Headers. We enumerated gadget headers using our template-based approach for maximum size sequence $n = 4, 8$ and 12. We found headers for two-machine combinations, when $n = 4$ and 8. We show the number of 4 byte headers found for different architecture combinations in Table 4, and the number of 12 byte headers in Table 5. Due to space, we leave out 8 byte headers for machine pairs. The first header we found for three architectures — x86, little-endian ARM, and little-endian MIPS — required 12 bytes. RG found 4×10^{14} total 3-architecture gadget headers. It took 0.07 seconds to find 4-byte gadget header templates, 16 seconds for 8-byte templates, and 7 hours for 12-byte templates.

We observe that there are a large number of headers. This again supports the idea that finding headers is not as difficult as previously believed. We can also find gadget headers even across different endian architectures. For example, Table 4 shows that there are 768 valid headers for big-endian ARM and little-endian MIPS. Finally, we observe that RG's template-based approach to finding headers is an important component in generating platform-independent programs. There are 2^{96} possible 12-byte instruction sequences. Enumerating all of them to find headers is computationally infeasible. Our template-based approach reduced the search space significantly, while still finding a large number of headers.

	x86	ARM(L)	ARM(B)	MIPS(L)	MIPS(B)
x86	N/A	66,092	0	774	0
ARM(L)	66,092	N/A	65,536	0	768
ARM(B)	0	65,536	N/A	768	0
MIPS(L)	774	0	768	N/A	6
MIPS(B)	0	768	0	6	N/A

Table 4: Number of 4-byte long gadget headers. L represents little endian and B represents big endian.

	x86	ARM(L)	ARM(B)	MIPS(L)	MIPS(B)
x86	N/A	7.9×10^{18}	4.3×10^{9}	5.5×10^{18}	4.3×10^{9}
ARM(L)	7.9×10^{18}	N/A	8.6×10^{15}	5.4×10^{16}	1.3×10^{19}
ARM(B)	4.3×10^{9}	8.6×10^{15}	N/A	1.3×10^{19}	5.4×10^{16}
MIPS(L)	5.5×10^{18}	5.4×10^{16}	1.3×10^{19}	N/A	1.7×10^{18}
MIPS(B)	4.3×10^{9}	1.3×10^{19}	5.4×10^{16}	1.7×10^{18}	N/A

Table 5: The number of 12-byte gadget headers.

6.3 Platform-Independent Programs

HelloWorld. One way to view gadgets is as a new programming language for multiple platforms. As such, it is fitting to create a "hello world" program, shown in Appendix, Figure 9. "hello world" demonstrates a large number of features, including stringing together multiple gadget bodies, a platform- independent `write` system call, etc. In our experiment, we confirmed that the program ran on x86, ARM, and MIPS. Note that the Figure shows a complete ELF file generated from our PI generator. In order to run on each architecture, we changed the one-byte ELF header of a CPU-type field to the appropriate value.

Prime Checker. The prime checker program is a CPU-intensive program that implements the Sieve of Eratosthenes to find all prime numbers up to 3,000,000. We compiled the C program for both x86 and ARM, and fed the resulting binaries to the PI generator in RG. Also, we used a variant of a gadget header that does not require nop instructions to minimize the size of the gadget. This is possible because we can ignore the change of the machine state if the gadget header is located at the beginning of a program (discussed in § 8). In addition, we measure the performance of PIPs in § 6.5.

Shellcode. We experimented with creating PI shellcode using 8 standard shell code examples from Exploit-DB [1]. The selected shellcodes include local shell, bindshell, and reverse bind shell. In particular, the bindshell and the reverse bindshell are extremely popular in practice because they allow an attacker to have remote access to the victim machine.

We first confirmed that RG could solve the PIP challenge for the 8 shellcodes. We manually created semantically equivalent shellcodes for ARM for each shellcode. We fed the 8 ARM, x86 shellcode pairs to the PI generator, and obtained 8 PI shellcodes as outputs. An overview of the size and generation time is shown in Table 6. We verified that each shellcode successfully executed the appropriate shell on both x86 and ARM. The PI shellcode generated for x86 and ARM that executes bind shell is listed in Appendix, Figure 8.

Name	Original Size (byte)		PIP Size (byte)	Gen. Time (sec)
	x86	ARM		
Local shell code (23 byte)	23	72	99	0.028
Local shell code 2 (32 byte)	32	84	120	0.024
Local shell code 3 (40 byte)	40	72	116	0.028
Binding shell code	171	244	428	0.028
Reverse binding shell	155	236	416	0.028
Killall5	34	96	135	0.028
Flush iptable	40	120	164	0.028
Fork bomb	7	12	23	0.028

Table 6: Shellcode PIP generation results.

Name	PIP Size (byte)	Generation Time (sec)
Local shell code (23 byte)	300	0.068
Local shell code 2 (28 byte)	320	0.080
Local shell code 3 (40 byte)	672	0.113
Binding shell code	2564	0.431
Reverse binding shell	2592	0.439
Killall5	388	0.088
Flush iptable	532	0.077
Fork bomb	196	0.076

Table 7: Shellcode PI translation results.

We next confirmed that RG could solve the PI translation challenge by automatically generating ARM shellcode from the x86 shellcode. Table 7 shows the size of each of the PI-shellcode and the generation time. The size is larger than in the PI generator's setting because, in the translation setting, RG translates each input instruction into a gadget. In the setting, we only need to generate a single gadget.

Vulnerabilities. We installed and created exploits and PI shellcode for two vulnerable programs: Snort 2.4 [21] and the iPhone's coreaudio library [22]. For Snort, we installed the vulnerable version on Debian Linux on our ARM and x86 machine. We then created an exploit and used our PI shellcode. We confirmed that the shellcode worked as intended on a real exploit.

In the iPhone experiment, we created an exploit and then used RG to generate ARM shellcode from the x86 remote bindshell shellcode. We confirmed that the PI shellcode, paired with the iPhone coreaudio exploit, gave us the expected shell.

6.3.1 *OS-Independent Shellcode*

We have used a variant of our technique to generate OS-independent shellcode for Linux, FreeBSD, and OS X. OS-independent shellcode uses a gadget header that identifies the running OS. We employ two heuristics. First, we have a gadget header that checks the address of the running process, which will differ for each OS. Second, we designed gadget headers that checked the set of defined system calls. For example, system call number 395 is not defined in Linux, whereas it is used as `getlcid` system call in FreeBSD and OS X. If we use this system call number in Linux, we get a negative return value in `eax` register, but we get positive return value on FreeBSD and Mac OS X.

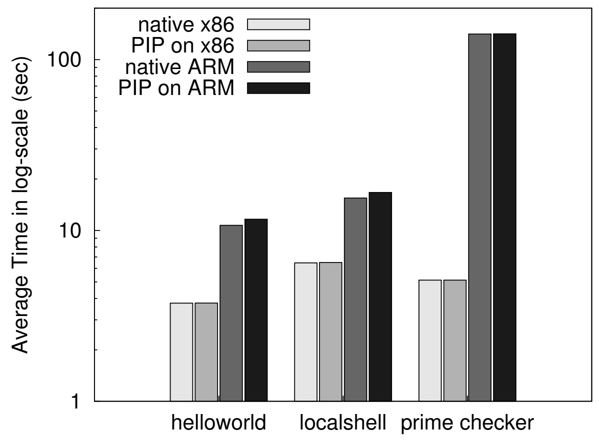

Figure 7: Performance comparison of PIPs.

We have used our platform to generate shellcode that works on all three platforms. Our experiments show that both OS-independent and platform-independent shellcode is possible. Our results indicate that attackers only need to care about their exploits, but not shellcodes, regardless of the targeting OS or architecture.

6.4 Execution-Based Steganography

Recall from § 1, the security scenarios where a PIP hides its execution behavior, e.g., execution-based steganography. In order to show that these scenarios are feasible, we generated a PIP that acts as a simple "hello world" on ARM, but as a virus on x86. We first created a virus for x86 (`RG.poc`) from scratch, which infects ELF files on the system whenever it executes. Then we fed the virus into our PI generator with a "hello world" program compiled for ARM to generate a PIP. The resulting PIP acts as a simple virus on x86. However, if we change the ELF header of the PIP to indicate the file is for ARM, then the program runs as a simple "hello world" program on ARM. It is trivial to generate a platform-independent virus by simply substituting ARM virus for the "hello world" program. Thus, we conclude that both hiding the execution behaviors of a program and creating platform-independent viruses by generating a PIP is possible.

6.5 Performance Comparison

We measured the running time for executing three PIPs on x86 and ARM: the prime checker, `helloworld`, and `localshell` (from the previous sections). We ran `helloworld` and the `localshell` 100 times on ARM, and 1000 times on x86 in a loop to measure the time, and all the results were averaged over 10 runs. Figure 7 shows the run-time performance on our x86 Pentium D machine running the native x86 as baseline. We ran the PIP natively, and ran the native ARM using a QEMU emulator on the same machine. The result shows 0.1% and 5.5% of average performance degradation on x86 and ARM, respectively. We think that the performance degrade on ARM is greater than on x86 due to the QEMU's emulation overhead. Thus, we conclude that PIPs do not degrade performance significantly.

7. RELATED WORK

Multi-Platform Executables. There has been considerable effort toward running an executable on multiple platforms. Our approach is significantly different from previous approaches because we do not require additional meta-data with OS support, and we do not use emulation. Instead, we create a single string that runs natively on each architecture with the instruction stream itself identifying platform-specific semantics.

Fat binaries are one approach where two independent program images are combined with special meta-data that is used at run-time to select the appropriate image [2, 4, 20]. Fat binaries require OS support to read in the appropriate meta-data and execute the appropriate image.

Sometimes architectures provide backward- and cross- compatibility with similar instruction sets, e.g., early VAX machine has PDP-11 compatibility mode [19], and modern Intel processors supports hardware-based virtualization for x86, IA64, and AMD64 [15]. In our work, we target completely different instruction sets, and even show that it is possible to automatically generate programs that work on architectures with different endianness.

All the above approaches use independent binary strings for each architecture, which are semantically the same program when they are run on a specific machine. To the best of our knowledge, the first PI program in the same sense of this paper that we are aware of was manually created by Drew Dean [8] in 2003. Two years later, Nemo also demonstrated a proof-of-concept shellcode for PowerPC and x86 [16]. However, those programs were manually generated, and did not offer a generalized algorithm for generating platform-independent programs.

Steganography. Simmons formulated steganographic security, in terms of the prisoners dilemma, which involves two prisoners whose conversations are monitored by a warden. The warden throws the prisoners in solitary confinement if he detects that they are planning escape, so the prisoners' goal is to talk about escape while evading detection by the warden. [18]. Cachin [6] approaches the problem of steganography from an information theoretic model. Hopper et al. [12] established a complexity-theoretic model for steganography. Mason et al. [14] have used steganographic techniques to generate shellcode that looks like English.

Previous approaches have focused on data hiding. While there are interpretations of our model that may be similar to data hiding, the fundamental goal is to hide execution behavior.

8. DISCUSSION

PIP Length. PIPs can be much longer than the input programs. This makes sense because the resulting program must be functionally equivalent to all input programs. In our implementation we make no significant effort to reduce the size of the PIP, though we recognize that it may be possible to perform optimizations to reduce the size. We leave this as future work. In addition, if PIPs are used in a steganographic setting, then one may also want to use additional steganographic techniques to mask other attributes, such as instruction and operand frequency.

More Gadget Headers. In § 3.1, we only considered gadget headers that did not change the system state other than the program counter: $(nop)^*(jump)$. This requirement was primarily a simplifying assumption so that the gadget header would not interfere with the semantics of the gadget body. We see three ways to relax this requirement. First, make sure any side-effects in the gadget header are "undone" in the gadget body, e.g., `push eax` in the gadget header can be undone by `pop eax` in the gadget body before the actual machine-specific logic. Second, it might be possible to perform semantic analysis of the input program to make sure any side effects in the header have no overall effect on the program. Third, the requirement of nop instructions does not apply to most of the shellcode because shellcode does not consider the effect of flags in general.

Large Input Programs. Given two input programs b_1 and b_2, one simple way to create a PIP is to generate $h||b_1||b_2$ where h is a gadget header that identifies the running platform. We stress that such a solution does not demonstrate a Turing-complete PIP scenario. In addition, one practical issue is that we may not be able to find a header h with a jump target large enough to skip over b_1 to execute b_2. This problem can be solved by inserting a long jump trampoline that eventually lead to executing b_1 on m_1, and b_2 on m_2. It is trivial to apply this technique in RG.

Indirect Jumps and Self-Modifying Code. Our current prototype does not handle indirect jumps (e.g., `jmp *eax`) and self-modifying code. Both cases would require an analysis, or set of techniques to ensure the jump target is to the correct gadget header. For example, indirect jumps can be handled if we know the jump targets, e.g., using an analysis such as VSA [5]. Alternatively, one could include a run-time monitor in the PIP itself that "fixed-up" jumps at run-time. Such extensions touch more on static and dynamic analysis than the fundamental possibility of automatic PIP generation and, thus, are left outside the scope of this paper.

Generating Platforms. In our approach, we create a single program that can exhibit different behaviors, depending upon which platform it is run on. A related problem is: given a program string, generate a new platform (e.g., emulator or instruction set update) such that the same program string has a predetermined different behavior. For example, Intel could use such a procedure to design a micro-code update to turn a pre-determined program into malware. We leave such questions as open problems for future work.

9. CONCLUSION

In this paper, we have developed techniques for automatically generating a single program string may run on multiple architectures. The central security implications of our algorithm is that the results of any static or dynamic analysis must be prefaced with the assumed platform. These implications lead directly to new security scenarios, such as execution-based steganography and rogue updates affecting security. Our techniques can also be used to ease cross-platform program (and shellcode) development. Finally, we show that, empirically, the amount of overlap between instruction sets means PIPs are likely hard to detect.

10. ACKNOWLEDGEMENTS

We thank Richard (Drew) Dean, Virgil D. Gligor, Melanie Thompson, and the anonymous reviewers for their helpful feedback. We also thank Jessica K. Hodgins, Lujo Bauer, and Raj Rajkumar for providing the variety of hardware needed for this paper. The authors were supported in part by funding by CMU CIT's Dean Fellowship.

References

[1] exploit-db. http://www.exploit-db.com/.

[2] FatELF. http://icculus.org/fatelf/.

[3] A. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[4] Apple. Universal binary programming guidelines. http://developer.apple.com/documentation/MacOSX/Conceptual/universal_binary/universal_binary.pdf.

[5] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 - a platform for analyzing x86 executables. In *Proc. of the International Conference on Compiler Construction*, Apr. 2005.

[6] C. Cachin. An information-theoretic model for steganography. In *Proc. of the Second International Workshop on Information Hiding*, pages 306–318, London, UK, 1998. Springer-Verlag.

[7] C. Cifuentes, M. V. Emmerik, and N. Ramsey. The design of a resourceable and retargetable binary translator. In *Proc. of the 6th Working Conference on Reverse Engineering*, pages 280–291, Oct. 1999.

[8] D. Dean. Personal email correspondence. Email exchange regarding prior work in multi-platform programs on August 4, 2009.

[9] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001.

[10] funkysh. Into my ARMs: Developing StrongARM/Linux shellcode. *Phrack*, 58, Dec. 2001.

[11] G. R. Gircys. *Understanding and using COFF.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1988.

[12] N. Hopper, L. von Ahn, and J. Langford. Provably secure steganography. *IEEE Trans. Comput.*, 58(5):662–676, 2009.

[13] O. Kolesnikov and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic. Technical Report GIT-CC-05-09, Georgia Institute of Technology, 2004.

[14] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proc. of the 16th ACM conference on Computer and Communications Security*, pages 524–533, New York, NY, USA, 2009.

[15] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel®virtualization technology: Hardware support for efficient processor virtualization. *Intel® Technology Journal*, 10(3):167–177, 2006.

[16] Nemo. Multi-arch shellcode. http://seclists.org/fulldisclosure/2005/Nov/387, 2005.

[17] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conference on Computer and Communications Security*, pages 552–561, New York, NY, USA, 2007.

[18] G. J. Simmons. The prisoners' problem and the subliminal channel. In *Proc. of CRYPTO '83*, pages 51–67. Plenum Press, 1984.

[19] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, 1993.

[20] A. Tevanian, M. DeMoney, K. Enderby, D. Wiebe, and G. Snyder. Method and apparatus for architecture independent executable files, 1993.

[21] C. Vulnerabilities and Exposures. CVE-2005-3252. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2005-3252, 2005.

[22] C. Vulnerabilities and Exposures. CVE-2010-0036. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0036, 2010.

[23] Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant code injection on ARM. In *Proc. of the 16th ACM conference on Computer and Communications Security*, pages 11–20, New York, NY, USA, Nov. 2009.

APPENDIX

```
shellcode = "\x06\xeb\x55\x0e\x0f\x60\xa0\xe1\x32\x6e\x86\xe2\x06\xd0\xa0\xe1\x24\x70\x8f\xe2\x32\
    x7e\x47\xe2\x06\x60\x26\xe0\x32\x6e\x86\xe2\x07\x20\xd6\xe7\x30\x20\x82\xe2\x07\x20\xc6\xe7\
    x01\x60\x86\xe2\x4b\x0e\x56\xe3\xf9\xff\xff\xda\x06\x60\x26\xe0\xd2\xf0\xf2\xb0\xd1\xe0\x52\
    xb2\xd1\xd0\x51\xb2\xd1\x4c\x70\xb3\xe9\x40\x57\xb2\xd0\xd0\xd0\xbf\xeb\xd0\xd0\xba\x31\xc9\
    x83\xe9\xeb\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xe8\x8e\x30\x01\x83\xeb\xfc\xe2\xf4\xd9\
    x55\x63\x42\xbb\xe4\x32\x6b\x8e\xd6\xa9\x88\x09\x43\xb0\x97\xab\xdc\x56\x69\xe7\xf2\x56\x52\
    x61\x6f\x5a\x67\xb0\xde\x61\x57\x61\x6f\xfd\x81\x58\xe8\xe1\xe2\x25\x0e\x62\x53\xbe\xcd\xb9\
    xe0\x58\xe8\xfd\x81\x7b\xe4\x32\x58\x58\xb1\xfd\x81\xa1\xf7\xc9\xb1\xe3\xdc\x58\x2e\xc7\xfd\
    x58\x69\xc7\xec\x59\x6f\x61\x6d\x62\x52\x61\x6f\xfd\x81\x0a\x90\x90\x90\xd0\x20\x70\xb1\xd1\
    xe0\xf1\xb0\xd2\xd0\xfd\xb9\xd0\xe0\x6f\xb5\xd0\xd0\xd0\xba\xd2\xd0\xf0\xd0\xd4\xe0\xfd\xb5\
    xdd\xe0\x70\xb1\xd2\xd0\x70\xb1\xe0\xf0\x70\xb3\xd1\x40\x57\xb2\xd0\xd0\xd0\xbf\xd5\xd0\x70\
    xb1\xd1\xeb\x70\xb3\xe0\xf0\x70\xb3\xd2\x40\x57\xb2\xd0\xd0\xd0\xbf\xd5\xd0\x70\xb1\xd0\xe0\
    x70\xb3\xd0\xf0\x70\xb3\xd1\x40\x57\xb2\xd0\xd0\xd0\xbf\xd0\x20\x70\xb1\xd2\xe0\x70\xb3\x0f\
    x40\x70\xb3\xd0\xd0\xd0\xbf\xd5\xd0\x70\xb1\xd1\xe0\x70\xb3\xd0\xd0\xd0\xbf\xd5\xd0\x70\xb1\
    xd0\xe0\x70\xb3\xd0\xd0\xd0\xbf\xd0\xd0\xf0\xb0\xd1\xe0\xf1\xb0\xd2\xf0\xf2\xb0\xd4\xf0\xfd\
    xb5\xd0\x90\x6f\xb5\xd0\xd0\xd0\xba\xff\xff\x43\x38\xd4\x90\xfd\xb5\xd0\x90\x6f\xb5\xd0\xd0\
    xd0\xba\xff\x32\x39\x3e\xd4\x90\xfd\xb5\xdd\xd0\x70\xb1\xdb\x40\x70\xb3\xd0\xd0\xd0\xbf\xd0\
    xd0\x70\xb3\xd1\x40\x70\xb3\xd0\xd0\xd0\xbf"
```

Figure 8: Example generated PI (ARM/x86) remote bind-shell shellcode.

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |........T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |........4. .....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |................|
00000040  00 80 04 08 f4 00 00 00  f4 00 00 00 05 00 00 00  |................|
00000050  00 10 00 00 90 eb 3e 20  17 00 00 2a 16 00 00 3a  |......> ...*...:|
00000060  07 00 00 10 00 00 04 24  21 28 e0 03 0c 00 06 24  |.......$!(.....$|
00000070  a4 0f 02 24 0c 00 00 00  a1 0f 02 24 0c 00 00 00  |...$.......$....|
00000080  f8 ff 11 04 00 00 00 00  48 65 6c 6c 6f 20 77 6f  |........Hello wo|
00000090  72 6c 64 0a 90 90 90 90  eb 17 31 db 43 8b 0c 24  |rld.......1.C..$|
000000a0  ba 0c 00 00 00 b8 04 00  00 00 cd 80 31 c0 40 cd  |............1.@.|
000000b0  80 e8 e4 ff ff ff 48 65  6c 6c 6f 20 57 6f 72 6c  |......Hello Worl|
000000c0  64 0a 90 90 01 00 a0 e3  18 10 8f e2 0c 20 a0 e3  |d............ ..|
000000d0  04 70 a0 e3 00 00 00 ef  00 00 a0 e3 01 70 a0 e3  |.p...........p..|
000000e0  00 00 00 ef 00 00 a0 e1  48 65 6c 6c 6f 20 57 6f  |........Hello Wo|
000000f0  72 6c 64 0a                                       |rld.|
```

Figure 9: Hexdump of a Hello World PI program for ARM, MIPS and x86 (244 byte).