Return Oriented Programming for the ARM Architecture

Tim Kornau

December 22, 2009

Diplomarbeit
Ruhr-Universität Bochum

Lehrstuhl für Netz- und Datensicherheit
Prof. Jörg Schwenk

Hiermit versichere ich, dass ich meine Diplomarbeit eigenständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

I hereby declare that the work presented in this thesis is my own work and that to the best of my knowledge it is original, except where indicated by references to other authors.

Bochum, January 4, 2010

Tim Kornau

Acknowledgements

I always wanted to be an engineer, the kind that brings people to the moon [Nieporte, 2009].

First I want to thank Prof. Jörg Schwenk for giving me the opportunity to write this thesis.

In particular my thanks go to Sebastian Porst and Thomas Dullien, who helped me whenever there where questions and suggestions about the development, and who provided insight to questions of design and structure of this work.

Special thanks go to my family who made it possible for me to study and always helped out if I was in need.

Finally I want to thank my fellow students and research assistants who helped me.

Contents

Abstract

This thesis describes the applicability of return-oriented programming on the ARM architecture. In the pursuit to defend against failures in software programs, defence mechanisms have been developed and are applied to almost all operating systems. One defence mechanism commonly used to defend against certain types of attacks is the use of non-executable memory regions. Return-oriented programming is a technique which circumvents this defence mechanism by using already existing code sequences, which can be chained to form an arbitrary program without the injection of code. In this thesis, a novel approach for the search for code sequences is presented that uses the REIL meta-language. With a focus on ARM as the target architecture, the novel approach presented here enables the analysis of library code to automatically identify code fragments for use in return-oriented programming. While the focus is on ARM, the work is largely independent of the underlying architecture. To the best of the author's knowledge there is no prior work that presents return-oriented programming on the ARM platform.

1. Introduction

1.1. Introduction

This thesis describes return-oriented programming on the ARM architecture. Furthermore, methods are discussed that help to automatically identify code fragments (gadgets) that are used in return-oriented programming. While there has been a reasonable amount of research on both return-oriented programming and offensive computing on ARM, no public work has combined the two so far.

Return-oriented programming is a recently-coined term [Shacham, 2007]. It has its origins in the well-known ”return-into-library”-technique, but extends it to allow the execution of arbitrary algorithms (including loops and conditional branches). While the ”return-into-library”-technique is well-known, the publications which have provided the most significant contributions are the works of Designer [1997a] and Wojtczuk [2001]. The work in Checkoway et al. [2009] shows that the technique of return-oriented programming is not confined to academic scenarios, but has practical applications and thus forms a significant addition to the offensive researchers tool chain.

The work presented in the field of offensive computing on the ARM architecture is dominated by research in the mobile phone area. The publications of Mulliner and Miller [2009a,b], Mulliner [2008] show a small number of examples of such research. Another field which received attention is the SOHO [1] router and small network devices area.

This thesis focuses on mobile phones, for which quite some previous offensive work exists [Hurman, San, 2005, Economou and Ortega, 2008], and furthermore Windows Mobile. Previous security analysis work on this platform was done in [Mulliner, 2006, 2005, Leidner, 2007, Becher et al., 2007].

1.2. Motivation

The ARM architecture is used in almost every mobile phone available today and even in some recently-popularized netbooks. The vast amount of mobile phones which are constantly powered on and have a constant network connectivity provide an interesting target for security research.

Desktop $! =$ **mobile:** Unlike the small number of mainstream operating systems used in modern desktop systems, the number and diversity of mainstream mobile operating systems is tremendous. Even though modern mobile operating systems are in many ways comparable to modern desktop operating systems, they still have unique characteristics. One of those characteristics is that they enable an adversary to generate revenue for himself after an successful attack just with using the phones capabilities to call premium numbers.

Security measures are often not adopted: Until now the necessity of defending mobile operating systems against adversaries has often been underestimated. This manifests itself in the limited use of defence mechanisms by mobile operating systems. Even though these defence mechanism are widely deployed in mainstream desktop operating systems, vendors of mobile operating systems often only implement one of the possible protection mechanisms into their devices.

[1] Small home and office

Defence mechanisms (Section 2.1) which are employed on operating systems include but are not limited to:

- stack cookies / heap cookies
- code and data separation (NX bit)
- address space layout randomization

The goal is to develop ”return-oriented programming” which allows to attacker to bypass NX bit protection.

Portability is a key factor: The focus of this thesis is the ARM architecture. But to be able to solve similar tasks on other architectures efficiently in the future, algorithms need to be portable across different architectures and adaptable to similar objectives. All previous work in the field of return-oriented programming has failed to address portability and adaptability. With the use of the platform-independent meta-language REIL [Dullien and Porst, 2008] it is possible to address these issues and provide algorithms which can be used platform-independently and which are adaptable to different problems.

1.3. Related work

Since the concepts in this thesis are easily ported to other operating systems, the related work section focuses on the return-oriented programming side.

The first return-into-library exploit: In a mail to the Bugtraq [Focus] mailing list the first public return-into-library exploit was presented by Solar Designer in August 1997 [Designer, 1997a]. The exploit, presented for the Linux lpr command, showed that return-into-library exploits are possible and may even prove to be simpler than exploits using injected shellcode. The most interesting aspect of the work presented by Solar Designer is that he has always been a researcher involved on both sides of the security game. Even though he provided the exploit and therefore proof that this technique works, he also provided defensive mechanisms [Designer, 1997b].

Advanced return-into-library exploits: In 2001 Nergal published an article [Wojtczuk, 2001] in the security magazine phrack which was devoted to the advancements of the return-into-library techniques. In his work he describes the ideas and improvements to the technique which developed out of the original approach, and adds new methods and ideas which further contributed to the field. In this article the unlimited chaining of functions within return-into-library exploits is described and its possible uses are shown. This work was the first work to include function chunks to shift the **esp** register, which is used to perform chaining of function calls, within return-into-library exploits.

Borrowed code chunks technique: With the introduction of hardware-assisted non-executable pages, common buffer overflow techniques became useless. Sebastian Krahmer postulated a possible way to circumvent the protection mechanism introduced in his work [Krahmer, 2005]. Furthermore, classic return-into-library exploits would cease to function on 64 bit Linux machines with proper page protection because the ABI [2] required the arguments of a function to be passed in registers. Therefore he developed an method to get arguments from the stack into registers and then call the desired function within the library. This enabled him to use the ”return-into-library”-technique with the new ABI.

[2] Application binary interface

Return-oriented programming on x86: In 2007 Hovav Shacham described the first Turing-complete set of code chunks which he named gadgets. These gadgets could be used to form an arbitrary program from code already present in the exploited target. The paper [Shacham, 2007] provides three contributions that have since been used to further research the area of return-oriented programming. He described an algorithm which is capable to recover code sequences in x86 libraries with the use of a disassembling routine. He described the first gadget set which became the starting point used in all later works in this field. He showed that return-oriented programming is not only possible on Linux but also on other x86 based operating systems. His last claim was that return-oriented programming on RISC machines would not be possible. He believed this because of the strict alignment requirements of the instruction set and the resulting scarcity of useful instruction sequences. This claim has been proven wrong in [Buchanan et al., 2008].

Return-oriented programming goes RISC: In 2008 Ryan Glenn Roemer presented his work [Buchanan et al., 2008, Roemer, 2009] in the field of return-oriented programming which was greatly inspired by the work of Hovav Shacham and is in part a joint work of both. The work presents the adoption of return-oriented programming to a RISC architecture (SPARC). This work demonstrated that return-oriented programming is possible on strictly aligned instruction sets and on machines that have completely different calling conventions compared to x86.

Return-oriented programming starts voting: In 2009 the paper [Checkoway et al., 2009] shows an attack against a voting machine which had been used for elections in the United States. This paper showed that the return-oriented programming technique was the only feasible way to reliably exploit the targeted machine in a real life scenario. The reason for this explicit conclusion is that the voting machine used a Harvard-type architecture which has code and data segments completely separated from each other. This prevents any other type of software exploitation technique. The main contributions of this paper in the field of return-oriented programming are:

- First real life example.
- Return-oriented programming in a scientific use case.
- Built a gadget set for a Harvard-type architecture.

Practical return-oriented approach Even though DEPlib [Sole] is not really a work in the field of return-oriented programming, it has one major advantage over the other papers which are listed as related work: It has a working implementation which is available. This work is important because it focuses on the application of the tool chain rather than the scientific side. The primary goal of the tool is not to provide a Turing-complete set of gadgets which can then be combined to a gadget set but to aid an attacker with a powerful interface to circumvent possible problems and to provide reliable exploitation. The main contribution of this work is to have built a tool around previously known ideas and to make this tool reliable and useful.

1.4. Thesis

Our thesis is as follows:

Return-oriented programming on the ARM architecture is possible. If the binary code of libraries for a given operating system can be analysed, there exists an algorithm which can determine whether the given code can construct the necessary gadgets for return-oriented programming. If the necessary gadgets for return-oriented programming exist, there exists an algorithm *which can extract the pre- and post-conditions necessary to craft an arbitrary program with the given gadgets.*

The purpose of this work is to investigate the above thesis and attempt to discover and implement a satisfying set of algorithms. Due to the sheer number of possible ways to perform specific tasks in return-oriented programming, it is necessary to limit the research to a subset of possible gadget types. In this investigation the following practical limits are imposed.

1. The search for certain functionality is performed by searching for particular sub-expressions in expression trees [3] generated from existing code. There exists a threshold of complexity for a given expression tree which is used to decide whether further analysis of the given tree should be performed.
2. The process of building a return-oriented program with the help of the automatically found gadgets is performed manually.

1.5. Contributions of this work

In the matter of return-oriented programming this thesis shows that return-oriented programming is possible on the ARM architecture. This thesis uses algorithms based on the REIL meta-language to perform the search for suitable gadgets in the given binaries. This shows that an alternative platform-independent way exists to search for gadgets automatically. No previous work on this subject uses platform-independent algorithms for return-oriented programming. This thesis therefore enables analysts to utilize one more tool for offensive computing on ARM based devices.

1.6. Overview

In Chapter 2 a definition of the objective of this thesis is given. Initially return-oriented is defined and its roots are explained. Then a description of the strategy to reach the given objective ”return-oriented programming for the ARM architecture” is given. Chapter 3 is a formalisation of the components required to build a return-oriented program for the ARM architecture. Chapter 4 contains the main description of the algorithms used in this thesis and the theory which they are based on. In Chapter 5 an outline of the implementation details related to the algorithms described in Chapter 3 is given. Chapter 6 shows the results of running the implemented algorithms against a set of binaries. The results are then used in a ”proof-of-concept” exploit which shows that the approach taken works. Finally, Chapter 7 gives a conclusion about the work performed in this thesis and discusses suggestions for further work.

[3] Expression trees represent mathematical expressions in binary tree form, where leaf nodes are variables and non-leaf nodes are operators.

2. Definition of objective

The following chapter describes the objective of return-oriented programming. To provide an introduction, the common protection mechanisms employed on operating systems are presented. The question in focus is which of the presented defence mechanisms can be circumvented by return-oriented programming. Then, the evolution of return-oriented programming is highlighted. It shows, which research and milestones have led to the approach of this thesis. Finally the strategy used in this thesis to solve the challenge of return-oriented programming for the ARM architecture is presented.

2.1. Protection mechanisms

Return-oriented programming is aimed to circumvent a certain class of protection mechanisms found in modern operating systems today. To be able to understand the impact of return-oriented programming for the ARM architecture, a basic knowledge about the common protection mechanisms is necessary. The following section briefly explains each defensive mechanism and provides the information whether return-oriented programming circumvents it.

2.1.1. Stack cookies

Stack cookies are special random values that are stored on the stack upon function entry. Upon function exit, the code checks if the value remains unchanged. Through this, sequential corruptions of stack frames can be detected on run-time. This does not provide any protection against modification of data structures in the stack frame of the local function, and only kicks in when the function exits. Attacks on structured-exception-handlers on x86-Windows exploited this (See Burrell [2009] for details). Return-oriented programming can not be used to circumvent stack cookies.

2.1.2. Address space layout randomisation

Address space layout randomisation randomises the addresses of executables, libraries, stacks, and heaps in memory. This technique prevents an attacker from using static addresses and static information in the attack, therefore lowering the reliability of an exploit or even rendering it useless. Return-oriented programming can not be used to defeat address space layout randomization.

2.1.3. Code and data separation

Code and data separation techniques are usually featured on Harvard-architecture based machines. Code and data separation is a technology where a certain memory area can either be used to write data to or execute code but not both. All major operating systems today have an implementation of this technique, most of them based on specific hardware support. In ARM the execute never (**XN**) bit was introduced in the virtual memory system architecture version 6. The feature was first introduced into mainstream processors in 2001 but was known as a technique as early as 1961 within the Burroughs B5000 [Wikipedia, 2009a]. One important aspect is that the NX bit for x86 machines is only available if PAE [1] is enabled. Return-oriented programming is aimed to defeat this protection mechanism.

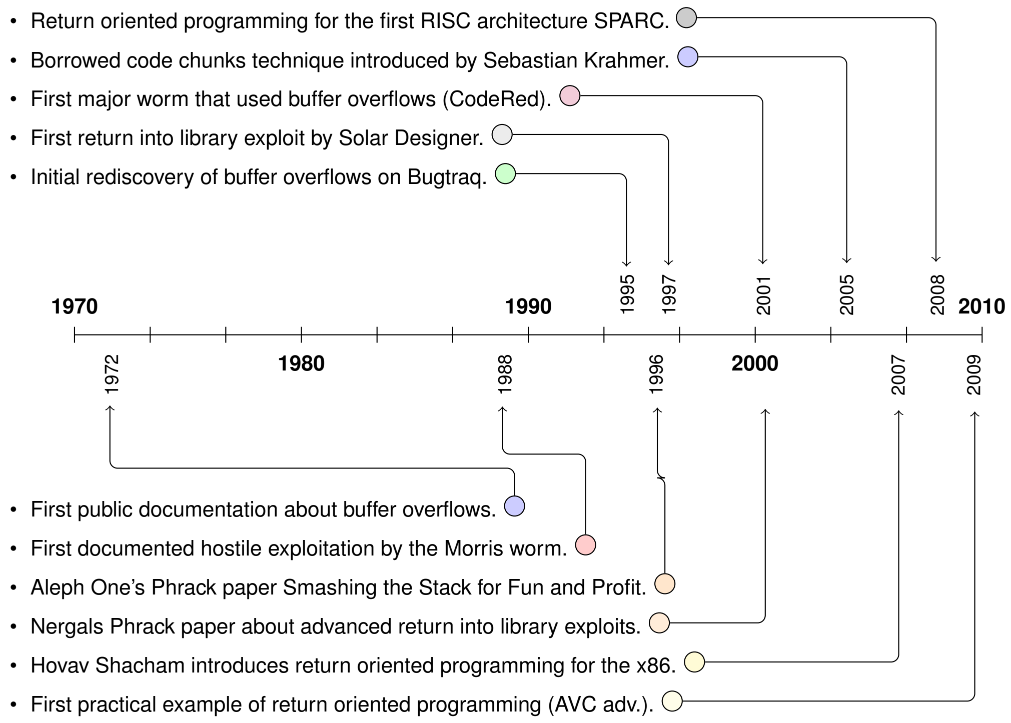
[1] Physical Address Extension

2.2. The evolution of return-oriented programming

This section describes the evolution of return-oriented programming and its applicability in various scenarios. Initially a time line is presented that provides an overview on the most important contributions in the field which have eventually led to return-oriented programming on the ARM architecture. Then these contributions are explained in detail.

2.2.1. The evolution time line

To be able to understand where return-oriented programming has evolved from and which steps eventually led to the first publicly available documentation, the following section provides a brief historical overview. As depicted in Figure 2.1, buffer overflows are a long known problem to the security of computer systems. But only after the first network infrastructures allowed attackers to reach many systems at once, the manufacturers of operating systems started to develop defensive mechanisms to counter the growing threat.



Return oriented programming for the first RISC architecture SPARC.
Borrowed code chunks technique introduced by Sebastian Krahmer.
First major worm that used buffer overflows (CodeRed).
First return into library exploit by Solar Designer.
Initial rediscovery of buffer overflows on Bugtraq.
1995
1997
2001
2005
2008
1970
1990
2010
1972
1980
1988
1996
2000
2007
2009
First public documentation about buffer overflows.
First documented hostile exploitation by the Morris worm.
Aleph One's Phrack paper Smashing the Stack for Fun and Profit.
Nergals Phrack paper about advanced return into library exploits.
Hovav Shacham introduces return oriented programming for the x86.
First practical example of return oriented programming (AVC adv.).

FIGURE 2.1.: TIME LINE FROM BUFFER OVERFLOWS TO RETURN-ORIENTED PROGRAMMING

2.2.1.1. Buffer overflows

As early as 1972 the first publicly available documentation of the threat of buffer overflows was presented in the Computer Security Technology Planning Study [Anderson, 1972]. One might ask why the necessary effective countermeasures have not been developed at this stage and why the information about the problems was not available more broadly. One reason for this was that only a small circle of people had access to this information at the time it was released, and that the policy to communicate with outsiders of these circles was strict [Dreyfus and Assange, 1997].

A buffer overflow is, in the original form, a very simple error that is introduced if a function does not perform proper bounds checking. Basically this means the function receives more input data than it can store. Assuming that the overflowed buffer was located on the stack, the attacker can now write a certain amount of data onto the stack where other variables and the return address might be located. Therefore the attacker can hijack the control flow of the current process and perform an arbitrary computation.

Even though the first worm which used a buffer overflow to spread dates back to 1988, the worms that changed the security mindset are not even a decade old. The CodeRed [CERT/CC, 2001] and SQL Slammer [CERT/CC, 2003] worms were the crossroad for introducing the initial security measures into Microsoft operating systems. Even though operating systems such as OpenBSD [OPE] had long before introduced software defences against this kind of attack, the first protection mitigating buffer overflows on Windows was not introduced until Windows XP SP 2 (2004).

2.2.1.2. Return-into-library technique

The return-into-library technique is the root on which all return-oriented exploit approaches are based.

A return-into-library exploit works as follows: After the attacker has hijacked the control flow, a library function he chooses is executed. The attacker has made sure that the stack pointer points into a memory segment he controls. The attacker has set up the data in the memory segment in a way that it provides the right arguments to the library function of his choice. Through this he can execute a function [2] with the needed arguments.

This technique was known as early as 1997 when Solar Designer initially posted the first publicly available proof-of-concept exploit [Designer, 1997a] to the Bugtraq mailing list. In this mail the groundwork for the offensive and defensive side of return-into-library exploits was presented. The development on both the offensive and the defensive side continued. The milestone article [Wojtczuk, 2001] discussed the wide range of available techniques up to its release. In his article Nergal presents advanced return-into-library attacks which where not known beforehand. One of these advanced techniques was the shifting of the **esp** register. This technique allows the unlimited chaining of function calls to be used in return-into-library exploits.

2.2.1.3. Borrowed code chunks technique

With the introduction of hardware-supported non-executable memory segments and 64 bit support in CPUs, the traditional return-into-library exploits ceased to work. This was due to an ABI change that now requires arguments to a function to be passed in registers instead of the stack. Sebastian Krahmer developed a new approach that uses chunks of library functions to still be able to exploit buffer overflows on machines that employed the newly introduced defences. His approach is designed around the idea to locate instruction sequences which pop values from the stack into the right registers for function calls. By using his approach an attacker can use return-into-library exploits with the new ABI.

2.2.1.4. Return-oriented Programming on x86

In his work [Shacham, 2007] ”The Geometry of Innocent Flesh on the Bone: Return-into-libc without function Calls (on the x86)”, Hovav Shacham has coined the term return-oriented programming. His work describes why he put effort into broadening the attack possibilities of return-into-library attacks and developed return-oriented programming. His argument was that the return-into-library technique does not use its full potential and that some of the proposed countermea-

[2]Usually an attacker chooses a function like *system()*, which executes the given argument in a new shell process of the system

sures are ineffective. Therefore he compiled a list of shortcomings and false assumptions which he addressed.

- The return-into-library technique has no support for loops and conditional branching.
- The removal of functions from libraries does not provide any security against return-oriented programming.

The approach Shacham uses to locate suitable instruction sequences works as follows: Initially he locates instruction sequences (gadgets) in x86 libraries. He does that by scanning the binary for the binary opcode which represents a return instruction (for example **0xC3**). From the address of the located return instruction he disassembles the binary backwards. The instruction set length of x86 is variable. Therefore a disassembly for each located return provides many possible instruction sequences. Each of the located instruction sequences is a possible gadget which can be used in the return-oriented-program. His work is the first work to define a gadget set of Turing-complete instruction sequences which can be used for return-oriented programming. It defines how these gadgets are constructed and combined to build an arbitrary computation with these gadgets.

2.2.1.5. Return-oriented programming on SPARC

Following the original work from Shacham, Ryan Roemer ported the return-oriented programming approach to the first RISC machine. His thesis [Roemer, 2009] shows the applicability of return-oriented programming on the SPARC architecture. The SPARC architecture is very different from the modern x86 architecture and has some characteristics that differentiate it from almost any other RISC machine as well. These differences lead to major changes in the approach to find gadgets in contrast to Shacham's original work:

- Due to the alignment that all RISC machines enforce for their assembly instructions, the original scanning method Shacham used to locate gadgets in x86 binaries can not be used on SPARC. The paper modifies the search algorithm to only consider existing instruction sequences for gadgets.
- As the SPARC architecture has a distinct calling convention and makes use of a register window for the exchange of data between functions, the gadget set and the instructions had to be adapted to work on SPARC.
- The thesis implements the gadget set as a memory to memory gadget set. Therefore registers are only used inside individual gadgets but not to transfer data between different gadgets.

Further contributions of the thesis are that not only a catalogue of gadgets is now available for the SPARC architecture, but also a gadget API has been developed which allows an attacker to develop exploitation code with the use of return-oriented programming in a convenient way. The specified contribution has not been verified by the author because the source for the API and the gadget search algorithms is not publicly available.

2.2.1.6. DEPlib

In an effort to completely automate the bypass of the non-executable stack technique "DEP" introduced by Microsoft, Pablo Sole presented his work [Sole] which is the most usable implementation of a return-oriented approach. The only drawback of his work is that he has no documented support for any conditional execution and therefore misses a Turing-complete gadget set. None the less, his work is the most practical work in this field and has some unique aspects which all of the works from academia are lacking. He introduces a complexity value for gadgets that focuses on the side effects of the located gadgets. Furthermore he does not rely on specific libraries but scans the whole address space of the executable for useful instruction sequences. One drawback of his implementation is that he only supports Windows because his software is an extension to the Immunity debugger which is only available for Windows.

2.2.1.7. Return-oriented programming on Harvard-type architectures

The most recent work which contributes to the general applicability of return-oriented programming is the work of Checkoway et al. [2009] which shows the use of return-oriented programming on a true Harvard-type architecture. The most important contribution of this work is that it shows a real-life use case for return-oriented programming in which no other exploitation technique would lead to results.

The paper presents an attack against the AVC Advantage voting machine, a machine which has been used for elections in the United States in the past. The machine uses a Zilog Z80 CPU. The Z80 has a variable length instruction set and is a Harvard-type architecture. The paper shows the applicability of a return-oriented programming attack against this architecture.

2.3. Strategy

This section describes the strategy used to solve the problem of return oriented programming for the ARM architecture. It presents the ideas that led to the decisions about data structures and algorithms as well as the dependencies which arose from them.

2.3.1. Problem approach

The goal of this thesis 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 a return oriented program. To build a return oriented program, parts which can be combined to build the program are necessary.

The parts to build a return oriented program are named gadgets. A gadget is a sequence of instructions which is located in the target binary and provides a usable operation, for example the addition of two registers. A gadget can therefore be thought of as a meta-instruction.

To be able to build a program from gadgets, they must be combinable. Gadgets are combinable if they end in an instruction that controlled by the user alters the control flow. Instructions which end gadgets are named ”free branch” instructions. A ”free branch” instruction must satisfy the following properties:

- The control flow must change at this instruction.
- The target of the control flow must be controllable (free) such that the input from a register or the stack defines the target.

It is necessary to search the set of all gadgets for the subset of gadgets which can be used for a return oriented program. The set of all gadgets is built by initially identifying all ”free branch” instructions followed by the analysis of the program paths ending in these instructions.

To be able to easily search for a specific operation within the set of all gadgets, the gadgets are stored in tree form. This tree form is named binary expression tree. A binary expression tree consists of operations with their operands. The tree is a result of multiple sequential native instructions and their effects. One binary expression tree only affects one target register. Therefore a single gadget always consist of more than one binary tree. The binary expression trees are searched for sub-trees, which specify a distinct operation, to find usable gadgets. The subtrees which are used to search for an operation 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.

3. Technical details

This chapter provides the technical background needed for return oriented programming on the ARM architecture. First the ARM architecture is explained, followed by a description of the operating system which is used as test subject. The description of the ARM architecture is provided because ARM has some unique characteristics that differentiate the architecture from other architectures. Also mobile operating systems differ in their architecture and design as much as desktop operating systems do. Therefore a short introduction to the specialities of the operating system used in this thesis is given. Then the REIL meta-language used for the analysis and matching algorithms is presented. A good understanding of REIL is necessary because it is the basis for every data structure and every algorithm used in this thesis. Thereafter the introduction to return oriented programming for the ARM architecture is presented and the gadget catalogue developed in this thesis is described. The gadget catalogue describes a comfortable gadget set with whom an analyst can build return-oriented programs on the ARM architecture.

3.1. architecture and operating system details

In this section an introduction to the ARM architecture is given. The necessary basics about the architecture are explained and the important aspects are highlighted. In the second half Windows Mobile, the reference platform for this thesis, is explained and its specifics are described. These basics are necessary because all of the work in this thesis is very closely related to the hardware and its particularities.

3.1.1. The ARM architecture

The ARM processors have been developed primarily for use in small scale systems such as mobile communication devices and small home and office network hardware. ARM processors are used in almost every new mobile phone which ships today. The widespread deployment of ARM makes the architecture an interesting target for offensive research in general and return oriented programming specifically.

With the introduction of the ARM9 processor core, the architecture of the ARM is a Harvard-type architecture. The primary difference between a Harvard architecture and a Von-Neumann architecture is that the instruction memory is physically separated from the data memory. Likewise both memory segments are addressed over distinct bus systems by the processor. In case of ARM an approach is used that slightly differs from a *true* Harvard architecture. Within ARM only the caches for data and instructions are separated.

Using a Harvard-type architecture has some side effects which have to be considered. The use of self modifying code on the ARM architecture is not possible without additional cache sync and flush code sequences. Also traditional stack overflows which inject code on the stack, and then adjust the control flow to execute it, always need cache syncing.

3.1.1.1. History

The ARM architecture has been changed quite frequently during its existence. The support for more instruction sets and extensions was added over time. Also as described the architecture was switched from a Von-Neuman type to a Harvard-type architecture with the introduction of the **ARM9** core. The first ARM processor which was widely available was the **ARM2** released in

1987. The ARM processors are always sold as **IP** [1] by ARM semiconductors, this means that they sell the specifications needed to fabricate an ARM processor but do not themselves build the chips. In table a brief overview on the wide range of ARM processors version is given.

YEAR	FAMILY	ARCHITECTURE VERSION
	ARM1	ARMv1
1987	ARM2	ARMv2
1989	ARM3	ARMv2
1991	ARM6	ARMv3
1993	ARM7	ARMv3
	ARM7TDMI	ARMv4T or ARMv5TEJ
	StrongARM	ARMv4
1995	ARM8	ARMv4
1997	ARM9TDMI	ARMv4T
	ARM9E	ARMv5TE or ARMv5TEJ
1998	ARM10E	ARMv5TE or ARMv5TEJ
	XScale	ARMv5TE
2002	ARM11	ARMv6 or ARMv6T2 or ARMv6KZ or ARMv6K
2005	Cortex	ARMv7-A or ARMv7-R or ARMv7-M or ARMv6-M

FIGURE 3.1.: ARM PROCESSOR TO ARCHITECTURE MAPPING

3.1.1.2. Registers

In the following paragraphs the available registers of the ARM architecture are described. Some of the available registers have a certain purpose which will be highlighted and explained. As the ARM architecture provides a subset of registers only in certain execution modes, these modes will be introduced shortly.

User mode registers: The ARM ISA provides 16 general-purpose registers in user mode (Figure 3.2). Register **PC**/**R15** is the program counter which can be manipulated as a general-purpose register. The general-purpose register **LR**/**R14** is used as a link register to store function return addresses used by the branch-and-link instruction. Register **SP**/**R13** is typically used as the stack pointer although this is not mandated by the architecture.

Flags and Modes: The current program status register **CPSR** contains four 1-bit condition flags (**N**egative, **Z**ero, **C**arry, and o**V**erflow) and four fields reflecting the execution state of the processor. Flag fields are used in a total of 16 possible condition combinations for the use in ARM instructions. The **T** field is used to switch between ARM and THUMB instruction sets. The **I** and **F** flags enable normal and fast interrupts respectively. The ”mode” field selects one of seven execution modes of the processor:

User mode is the main execution mode. By running application software in user mode, the operating system can achieve protection and isolation. All other execution modes are privileged and are therefore only used to run system software.

Fast interrupt processing mode is entered whenever the processor receives an interrupt signal from the designated fast interrupt source.

Normal interrupt processing mode is entered whenever the processor receives an interrupt signal from any other interrupt source.

[1] intellectual property

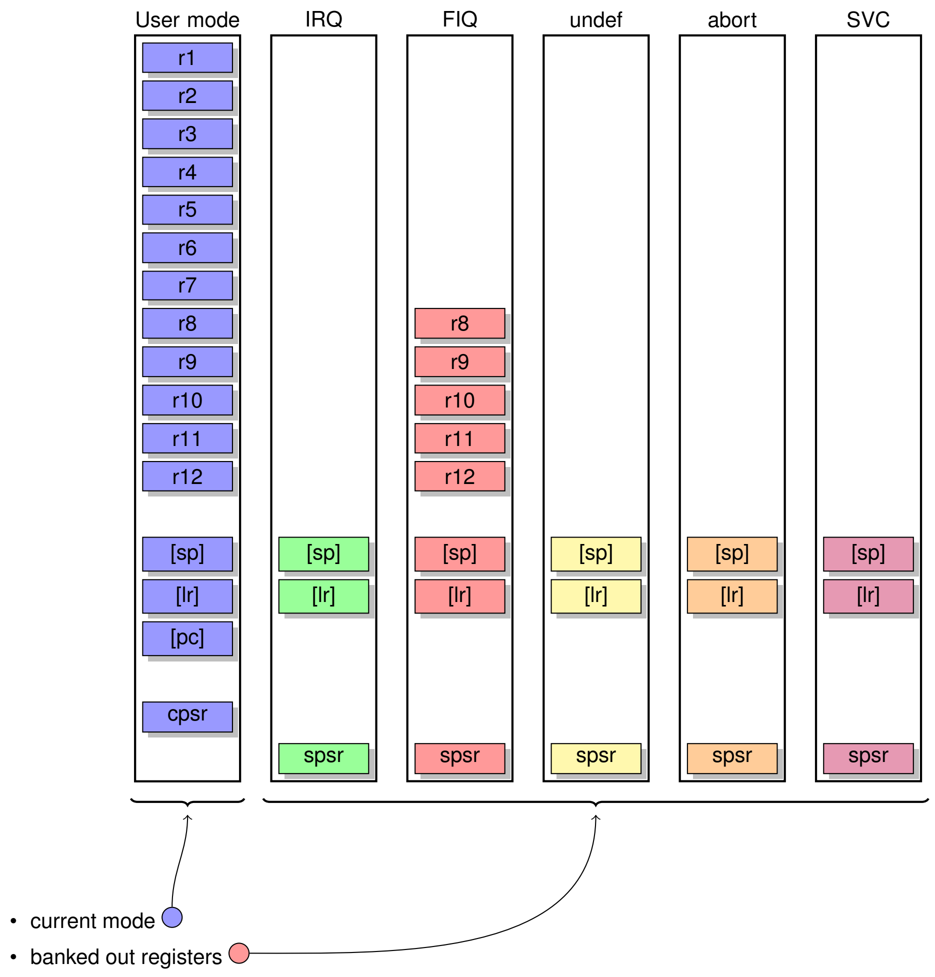


FIGURE 3.2.: ARM REGISTER OVERVIEW

Software interrupt mode is entered when the processor encounters a software interrupt instruction. Software interrupts are a standard way to invoke operating system services on ARM.

Undefined instruction mode is entered when the processor attempts to execute an instruction that is supported neither by the main integer core nor by one of the coprocessors.

System mode is used for running privileged operating system tasks.

Abort mode is entered in response to memory faults.

Privileged mode registers: In addition to registers visible in user mode, ARM processors provide several registers available in privileged modes only (Figure 3.2). **SPSR** registers are used to store a copy of the value of the **CPSR** register before an exception is raised. Those privileged modes that are activated in response to exceptions have their own **SP**/**R13** and **LR**/**R14** registers. These are provides to avoid the need to save the corresponding user registers on every exception. In order to further reduce the amount of state that has to be saved during handling of fast interrupts, ARM provides 5 additional registers available in fast interrupt processing mode only.

3.1.1.3. Instruction set

The ARM architecture can support several extensions to the normal ARM 32 bit instruction set. These extensions are labelled through the architecture type description: Extension **T** specifies THUMB support, **J** specifies Jazelle support, and **T2** specifies THUMB2 support. The THUMB instruction set is a 16 bit mapping of the 32 bit ARM instruction set but there are some differences between the instruction sets which will be covered in 3.1.1.4. The Jazelle extension is an implementation of a Java byte-code machine and allows the processor to execute Java byte-code natively in hardware. The THUMB2 instruction set adds a limited set of 32 bit instructions to the normal THUMB instruction set.

3.1.1.4. ARM and THUMB

The two instruction sets which are widely available on almost all the ARM devices are the 32 bit ARM instruction set and the 16 bit THUMB instruction set. Therefore these two instruction sets are explained and their differences are described.

The ARM instruction set uses 32 bits for every instruction it supports. It can make use of all features the specific processor has. The THUMB instruction set uses 16 bits for every instruction and is limited in the features it can use. The code density of THUMB mode is much higher than the code density of ARM mode. Due to the limitations of THUMB code it is generally executed slower then ARM code. Almost all 32 bit ARM instructions are conditional. The 16 bit THUMB extension does not support conditional execution. Conditional execution of instructions extends instructions with an optional condition field. This condition field is evaluated by the processor prior to the execution of the instruction. If the condition is true, the instruction is executed. If the instruction is false, the instruction is not executed. Conditional execution leads to more efficient code in terms of CPU pipeline usage and size. For example, the GCD instruction in Listing 3.1 uses 7 instructions without conditional execution while the implementation in Listing 3.2 uses only 4 instructions with conditional execution.

LISTING 3.1: ARM GCD EXAMPLE WITHOUT CONDITIONAL EXECUTION

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUBS    r0, r0, r1   ; could be SUB r0, r0, r1 for ARM
        B       gcd
less
        SUBS    r1, r1, r0   ; could be SUB r1, r1, r0 for ARM
        B       gcd
end
```

LISTING 3.2: ARM GCD EXAMPLE WITH CONDITIONAL EXECUTION

```
gcd
        CMP     r0, r1
        SUBGT   r0, r0, r1
        SUBLE   r1, r1, r0
        BNE     gcd
```

The instructions used in the examples Listing 3.1 and Listing 3.2 are described in Figure 3.3. For a more detailed explanation of the ARM instruction set refer to Ltd. [2005].

All 32 bit arithmetic instructions are able to use a barrel shifter which provides multiple shift operations to the last operand. This barrel shifter is not available with 16 bit instructions. ARM supports different addressing modes with pre- and post-indexed register updates for all memory operations. The switch between ARM and THUMB instructions is indicated with the **T** flag within the **CPSR** register. THUMB code is used if the size of available memory is small and execution speed is not a critical asset.

INSTRUCTION	DESCRIPTION
CMP	compare instruction, sets flags accordingly
BEQ	branch equal
BLT	branch less than
BNE	branch not equal
B	unconditional branch
SUBS	subtract and set flags
SUBGT	subtract if greater than condition met
SUBLE	subtract if less or equal condition met

FIGURE 3.3.: SHORT INSTRUCTION DESCRIPTION

3.1.1.5. Endianness of memory

Usually a certain operating system uses only one specific endianess for storing of data but in the case of ARM this can vary.

In contrast to other architectures the ARM architecture supports multiple modes for the endianness of the system. It supports little-endianess, big-endianess, and a supplemental mixed mode. And these are only the most common modes of endianess used, even though the ARM architecture supports several more. The variable endianess is only available for data access, the endianess of instructions is always little-endian mode and can not be changed.

For more information about memory endianess please refer to [Ltd., 2005].

3.1.1.6. Stack modes

The stack is in the case of return-oriented programming an important factor as it might be used to store information which is used within the gadgets. Therefore the stack modes of the ARM architecture are explained and the constraints which are crucial for successful exploitation are presented.

ARM has four different stack modes. These are used in the **LDM** (Figure 3.4) and **STM** (Figure 3.5) instructions. The stack mode used is controlled by the **L**, **P**, and **U** bits of the instruction encoding. If the **L** bit is set the instruction is an **LDM** instruction. If the bit is cleared the instruction is an **STM** instruction. The **P** bit controls whether the stack pointer points to the last ”full” element pushed onto the stack or the next ”empty” stack slot after the element. The **U** bit indicates in which direction the stack grows.

STACK ADDRESSING MODE	L BIT	P BIT	U BIT
LDMFA (Full Ascending)	1	0	0
LDMFD (Full Descending)	1	0	1
LDMEA (Empty Ascending)	1	1	0
LDMED (Empty Descending)	1	1	1

FIGURE 3.4.: ARM LDM ADDRESSING MODES

STACK ADDRESSING MODE	L BIT	P BIT	U BIT
STMED (Empty Descending)	0	0	0
STMEA (Empty Ascending)	0	0	1
STMFD (Full Descending)	0	1	0
STMFA (Full Ascending)	0	1	1

FIGURE 3.5.: ARM STM ADDRESSING MODES

In any operating system usually only one stack addressing mode is used, but there are also cases when the frame pointer is used to access data on the stack instead of the stack pointer. In these cases the direction of stack growth is usually switched.

Universal stack constraints The AAPCS (Procedure Call Standard for the ARM Architecture) Ltd. [2008] defines basic constraints that must hold at all times:

Stack-limit < SP <= stack-base The stack pointer must lie inside the stack.

SP mod 4 = 0 The stack must at all times be aligned to word boundaries.

Access limit A process may only access (either read or write) the closed interval of the entire stack delimited by [**SP**, stack-base - 1].

3.1.1.7. Subroutine calling convention

Both instruction sets, ARM and THUMB, contain a primitive subroutine call instruction (**BL**) which performs a branch with link operation. The effect of the **BL** instruction is that the sequentially next value of the program counter (*current address + 4* for ARM and *current address + 2* for THUMB) is saved into the link register **LR** and the destination address is stored in the program counter **PC**. In case of **BL** the least significant bit of the link register is set to one. If the instruction was called from THUMB code. Otherwise the least significant bit is set to zero. The result is that control is transferred to the destination address and the return address is passed to the subroutine as an additional parameter in the link register. The ARM architecture also provides the **BLX** instruction that can use a register to hold the destination address to pass control to. This instruction also handles ARM / THUMB interworking.

If the **BL** instruction is used, far jumps are not possible. In this case a stub function must be used to pass the control to the called function. An example of such a stub can be seen in Figure 3.8. For a more in-depth explanation of the subroutine calling convention refer to [Ltd., 2008].

3.1.2. Operating system

The following sections will focus on the operating system which has been used in this thesis. The important aspects which are in part specific this operating system and in part generic to operating systems are explained. This description is important because it will explain details necessary to understand the limitations and problems a researcher encounters on mobile operating systems. A larger part of the problems and concepts presented can be applied to almost any embedded operating system.

3.1.2.1. Operating system overview

The operating system used as the research target in this thesis is Windows Mobile 6.x. Windows Mobile is based on Windows CE 5. Windows Mobile is used in a wide range of consumer devices such as mobile phones and personal digital assistants. The Windows CE API which can be used within Windows Mobile is a subset of the Win32 API for Windows.

3.1.2.2. Memory architecture

This section describes the differences of virtual and physical memory and tries to clear out some misconceptions that can lead to false assumptions in case of memory definitions.

Virtual Memory Virtual memory is the addressable memory space. This can be understood as the *work area* for a process. On 32 bit Windows desktop systems each user land application has 2 gigabytes private virtual address space Sanderson. The addressable virtual memory space is 4 gigabytes. On Windows Mobile each application has a 32 megabytes private slot of virtual memory.

RAM Random access memory is the physical resource each process consumes to fulfil memory requests. A process has a 32 megabytes virtual memory address space but will not consume 32 megabytes of RAM initially when the process is started. RAM is consumed when the application allocates objects.

RAM vs. Virtual Memory As described, RAM and virtual memory are two different aspects of memory which should not be confused. The failure characteristics are different when one of the two runs out. If RAM runs out, there is no physical memory left. If virtual memory runs out, there is no *usable* memory left.

Address space Windows Mobile 6 has the memory architecture of Windows CE 5.2. It has 32 bits of addressable virtual memory. The upper 2 gigabytes of virtual memory are used for the kernel and system space. The lower 2 gigabytes are used for user space.

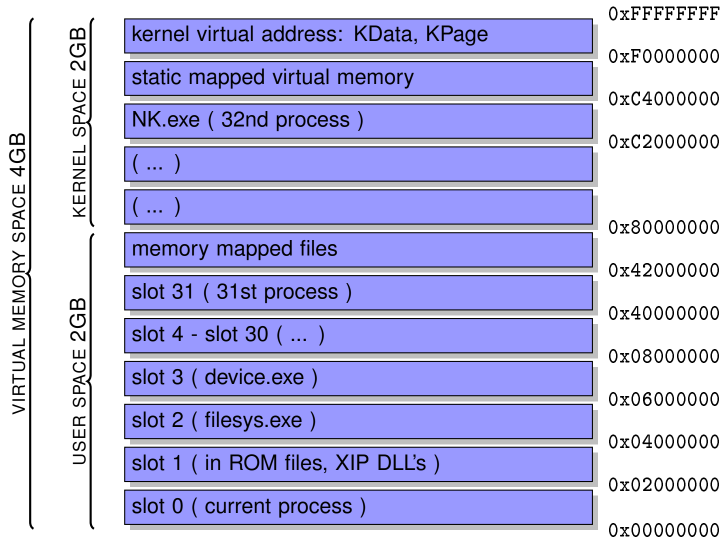


FIGURE 3.6.: WINDOWS MOBILE VIRTUAL MEMORY

The user space is divided into memory regions. The larger part of the memory regions is defined as the *large memory area*. This area is used to allocate large blocks of memory usually used for memory mapped files. The smaller part of the memory region is divided into small sections named slots. A slot is a basic unit for maintaining virtual memory within the Windows CE kernel.

There are 33 slots available on Windows Mobile of which 31 slots can be used by processes. Therefore a total of 31 simultaneous processes can be started on a Windows CE based system. The kernel process is counted as the 32nd process. The process with its currently running thread is cloned into slot 0. Slot 1 (XIP section) is used exclusively for in-ROM [2] components that have been included in the device image.

[2]Read Only Memory

3.1.2.3. XIP DLLs

Slot 1 is the XIP section. XIP stands for ”e**X**ecute **I**n **P**lace” as the binaries in this section are not relocated on execution. The XIP section was introduced with Windows CE 3 to provide a relief for the memory constraints in Windows CE. DLLs located in the XIP section are loaded from address `0x03FFFFFF` (64 megabytes) down to address `0x02000000` (32 megabytes). Only DLLs that are part of the original ROM shipped by the OEM [3] are placed in the XIP section. No non-XIP DLLs may be loaded in this memory area. Common DLLs for the inclusion into the XIP section are for example ”coredll.dll” and ”ws2.dll”.

3.1.2.4. DLL loading

The loading of DLLs which are part of 3rd party programs is done in slot 0 of the memory layout. Different DLLs under Windows CE 5.2 may not be loaded at the same address range in different processes and the same DLL may not occupy different address ranges in different processes. This implies that a DLL that is loaded in one process occupies space in all applications and not only the one that has loaded the DLL. This loading procedure is one of the reasons for memory exhaustion on Windows Mobile devices.

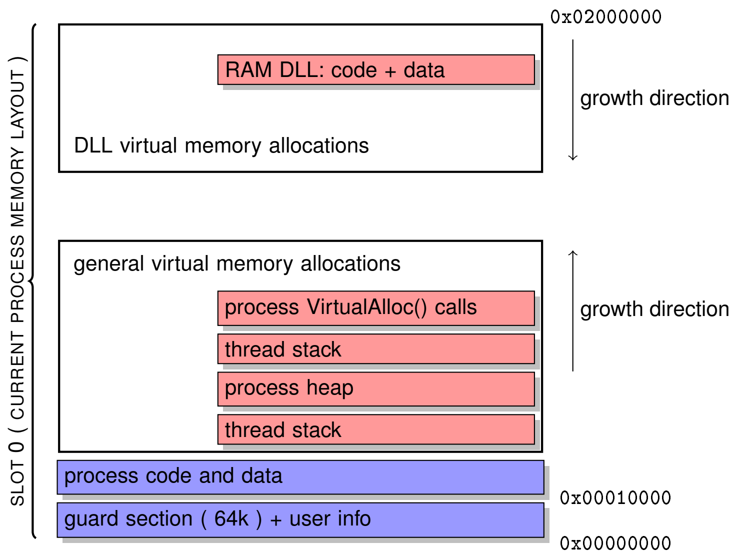

SLOT 0 (CURRENT PROCESS MEMORY LAYOUT)
0x02000000
RAM DLL: code + data
growth direction
DLL virtual memory allocations
general virtual memory allocations
process VirtualAlloc() calls
growth direction
thread stack
process heap
thread stack
process code and data
0x00010000
guard section (64k) + user info
0x00000000

FIGURE 3.7.: WINDOWS MOBILE SLOT 0 MEMORY LAYOUT

The application code is loaded into the virtual memory at address `0x00010000`. This section is followed by the read only section and then the read write space. Then the heap and the stack are the last sections which grow upwards towards higher addresses. The DLL space starts at the top of slot 0 and grows downward towards lower addresses.

3.1.2.5. Registers

Even though the registers and their meaning have already been discussed for the ARM architecture in general, The specific use of the registers in Windows Mobile is important to understand some of the gadgets later described.

There are 16 general-purpose registers in the ARM processor specified for use with Windows Mobile. How they are used within Windows Mobile is presented in table 3.8.

[3] Original Equipment Manufacturer

Register	Affinity	Aliases	Description
R0	Temporary		Argument 1, Return Value
R1	Temporary		Argument 2, Second 32-bits
			if double / int Return Value
R2, R3	Temporary		Arguments
R4–R10	Permanent		General registers,
			R7 is THUMB frame pointer
R11	Permanent	**FP**	ARM frame pointer
R12	Temporary		Scratch register
R13	Permanent	**SP**	Stack pointer
R14	Permanent	**LR**	Link register
R15	Permanent	**PC**	Program counter
CPSR			Flags

Figure 3.8.: Register description for Windows Mobile

Arguments for function calls are held in the registers **R0** through **R3**. Remaining arguments are placed in the calling function's argument build area. The area does not provide space into which **R0** through **R3** can be spilled.

3.1.2.6. The stack

Windows Mobile uses the little-endian mode of the ARM processor. The stack mode used by Windows Mobile is *full descending* which means that the stack pointer **SP**/**R13** is pointing to the last *full* entry of the stack and grows towards decreasing memory addresses [4]. Even though Windows Mobile specifies the frame pointer to be located as shown in Figure 3.9, experiments have shown that this must not be true in all cases. Therefore the location of the frame pointer should not be relied upon.

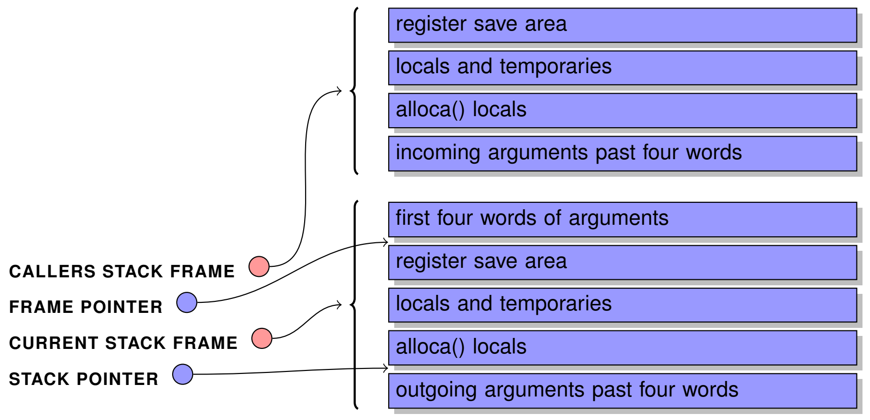


Figure 3.9.: Windows Mobile stack layout

The following list specifies additional information about the stack specifications on the ARM platform used by Windows Mobile. The information has been extracted from Corporation [2004].

Register Save Area (RSA) holds the preserved values of permanent registers used by the function. It also contains the function return address.

[4]towards the bottom of memory.

Locals and temporaries area represents the stack space allocated for local variables and temporaries.

First four words on top of the stack can contain the values passed in registers **R0–R3**. Any of these values may be missing. The values should be stored in the registers **R0–R3** if registers can not hold the arguments for the entire function or if the addresses for the arguments are in use.

Storage at the top of the called function stack is initialized if a routine needs storage space for the first four words of arguments. If a register keeps an argument for the argument live range, the argument has no associated storage in the stack frame.

Separate frame pointer If a routine has alloca() locals, the ARM specification requires a separate frame pointer to access the incoming arguments and locals. The frame pointer assigned for 32 bit ARM code is register **R11**, the register **R7** is used as frame pointer for 16 bit THUMB code.

Leaf vs. non-leaf routines In a leaf routine [5] any free register can be used as frame pointer. A non-leaf routine must use a permanent register as the frame pointer. The routine must not modify the frame pointer register between the prologue and the epilogue.

References with use of alloca() In a routine that uses alloca(), everything in the frame at a lower address than the alloca() area is referenced relative to the stack pointer and never contains a defined value at the time of an alloca() call. Everything in the frame below an address higher than the alloca() area is referenced relative to the frame pointer.

Efficient access in large stack frames A routine that needs to access data in a large stack frame can established another frame pointer. The establish frame pointer usually points to a fixed offset in the register save area or the locals and temporaries area of the stack frame but can point to any offset in the frame.

Stackless routines If a routine does not need to save permanent registers or allocate space for locals or outgoing arguments larger than four words, it does not need to set up a stack frame.

Strict alignment The stack pointer and the frame pointer are 4-bytes aligned on the ARM architecture.

3.1.2.7. ARM prologue and epilogue

Windows Mobile supports the virtual unwinding of stack frames. ARM prologue and epilogue code segments are required to implement structured exception handling (SEH) for ARM microprocessors. The ARM prologue is a code segment that sets the up the stack frame for a routine. The ARM epilogue is a code segment that removes the routine's stack frame and returns from the routine.

ARM prologue The ARM prologue for Windows Mobile has three parts. The three parts are directly continuous and there are no interleaved instructions. If the function prologue follows this guideline, the virtual un-winder can virtually reverse execute the prologue.

The three important parts of the ARM prologue are:

1. Zero or one instructions that push the incoming arguments in the registers **R0-R3** to the argument locations and update the stack pointer accordingly. If present, this instruction saves all the permanent registers in descending order at the top of the stack frame after any saved argument registers.

[5] A leaf routine is a routine that does not call any other routine, and does not have variables passed on the stack

2. Set up the additional frame pointer if necessary. If a frame pointer is established, the stack pointer is copied to the scratch register **R12** before the initial register saves. The scratch register **R12** is then used to compute the value of the frame pointer.

3. A sequence of zero or more instructions is used to allocate the remaining stack frame space for local variables, the compiler generated temporaries, and the argument build area. This is achieved by subtracting a 4-bytes aligned offset from the stack pointer. If an offset is too wide to be represented in the immediate section of the instruction used to subtract the offset, the scratch register **R12** is used to hold the offset. The offset used within **R12** is computed using a different instruction.

Listing 3.3: ARM routine prologue with frame pointer setup

```
MOV    r12, SP          ; Save stack on entry if needed.
STMFD  SP!, {r0-r3}      ; As needed
STMFD  SP!, {r4-r12, LR}   ; As needed
SUB    r11, r12, #16     ; Sets frame past args
<stack link if needed>
```

Listing 3.4: ARM routine prologue without frame pointer setup

```
MOV    r12, SP
STMFD  SP!, {r0-r3}  ; As needed
STMFD  SP! {[r4-r12,]|[SP,]LR} ; As needed
<stack link if needed>
<note: r12 is not used if the stack (SP) is the first register saved>
```

A short description of the instructions used in the examples (Listings 3.3, 3.4, 3.5, 3.6 and 3.7) is provided in Figure 3.10. For a more-in depth description of the specific instructions refer to Ltd. [2005]. The extensions to the *LDM* and *STM* instructions have been omitted from the description because they have been explained in Figure 3.4 and Figure 3.5 respectively.

Instruction	Description
MOV	move the contents of a register or integer to a register
STM	memory store multiple registers, first register is memory location start
SUB	subtraction
LDM	memory load multiple registers, first register is memory location start
BX	branch with interworking support for THUMB

Figure 3.10.: Simple ARM mnemonics

ARM epilogue The ARM epilogue for Windows Mobile is a sequence of continuous instructions that perform the unwinding of the current routine. The saved permanent registers are restored. The stack pointer is reset to the value before the routine entry and control is handed to the calling function.

The guidelines which are applied in epilogues used by Windows Mobile are the following. The instructions which form the epilogue are immediately continuous and no interleaving instructions are present.

If a frame pointer was set up, the epilogue is a single instruction (Listing 3.5) that uses the frame pointer as the base and updates all non-volatile registers. This includes the program counter and the stack pointer.

Listing 3.5: ARM routine epilogue with frame pointer

```
<no stack unlink>
LDMEA  r11, {r4-r11, SP, PC}
```

If no frame pointer was set up, the epilogue is comprised of a stack unlink, if needed, followed by an instruction that restores multiple registers or copies the link register into the program counter (Listing 3.6).

LISTING 3.6: ARM ROUTINE EPILOGUE WITHOUT FRAME POINTER

```
<stack unlink if needed>
LDMFD  SP, {r4-R11, SP, PC}
```

If a routine has not modified any non-volatile registers and there is no interworking between ARM and THUMB required, only a copy of the link register to the program counter is performed.

If interworking between ARM and THUMB is possible after the current routine returns, the epilogue needs to support interworking (Listing 3.7).

LISTING 3.7: ARM ROUTINE EPILOGUE WITH INTERWORKING SUPPORT

```
<stack unlink if needed>
LDMFD   SP, {r4-r11, SP, LR}
BX      LR
```

If a routine only branches to another routine in its last instruction and does not modify any non volatile registers, the epilogue of the function that performs the branch can be empty. If a routine establishes a frame pointer in the register **R11**, this routine must not modify the pointer value during the interval between the completion of the routine's prologue's last instruction and its epilogue's first instruction. If a routine has not established a frame pointer, this routine must not alter the stack pointer during the interval of the last instruction in the routines prologue and the execution of the first instruction of the routines epilogue. The address which is contained in the stack pointer must never be greater than the lowest address of any not restored register value in the register save area. This prevents that the preserved values of saved permanent registers are being corrupted by a context switch or any other asynchronous event that might occur during the execution of a prologue or epilogue.

3.1.2.8. Function calling

ARM calling convention requires that a full 32 bit address is called when calling a function. The maximum space which can be used to specify the address to call is 12 bits wide (8 address bits, 4 shifter bits). Therefore Windows Mobile uses function stubs to call routines in libraries outside the reachable address space Listing 3.8.

LISTING 3.8: ARM FUNCTION CALLING STUB

```
accept:
0x00011BDC ldr r12, [PC, #4]
0x00011BE0 ldr r12, [r12]
0x00011BE4 bx r12
0x00011BE8  [0x000140C4]  ; only data

__imp_accept:
0x000140C4  [0x02D365FC]  ; only data
```

The function stub loads the **PC**-relative address into the scratch register **R12**. The address to which the data in the stub points to is then loaded into the scratch register by dereference. Now the address of the function to be called is present in register **R12** and can be called. The stub uses the same instructions for all functions to be called but differs in the address.

3.1.2.9. System calls

In the ARM architecture there exists the **SWI** instruction to perform a system call or software interrupt. This instruction can be used to implement system calls although it is not used by Windows Mobile.

Windows Mobile implements system calls differently with a call to an invalid address in the range of `0xF0000000 - 0xF0010000`. The call to this address causes a pre-fetch abort trap which is then handled by PrefetchAbort in the file *arptrap.s*. The actual fault is a permission fault. If the address provided is located in the specified trap area, then the function ObjectCall, located in the file *objdisp.c*, will locate the actual system call function. If the address is not part of the trap area, the function ProcessPrefAbort, also located in the file *armtrap.s*, will deal with the exception.

To be able to locate a specific system call the following formula can be used.

- BASE ADDRESS

$$systemcall = 0xF0010000 - (256 * apiset + apinr) * 4 \tag{3.1}$$

- APISET in *kfuncs.h* and *psyscall.h*
- APINR in several files.

The really interesting part in respect to return oriented programming and exploitation of Windows Mobile in general is that the addresses used for system calls are static.

Additional information about system calls can be accessed in Becher and Hund [2008], Hengeveld and Loh [2006].

3.1.2.10. Cache synchronisation and buffers

With traditional shell code injection techniques, the Harvard architecture used in the ARM processor and its separate caches for data and code pose a problem for reliable exploitation. Even though this is not the case with return oriented programming the basics of circumventing the issue are explained in the following.

On ARM, data and instructions are separated into two buses, each with a separate cache [6]. In between the data cache and the main memory a write buffer in *write back* mode is used. This is a problem because data that has been injected has not yet been written back to memory and therefore can not used as instruction. With traditional injection, this can lead to a case where *old* data from the area where the injection was performed is executed leading to unpredictable results.

In Figure 3.9 the instruction sequence needed to perform this type of cache invalidation is shown. This sequence will work for any ARM operating system if the required privilege level is available to the program being exploited.

Listing 3.9: ARM cache invalidation instruction sequence

```
mcr p15, 0 , r0, c7, c10, 4 ; Instruction to drain the write buffer.
mrc p15, 0 , r0, c2, c0 , 0 ; Arbitrary read of CP15.
mov r0 , r0   ; Wait for the drain to complete.
```

In the specific case of Windows Mobile a function (Listing 3.10) is provided in the *coredll.dll* library that performs this function based on the arguments passed.

Listing 3.10: Windows Mobile cache sync function

```
VOID CacheSync(
  int flags
);

/*
 * where flags can be
 * CACHE_SYNC_DISCARD : Writes back and discards all cached data.
 * CACHE_SYNC_INSTRUCTIONS : Discards all cached instructions.
 * CACHE_SYNC_WRITEBACK : Writes back, but does not discard, cached data.
 */
```

[6]hybrid between a fully-associative cache, and direct-mapped cache

Even though the MSDN library specifies this function only for Windows CE and not for Windows Mobile, reverse engineering of coredll.dll has provided proof that this function exists for Windows Mobile as well.

3.1.2.11. Dumping ROM and XIP

In order to conduct return oriented programming on an analysis target, one has to determine the linked libraries of the specific target. In case of Windows Mobile the primary library that is linked to almost all executables is *coredll.dll* which is located in the XIP section of memory. But the use of the XIP section poses a set of difficulties one has to solve to gain access to the files in question.

Problems with dumping and extracting files

Copy protection Elements located in the XIP section can not be copied off a device or emulator with normal procedures such as the *copy* command on the command line.

File fragmentation In an image provided for updating or initial installation of a device, the files in the XIP section are fragmented into different parts.

Base address The address which is essential to return oriented programming may not be relocated correctly after extraction. This can lead to wrong offsets in the gadget search process.

File version As an OEM builds the operating system for a specific mobile device, libraries can contain subtle differences. Therefore changes from target device to target device must be taken into account.

Dumping XIP files from a running device Extracting XIP files from a running device either in the emulator or on a handheld is possible with the free tool *ROMExtractor* which is provided as binary executable Cortulla [2007]. The tool can extract all files with the file property *FILE ATTRIBUTE ROMMODULE* from a running device. Observations have shown that the files are relocated properly to the base address which is present in the running system. Therefore this seems to be the most efficient way of extracting files from XIP sources.

Extracting XIP files from an image There exists no general way to extract XIP files from an image because the OEM chooses the locations of the files in the XIP location and therefore one needs adapted tools for any device which is to be analysed. For a general approach to analyse an off-line image the following steps must be performed:

1. Download the image of choice from developer members.
2. Search the *kitchen* [7] for the device to be analysed with.
3. Locate the extraction functionality of the *kitchen*.
4. Extract the contents of the XIP.
5. Locate the folder where the dump has been placed.
6. Download the tool *recmod.exe* form developer members and place it in the directory above the dump directory.
7. Reconstruct the original files from the sections using *recmod.exe*.
8. Disassemble New File with IDA Pro.

[7] A kitchen is a compilation of tools used for unpacking and repacking of Windows Mobile images and software.

9. Select the option for Pocket PC ARM dynamic libraries for the file type.
10. Be sure to go through the advanced options and deselect the *simplify instructions* check box.
11. Start the disassembly.
12. Usually, using the dump of a specific file from XIP the relocation information is present in the files directory in the file *imageinfo.txt*. This information can be used to relocate the image to the original addresses on the device.

Finding the right set of tools for a specific device can prove to be a difficult task. The specified forum provides a lot of information but this information is sometimes hidden under a pile of non-informative or even misleading information.

The step to deselect the option to simplify instructions is very important if you plan to use REIL later in your analysis. This is due to how IDA Pro handles the simplification. It merges certain instructions into one instruction but the length of the instruction is not 16 or 32 bits any more. Therefore the REIL translators can not decide upon the instruction which was originally located at this specific position and translation fails.

Problems with extraction If one has no access to the real phone and extracts the needed libraries from an image, relocation is always the main issue. The file that accompanies the dynamic libraries gives hints about what base address and what size the original file had but there is always room for errors. This is a very serious issue with return oriented programming due to the fact that one will rely on the exact instructions at a specific location. If this information is just off by one, the results can greatly vary from the intended result.

3.1.2.12. Debugging Windows Mobile

Debugging Windows Mobile is as described difficult if one wants to debug system code and comparable to desktop Windows debugging if user land code, excluding system libraries, is to be debugged. Currently no software tool can be purchased which enables system code debugging on Windows mobile. The reason is that the system files, like most of the dynamic linked libraries, and the kernel are part of the XIP. All of the available Windows debuggers seem not to allow the usage of breakpoints within the XIP area of memory.

Even though this is a major limitation, debugging is still possible and in most cases it is not really necessary to break into a system library.

Debugging with Visual Studio If the source of a program is available, debugging with Visual Studio is easy and efficient. The tools which are required to enable development and debugging of Windows Mobile are.

- Visual Studio 2008 / 2005.
- Windows Mobile 6.1 refresh SDK.
- Microsoft Virtual PC.
- Active Sync.
- Emulator images for the desired Windows Mobile release.
- A Windows Mobile device (optional).

To be able to debug a piece software it has to be added to a Visual Studio project. In the Visual Studio project the settings have to be adjusted where to debug the project. The first option is to test the software on a real device. The second option is to test it on the emulator. Debugging on the real device is a poor choice for exploit and attack development because some devices crash irrecoverably upon memory corruption.

This does not imply that the final exploit will not work against the device but to save time debugging with the emulator initially is a better choice. The emulator has a major disadvantage. It does not emulate all of the hardware of the device. If one wants to develop an exploit or attack against a specific type of hardware / software combination the only option is to use the device itself.

Breakpoints and watchpoints work just like on desktop Windows software. The only difference is that the register window or the assembly code window displays ARM specific registers and assembly code in contrast to x86 specific information.

Another point which reduces the ability to debug the core libraries in Windows Mobile is that no PDB files are available from Microsoft for the libraries. The reasoning behind not providing the user with those files is that an OEM can change the libraries and therefore debugging symbols would not match the device specific libraries. However, this reasoning does not explain why there are no debugging symbols for the emulator images.

Debugging with IDA Pro IDA Pro provides a Windows CE debugger which is capable of debugging Windows Mobile devices. IDA Pro uses a debugger server which is copied to the device via the Active Sync protocol. For some versions of Windows Mobile two registry keys (Listing 3.11) have to be changed to lower the device security to a point where the debugger can be invoked remotely.

LISTING 3.11: WINDOWS MOBILE REGISTRY FIXES

```
Key: 'HKLM\Security\Policies\Policies001001' change to value DWORD:1
Key: 'HKLM\Security\Policies\Policies00100b' change to value DWORD:1
```

Like the debugger which is shipped with Visual Studio some address ranges are blocked for breakpoints if you use IDA Pro as debugger. The primer about Windows Mobile debugging with IDA lists the address range above *0x80000000*, which is the kernel memory space, and the address range of coredll.dll, as blocked ranges.

In contrast to the Visual Studio debugger, IDA Pro supports hardware breakpoints in data memory regions. This is especially useful if you consider the approach of return oriented programming where the future variables and parameters to functions all reside in the data region of the stack. But even though hardware breakpoints seems to be the perfect debugging solution, breaking into system functions is not possible. For a primer on debugging Windows Mobile with IDA Pro see Hex-Rays.

Debugging with BinNavi The debugger in BinNavi has been rewritten for this thesis by the author. The debugger is based upon the Windows Mobile debugging interface provided by the Microsoft libraries. In contrast to debuggers on desktop Windows versions some features of the debugging API can not be easily transferred to Windows Mobile. One example of these types of problems that was encountered is the use of the *Toolhelp32* API for the enumeration of loaded libraries by the debugged process. With a desktop Windows using this function is no problem. In Windows Mobile after some calls to the function have been performed it just stops working. In the process of rewriting the debugger this problem could not be traced to any specific cause but could be reproduced every single time.

The major difference between the BinNavi debugger and the two other debuggers is that it's not based on the Active Sync protocol. This leads to two important points. The debugger is not able to load the program which is to be debugged onto the device, but its not bound to the limitations and quirks that Active Sync has. The debugger is used over a simple TCP/IP connection which can be started for any networking device the Windows Mobile device has.

The other difference is that the debugger has to be installed on the device prior to debugging. It provides a graphical user interface where the process to be debugged can be selected from the list of running processes or from the disk of the device.

The selection of breakpoints and the presentation of the information from the debugger is shown in the BinNavi debugger window.

Problems with debugging As already mentioned in the previous paragraphs, debugging a Windows Mobile machine is not as easy as debugging a native Windows desktop machine. The primary reason for this is the XIP section and the limitations that result in a pre-relocated read only memory section. Another reason is that the tools which may be able to surround these issues are only available to device OEMs (Platform Builder for Windows Mobile).

3.2. The REIL meta-language

For the last decades, a wide range of people had only access to personal computers - generally x86 machines - at home. However, the recent evolution of consumer electronics like modern cell phones, PDAs, SOHO routers, and wireless devices, have brought people in contact with a wide range of other architectures. Even though Intel very recently introduced chips which focus on this specific market, architectures like PowerPC, ARM, and MIPS are used instead in most of the devices. Most of these devices store sensitive data and have the ability to connect to a wide range of networks and services. The progression towards consumer device architecture diversity is affecting the work of security researchers. The average security researcher now must deal with larger programs on a multitude of platforms. The ever-growing complexity of the software that runs on these devices naturally leads to more bugs in the code.

Auditing these larger code bases on multiple platforms, becomes more expensive as more analysts are needed which must have a diverse skill set to be able to analyse the platforms in question. To counteract the growing complexity and costs, methods need to be found to reduce complexity and make analysis tools portable across different platforms. zynamics [zynamics GmbH] created a low-level intermediate language to do just that. With the help of the Reverse Engineering Intermediate Language (REIL) an analyst can abstract various specific assembly languages to facilitate cross-platform analysis of disassembled binary code. In this thesis REIL was chosen as base for all analytic algorithms to allow a later adoption of the acquired results to other architectures.

3.2.1. A brief description of REIL cornerstones

Intermediate language design appears to be more art than science and has been proven to be only effective if it evolved from an iterative process of trial and error. The most important influence to the design of REIL were the experiences made designing previous intermediate representations, amongst others those discussed in a presentation by Halvar Flake at Black Hat Asia 2003. In that presentation, "Automated Reverse Engineering", he follows a first attempt at using intermediate representations for static bug detection [Dullien, 2003].

One of the key elements for REIL was the insight that an initial intermediate language needs to be extremely simple. Therefore complexity or over-dependency on correct disassembly needs to be avoided because it easily leads to mistranslations and difficulties later in the analysis chain. REIL does a simple one-to-many mapping in the translation step without trying to understand or recognize structures in the translated assembly source. Is has explicit modelling of the contents of the flags registers as results of the underlying arithmetic. For example, when translating ARM code to REIL, the ARM flags are modelled independently instead of being grouped into the CPSR register like they are on the real ARM CPU. Memory accesses in REIL are as explicit as possible. This is in contrast to the x86 instruction set where many different instructions can implicitly access memory. REIL has dedicated memory access instructions.

In general, one of the main goals was to create a language where every instruction has exactly one effect on the program state and this effect is immediately obvious when looking at the instruction. This contrasts sharply to native assembly instruction sets where the exact behaviour of instructions is often influenced by CPU flags or other pre-conditions. Real instructions often have an effect on the program state that is not immediately obvious without a deeper understanding of the instruction set and the underlying architecture.

3.2.2. REIL architecture and instruction set

The purpose of REIL is to provide a platform-independent intermediate language which makes it as easy as possible to write static code analysis algorithms such as the gadget finding algorithm for return oriented programming presented in this thesis.

This specific focus of REIL has led to design differences in comparison to other intermediate languages. REIL was designed to be easily understandable in a few minutes by an average security researcher. Also, algorithms designed for REIL should be shorter and simpler than algorithms designed for a native assembly language.

REIL is syntactically and semantically simple.

Syntactic simplicity means that all REIL instructions have the same general format. This makes them easy to parse and comprehend.

Semantic simplicity means that the REIL instruction set is as small as possible and that the effects of an instruction on the program state are clear and explicit.

REIL has a common instruction format which is shared among all instructions. Each REIL instruction has a unique REIL address which identifies the relative position of the current REIL instruction to other REIL instructions. A REIL address is composed of two parts. This is different to native assembly addresses.

- Native assembly instructions address which has been translated to REIL.

$$REILaddress = 0x03F0A0B0C\,.\,05 \tag{3.2}$$

- Zero based offset to the first REIL instruction.

Each REIL instruction has exactly one mnemonic that specifies the effects of an instruction on the program state. In total there are 17 different REIL mnemonics. Each REIL instruction takes exactly three operands. For instructions where some of the three operands are not used, place-holder operands of a special type called *Empty* are used where necessary.

Additionally it is possible to associate a list of key-value pairs, the so called meta data, with each REIL instruction. This meta data can be used by code analysis algorithms to do a more precise program analysis.

Prior to the description of the 17 different REIL instructions, it is necessary to give an overview of the REIL architecture. Listing the instructions is not sufficient to define the runtime semantics of the REIL language. It is also necessary to define a virtual machine (REIL VM) that defines how REIL instructions behave when interacting with memory or registers.

3.2.2.1. The REIL VM

The REIL VM is a register-based virtual machine without an explicit stack. This decision was made because the most-targeted native CPUs (x86, PowerPC, ARM) are also register-based machines.

This close proximity between the native architectures and the REIL architecture make it easy to translate native instructions to REIL.

Unlike native architectures, the REIL architecture has an unlimited set of registers. The names of REIL registers all have the form t-number, like t0, t1, t2. Furthermore, REIL registers are not limited in size. This means that in one instruction, the register t17 can be four bytes large and in the next instruction it can be 120 bytes large. In practice only register sizes between 1 byte and 16 bytes have been used. Due to REIL translation conventions REIL registers are local to one native instruction. This means that REIL registers can not be used to transfer values between two native instructions.

Native registers are also used in REIL instructions. This does not violate the principle of platform-independence because native registers and REIL registers can be treated completely uniformly in analysis algorithms.

The memory of the REIL VM is also not limited in size. It is organized using a flat memory model where individual bytes are addressable without alignment constraints. Memory segments or memory selectors are not used by the REIL memory. The endianness of REIL memory accesses equals the endianness of memory accesses of the source platform. For example, in REIL code that was created from ARM code, all memory accesses use little endian mode, while REIL code created from PowerPC code uses big endian by default.

3.2.2.2. REIL instructions

Now that the REIL architecture has been described, the 17 different REIL instructions can be introduced. These instructions can loosely be grouped into five different categories according to the type of the instructions. These categories are

- Arithmetic instructions
- Bitwise instructions
- Logical instructions
- Data transfer instructions
- Other instructions

Arithmetic instructions The general structure of all arithmetic instructions is the same. They all take two input operands and one output operand. The values of the two input operands are connected using the arithmetic operation specified by the instruction mnemonic. The result of the arithmetic operation is stored in the third operand, the output operand. The input operands can be either integer literals or registers. The output operand must be a register. None of the operands have any size restrictions but the arithmetic operations can impose a minimum output operand size or a maximum output operand size relative to the sizes of the input operands.

[ADD OP1, OP2, OP3] The ADD (Addition) instruction is used for addition computed in two's complement. To account for potential overflows the size of the output operand must be larger than the biggest size of the input operands.

[SUB OP1, OP2, OP3] The SUB (Subtraction) instruction is the exact opposite of the ADD instruction. It subtracts the second input operand from the first input operand and stores the result of the subtraction in the output operand. Subtraction can also overflow or underflow on fixed-size registers and the size of the output register must be adjusted accordingly.

[MUL OP1, OP2, OP3] The MUL (Multiplication) instruction is the unsigned multiplication instruction of REIL. It takes two input operands, interprets them in an unsigned fashion and multiplies them. The result of the operation is stored in the output operand. Multiplication can overflow too and the size of the output operand must be large enough to hold all potential results. In REIL, signed multiplication is simulated using unsigned multiplication followed by an explicit adjustment of the sign bit of the multiplication result.

[DIV OP1, OP2, OP3] The DIV (Division) instruction is the unsigned division instruction of REIL. It divides the first operand by the second operand. The result is stored in the output operand. The DIV operation is an integer division, meaning that the fractional part of the division result is truncated. Since the minimum absolute value of the second input operand is 1 (dividing by 0 is invalid), the result can never be bigger than the first input operand. The size of the output operand can therefore always be set to the size of the first input operand

[MOD OP1, OP2, OP3] The MOD (Modulo) instruction is the unsigned modulo instruction of REIL. It calculates the same operation as the DIV instruction but stores the remainder of the division in the output operand. Since the calculated remainder must be somewhere between 0 and the second input operand less one, the size of the output operand can be set to the size of the second input operand.

[BSH OP1, OP2, OP3] The BSH (Bitwise Shift) instruction is used for logical bit-shifting. The first input operand contains the value to shift. The second input operand contains the shift-mask that specifies how far the first input operand is shifted. Furthermore, the second input operand specifies the direction of the shift. If the value in the second operand is negative, the shift is a left-shift. Otherwise it is a right-shift. The BSH instruction is a logical shift instruction. Arithmetic shifting is simulated using BSH and explicit adjustment of the most significant bits of the shifted value.

Bitwise instructions The structure and behavior of bitwise instructions is comparable to that of arithmetic instructions. Each bitwise instruction takes two input operands and connects all bits of the input operands using the truth table of their respective underlying Boolean operation. The result of the operation is stored in the output register. Since none of the bitwise operations can overflow or underflow, the size of the output operand can be set to the size of the bigger input operand.

[AND OP1, OP2, OP3] The AND operation executes a bitwise AND on the two input operands and stores the result of the operation in the output operand.

[OR OP1, OP2, OP3] The OR operation executes a bitwise OR on the two input operands and stores the results of the operation in the output operand.

[XOR OP1, OP2, OP3] The XOR operation executes a bitwise XOR on the two input operands and stores the results of the operation in the output operand.

Logical instructions Logical instructions are used to compare values and to conditionally branch. The instructions of this category are the first instructions which do not use all three operands. The special operand type *Empty* is inserted where necessary instead. When writing down instructions, the *Empty* operands are not written.

[BISZ OP1, , OP3] The BISZ (Boolean is zero) instruction is the only way to compare two values in REIL. In fact, the BISZ instruction can only be used to compare a single value to zero. BISZ takes a single input operand and tests whether the value of the input operand is zero. If the value of the input operand is zero, the value of the output operand is set to one. Otherwise it is set to zero. The second operand is always unused. More complex comparisons must be modeled using a series of REIL instructions followed by an (optional) BISZ instruction.

[JCC OP1, , OP3] The JCC (Conditional jump) instruction is used to branch conditionally. If the value given to the instruction in the first operand is anything but zero, the jump is taken and control flow is transferred to the address specified in the third operand. The second operand is always unused.

Data transfer instructions Data transfer instructions are used to access the REIL memory or to copy values between registers.

[LDM OP1, , OP3] The LDM (Load Memory) instruction is used to load a range of bytes from the REIL memory into a register. The start address from where bytes are loaded is given in the first operand of the instruction. The register where the loaded value is stored is given in the third operand. The number of bytes which are read from REIL memory equals the size of the output operand. For example, if the output operand is four bytes large, four bytes are read from memory. Since there is no limit to the size of REIL registers, the number of bytes that can be loaded by a single LDM instruction is also not limited.

[STM OP1, , OP3] The STM (Store Memory) instruction is used to store a value in the REIL memory. The value to store is given in the first operand of the instruction. It can be either an integer literal or the content of a register. The third operand contains the start address the value is written to. The STM instruction always stores all the bytes of the input operand in the REIL memory. Since both integer literals and registers can be arbitrarily large, the number of bytes written by a single STM instruction is not limited.

[STR OP1, , OP3] The STR (Store Register) instruction is used to store a value in a register. This instruction can be used to store an integer literal in a register or to copy the content of one register to another register. The first operand contains the value to be copied; the third operand contains the target register where the value is stored.

Other instructions The last category of REIL instructions is the category of instructions that do not fit into any other category.

[NOP , ,] The NOP (No Operation) instruction takes no operands and does not have an effect on the program state.

[UNDEF , , OP3] Certain native assembly instructions, like the x86 instruction MUL leave registers or flags in an undefined state. To be able to simulate this behaviour in REIL it was necessary to add the UNDEF (Undefine Register) instruction. This instruction takes a single operand, always a register, and marks the content of that register as undefined.

[UNKN , ,] The UNKN (Unknown Mnemonic) instruction is a place-holder instruction that is emitted by REIL translators if they come across an instruction which they cannot translate.

3.2.3. Limitations of REIL

At this point REIL cannot yet completely translate all native assembly instructions. This is partly caused by limitations in the REIL language or the REIL architecture itself and partly because there has not been yet time to implement everything planned. The first limitation is that certain native instructions cannot be translated to REIL code yet. For example, FPU instructions and x86 instruction set extensions like MMX or SSE are not yet translated to REIL code because REIL is specifically made for finding security-critical bugs in binary code and these instructions are rarely involved in such bugs. Another limitation is that instructions which are close to the underlying hardware can often not be translated to REIL code without extending the REIL instruction set. For purposes of analysis, keeping the REIL instruction set small is more important than having the ability to translate all seldom used native assembly instructions to REIL code. A limitation that is significantly more important in practice is that exceptions are not handled properly yet. To handle exceptions in REIL, it is necessary to create a platform-independent model of exception handling first. This has yet to be done.

3.3. Return-Oriented Programming on ARM

In this section the answer to the question of feasibility of return oriented programming on the ARM architecture is given. Like other modern architectures, the ARM architecture has a non-executable stack.

In the original work by Shacham, unintended instruction sequences were the core of all analysing algorithms and the resulting gadgets. While this is a valid approach for any variable length instruction set like the x86, in fixed-length instruction sets this is not possible. It has been shown that return oriented programming is feasible on RISC machines with a fixed-length instruction set.

As described in Section 3.1 the ARM architecture is structurally different from both the x86 architecture and the SPARC architecture. In contrast to the x86 architecture it has strictly aligned 4 byte or 2 byte instructions. In contrast to SPARC it does not use a register window shift mechanism for parameter passing.

The contributions of this thesis to the research in the field of return oriented programming are:

- That a platform independent meta-language can be used as the basis for locating interesting instruction sequences.
- That function epilogues of leaf functions can be used as gadgets.
- That it is possible to construct a gadget set which can use registers and memory across different gadgets.

A return oriented program is defined by a distinct sequence of gadget frames which are placed in a attacker controlled memory segment of the exploited process, for example the stack or the heap. A gadget frame consists of one or more variables which are used as data source. The data from the variables provides the input for the gadgets and the information where the control flow will continue. A gadget is a short sequence of instructions located in a library or executable accessible at runtime. A single gadget provides a single operation for the return oriented program. Gadgets can be understood as meta-instructions and can, if carefully combined, form a return oriented program.

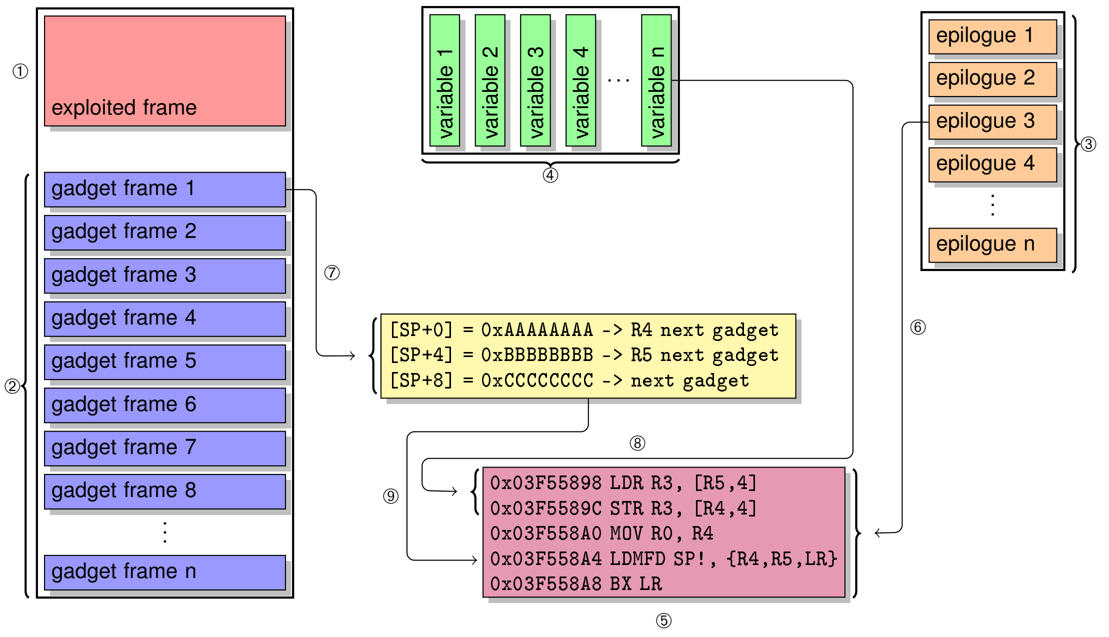
A return oriented program operates as follows: After the attacker has hijacked the control flow, the first gadget he chooses is executed. The attacker has made sure that the stack pointer points into the memory he controls. The first gadget is executed and eventually ends in a "free branch", e.g. a branch whose target address is determined during runtime. The attacker has set up the data in a way that allows branching to the next gadget he wishes to execute. Through this, he can execute one gadget after the other, where arguments for each code sequence can come from either the memory he controls or register values set by previous gadgets.

To be able to built a return oriented programming for the ARM architecture, a comfortable set of gadgets is explained in depth in Section 3.4. In Chapter 4, the algorithms to locate these gadgets are described and the theoretical ideas on which they build are explained. The algorithms are able to find and categorize gadget types and to measure their respective complexity to choose the least complex gadget.

3.3.1. A note on Turing-completeness

In this thesis the term ”Turing-complete” is often used in the context of the instruction set which is available through the use of return-oriented programming. The term Turing-completeness is named after Alan Turing. It states that every plausible design for a computing device can be emulated by a universal Turing machine. Therefore, a machine which can act as a universal Turing machine can perform any calculation of any other computing device.

However, this does not state anything about how complex it is to build a program for this computing device.



```
[SP+0] = 0xAAAAAAAA -> R4 next gadget
[SP+4] = 0xBBBBBBBB -> R5 next gadget
[SP+8] = 0xCCCCCCCC -> next gadget
```

```
0x03F55898 LDR R3, [R5,4]
0x03F5589C STR R3, [R4,4]
0x03F558A0 MOV R0, R4
0x03F558A4 LDMFD SP!, {R4,R5,LR}
0x03F558A8 BX LR
```

① An exploited frame on the stack gives the attacker initial control over the control flow and provides the arguments for the first gadget.

② After the initial control hijack, all data which has been stored on the stack below the initially exploited frame is used as a gadget frame. A gadget frame uses the stack to pass variables to the next gadget.

③ The gadgets itself are located in the runtime image of the exploited program. The runtime image consists of the program itself and all the libraries which have been loaded into the program.

④ The variables which are needed across multiple gadgets can be stored in a dedicated memory section. The exploited process must have read and write access to the memory area.

⑤ Example instruction sequence which forms a gadget.

⑥ The gadget is located in a function epilogue in the current runtime image of the exploited binary.

⑦ A gadget frame is needed for each gadget which has stack usage. The gadget frames are located on the stack.

⑧ The first two instructions of the example gadget need the memory locations [R4+4] and [R5+4] to be accessible memory.

⑨ The gadget frame provides the input for the LDMFD instruction. Therefore the registers R4 and R5 are set to the values present in the frame and the control flow is passed to the value which has been stored in LR.

FIGURE 3.11.: RETURN ORIENTED PROGRAM ENVIRONMENT OVERVIEW

Even though the term Turing-complete is used to describe the basic capabilities of the gadget set, a much more comfortable set of gadgets is searched for. With a comfortable gadget set it is not only theoretically possible to program, it is practically possible.

3.3.2. Finding ARM Instruction Sequences in libraries

The search for *useful* instruction sequences in ARM libraries was performed with a set of conditions in mind that have to be fulfilled by the instruction sequence.

- It should completely perform one of the desired operations which are needed for a comfortable set of gadgets.
- It should partially perform one of the desired operations and there are other sequences available which perform the missing parts to fulfil the operation.

- The sequence must have the result of the operation stored in memory or in a register.
- The sequence must not have unintended side effects which might jeopardize the operation of another sequence and therefore the entire program.

All experiments performed in this thesis have used the Windows Mobile version 6.1 library *coredll.dll* in multiple versions. The file is approximately 592 kilobytes large and provides about 3000 ”free branches”, which are terminating potentially usable instruction sequences.

The algorithms which are used to perform the experiments are explained in depth in Section 4.

One of the primary concerns within the gadget search process is passing variables between instruction sequences. On the ARM architecture passing variables is possible with registers and with memory. The overview in Figure 3.11 shows the use of memory locations as parameters to a function ⑧ and registers as parameters ⑥.

3.3.3. Construction of ARM Gadgets

A gadget is the combination of one or more instruction sequences located in the library. It can either read from registers or memory, perform its operation and store the result to a register or a memory location. The ARM gadget catalogue describes gadgets which can perform a basic set of computational methods, such as memory read and writes, register read and writes, arithmetic on registers or memory, bitwise arithmetic on registers or memory, control flow operations, function calls and system calls. The operations performed by the gadgets are described in a simple descriptive way using three parts:

- Tree form, to display what has been searched for.
- Assembly form, to display what has been found.
- Gadget form, to explain what is needed to use the gadget and what its results are.

3.3.4. Crafting a Return-Oriented Program

A return oriented program is nothing more and nothing less than a carefully crafted buffer of consecutive addresses, variables, and place-holder values which is placed on the stack or heap of the exploited program. It needs to fulfil the requirement that upon return of the initially exploited function, in case of a buffer overflow on the stack, the stack pointer **SP** and the link register **LR** are set to the right values for the initial gadget and the necessary arguments are provided in the right registers.

3.3.5. Generating a return oriented program with a compiler

With these prerequisites it will be possible to construct a compiler which is able to use these gadgets to build a return oriented program automatically. Earlier papers in the field of return oriented programming have provided such an compiler and an API for return oriented programming. This approach abstracts the creation of a return oriented program even more and provides an even simpler interface for the user. None of the described compiler extensions are publicly available.

3.4. ARM gadget catalogue

For every programming language there must be a definition about the possible commands and structures present in the language. For return-oriented programming the gadgets are the commands. Therefore it is necessary to define and describe them.

The gadget catalogue is a compilation of the gadgets used in this thesis. In this section the following information is provided: A description for each gadget is provided and the operation it performs is explained. The **tree form** (search string) which is used to find the particular gadget is presented. An exemplary assembly listing for each gadget is given and the effects for the presented listings are described.

The in this section provided information is used in the algorithms for gadget searching which are explained in Chapter 4.

3.4.1. Description

What does the tree form do and why is it needed ? The set of all possible gadgets is searched for usable gadgets. A single gadget in the set consists of multiple binary expression trees, which are generated from the code sequences. Therefore operations must be located within these binary expression trees. The **tree form** is the "search string" used for the search. A search is performed with a binary expression trees that matches only a certain operation, for example an addition. The expression trees used for searching have been constructed manually by looking at the REIL translation of an instruction sequence which performs a desired operation. Once a tree for a specific operation has been constructed it can be used in the algorithms for automatic gadget finding (Chapter 4).

What information does the assembly form provide ? A single gadget for a specific operation is usually present more then once in a binary, also it need not match an exact instruction sequence. Therefore an example is provided to show how a representative of this specific gadget might look like. The short assembly listing shows the instructions which have been found by using the binary expression tree of one specific gadget in the algorithms from Chapter 4. They are provided as a basis to explain what is necessary to use this specific sequence of instructions.

Why is the gadget form presented ? As the goal is to build a program from gadgets, these gadgets must be combinable. A gadget can be thought of as a meta-instruction which provides a specific operation in form of an instruction sequence. This instruction sequence need not be the same for two gadgets that provide the same operation. Therefore not all of the gadgets for one operation are the same and might require different conditions to be combinable. To provide an abstraction for the information about what a certain gadget needs as input and provides as output, as well as its side effects, the **gadget form** is given. The **gadget form** is used in combination with the **assembly form** as it represents this exact sequence of instructions.

What about side effects ? A gadget might taint registers or memory cells which are not part of the core functionality that was searched for. These tainted registers or memory cells are the **side effects** of the gadget. Side effects that taint registers can in almost any case be ignored as the register can just be marked as tainted until a known value is stored into the register. For a memory cell this is not the case. This is because the memory cell can be addressed in various ways other then a register which is always addressed by its name. Therefore gadgets which have memory cell side effects should be avoided if possible. The side effects for each gadget are automatically extracted by the algorithms in Chapter 4.

How does the combination of gadgets work ? The calling convention of the ARM architecture describes that a functions epilogue must restore the registers of the caller function. This behaviour is used to combine gadgets to form a program. The registers which must be restored are usually saved onto the stack in the function prologue. In a return oriented program the prologue of a function is not used, but the stack contains the data placed there for the return oriented program. Therefore the function epilogue which would usually return to the caller function and restore its registers now returns to the next gadget and provides it with its registers. The registers are restored by a single assembly instruction (**LDMFD**) which usually has the stack pointer as first argument. Therefore all of the registers are loaded from a stack offset. Not all gadgets restore the same registers. Therefore gadgets can only be combined if the pre- and post-conditions specified in the **gadget form** of the gadgets match each others requirements.

3.4.2. Nomenclature

To avoid misunderstanding and uncertainty about the following gadget specifications the nomenclature provides the information to interpret them correctly.

Tree form The tree form has two types of nodes which are differentiated by their color. Light blue defines mandatory nodes which must be present for the search. Light green nodes define sub-trees which must be present but the actual form in which they exist in a certain gadget can vary.

Gadget form The gadget form has four different node types also differentiated by color. The operation(s) performed by the gadget are coloured green. The chaining variables which are used as input for the next gadget subsequent to the current gadget are coloured blue. The gadget chaining (through the PC register) is coloured in orange. The side effects of the gadget are coloured in red.

register access Register access is denoted through the name of the register states without any brackets (for example **R3 = 5**).

memory access Memory access is specified through a set of brackets which contain the name of the register and a possible offset (for example **[R5] = R2**).

offsets Offsets (+ || - are the possible operators) exist only for memory locations and are presented in the form register operand offset (for example **[R5+4]**).

mem prefix The **mem** prefix followed by a bracket with the possible operand types (register, immediate, register with immediate offset or register with register offset) states a memory access to the memory location within the brackets. If the **mem** prefix is followed by another **mem** prefix, an access to a dereferenced memory location is indicated (for example **mem[mem[R4]**).

$\perp$ The $\perp$ symbol is used to define that no information about the state of the register or memory cell can be given. The $\perp$ symbol is used in the side effects for each gadget.

3.4.3. Memory gadgets

Memory gadgets are gadgets that store the result of the computation in memory. The source of the computation can either be a register or a memory location. A computation in this context is understood as a change in value of the target memory location. Therefore a simple move from a register to a memory location is a computation as well as an addition of two registers with a downstream store of the result in a memory location.

LISTING 3.12: MEMORY TO REGISTER GADGET EXAMPLE

```
0x03F8E3F8    STR         R0, [R4,12]
0x03F8E3FC    LDMFD       SP!, {R4,R5,LR}
0x03F8E400    BX          LR
```

3.4.3.1. Gadget: memory to register

The ”memory to register” gadget copies the value stored in a register to a memory location. The source operand must be a register and the target operand must be a memory location. This gadget can be used in combination with other gadgets if the desired operation needs to store its result to a memory variable but the computing gadget can not access the memory itself.

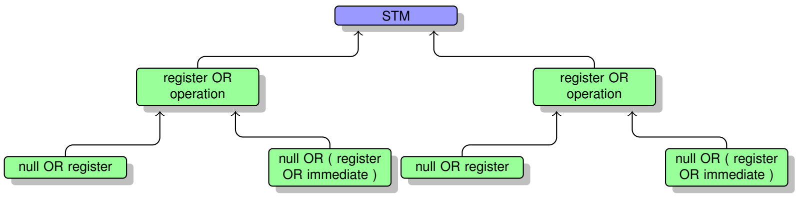


FIGURE 3.12.: MEMORY TO REGISTER TREE FORM

The tree in Figure 3.12 has one light blue mandatory node which must be present as root node. Without this node an expression tree from an instruction sequence does not match the specific gadget. The key under which such an expression tree is stored in the operand tree map always starts with a **MEM** prefix. The light green nodes in the figure represent the optional nodes. Optional means that there can be multiple possible trees present for the tree to match but the tree must be empty. For example the left-hand tree can either be only a register node, which must be present as the left-hand node of the light blue **STM** node, or an operation of the type **ADD** or **SUB**. If the left-hand node is a **ADD** or **SUB** then their respective left- and right-hand nodes must be present, and at least the left-hand node must be a register. The right-hand node can either be a register or an immediate integer. All of the following trees meet this specification.

The assembly code (Listing 3.12) shows an example for the ”memory to register” gadget type. On the left-hand side of the listing the original addresses of the assembly code from the analysed binary are shown. On the right-hand side of the listing the ARM assembler is shown. All of the assembly listings presented in this section follow this specification. They are meant to show for which assembler code the tree in Figure 3.12 will provide a match.

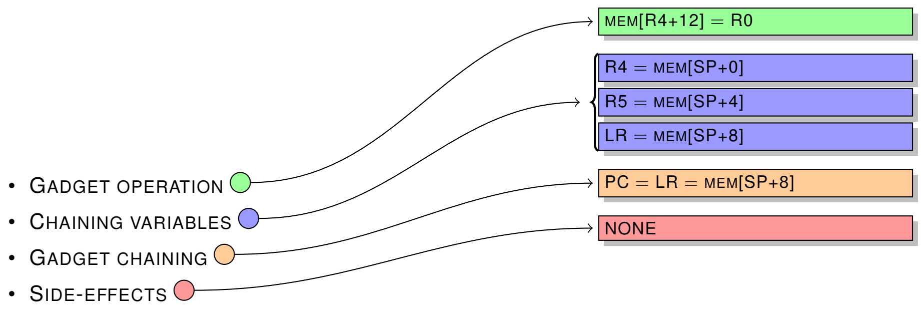


FIGURE 3.13.: MEMORY TO REGISTER GADGET FORM

The gadget form in Figure 3.13 shows the conditions which have to be met for the assembler

LISTING 3.13: MEMORY TO MEMORY GADGET EXAMPLE

```
0x03F55898    LDR         R3, [R5,4]
0x03F5589C    STR         R3, [R4,4]
0x03F558A0    MOV         R0, R4
0x03F558A4    LDMFD       SP!, {R4,R5,LR}
0x03F558A8    BX          LR
```

code (Listing 3.12) to work as the intended gadget. In the register **R0** the value which is to be stored in the memory location **mem[R4+12]** can be provided. The memory location **mem[R4+12]** must point to an accessible memory location. The memory locations **mem[SP]**, **mem[SP+4]**, and **mem[SP+8]** must also point to accessible memory locations. In the special case of the memory locations which are referenced with an offset from the stack pointer, accessibility is almost always possible. The values from the stack will be used to load values into the registers **R4**, **R5**, and **LR**. The value for **LR** has to be treated with special care because the control flow will continue at the given address as indicated in the highest light orange node. The memory locations and registers which are tainted by the gadget are shown in the light orange node below the control flow node. All of the gadget type figures in this thesis follow this specification.

3.4.3.2. Gadget: memory to memory

The ”memory to memory” gadget copies the value stored at the memory location the source register points to, to the memory location the target register points to.

To be able to have more matches for this gadget type within a specific binary, the source and target registers which specify the memory locations can have positive or negative offsets. These offsets can be treated as they were normal memory cells because the register value which is used in combination with the offset can always be adjusted. Integer values as source or target are not allowed because this would reference static values within the running application which are not needed as all the values can be loaded into a register.

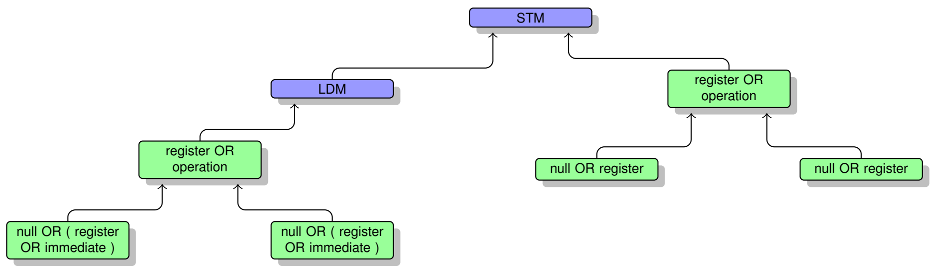


FIGURE 3.14.: MEMORY TO MEMORY TREE FORM

A ”memory to memory” gadget search tree has two mandatory nodes. The root node which must be present is a **STM** node. The left child node of the **STM** node must be a **LDM** node. The light green sub-trees one on the left side of the **LDM** node and one on the right node of the **STM** node. Both light green trees can either be registers only or an **ADD** or **SUB** operation. In the case of an **ADD** or **SUB** node present as the root of the tree the left-hand side of the operation must be a register while the left-hand side can either be a register or an immediate integer.

Listing 3.13 shows a possible match for the described tree in Figure 3.14. The two instructions at the beginning of the listing are the instructions which, after they have been translated to REIL, get matched by the search tree.

Figure 3.15 shows the pre- and post-conditions which must be met for the gadget example in Listing 3.13.

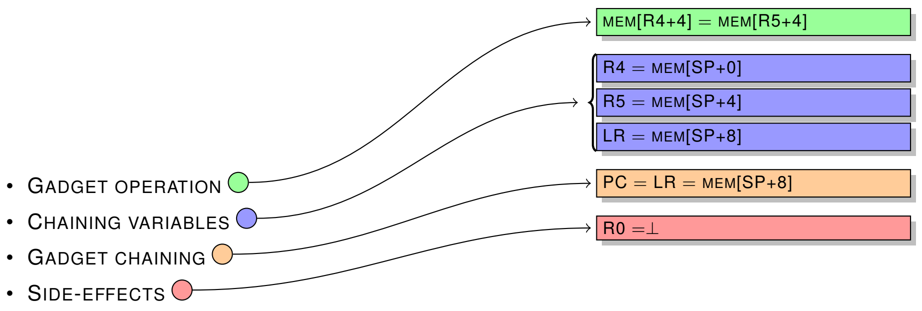


FIGURE 3.15.: MEMORY TO MEMORY GADGET FORM

LISTING 3.14: ARITHMETIC MEMORY OPERATION GADGET EXAMPLE

```
0x03F86EB8    LDR         R1, [R6]
0x03F86EBC    MOV         R0, 1
0x03F86EC0    ADD         R3, R1, R4
0x03F86EC4    ADD         R2, R1, 0x10000
0x03F86EC8    STR         R3, [R6,8]
0x03F86ECC    STR         R2, [R6,12]
0x03F86ED0    STR         R1, [R6,4]
0x03F86ED4    LDMFD       SP!, {R4,R5,R6,LR}
0x03F86ED8    BX          LR
```

3.4.3.3. Memory arithmetic operation gadget

Gadgets for memory arithmetic provide basic arithmetic operations. The target operand for the gadget must be a memory location, the source operands can either be registers or memory locations. It is possible to specify immediate values as operands for the arithmetic operation as right-hand operands. The supported arithmetic operations are:

- MEMORY ADDITION
- MEMORY SUBTRACTION

The search trees for both the memory arithmetic gadgets and the bitwise operation gadgets are identical. The operation specified on the left-hand side of the root node is mandatory for this type of gadget. It differs between all arithmetic and bitwise functions.

The tree to locate a memory operation gadget within the runtime image of the binary introduces a new aspect which is now explained. The left-hand side of the root node is the mandatory operation which can be any of the operations named above. The two trees the left-hand sub-tree of the operation node and the right-hand sub-tree, can be memory load operations. The option that these trees are memory load operations is indicated through the **LDM** instruction present in both child nodes of the operation node. If a **LDM** instruction was matched the possible sub-trees are register only, register with immediate offset, or register with register offset.

Example 3.14 shows a memory addition with two important aspects of the gadget finding process. The first aspect is the complexity of the gadget itself. It computes more than what would be sufficient to match the search tree. The second aspect is that the gadget, even though it is complex and performs instructions which are not needed, is still found. Therefore, even in cases where the number of usable functions is quite small, the algorithm can still find complex combinations of instructions to perform a specific task.

In Figure 3.17 the complexity of the given Listing 3.14 is also present. The gadget taints a wide range of memory cells and registers. Therefore the gadget needs all of the memory location to

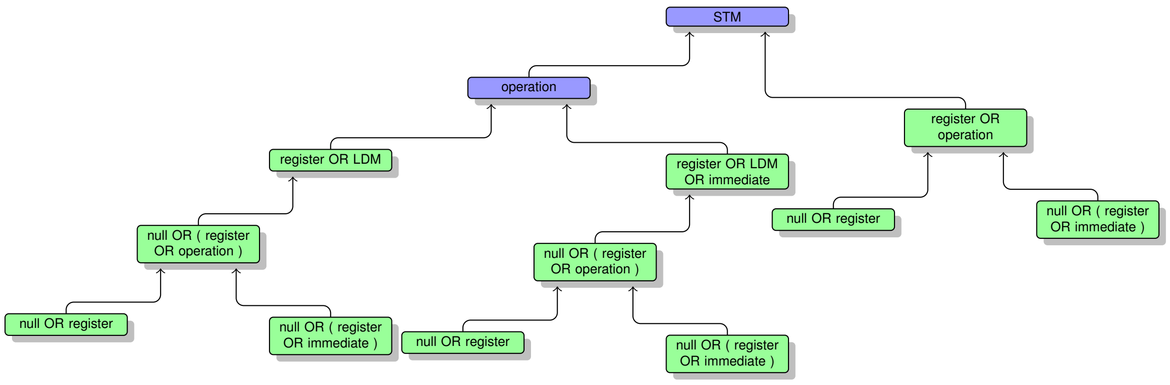

STM
operation
register OR operation
register OR LDM
register OR LDM OR immediate
null OR register
null OR (register OR immediate)
null OR (register OR operation)
null OR (register OR operation)
null OR register
null OR (register OR immediate)
null OR register
null OR (register OR immediate)

FIGURE 3.16.: MEMORY OPERATION GADGET TREE FORM

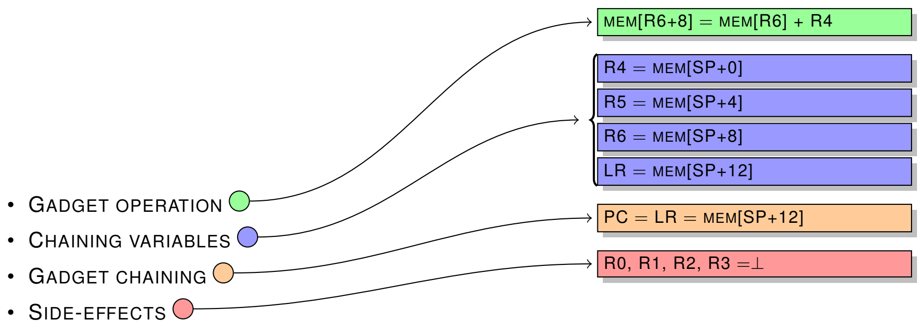


FIGURE 3.17.: ARITHMETIC MEMORY OPERATION GADGET FORM

be accessible. In the context of a return oriented program using this gadget, the memory location should not contain important information which shall not be overwritten.

3.4.3.4. Memory bitwise operation gadgets

The bitwise operation memory gadgets perform the basic bitwise operations. The source operands of the gadget can either be registers or memory locations, the target operand needs to be a memory location. The bitwise operations found by the algorithm are:

- MEMORY AND
- MEMORY OR
- MEMORY XOR
- MEMORY NOT

The tree structure to locate memory bitwise operations is the same as the tree structure for memory arithmetic operations. The operation node in Figure 3.16 would in this case match one of the bitwise arithmetic operations instead of a normal arithmetic operation.

The bitwise memory operation gadget in Listing 3.15 is a leaf type function. Leaf type functions in ARM do only work with function-local registers and do not use the stack. This specific gadget can not work all by itself because the **LR** register which the control flow is transferred to must be set to the next gadget in the gadget chain. To use this gadget we need to use the gadget described as leaf function call gadget in Section 3.4.8.2.

LISTING 3.15: BITWISE MEMORY OPERATION GADGET EXAMPLE

```
0x03FB1D50    LDR         R3, [R2]
0x03FB1D54    ORR         R3, R3, R0
0x03FB1D58    STR         R3, [R2]
0x03FB1D5C    BX          LR
```

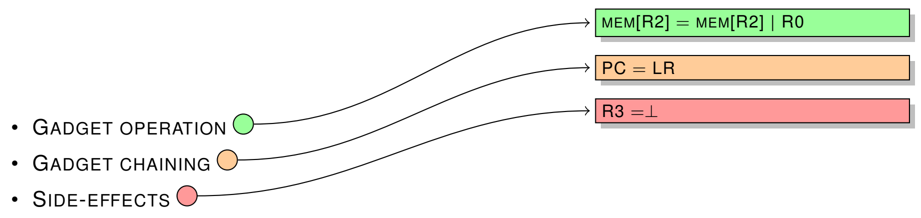


FIGURE 3.18.: BITWISE MEMORY OPERATION GADGET FORM

3.4.4. Memory dereference gadgets

This section describes the gadgets which can be used for memory dereference load and store operations. Basically a memory dereference on the source side of an expression tree is a pointer read and a memory dereference on the target side is a pointer write.

3.4.4.1. Gadget: register to memory dereference (pointer read)

The tree form of both the ”register to memory dereference” and the ”memory to memory dereference” gadgets rely on the same combination of two consecutive **LDM** instructions. The only difference is the operand storage type. The definition of a pointer read in this context can be misleading and is not to be confused with a pointer read in a normal, for example C program. In return oriented programming variables can only be stored in a dedicated memory area present somewhere in the accessible memory as shown in Figure 3.11 item ④. A variable is loaded from memory with a REIL **LDM** instruction and would therefore in the normal convention already be a pointer read. In return oriented programming this would be a normal assignment and only if a pointer has been stored in the variable location, and the value it points to is loaded, a return oriented pointer read is performed. The return oriented pointer read therefore needs two **LDM** instructions one to load a variable from the dedicated memory space and one to load the value where the variable points to into the desired location.

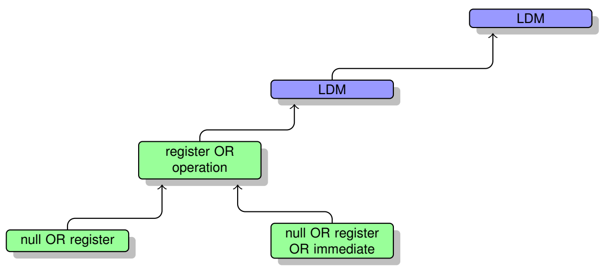


FIGURE 3.19.: REGISTER TO MEMORY DEREFERENCE GADGET TREE FORM

Figure 3.19 shows the two **LDM** instructions necessary for the ”register to memory dereference” gadget. The lower **LDM** instruction can either have a register as child node or an offset tree as

Listing 3.16: Register to memory dereference gadget example

```
1 0x03F85CC0    LDR         R3, [R5,0x10]
2 0x03F85CC4    LDR         R2, [R5]
3 0x03F85CC8    LDR         R1, [R3]
4 0x03F85CCC    ADD         R3, R2, R6LSL2
5 0x03F85CD0    STR         R1, [R3,4]
6 0x03F85CD4    LDMFD       SP!, {R4,R5,R6,LR}
7 0x03F85CD8    BX          LR
```

Listing 3.17: Memory to memory dereference gadget example

```
1 0x03FA75E0    LDR         R3, [R4,12]
2 0x03FA75E4    LDR         R2, [R4,4]
3 0x03FA75E8    MOV         R0, R3
4 0x03FA75EC    LDR         R3, [R3]
5 0x03FA75F0    STR         R3, [R4,8]
6 0x03FA75F4    MOV         LR, PC
7 0x03FA75F8    BX          R2
```

sub-tree.

Listing 3.16 shows an example of a ”register to memory dereference” gadget. Instruction 1 loads a memory cell (**mem[R5+0x10]**) into the the register **R3**. In instruction 3 the content of register **R3** loaded before is now the source for the memory read, which is stored in register **R1**. This specific code sequence could also be used as a ”memory to memory dereference” gadget if the instructions 4 and 5 are taken into account.

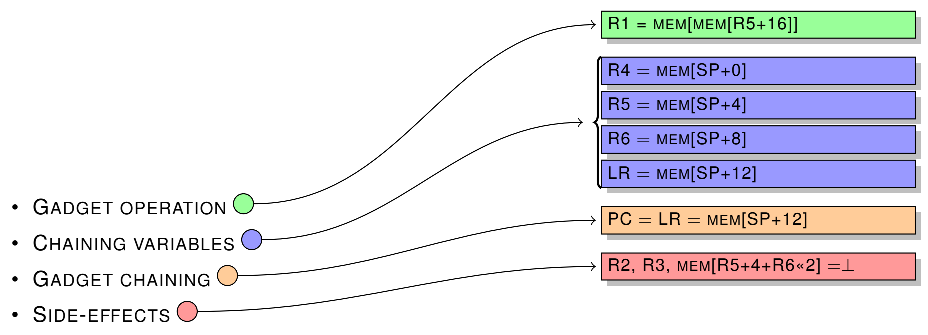


Figure 3.20.: Register to memory dereference gadget form

3.4.4.2. Gadget: memory to memory dereference (pointer read)

The ”memory to memory dereference” gadget closely matches the ”register to memory dereference” gadget, but has a different source for the read pointer, a memory location. The gadget follows the same ideas the ”register to memory dereference” gadget does. Generally all of the ”memory to memory dereference” gadgets which can be found with the algorithms in this thesis are also ”register to memory dereference” gadgets.

Figure 3.16 shows that the target is a memory location. The source side of the search tree has the same characteristics as the ”register to memory dereference” tree. The target side can either be a memory location specified as a register or a memory location which is addressed by an offset to a register.

Listing 3.17 also presents a leaf function but with a different set of restrictions that apply in

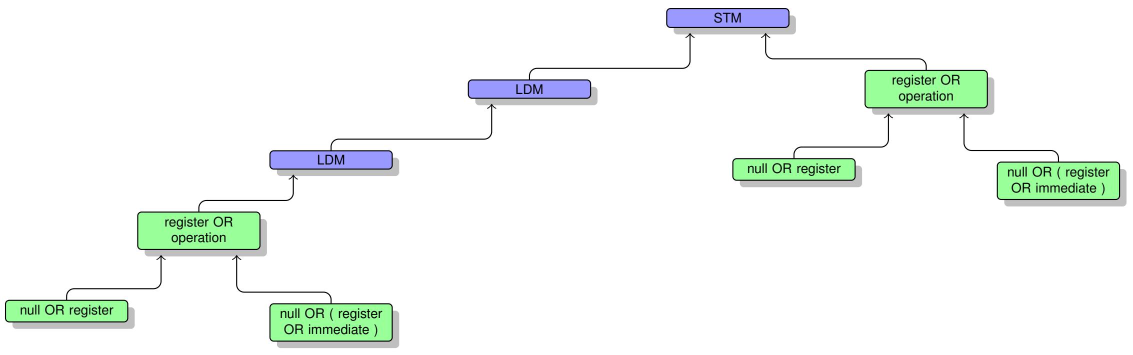


FIGURE 3.21.: MEMORY TO MEMORY DEREFERENCE GADGET TREE FORM

contrast to the earlier example. This time the register **R2** is used in the branch instruction which lowers the usage restrictions of the gadget. Because this specific register is usually user-settable across multiple instruction sequences which is not the case for the **LR** register. In instruction 7 the control flow is passed to the address present in register **R2** which has been loaded from the memory cell addressed by **mem[R4+4]**. Therefore, to use this gadget a program must provide a valid address for control flow passing in the memory location, otherwise the program will execute unintended data.

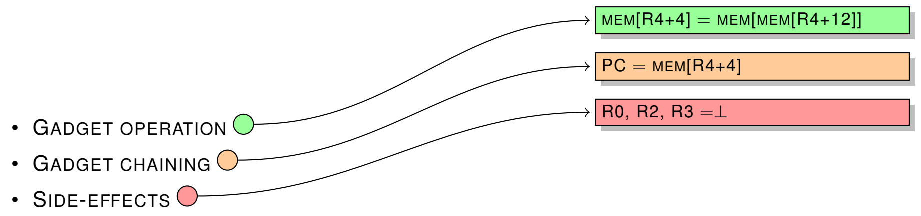


FIGURE 3.22.: MEMORY TO MEMORY DEREFERENCE GADGET FORM

3.4.4.3. Gadget: memory dereference to memory or register (pointer write)

The memory dereference gadgets store a memory or register value into a memory cell. A pointer write in return oriented programming is, as the pointer read operation, different from pointer writes in C code. In return oriented programming a pointer write means that the target where a memory location or a register value will be written to has been loaded from memory prior to the write. The process is, as follows, an address of a variable from an accessible memory location is loaded into a register. Now the **STM** instruction stores a new value in the memory cell were the address points to. Therefore this operation is named pointer write.

The tree presented in Figure 3.23 describes what has to be present in an operand tree map for this type of gadget to be located. On the right-hand side can be seen what differentiates this gadget from all other gadget types. It locates instructions, which have a memory load instruction **LDM** from a location prior to the write memory operation **STM** to this location.

In Listing 3.18 instruction 1 loads a variable from memory into the register **R2**. Instruction 3 then stores the contents of register **R3** into the variable loaded from memory. Two important aspects must be taken into account with the example gadget. The first aspect is that the initial load is performed from a stack offset, which might be used in the next gadgets as input. The second aspect is that an addition was performed in instruction 2 which affects the source register. Both

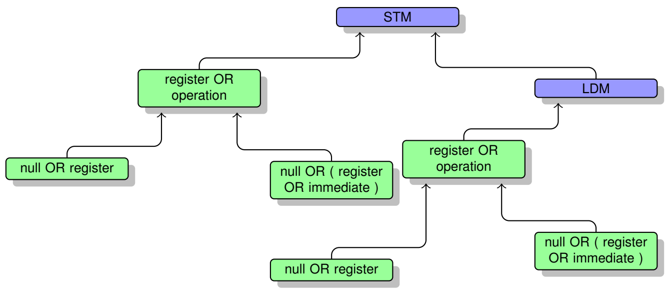


FIGURE 3.23.: MEMORY DEREFERENCE TO REGISTER GADGET TREE FORM

LISTING 3.18: MEMORY DEREFERENCE TO MEMORY GADGET EXAMPLE

```
0x03F5F11C    LDR          R2, [SP,8]
0x03F5F120    ADD          R3, R3, 1
0x03F5F124    STR          R3, [R2]
0x03F5F128    LDMFD        SP!, {R4,LR}
0x03F5F12C    BX           LR
```

of the mentioned aspects must be considered when this gadget is to be used in a return oriented program.

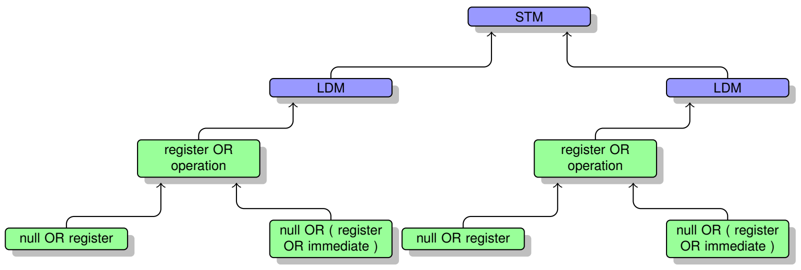


FIGURE 3.24.: MEMORY DEREFERENCE TO MEMORY GADGET TREE FORM

The tree in Figure 3.24 has in contrast to Figure 3.23 a memory load for both sides of the memory store instruction. Therefore the tree locates instruction sequences where a memory location is written to a memory dereference.

Listing 3.19 shows a ”memory dereference to memory” gadget. In Instruction 1 the memory location addressed by register **R5** is loaded into register **R3**. In instruction 2 a stack offset is loaded into the register **R2**. Instruction 4 stores the value in register **R3** into the memory location addressed by register **R2**. Therefore the memory location where **R5** initially pointed to is now stored at the address where the stack offset pointed too, which is a return oriented pointer write.

Like prior gadgets, the conditions of this gadget 3.25 need to be analysed closely for potential pitfalls. The reference to a stack offset must always be taken into account when using such a gadget as following gadgets might use the value as input. Not all gadgets of this type have this limitation but as an example a problematic candidate does provide more insight then a perfect candidate.

3.4.5. Register gadgets

Register gadgets are gadgets where the result of the computation of the gadget is stored in a register. The source of the computation can either be a register or a memory location. A

LISTING 3.19: MEMORY DEREFERENCE TO MEMORY GADGET EXAMPLE

```
0x03F768D0    LDR        R3, [R5]
0x03F768D4    LDR        R2, [SP,0x50]
0x03F768D8    MOV        R0, R10
0x03F768DC    STR        R3, [R2]
0x03F768E0    ADD        SP, SP, 0x18
0x03F768E4    LDMFD      SP!, {R4,R5,R6,R7,R8,R9,R10,R11,LR}
0x03F768E8    BX         LR
```

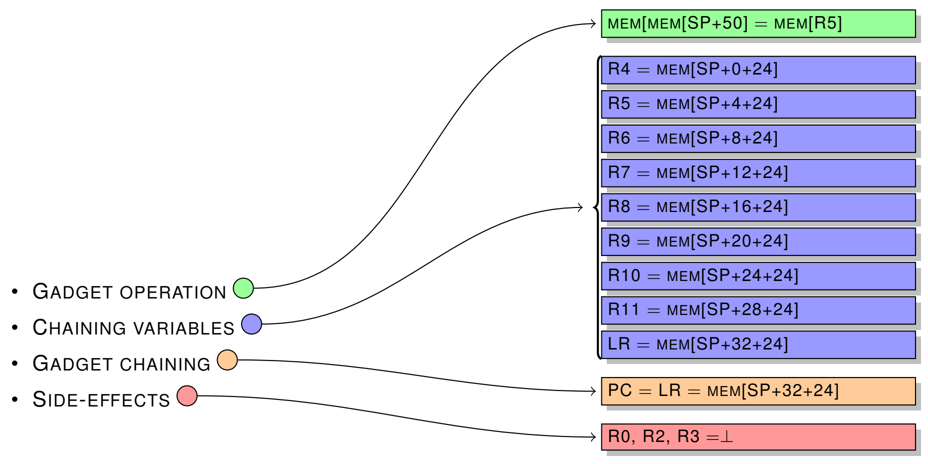


FIGURE 3.25.: MEMORY DEREFERENCE TO MEMORY GADGET FORM

computation is understood as a change in value of the target register.

3.4.5.1. Gadget: Register to register

The ”register to register” gadget copies the contents of the source register into the target register. This gadget is very simple but useful, especially on the ARM architecture. Because the encountered calling conventions take arguments from the registers **R1-R3**, but they are almost always local to the current function and do not get overwritten by the stack restore in the functions epilogue. Also, these registers are often used inside the function to perform function local tasks. Therefore the register to register gadget can be used to copy values from a register that can be set by the stack restore instruction into the function local registers. The layout of the gadget is very simple as it allows no other operands besides registers as source and target.

The ”registers to registers” gadget can be used if multiple register value copies are needed, this is the case for the function call gadget which is discussed in 3.4.8.

FIGURE 3.26.: REGISTER TO REGISTER GADGET TREE FORM

The ”register to register” gadget only needs a very simple search tree as the gadget itself is very simple. It is sufficient, if an entry in the operand tree map only has a register as root node and no other nodes.

Listing 3.20 shows an example finding which is more complex then the average gadget present in almost all binaries analysed. Instruction 1 performs the desired operation that is searched for.

LISTING 3.20: REGISTER TO REGISTER EXAMPLE

```
0x03FAF4C0    MOV        R5, R4
0x03FAF4C4    MOV        R4, 0
0x03FAF4C8    SUB        R3, R3, 0x1F
0x03FAF4CC    SUB        R3, R3, R12
0x03FAF4D0    LDMFD      SP!, {R1,PC}
```

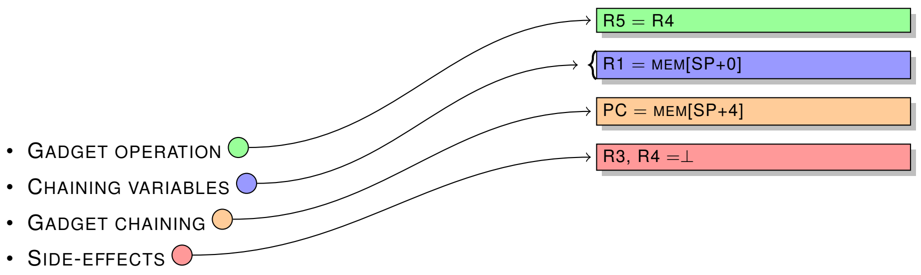


FIGURE 3.27.: REGISTER TO REGISTER GADGET FORM

3.4.5.2. Gadget: Register to constant

The ”register to constant” gadget is present in two different types. The first type searches for instruction sequences where a register is set to zero. The second type searches for all other constants besides zero. The layout of the gadget is simple as it allows only registers as targets and integers as sources. The differentiation between zero and other constants is made because zero constants are of much greater use then almost any other constant.

FIGURE 3.28.: REGISTER TO CONSTANT GADGET TREE FORM

Like the tree to locate a ”register to register” gadget, the tree to locate a ”register to constant” gadget is also very simple.

Listing 3.21 shows an example of how a ”register to constant” gadget looks in native assembly code. Instruction 1 is matched by the tree in Figure 3.28. This gadgets also shows that not all found gadgets really make sense for the use in a program but are still matched.

Stack pointer relative register restores have been omitted for Figure 3.29 to get the correct offsets add 0x63C to the stack pointer. This specific behaviour is explained in the next gadget.

3.4.5.3. Gadget: register to memory

The ”register to memory” gadget copies a value from a memory location and stores it into a register. The source operand of this gadget must be a memory location and the target operand must be a register. This gadget can be used in combination with a register operation gadget if the gadget itself does not load values from memory but a memory variable is the desired input for the operation.

Often all trees which provide similar functionality to the trees explained in the memory location only differ in the target operand. While in the memory section the target is always a memory location in the register section the target is a register. The tree in Figure 3.30 shows the tree which is used to locate a ”register to memory” gadget.

LISTING 3.21: REGISTER TO CONSTANT EXAMPLE

```
0x03F8AA24    MOV          R12, 0x63C
0x03F8AA2C    ADD          SP, SP, R12
0x03F8AA30    LDMFD        SP!, {R4,R5,R6,R7,R8,R9,R10,R11,LR}
0x03F8AA34    BX           LR
```

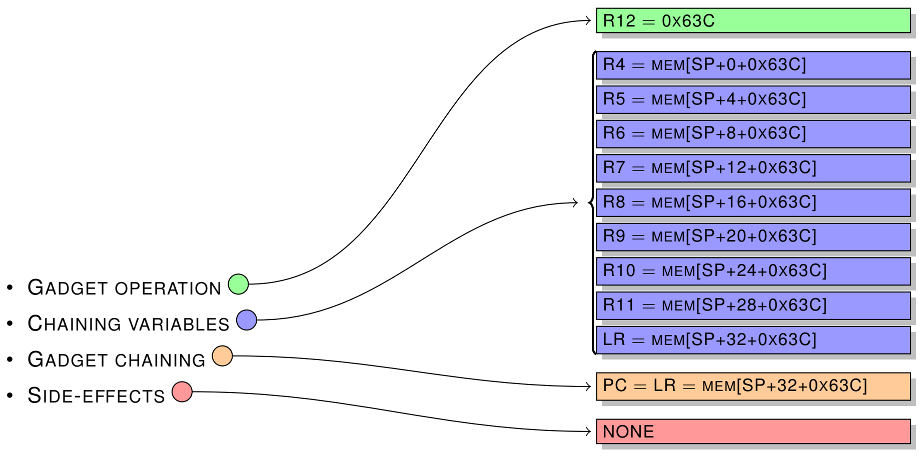


FIGURE 3.29.: REGISTER TO CONSTANT GADGET FORM

LISTING 3.22: REGISTER TO MEMORY EXAMPLE

```
0x03F5EC9C    LDR          R0, [R4,4]
0x03F5ECA0    ADD          SP, SP, 8
0x03F5ECA4    LDMFD        SP!, {R4,R5,LR}
0x03F5ECA8    BX           LR
```

Listing 3.22 shows an example of a ”register to memory” gadget. The register **R0** is loaded with the value from the memory location addressed by **mem[R4+4]**. The example has been chosen because it shows an often encountered stack shift. Stack shifts are curse and blessing: On the one hand they might enable certain stack offsets to be used in other gadgets, on the other hand stack shifts waste precious stack space which (especially on Windows Mobile) is very sparse.

3.4.5.4. Register arithmetic gadgets

The arithmetic gadgets for registers can be used to perform basic arithmetic operations. The target operand type of this gadget must be a register whereas the source operands can either be registers or memory locations. The available arithmetic operations are:

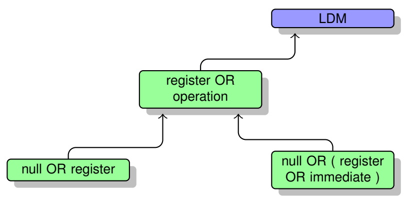


FIGURE 3.30.: REGISTER TO MEMORY GADGET TREE FORM

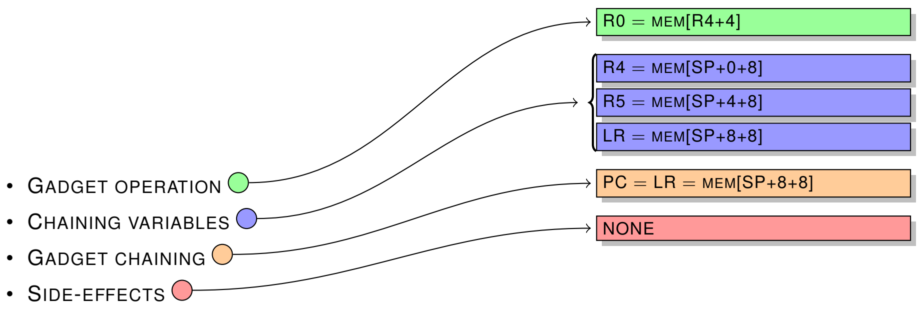


FIGURE 3.31.: REGISTER TO MEMORY GADGET FORM

LISTING 3.23: REGISTER ADDITION EXAMPLE

```
0x03F79EDC    LDR         R3, [R4,0xA8]
0x03F79EE0    ADD         R3, R1, R3
0x03F79EE4    CMP         R3, R2
0x03F79EE8    MOVEQ       R0, 2
0x03F79EEC    MOVNE       R0, 0
0x03F79EF0    ADD         SP, SP, 0x10
0x03F79EF4    LDMFD       SP!, {R4,R5,LR}
0x03F79EF8    BX          LR
```

- REGISTER ADDITION
- REGISTER SUBTRACTION

All of the arithmetic gadgets have the same tree form which is displayed in Figure 3.32.

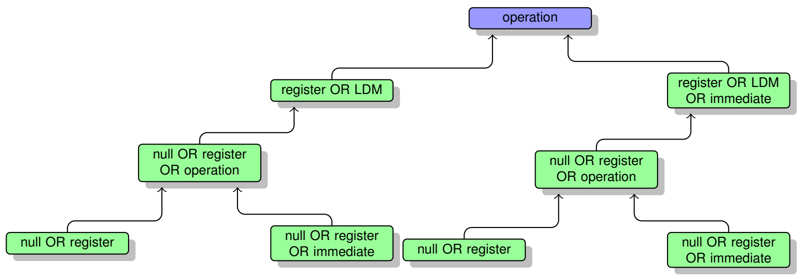

operation
register OR LDM
register OR LDM OR immediate
null OR register OR operation
null OR register OR operation
null OR register
null OR register OR immediate
null OR register
null OR register OR immediate

FIGURE 3.32.: REGISTER TO OPERATION GADGET TREE FORM

The tree to search for register operation gadgets in Figure 3.32 allows both sides of the operation to either be registers or variables loaded from memory, and also that the right-hand side of the operation is an immediate integer. The target of the tree is always a register stored in the operand tree map.

Register addition has been selected for the register arithmetic gadget.

Listing 3.23 shows an example of a ”register arithmetic” gadget where the left-hand side of the operation is a memory location. Instructions 1 and 2 in combination are matched by the search tree.

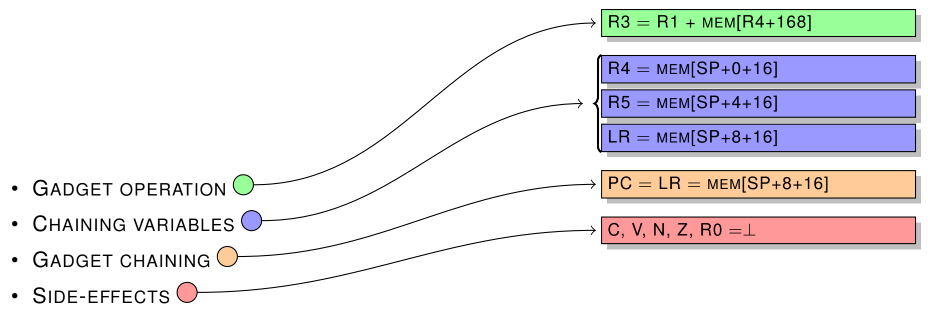


Figure 3.33.: Register to operation gadget form

Listing 3.24: Register **XOR** example

```
1 0x03F953C4   MOV     R5, 0xFF00
2 0x03F953C8   EOR     R2, R3, R7
3 0x03F953CC   ORR     R5, R5, 0xFF
4 0x03F953D0   AND     R3, R2, R5
5 0x03F953D4   EOR     R0, R3, R2LSR0x10
6 0x03F953D8   ADD     SP, SP, 8
7 0x03F953DC   LDMFD   SP!, {R4,R5,R6,R7,LR}
8 0x03F953E0   BX      LR
```

3.4.5.5. Register bitwise operation gadgets

The bitwise operation gadgets for registers perform bitwise arithmetic operations. The target operand type for bitwise gadgets must be a register while the source operands can either be registers or memory locations. The available bitwise operations are:

- REGISTER AND
- REGISTER OR
- REGISTER XOR
- REGISTER NOT
- REGISTER NEGATION

Bitwise operation gadgets are located with the same algorithms which are used to locate arithmetic gadgets. This leads to a similar tree form for both types of gadgets. The **XOR** operation has been selected as an example for the register bitwise operation.

In Listing 3.24 instruction 2 is the instruction which is matched by the search tree. The **XOR** instruction was chosen as an example because it shows that bitwise operations are more complicated to locate inside ARM binaries then other instructions. Although this is not true for all bitwise arithmetic operations, **XOR** instructions are a difficult target at least for Windows Mobile libraries. It is assumed that this is due to the Visual Studio compiler.

Even though the complexity of the selected assembly listing is quite high, the resulting pre- and post-conditions are surprisingly simple as only registers get used inside the gadget and the only memory usage is the restoring of stack variables.

3.4.5.6. Shift gadgets

Shift operation gadgets are treated different on the ARM architecture as there are no *real* shift instructions present in the 32 bit instruction set. Shift operations always take place in the ARM

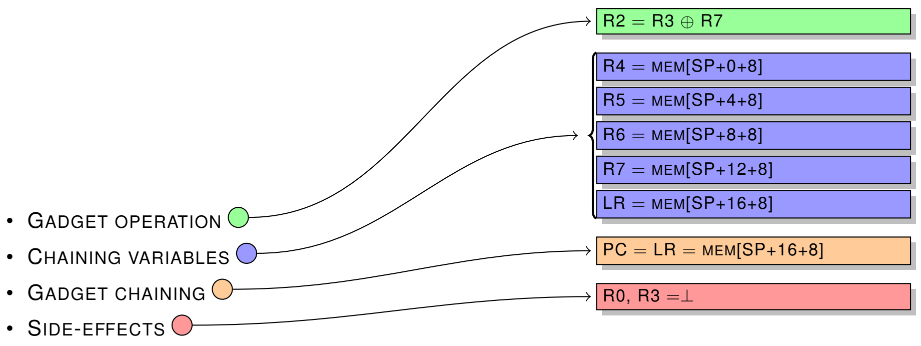


FIGURE 3.34.: REGISTER TO BITWISE OPERATION GADGET FORM

barrel shifter operand. To find a shift instruction for the 32 bit instruction set one has to really find a **MOV** instruction which uses the barrel shifter. This leads to a gadget searching routine for shifts that is more complex than the normal register operation gadgets which have been presented in Section 3.4.5.4 and Section 3.4.5.5.

Another problem that was encountered while determining the search tree of a shift gadget was that two different matching trees are needed to locate all possible shift operations.This is due to the fact that REIL, in its current release, does not differentiate between right and left shifts by mnemonic, but by the sign of the second REIL operands value. With an immediate shift parameter this does not cause problems in the analysis process. However if the shift parameter is a register the sign has to be set with one more REIL instruction which then has to be matched by the algorithm as well.

Due to the discussed issues with shift operations both trees for the two locatable shifts are presented.

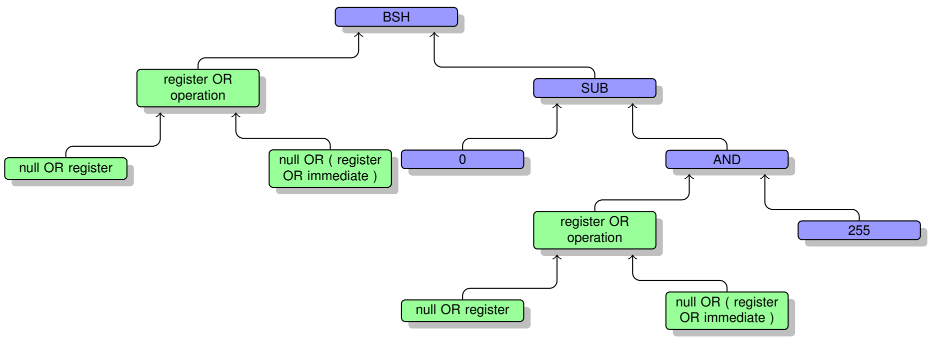


FIGURE 3.35.: REGISTER RIGHT SHIFT GADGET TREE FORM

Both trees which are used to search for shift gadgets require special mandatory nodes to be present. The special mandatory nodes are in case of Figure 3.35 the **SUB** node and its left child node with the value zero. This combination is used to adjust the sign in REIL and therefore must be matched by the tree. In Figure 3.36 the nodes needed for the ”right shift” are missing as the operation which is to be located is a ”left shift”.

As an example, a register left shift operation is shown in Listing 3.25. As explained above, a register shift does not exist as a standalone instruction in the 32 bit ARM instruction set. Therefore the located instruction is a **MOV** instruction which uses the **LSL** option of the barrel shifter.

The listing also provides an argument to the **LDMFD** instruction, which has not been explained yet, the **PC** register. If the **PC** register is present as argument for the **LDMFD** instruction no ARM

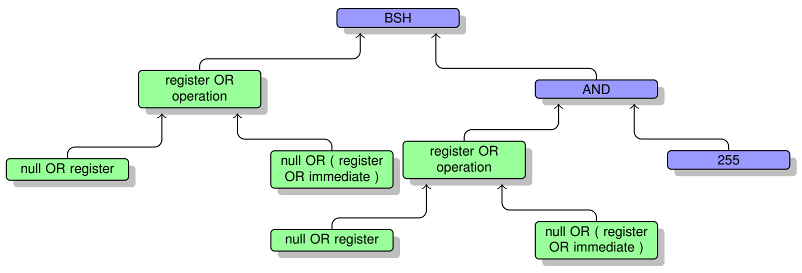


FIGURE 3.36.: REGISTER LEFT SHIFT GADGET TREE FORM

LISTING 3.25: REGISTER LEFT SHIFT EXAMPLE

```
0x03FAF3B0    MOV         R2, R2LSLR12
0x03FAF3B4    SUB         R1, R1, R12
0x03FAF3B8    ADD         R1, R1, 1
0x03FAF3BC    LDMFD       SP!, {R3,PC}
```

/ THUMB interworking is possible, because the **LDMFD** instruction does not set the necessary bits which would get set if a **BX** instruction was present.

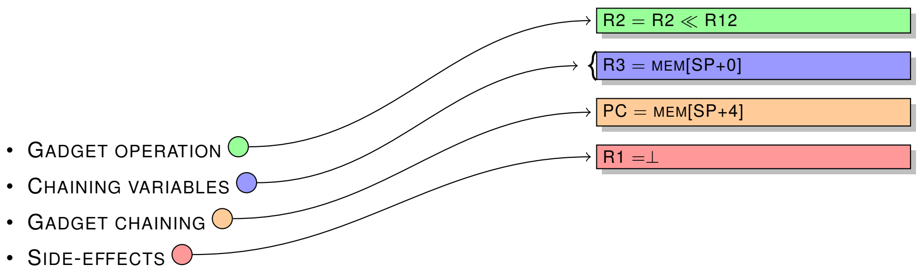


FIGURE 3.37.: REGISTER LEFT SHIFT GADGET FORM

3.4.6. Flags

The gadget in the flags section locates instruction sequences where one or more flags are modified. This gadget can be used in combination with conditional call gadgets to build a conditional control flow gadget. The operation of the gadget is basically a comparison between different source operands. A source operand can either be a register, memory location, or an immediate integer.

Even though it seems as if the tree should search for a compare instruction, it does search for the result of a compare instruction. The reason for this is that several instructions, in most of the assembly languages today, may set flags even if they are not compare instructions. In the ARM architecture all arithmetic instructions which are suffixed with a **S** set flags according to the result of the arithmetic.

As an example Listing 3.26 shows a match of the search tree in Figure 3.38. In the example a real compare instruction (**CMP**) was found.

Figure 3.39 shows the pre- and post-conditions of Listing 3.26. The important aspect which has to be kept in mind when a compare gadget has been located is that all of the flags available in the ARM architecture are set. While different instructions might set flags differently, all conditional

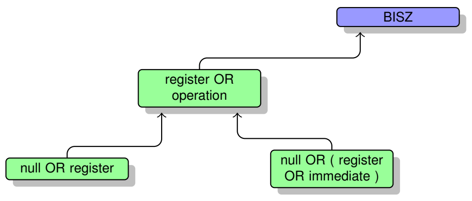


FIGURE 3.38.: COMPARE GADGET TREE FORM

LISTING 3.26: COMPARE GADGET EXAMPLE

```
0x03FAE6AC    CMP       R1, R3
0x03FAE6B0    STRHIB    R1, byte [R2,11]
0x03FAE6B4    LDMFD     SP!, {R4,LR}
0x03FAE6B8    BX        LR
```

executions that might be present in instruction sequences, which are executed after a compare gadget, are influenced by the flags set here. Therefore crafting a return oriented program with compare gadgets is more difficult then crafting one without conditional execution.

3.4.7. Control Flow gadgets

The control flow gadgets section describes a gadget type which is used to alter control flow directly. Normal gadgets alter the control flow with the final return statement. The gadgets described here alter the control flow conditionally or unconditionally.

3.4.7.1. Gadget: branch Always

The branch always gadget is a gadget which basically pops registers off the stack and transfers control to the next gadget. This is what every gadget, which is not part of a leaf function, performs in the last two instructions. As there is no real need to find a "branch always" gadget, no search pattern has been introduced to locate it specifically but if it would be needed later on, adding it could easily be done.

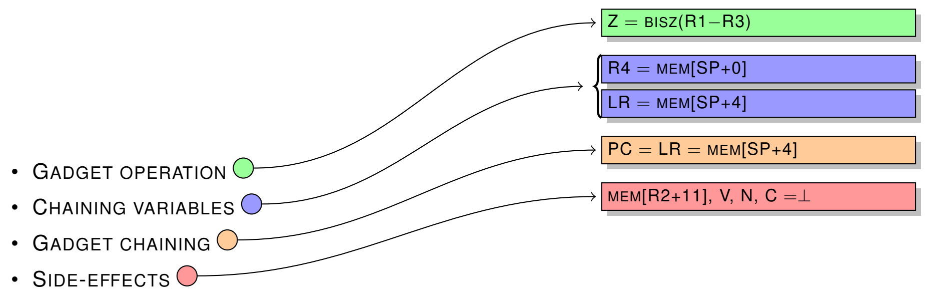


FIGURE 3.39.: COMPARE GADGET FORM

LISTING 3.27: CONDITIONAL BRANCH GADGET EXAMPLE

```
0x03F632E4    BXNE         R4
0x03F632E8    LDMFD        SP!, {R4,R5,R6,R7,LR}
0x03F632EC    BX           LR
```

3.4.7.2. Gadget: branch conditionally

The branch conditionally gadget alters the control flow of the program based on the current state of the flags. Multiple flags exist in the ARM architecture which specify different conditions. These are explained in Section 3.1.1.2.

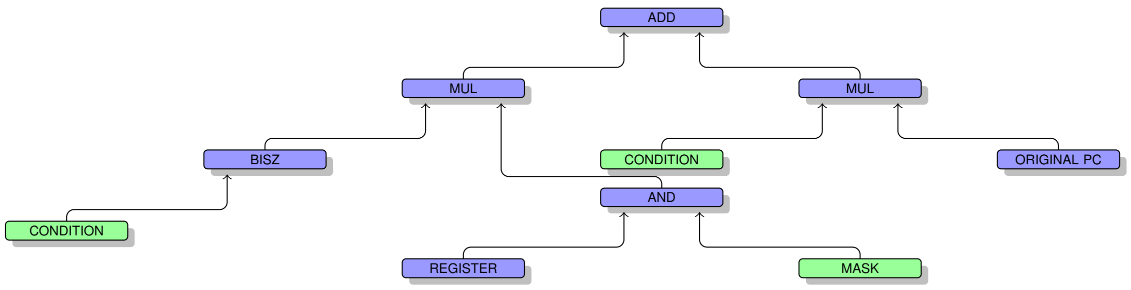


FIGURE 3.40.: CONDITIONAL BRANCH TREE FORM

Tree 3.40 is currently the most complex tree to search for a specific gadget type. This is due to the general concept of how conditional instructions are handled by the algorithms in this thesis. For every conditional instruction a true and a false tree are generated each with the according condition which has to be fulfilled. Both of the trees must be present to locate a conditional instruction. In the special case of locating a ”conditional branch” gadget the original register **PC-ORIG** must be present in the left-hand tree.

As an example of a ”conditional branch” gadget, Listing 3.27 has been selected. For Windows Mobile only suitable gadgets which either branch on equal or not equal could be located.

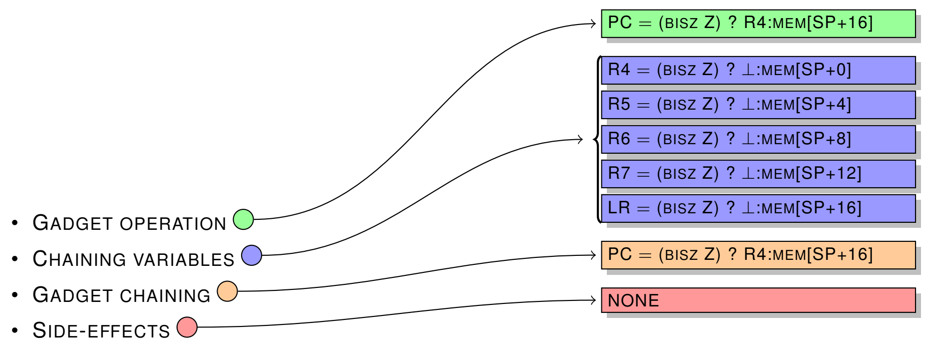

GADGET OPERATION
CHAINING VARIABLES
GADGET CHAINING
SIDE-EFFECTS
PC = (BISZ Z) ? R4:MEM[SP+16]
R4 = (BISZ Z) ? ⊥:MEM[SP+0]
R5 = (BISZ Z) ? ⊥:MEM[SP+4]
R6 = (BISZ Z) ? ⊥:MEM[SP+8]
R7 = (BISZ Z) ? ⊥:MEM[SP+12]
LR = (BISZ Z) ? ⊥:MEM[SP+16]
PC = (BISZ Z) ? R4:MEM[SP+16]
NONE

FIGURE 3.41.: CONDITIONAL BRANCH GADGET FORM

Figure 3.41 shows the pre- and post-conditions of Listing 3.27. As the outcome of the gadget is not determined upon invocation, all registers that might only be changed if the condition is false have been placed in round brackets.

LISTING 3.28: FUNCTION CALL GADGET EXAMPLE

```
0x03F91064    MOV          R3, R5
0x03F91068    MOV          R2, R6
0x03F9106C    MOV          R1, R7
0x03F91070    MOV          R0, R8
0x03F91074    MOV          LR, PC
0x03F91078    BX           R4
```

3.4.8. Function call gadgets

The function call gadgets are instruction sequences which enable the use of any native function in the current library. They set up a specific amount of registers which can be passed to the native function.

There are two different function call gadgets, the normal function call gadget and the leaf function call gadget. The difference between normal functions and leaf functions is that normal functions, at least for the encountered calling conventions, take care of the stack within the called function. This implies that the stack of the caller must be saved within the functions prologue and restored within the functions epilogue. The normal function call gadget is explained in 3.4.8.1.

Leaf functions on the other hand do not use the stack. Therefore they don't need to save or restore the stack frame of the caller. The leaf function call gadget is explained in 3.4.8.2.

3.4.8.1. Gadget: normal function call

The normal function call gadget is used to call functions which use the stack and restore the caller's stack in their function epilogues. The function call gadget enables the use of any function present in the library with up to four arguments. This is the maximum number of parameter usually given to a ARM subroutine. To be able to use the library function, the target function address must be set by the stack frame which is restored by the previous gadget and the target function prologue must be omitted from execution by taking an entry point below the stack frame save instruction. An example how the gadget is supposed to be used is presented in Chapter 6.3.

This jump to the instruction below the stack frame save instruction prevents the program from being lost in an endless loop or continuing execution at a location where control for the program is lost.

Upon termination of the called function, the stack frame will be restored as it is with normal gadgets. Therefore any function that is to be used in the return oriented program must be analysed with regard to the possible stack frame use and optional memory writes which can corrupt memory locations in use by the return oriented programs.

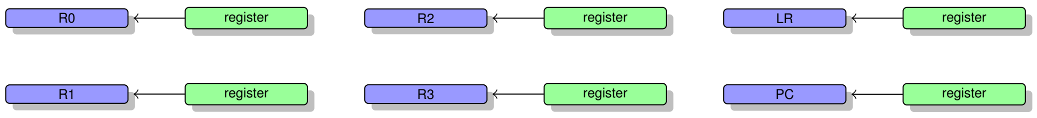


FIGURE 3.42.: FUNCTION CALL TREE FORM

The tree form in Figure 3.42 is basically an extension to the "registers to registers" gadget described earlier. The major difference is that certain registers must be present for the gadget while for the "registers set to registers" gadget this was not mandatory.

Listing 3.28 shows how a possible candidate for the "function call gadget" looks like.

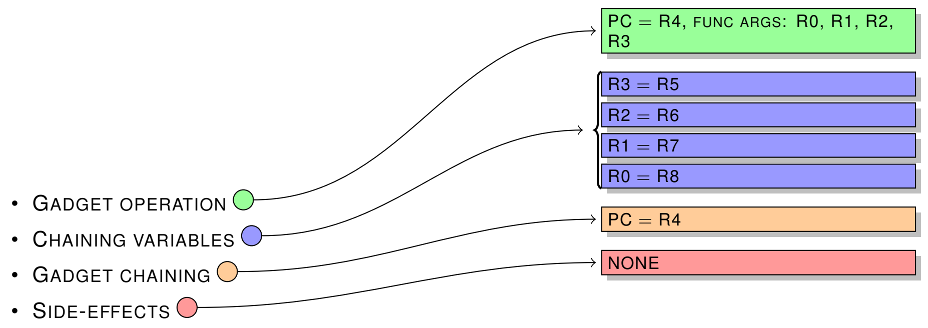


FIGURE 3.43.: FUNCTION CALL GADGET FORM

3.4.8.2. Gadget: leaf function call

The leaf function gadget is very similar to the conditional branch gadget presented in 3.4.7.2 with the major difference that in case of a leaf function call no conditional initial call instruction is permitted. One might ask if a normal function call gadget is not enough for a return oriented program to have a reasonable amount of computational power. While this can be true, it must not be the case for all encountered binaries. Therefore this gadget makes it possible to use leaf functions and leaf type gadgets.

As a leaf function or gadget does not restore the stack and just returns back to the address specified in the link register, the leaf function call gadget must set the link register to the instruction following the initial call to the gadget. When the called function or gadget has completed, control is passed back to the gadget. Therefore, the next instructions in the caller function must be the normal stack restoring instructions which build the function epilogue.

Hence, the leaf function call gadget can be understood as a wrapper which wraps the normal function epilogue around a set of instructions which do not normally have the function epilogue. An example for the use of the leaf function call gadget is presented in Chapter 6.3.

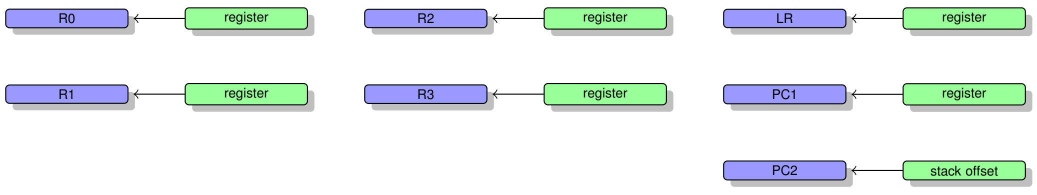


FIGURE 3.44.: LEAF FUNCTION CALL TREE FORM

The ”leaf function call” gadget can be matched with a tree similar to the function call gadget. The major difference is that the execution after the initial function call must be terminated only by the normal function epilogue instructions. Even though one could think of possible cases where this strict rule does not make perfect sense, to avoid side effects after the execution of a leaf function the rule makes calling leaf functions straightforward and predictable.

Listing 3.29 shows an example for the ”leaf function call” gadget.

3.4.9. System Call gadget

System call gadgets are special because they are needed to have a complete gadget set which is able to perform all the tasks of a normal program. These gadgets are implemented differently

LISTING 3.29: LEAF FUNCTION CALL GADGET EXAMPLE

```
0x03F91064    MOV        R3, R5
0x03F91068    MOV        R2, R6
0x03F9106C    MOV        R1, R7
0x03F91070    MOV        R0, R8
0x03F91074    MOV        LR, PC
0x03F91078    BX         R4
0x03F9107C    LDMFD      SP!, {R4,R5,R6,R7,R8,LR}
0x03F91080    BX         LR
```

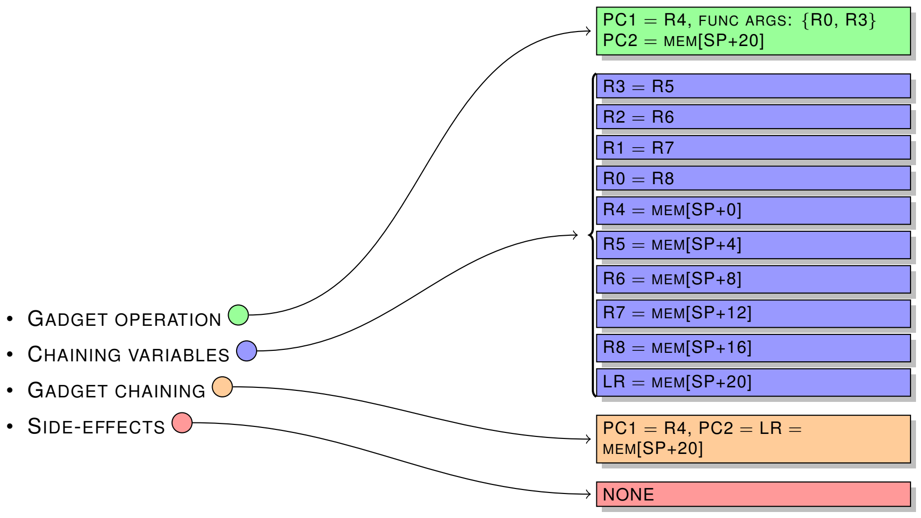


FIGURE 3.45.: LEAF FUNCTION CALL GADGET FORM

across the various operating systems. Therefore, in contrast to the gadgets which have been discussed so far, the system call gadget described here is only useful for Windows Mobile.

As described in Section 3.1.2.9, Windows Mobile does not use the **SWI** instruction to implement system calls. It rather uses a call to an invalid address in the trap area to perform a system call. For making a system call with return oriented programming the 3.4.8.2 gadget is used to perform the system call.

4. Algorithms for automatic gadget searching

To be able to build a return-oriented program the necessary parts for this program must be located in the binaries of the application. In this thesis the search for gadgets is performed automatically by a set of algorithms. This chapter describes these algorithms for automatic gadget searching developed in this thesis. The algorithms are divided into logical stages which can be roughly categorized as follows:

The first stage is the data collection stage in which the binaries are analysed. The second stage is the data merging stage in which the collected data is refined. The third stage is the matching stage where specific gadgets are located by comparing them against a set of expression trees.

4.1. Stage I

In order to be able to locate gadgets within a binary, this binary needs to be analysed and all necessary data the binary provides must be collected. The algorithms of the first stage collect the necessary information and store them in such a way, that the subsequent algorithms can access the data.

Collecting data from the binary is performed by two algorithms. The first algorithm is used to extract expression trees for a single native instruction. The second algorithm is used to extract path information.

4.1.1. Reverse walker algorithm

Both of the two algorithms are built upon a stack based reverse walker algorithm. This algorithm starts at a **free branch** instruction. It then traverses the instructions and basic blocks in reverse execution order, until either the user defined threshold is reached or no more predecessor instructions exist.

Each algorithm by itself uses the same code for traversing the instructions but has its own callback function where the logic resides in.

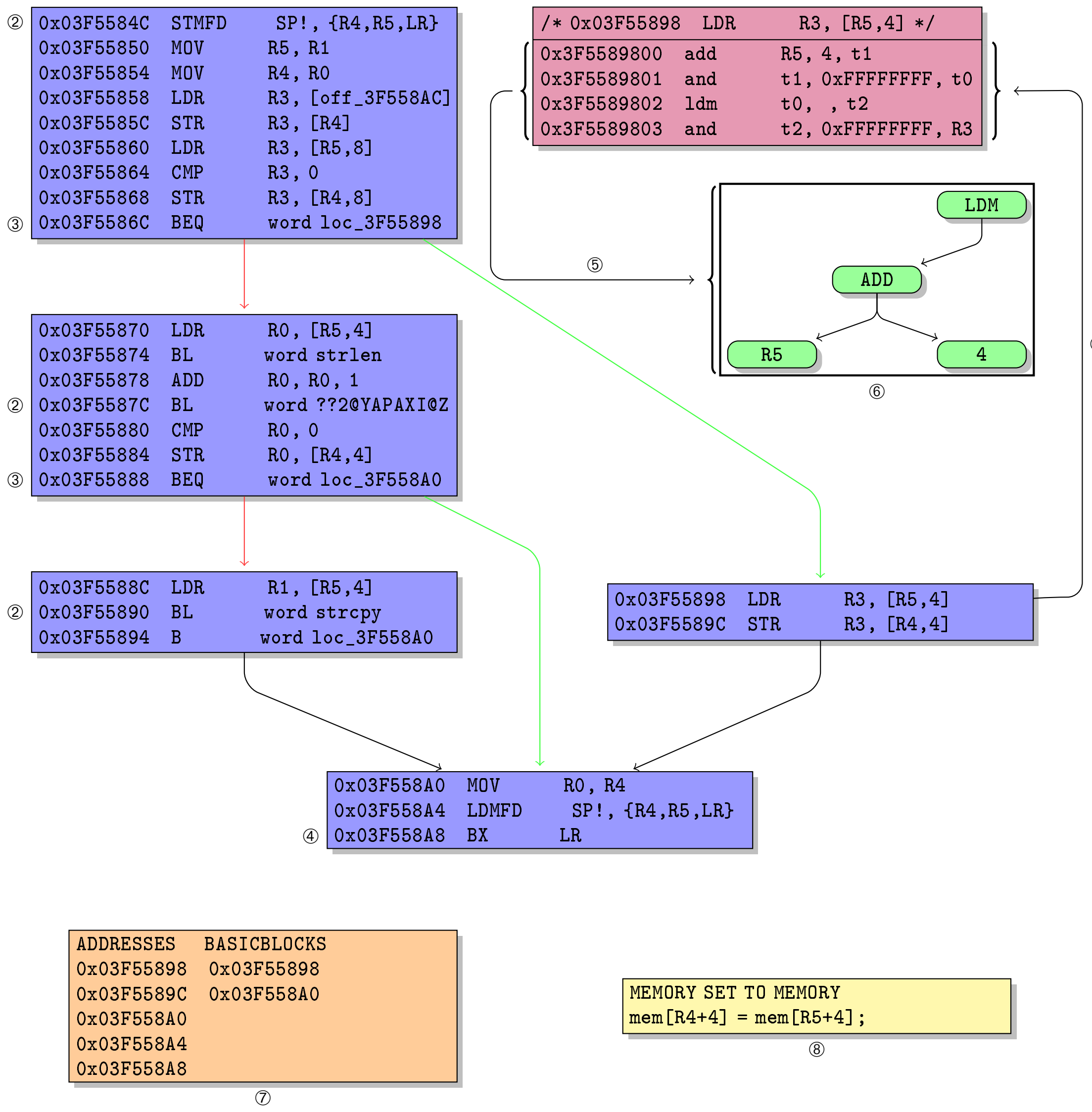
4.1.2. Algorithm: Expression tree extraction

To be able to construct gadgets it is necessary to have precise knowledge about how a single native instruction influences registers, flags and memory cells. Also specifically in case of the ARM architecture, where instructions can be conditional. Therefore it is necessary to be able to decide whether a certain native instructions effects have to be taken into account or not. This knowledge is gained by using the expression tree extraction algorithm. It builds a single map for each native instruction which holds the trees for all influenced registers, flags and memory cells.

The expression tree extraction algorithm works as follows: A native instruction is translated into the REIL meta-language. The translation results in a graph (*ReilGraph*) consisting of nodes (*ReilNodes*). Each node in the graph holds one or more instructions (*ReilInstructions*).

This graph is traversed from top to bottom, instruction by instruction. For each instruction the mnemonic of the instruction determines the handler for further processing.

The result of each handler is a tree that contains information about the effects the REIL instruction had on one specific register, flag, or memory cell. This tree is stored in a map which uses



```
0x03F5584C  STMFD     SP!, {R4,R5,LR}
0x03F55850  MOV       R5, R1
0x03F55854  MOV       R4, R0
0x03F55858  LDR       R3, [off_3F558AC]
0x03F5585C  STR       R3, [R4]
0x03F55860  LDR       R3, [R5,8]
0x03F55864  CMP       R3, 0
0x03F55868  STR       R3, [R4,8]
0x03F5586C  BEQ       word loc_3F55898
```

```
0x03F55870  LDR       R0, [R5,4]
0x03F55874  BL        word strlen
0x03F55878  ADD       R0, R0, 1
0x03F5587C  BL        word ??2@YAPAXI@Z
0x03F55880  CMP       R0, 0
0x03F55884  STR       R0, [R4,4]
0x03F55888  BEQ       word loc_3F558A0
```

```
0x03F5588C  LDR       R1, [R5,4]
0x03F55890  BL        word strcpy
0x03F55894  B         word loc_3F558A0
```

```
/* 0x03F55898  LDR       R3, [R5,4] */
0x3F5589800  add       R5, 4, t1
0x3F5589801  and       t1, 0xFFFFFFFF, t0
0x3F5589802  ldm       t0, , t2
0x3F5589803  and       t2, 0xFFFFFFFF, R3
```

```
0x03F55898  LDR       R3, [R5,4]
0x03F5589C  STR       R3, [R4,4]
```

```
0x03F558A0  MOV       R0, R4
0x03F558A4  LDMFD     SP!, {R4,R5,LR}
0x03F558A8  BX        LR
```

```
ADDRESSES   BASICBLOCKS
0x03F55898  0x03F55898
0x03F5589C  0x03F558A0
0x03F558A0
0x03F558A4
0x03F558A8
```

```
MEMORY SET TO MEMORY
mem[R4+4] = mem[R5+4];
```

① Exemplary REIL translation which is performed for each native instruction for use in algorithm 1.

② Instruction or condition which terminates the path search algorithm 8.

③ Conditional branch instruction, which leads to a "COND" prefixed expression tree in algorithm 10.

④ Controllable control flow altering instruction where the extraction process 4.1.2 begins.

⑤ Expression tree extraction from REIL translation performed in algorithm 1.

⑥ Multiple expression trees get merged with the path information in algorithm 9.

⑦ Possible path for this function which has been extracted.

⑧ Gadget candidate for the extracted path in ⑦ located by algorithm 12.

FIGURE 4.1.: ALGORITHM OVERVIEW

the register-, flag-, or memory cell-name as key. The name of the map where the trees are saved in is ”operand tree map”.

The entries in the map are constantly updated. Therefore the contents of the map represent all effects of the native instruction, by the time the last REIL instruction updated the map with its tree.

The translation result of a single native instruction is a graph. In every graph multiple paths are possible. While a single native instruction is executed in any case, the effects of the instruction vary depending on which path in the graph is taken. The expression tree extraction algorithm uses the **JCC** handler to construct a formulae which represents all possible effects in a single tree (See Section 6 for details) to solve this problem.

The following sections will explain in detail how the expression tree extraction algorithm work. Initially the handlers for the REIL instructions are presented. Then the part of the algorithm is explained which combines the information of the handlers and updates the map. Finally an example which uses the algorithm on a single instruction is shown.

```
Require: valid $ReilGraph$, valid $currentNativeAddress$.
 1. operandTrees = new OperandTreeMap()
 2. skippedInstructions $= 0$
 3. for all $ReilBlocks$ in $ReilGraph$ do
 4.     for all $ReilInstructions$ in $ReilBlock$ do
 5.         if $ReilInstruction$ is binary then
 6.             skippedInstructions $\leftarrow$ handleBinaryInstruction()
 7.         else {$ReilInstruction$ is unary}
 8.             skippedInstructions $\leftarrow$ handleUnaryInstructions()
 9.         else {$ReilInstruction ==$ STM}
10.             skippedInstructions $\leftarrow$ handleSTMInstruction()
11.         else {$ReilInstruction ==$ STR}
12.             skippedInstructions $\leftarrow$ handleSTRInstruction()
13.         else {$ReilInstruction ==$ JCC}
14.             skippedInstructions $\leftarrow$handleJCCInstruction()
15.         end if
16.     end for
17. end for
18. remove temporary registers
19. restore native register names
20. return operandTrees
```

ALGORITHM 1: EXPRESSION TREE EXTRACTION FOR A SINGLE NATIVE INSTRUCTION

4.1.2.1. Instruction handler

For each instruction (*ReilInstruction*) a specific handler is used to extract the effects. Basically the handler takes the instruction which is present in text representation and transforms it into a tree. While this simple transform is true for most of the instructions the **JCC** and **STM** handlers have to be explained in more detail.

Binary instruction handler Each binary REIL instruction is handled by algorithm 2. In case of a binary instruction the expression tree is built from the first and the second operand of the instruction. Both operands are retrieved from the map where the expression trees are stored. Therefore already stored registers will not be created but fetched, which leads to a single expression tree for a native register even in case of very complex REIL translators.

```
Require: valid $ReilInstruction$, valid $skippedInstructions$, valid $operandTrees$, valid $currentNativeAddress$.
 1. operandTree1 $\leftarrow$ operandTrees.getTree(reilInstruction.getFirstOperand())
 2. operandTree2 $\leftarrow$ operandTrees.getTree(reilInstruction.getSecondOperand())
 3. expressionTree $\leftarrow$ createBinaryTree( reilInstruction.getMnemonic(), operandTree1, operandTree2 )
 4. return updateOperandTrees( reilInstruction.getThirdOperand(), expressionTree )
```

ALGORITHM 2: HANDLER FOR BINARY INSTRUCTIONS

For a binary instruction the mnemonic of the REIL instruction is the root of the expression tree. The key for the map where the expression tree will be stored is the result register. In case of a binary REIL instruction this is the third operand. Figure 4.2 shows what the expression tree of a binary REIL instructions looks like.

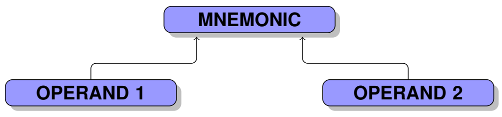


FIGURE 4.2.: EXPRESSION TREE FOR A BINARY INSTRUCTION

Unary instruction handler Some of the REIL instructions with only two operands are handled through algorithm 3. In the current implementation only the REIL instructions **LDM** and **BISZ** use this handler. The handler is a simpler handler because it only updates one side of the expression tree.

```
Require: valid $ReilInstruction$, valid $skippedInstructions$, valid $operandTrees$, valid $currentNativeAddress$.
1. operandTree1 ← operandTrees.getTree(reilInstruction.getFirstOperand())
2. expressionTree ← createBinaryTree( reilInstruction.getMnemonic(), operandTree1, null)
3. return updateOperandTrees( reilInstruction.getThirdOperand(), expressionTree )
```

ALGORITHM 3: HANDLER FOR UNARY INSTRUCTIONS

The handler works almost equivalent to the handler for binary REIL instructions. The root of the generated expression tree is the mnemonic of the current REIL instruction. The right side of the expression tree is always null. The left side of the expression tree is the first operand of the REIL instruction fetched from the map. An example for a unary instruction expression tree is presented in Figure 4.3.

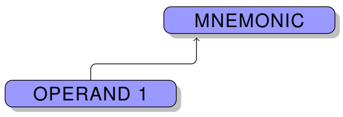


FIGURE 4.3.: EXPRESSION TREE FOR A UNARY INSTRUCTION

STM instruction handler The goal with each translation is that no information gets lost. For registers this is an easy task because every write to a register overwrites old information with new one. Therefore only the last state of the register can be taken into account as result state for a gadget. With memory on the other hand this is different as memory is addressed by registers. An overwrite of the register which addresses the memory does not mean that the information stored in the memory cell is lost. Therefore the handler for memory writes must save its information in such a way that its not lost if the register used to address it is overwritten.

This leads to the following behaviour: An **STM** instruction is not stored in the map using the target register as key but under a key that indicates memory access. Also the source and the target of the memory access are included into the tree which is saved. Therefore all information about the memory write is saved and can be used. The expression tree generated by the handler is shown in Figure 4.4.

```
Require: valid $ReilInstruction$, valid $skippedInstructions$, valid $operandTrees$, valid $currentNativeAddress$.
1. operandTree1 ← operandTrees.getTree(reilInstruction.getFirstOperand())
2. operandTree2 ← operandTrees.getTree(reilInstruction.getThirdOperand())
3. expressionTree ← createBinaryTree( reilInstruction.getMnemonic(), operandTree1, operandTree2 )
4. memoryReilOperand ← new ReilOperand( DWORD, "MEM_" + currentReilAddress )
5. return updateOperandTrees( memoryReilOperand, expressionTree )
```

ALGORITHM 4: HANDLER FOR STM INSTRUCTIONS

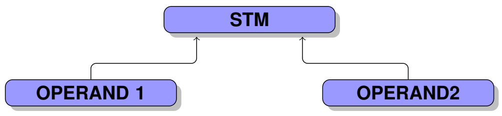


FIGURE 4.4.: EXPRESSION TREE FOR A STM INSTRUCTION

STR instruction handler A **STR** instruction is only a move of information into the target register therefore handler 5 simply fetches the information of the source register from the map and stores it with the key of the target register. An example for the generated expression tree can be seen in Figure 4.5.

```
Require: valid $ReilInstruction$, valid $skippedInstructions$, valid $operandTrees$, valid $currentNativeAddress$.
1. expressionTree←operandTrees.getTree(reilInstruction.getFirstOperand())
2. return updateOperandTrees( reilInstruction.getThirdOperand(), expressionTree)
```

ALGORITHM 5: HANDLER FOR STR INSTRUCTIONS

FIGURE 4.5.: EXPRESSION TREE FOR A STR INSTRUCTION

JCC instruction handler The **JCC** instruction in REIL can be conditional but it must not be conditional. In case of an unconditional **JCC** instruction the handler behaves as the **STR** handler with the **PC** register as target (Figure 4.6). But in the case of a conditional **JCC** instruction the handler is more complex as it has to solve the following problem:

The translation of a single native instructions leads to a graph which is comprised of nodes. These nodes hold instructions. In most cases the graph is just a single block with instructions. But in the case of a conditional instruction (for example **MOVEQ** or **BNE**) it consists of more nodes which hold instructions. One set of these nodes is only executed if the condition is true and the other set is only executed if the condition is false. The conditional **JCC** handler is used to construct a single tree which reflects both possible conditions.

The handler for a conditional **JCC** instruction is used in combination with the update algorithm described in section 4.1.2.2 and works as follows: Initially the number of REIL instructions which

JUMP TARGET

FIGURE 4.6.: EXPRESSION TREE FOR AN UNCONDITIONAL JUMP

will not be executed if the condition is true (as true means skip) is calculated. This is done by subtracting the address of the **JCC** instruction from the target address of the **JCC** instruction. The value where the number of skipped instructions is stored is called *skippedInstructions* and is used by the algorithm described in section 4.1.2.2 for all instruction handlers.

Then the **JCC** condition operand (first REIL operand OP1) is fetched from the map. With this operand two trees are generated: The **TRUE** tree (Figure 4.7) and the **FALSE** tree (Figure 4.8).

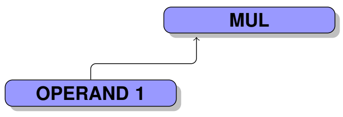


FIGURE 4.7.: EXPRESSION TREE FOR THE TRUE SIDE OF A CONDITIONAL JUMP

The **TRUE** tree consists of a multiplication as root of the tree and the condition operand as left child node of the multiplication node. The right child of the multiplication is empty and will be filled by algorithm 7. If the condition is now true (means it equals one [1]) then the result of the multiplication equals the tree on the right side of the multiplication of the **TRUE** tree.

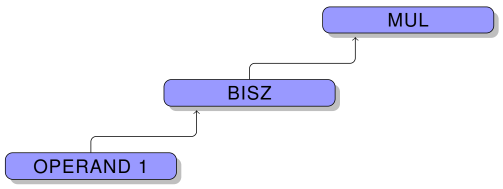


FIGURE 4.8.: EXPRESSION TREE FOR THE FALSE SIDE OF A CONDITIONAL JUMP

The **FALSE** tree in contrast to the **TRUE** tree uses the inverse of the condition. The **BISZ** instruction is used to invert the condition operand. This leads to the following behaviour: If the condition is now false (means it equals zero) then the result of the **BISZ** instruction equals one. Therefore the result of the multiplication equals the tree on the right side of the multiplication of the **FALSE** tree.

As shown it is not possible for both trees to yield a result $\neq 0$ at the same time. Therefore its possible with the **JCC** handler to combine both sides of a conditional execution within a single native instruction into one single tree.

The information how the combination of the **TRUE** and the **FALSE** trees is performed is provided in section 4.1.2.2.

[1] **JCC** instructions conditional operands can either be zero or one.

```
Require: valid $ReilInstruction$, valid $skippedInstructions$, valid $operandTrees$, valid $currentNativeAddress$.
 1. if is unconditional jump then
 2.     jumpTarget ← operandTrees.getTree(reilInstruction.getThirdOperand())
 3.     return updateOperandTrees( reilInstruction.getThirdOperand(), jumpTarget)
 4. else
 5.     skippedInstructions ← calculateSkippedInstructions()
 6.     operandTree1 ← operandTrees.getTree(reilInstruction.getFirstOperand())
 7.     falseTree ← createBinaryTree( "mul", "bisz", null )
 8.     falseTree ← falseTree.attachLeft( operandTree1 )
 9.     storeTree("FALSE", falseTree)
10.     trueTree ← createBinaryTree( "mul", null, null )
11.     trueTree ← trueTree.attachLeft( operandTree1 )
12.     storeTree("TRUE", trueTree)
13. end if
14. return skippedInstructions
```

ALGORITHM 6: HANDLER FOR JCC INSTRUCTIONS

4.1.2.2. Algorithm to update the operand tree map

The translation of a single native instruction leads in almost any cases to more then one REIL instruction. Some of the instructions which are translated are conditional. To be able to have all effects which a single native instruction has in one single map an update step has to be performed. This update step has to decide whether an instruction is to be simply stored in map or if it has to be combined with a condition tree from the **JCC** instruction. The algorithm which performs this work is the update operand the map algorithm 7.

The algorithm works as follows: Each handler aside from the **JCC** handler calls the update algorithm. If a native register is the result operand of the received tree from the handler the name of the register is suffixed with the current address of the native instruction. This is done because it is possible that the register which just got updated will be referenced by a later REIL instruction. If the name would not be suffixed then the newly written tree would be wrongly used as input.

Then the *skippedInstructions* variable and the register type of the result operand are evaluated. This leads to three possible outcomes:

If *skippedInstructions* equals zero then the tree can just be saved into the map regardless of the register type. If *skippedInstructions* does not equal zero and the register type is a temporary register (e.g. **t0**) the tree is also just saved to the map. But in this case the reason is that a temporary REIL register can never be the final result register. Therefore to keep the conditional trees small only the *skippedInstructions* variable is decremented and the tree is not combined with the conditional trees. If *skippedInstructions* does not equal zero and the register type is a native register (e.g. **R2**) the tree will be combined with the conditional trees from the **JCC** instruction. This is done by attaching the tree from the handler to the right side of the **FALSE** trees multiplication node, and the original value of the register, which is the key of the tree from the handler, to the right side of the **TRUE** trees multiplication node. Then an addition node is introduced as the new root of the result tree and the **TRUE** and **FALSE** tree are attached to it. This tree is then stored in the map with the result register of the tree as key.

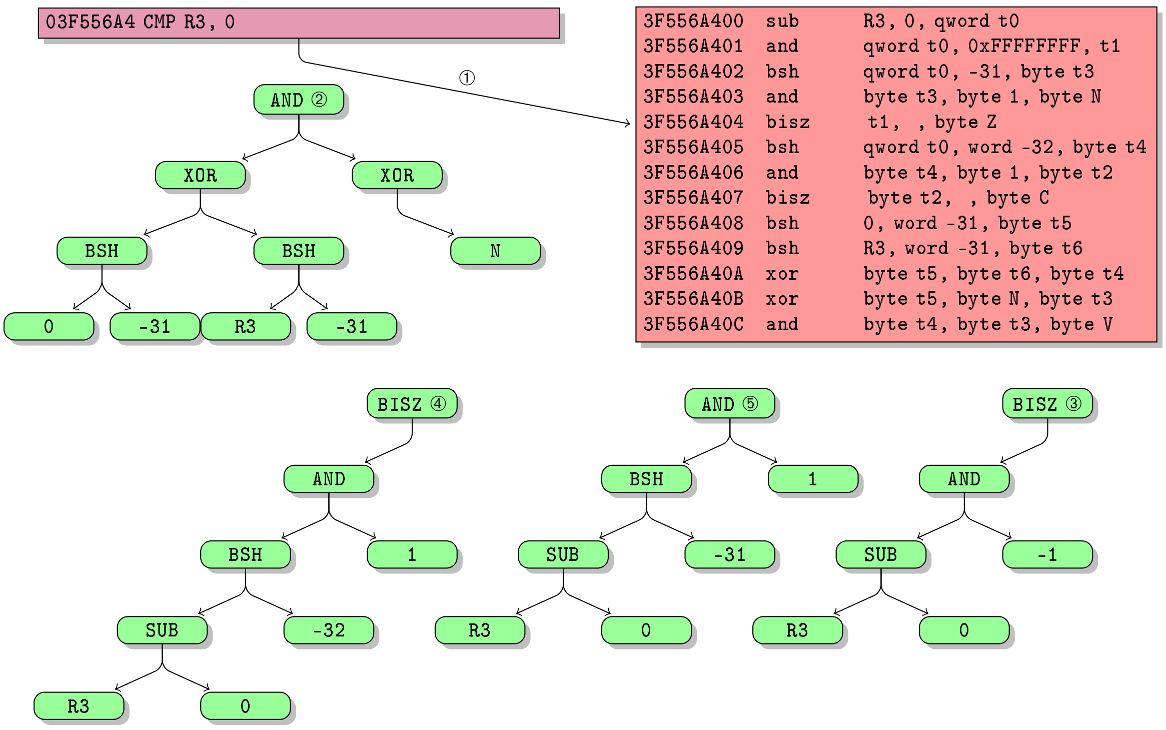
If all of the described algorithms are finished the native instructions effects are stored in a single map (operand tree map) which can be used by the subsequent algorithms.

4.1.2.3. Example for a single native instruction

To be able to comprehend the above described algorithms an example for a single native instruction expression tree extraction is given in Figure 4.9. The translation of a single instruction can vary from very simple to very complex depending on the instruction and the REIL translator. For example the ARM instruction **MOV** can, in a simple case, be only a single expression tree with one node that gets stored. But in a complex case when the barrel shifter is used, flags are set, and the instruction itself is conditional, multiple trees will be generated and have to inserted into the operand tree map.

```
Require: valid $resultReilOperand$, valid $expressionTree$, valid $skippedInstructions$, valid $operandTrees$, valid
    $NativeInstructionAddress$
 1. if $resultReilOperand$ is native platform register then
 2.     $resultReilOperand$.setValue(getValue() + ”-” + $NativeInstructionAddress$)
 3. end if
 4. if $skippedInstructions == 0$ then
 5.     storeTree($resultReilOperand$, $expressionTree$)
 6. else {$resultReilOperand$ is temporary REIL register}
 7.     storeTree($resultReilOperand$, $expressionTree$)
 8.     $skippedInstructions - -$
 9. else
10.     $skippedInstructions - -$
11.     falseTree = operandTrees.getTree(”FALSE”)
12.     trueTree = operandTrees.getTree(”TRUE”)
13.     falseTree.attachRight($expressionTree$)
14.     trueTree.attachRight($resultReilOperand$.original())
15.     conditionTree ← createBinaryTree( ”add”, falseTree, trueTree )
16.     storeTree($resultReilOperand$, conditionTree )
17. end if
18. return skippedInstructions
```

ALGORITHM 7: UPDATE OPERAND TREES ALGORITHM



```
3F556A400  sub     R3, 0, qword t0
3F556A401  and     qword t0, 0xFFFFFFFF, t1
3F556A402  bsh     qword t0, -31, byte t3
3F556A403  and     byte t3, byte 1, byte N
3F556A404  bisz     t1,  , byte Z
3F556A405  bsh     qword t0, word -32, byte t4
3F556A406  and     byte t4, byte 1, byte t2
3F556A407  bisz     byte t2,  , byte C
3F556A408  bsh     0, word -31, byte t5
3F556A409  bsh     R3, word -31, byte t6
3F556A40A  xor     byte t5, byte t6, byte t4
3F556A40B  xor     byte t5, byte N, byte t3
3F556A40C  and     byte t4, byte t3, byte V
```

FIGURE 4.9.: SINGLE NATIVE INSTRUCTION EXPRESSION TREE EXTRACTION

In Figure 4.9 the **CMP** ARM instruction is presented. The instruction compares the first operand with the second operand. Before the expression trees can be extracted the instruction is translated (item ①). This translation leads to the REIL assembly listing on the left-hand side of the figure. In the extraction process this listing is analysed. The result of the compare influences all flags which exist on the ARM architecture. The tree for the **V** flag is the tree ②. The tree for the **Z** flag is the tree ③. The tree for the **C** flag is the tree ④, ⑤ is the tree for flag **N**. The operand tree map where the trees are stored has registers as keys. Therefore after the **CMP** instruction is translated four trees will be inserted into the operand tree map.

4.1.3. Path extraction

Usually a function of a binary consists of multiple basic blocks which are organized in a graph. To be able to locate useful gadgets in such a graph its necessary to find paths comprised of instructions which are executed consecutively. The path extraction algorithm is designed to perform this search. The information which is extracted by the algorithm 8 is used in combination with the operand tree maps for single native instructions in 4.2.

The path extraction algorithm works as follows: Start at a **free branch** instruction. Traverse the graph in reverse execution order and for each native instruction found save the path to it in a map. A path consists of the list of traversed instructions and traversed basic blocks. Stop the traversing if either the user defined threshold is reached, no more predecessor instructions exist, or a function call instruction has been found.

```
Require: valid $currentPath$, valid $currentBasicBlock$, valid $currentInstruction$.
 1. isStart = ( $currentInstruction$ == startInstruction )
 2. if isCallInstruction($currentInstruction$) or ( !isStart and !specialInstructionAddress ) then
 3.     return false
 4. end if
 5. previousInstruction = isStart ? null : getPreviousInstruction()
 6. path = previousInstruction == null ? new Path : getPath()
 7. if path.size() == iterationDepth then
 8.     return false
 9. end if
10. path.add(current address)
11. savePath()
12. return true
```

Algorithm 8: Path extraction algorithm

4.2. Stage II

The overall goal is to be able to automatically search for gadgets. The information extracted in the first stage does not yet enable an algorithm to perform this search. The extracted information is a collection of all possible paths starting from the **free branch** instructions and a representation of the effects of each native instruction stored in a map. The second stage is a set of algorithms which merges the information of the first stage to enable the third stage to locate gadgets. Basically the second stage algorithms combine the effects of single native instructions along all possible paths. The second stage therefore has all effects of multiple instructions along a path as result.

The following Sections 4.2.1, 4.2.1.2, 4.2.1.3, and 4.2.2 explain for each of the algorithms. Section 4.2.1 describes the core functionality which uses the algorithms described in Section 4.2.1.2 and Section 4.2.1.3. Section 4.2.2 describes a simplification step which is run once all information has been merged.

4.2.1. Algorithm to merge expression trees with path information

Problem description: Almost any function on assembly level is a graph of interconnected basic blocks which hold instructions. The effects of these native instructions were saved as binary expression trees in operand tree maps in stage I. The control flow through a function is determined by the branches which connect the basic blocks. There are two types of branches: Unconditional branches which just pass control to the first instruction in the target basic block and conditional branches which determine if the jump to the target basic blocks first instruction is preformed based upon the given condition. In the path extraction algorithm from stage I all possible paths were extracted. This includes paths that have a conditional branch in them. As described the path extraction algorithm walks the graph in reverse execution order. Therefore the condition for the particular branches needs to be determined. All effects of the instructions in one path need to be merged into a single map such that a statement about the sum of effects for an executed path can be given.

```
Require: valid $addressToPath$, valid $addressToForests$, valid $pathToOperandTreeMap$.
 1. for all $addressList$ in $addressToPath$ do
 2.     currentPathOperandTreeMap = new OperandTreeMap()
 3.     for all addresses in $addressList$ do
 4.         currentAddressOperandTreeMap = jumpConditionDeterminator()
 5.         traverseAndUpdateAddressOperandTreeMap(currentAddressOperandTreeMap, currentPathOperandTreeMap)
 6.         tempOperandMap = buildTemporaryOperandTreeMap(currentPathOperandTreeMap)
 7.         fixAddressSuffix(tempOperandMap, currentPathOperandTreeMap)
 8.     end for
 9.     PathToOperandTreeMap.put(pathToOperandTreeMapKey, currentPathOperandTreeMap)
10. end for
```

ALGORITHM 9: MERGE PATH OPERAND TREE FUNCTION

Problem solution: The algorithm that merges expression trees with path information addresses the problem as follows: For each saved path (stored in control flow order) a new operand tree map is created. This operand tree map will hold the merged expression tree information. All instruction addresses of the path will be traversed from top to bottom. For each of the addresses it is determined if a condition needs to be generated (Described in Section 4.2.1.2). The result of the condition determination is the operand tree map for the current instruction. This operand tree map is then merged with the already merged operand tree maps (Described in Section 4.2.1.3). After all instructions have been traversed the merged information is stored. This information is then simplified by the algorithm described in Section 4.2.2.

4.2.1.1. Merging example

To further elaborate the concept of merging the following example is provided. The example shows the merging of a path with the expressions trees which have been extracted from the instructions whose addresses make up the path.

4.2.1.2. Jump condition determination algorithm

As described for each encountered conditional jump (for example **BEQ**) the condition needs to be determined which fits the path currently traversed. This condition determination is done by the jump condition determination algorithm.

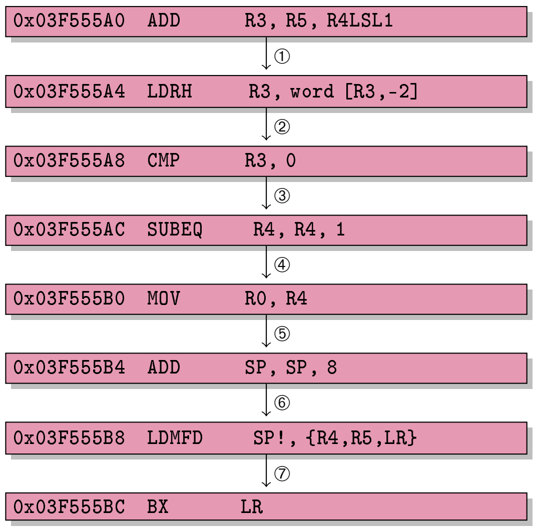
The algorithm works as follows: The operand tree map for the current instruction is loaded. For each of the expression trees within this operand tree map the keys are searched for the existence of a conditional branch. If this conditional branch is present the next address in the path is determined and compared to the branch target in the conditional branch. If the address is present, the condition ”jump is taken” is generated, which includes the conditional operand of the branch. If the address is not present the condition ”jump not taken” is generated. The conditions are also saved in the final operand tree map for the path with the prefix **COND**. This condition indicates that the path can only be used if the condition is satisfied. Therefore the path dictates the condition value.

If no conditional branch is present in the current instruction the operand tree map of the instruction is returned.

4.2.1.3. Traverse and update operand tree map algorithm

Similar to the expression tree extraction algorithm in the second stage multiple operand trees need to be merged according to the instruction addresses in the current path.

The algorithm to merge the operand tree of a single instruction with the information of the already merged operand trees works as follows:



```
0x03F555A0  ADD      R3, R5, R4LSL1
0x03F555A4  LDRH     R3, word [R3,-2]
0x03F555A8  CMP      R3, 0
0x03F555AC  SUBEQ    R4, R4, 1
0x03F555B0  MOV      R0, R4
0x03F555B4  ADD      SP, SP, 8
0x03F555B8  LDMFD    SP!, {R4,R5,LR}
0x03F555BC  BX       LR
```

① The first tree has been inserted into the path operand tree map. The key to access the newly stored tree is the register **R3**.

② The tree which was stored with key **R3** is now referenced as source of an operation and will therefore be merged with the tree from this instruction.

③ Only read access to already defined trees is performed therefore no merge will take place. The newly created trees are the trees for the flags **N**, **V**, **C**, and **Z**.

④ An new tree has been inserted and stored with the key **R4**. All previous accesses to the register have been read accesses therefore no tree was put in the map until now.

⑤ The tree with key **R0** is merged with the tree of register **R4** and stored in the map.

⑥ The **SP** tree is updated and stored in the map.

⑦ The trees for **SP**, **R4**, **R5**, **LR** are updated with new trees the old values for all trees but the **SP** tree are lost.

FIGURE 4.10.: EXAMPLE FOR THE MERGING FUNCTIONS

```
Require: valid $addressToForests$, valid $addressList$, valid $currentAddress$.
 1. $currentAddressOperandTreeMap$= $addressToForests$.get($currentAddress$)
 2. if $currentAddressOperandTreeMap$ contains tree with key "PC-" + $currentAddress$ then
 3.    $pcTree$ = getTree()
 4.    if $pcTree$.size() $= 1$ then
 5.       return $currentAddressOperandTreeMap$
 6.    end if
 7.    if $currentAddress$ is last element in path then
 8.       return $currentAddressOperandTreeMap$
 9.    end if
10.    $nextAddress$ = $addressToPathElementPath$.get($currentAddress + 1$)
11.    if $pcTree$ does not contain $nextAddress$ then
12.       generateConditionJumpNotTaken()
13.    else
14.       generateConditionJumpTaken()
15.    end if
16.    return $currentAddressOperandTreeMap$
17. end if
18. return $currentAddressOperandTreeMap$
```

ALGORITHM 10: JUMP CONDITION DETERMINATION FUNCTION

For each expression tree, Algorithm 11 locates all the registers which are referenced in the current expression tree and the operand tree map of the current path. If matches exist, the expression tree which will be stored, is updated with the already available information about the referenced register. After this step the trees of a single instruction are merged with the trees of all predecessor instructions in the path.

The following information is available in the resulting operand tree map after the three algo-

```
Require: valid $currentExpressionTree$, valid $currentPathOperandTreeMap$, valid $operandTreeKey$.
1. $registers \Leftarrow$ find all registers referenced in the expression tree and the map
2. for all $registers$ do
3.     replace all found operands with stored tree if available
4. end for
5. $currentPathOperandTreeMap$.storeTree($operandTreeKey$, $currentExpressionTree$)
```

ALGORITHM 11: TRAVERSE AND UPDATE OPERAND TREE MAP FUNCTION

rithms have finished processing a single path and its instructions.

- Merged information exists for all written registers.
- All leaf registers in the saved trees are in original state (original state means state equal to gadget entry).
- All memory locations written are available in an **MEM** statement.
- All conditions which need to be met are referenced by a **COND** statement.
- There are only native registers in the map and no temporary REIL registers.
- All entries are unoptimized (which means that there are still redundant instructions present).

4.2.2. Algorithm to simplify expression tree

All native instructions that are translated into REIL instructions have redundant instructions. This is due to the fact that REIL registers in contrast to native registers do not have a size limitation. Therefore 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 as a certain path can use immediate values as input for registers, in some cases it is possible to calculate sub-trees of the result trees, thus further reducing the redundant information present. The last algorithm in the second stage is the simplification of the merged map which performs the above steps.

The algorithm uses the simplification steps in Table 4.11.

SIMPLIFICATION OPERATION	DESCRIPTION
remove all register truncation operands	removes `AND 0xFFFFFFFF` operands
remove neutral element right	$Xop0 \Rightarrow X$ for op (ADD, SUB, BSH, XOR, OR)
	and $Xop0 \Rightarrow 0$ for op (MUL, AND)
remove neutral element left	$0opX \Rightarrow X$ for op (XOR OR ADD)
	and $Xop0 \Rightarrow 0$ for op (AND, MUL, BSH, DIV)
merge bisz operations	eliminate two consecutive bisz instructions.
merge add operations	merge consecutive add instructions into one.
calculate add operations of integer operands	$X + Y$
calculate and operations of integer operands	XY
calculate bisz operations of boolean operands	$X = 0$
calculate bsh operations of integer operands	$X \ll Y \| X \gg Y$
calculate div operations of integer operands	X/Y
calculate mod operations of integer operands	$XmodY$
calculate mul operations of integer operands	$X \times Y$
calculate or operations of integer operands	$X \| Y$
calculate sub operations of integer operands	$X - Y$
calculate xor operations of integer operands	$X \oplus Y$

FIGURE 4.11.: SIMPLIFICATIONS PERFORMED BY THE SIMPLIFICATION ALGORITHM

The simplification algorithm works as follows:

Each tree in the result operand tree map is passed to every possible simplification method. In the simplification method the tree is tested in regard to the applicability of the current simplification. It the simplification is applicable it is performed an 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 of the trees in the operand tree map have been simplified according to the rules in Figure 4.11.

4.3. 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. The goal is to locate specific functionality within the set of all possible gadgets that were collected in the first two stages. To locate this functionality multiple algorithms which locate specific functionalities are used. This section describes these algorithms.

Initially the core function for gadget search in described (Section 4.3.0.1). Then the actual locator functions are explained. Finally a complexity estimation algorithm is presented which helps with the decision which gadget to use for one specific gadget type.

4.3.0.1. Locate gadgets core function

The goal is to locate gadgets which perform a specific operation. All of the gadgets are organized as operand tree maps which are comprised of binary expression trees. These binary expression trees carry the information about what operation the gadget performs. Therefore an algorithm is needed which compares the expression trees of the gadget to expression trees which reflect a specific operation. The core algorithm which controls all the gadget locator functions (Section 4.3.0.2) works as follows:

To locate the gadgets in the operand tree maps a central function is used which consecutively calls all gadget locator functions for a single operand tree map and then parses the result for a possible inclusion into the gadget type operand tree map.

```
**Require:** valid $operandTreeMap$, valid $pathOperandTreeMapKey$, valid $gadgetTypeOperandTreeMap$.
 1. $currentAddress \Leftarrow pathOperandTreeMapKey$.first()
 2. **if** Controlflow is function of register **then**
 3.     **if** $operandTreeMap$ does not contain condition **then**
 4.         $gadgetCandidate \Leftarrow$ perform gadget search for each gadget type present.
 5.         **if** $gadgetCandidate$ != NULL **then**
 6.             $gadgetTypeOperandTreeMap$.store($gadgetCandidate$)
 7.         **end if**
 8.     **end if**
 9. **end if**
10. $currentPathOperandTreeMap$.storeTree($operandTreeKey$, $currentExpressionTree$)
```

ALGORITHM 12: LOCATE GADGETS IN OPERAND TREE MAP

4.3.0.2. Gadget locator functions

To be able to locate a specific functionality in a gadget candidate every operation that is searched for uses a specific tree which resembles exactly this operation. The trees which are used to match have been described in Section 3.4. There exist 32 gadget locator functions which locate the 32 possible gadget types.

The method of operation is the same for all of them and works as follows:

The core function passes the operand tree map of the current gadget candidate to a gadget locator function. The gadget locator function then traverses all of the keys of the trees stored in the map (keys can be registers, flags, conditions, or memory writes). For each of the keys it is checked if the initial condition of the tree is matched (for example Algorithm 13 searches for a memory write (Line 3)). If the initial condition is matched the tree from Section 3.4 which is used by the current gadget locator is compared to the tree of the matched key. If the tree matches the gadget information is passed to the core algorithm for inclusion into the gadget type map which stores which locator has produced a match for the current gadget. If the current operand tree map produces no match nothing is returned.

```
Require: valid $operation$, valid $optionalRightSourceOperand$, valid $operandTreeMap$.
 1. $operands$ $\Leftarrow$ get all operands from the operandTreeMap
 2. for all $operands$ do
 3.     if $operand$ indicates memory store then
 4.         if $rootNode$ from $operandTreeMap$.getTree($operand$) == "STM" then
 5.             if $rootNode$.left() == $operation$ then
 6.                 $targetSide$ $\Leftarrow$ checkValidOperands($rootNode$.right())
 7.                 $leftSourceSide$ $\Leftarrow$ checkValidOperands($rootNode$.left().left())
 8.                 if $optionalRightSourceOperand$ == NULL then
 9.                     $rightSourceSide$ $\Leftarrow$ checkValidOperands($rootNode$.left().right())
10.                 else
11.                     $rightSourceSide$ $\Leftarrow$ checkIsNodeValueEqual($optionalRightSourceOperand$)
12.                 end if
13.                 if $targetSide$ and $leftSourceSide$ and $rightSourceSide$ valid then
14.                     return $targetSide, leftSourceSide, rightSourceSide$
15.                 end if
16.             end if
17.         end if
18.     end if
19. end for
20. return null
```

ALGORITHM 13: MEMORY ARITHMETIC GADGETS CORE

4.3.0.3. Gadget complexity calculation

To be able to select the gadget with the least side effects, a routine is used to calculate the complexity of any given gadget candidate stored in the gadget type operand tree map.

This routine works as follows: It first performs a tree size check against the given gadget and tests whether a gadget of the same type with the same functionality in combination with the same registers has been already stored. It then chooses the gadget candidate with the smallest tree size as the gadget representative for this specific type and input values.

```
Require: valid $gadgetType$, valid $gadgetTypeOperandTreeMap$.
 1. $complexityMap$ $\Leftarrow$ HashMap<String, Integer>
 2. $leastComplexGadget$ = NULL
 3. for all $elements$ in $gadgetTypeOperandTreeMap$.keys() do
 4.     if ! $complexityMap$ contains $element$ then
 5.         $complexityMap$.add($element$, MAXINT)
 6.     end if
 7.     if $element$ == gadgetType then
 8.         if treeComplexity($element$) < $complexityMap$.get($gadgetType$) then
 9.             $leastComplexGadget$ = $element$
10.             $complexityMap$.add($element$, treeComplexity($element$))
11.         end if
12.     end if
13. end for
14. return $leastComplexGadget$
```

ALGORITHM 14: GADGET COMPLEXITY CALCULATION

5. System implementation

The implementation of the algorithms described in Chapter 4 consist of approximately 5000 logical lines [Wikipedia, 2009c] of Java code. The code is divided into logically structured components with a strict separation between the data and the algorithmic functions. In this chapter the details of the implementation are explained, focusing on the integration into BinNavi, the analysing algorithms, and the data structures which build the core of the system.

5.1. Integration into BinNavi

The software developed in this thesis is implemented as a plug-in for the reverse engineering platform BinNavi. Developed by zynamics GmbH, BinNavi focuses on static reverse engineering. In contrast to comparable tools it displays disassembled code as graphs rather than text based listings. One part of BinNavi is REIL which is described in Section 3.2. REIL is the basis upon all of the algorithms presented in this thesis are build on.

5.2. Initial data extraction

Initially the algorithms described in Section 4.1 need the information about all control flow altering instructions. In ARM assembly there exist many instructions which are able to alter the control flow because the **PC** register can be manipulated directly rather than only by a return, call, or jump instruction. As BinNavi stores the information about the disassembly in a relational database there are basically two ways to extract the information about **PC** altering instructions. The initial idea was to load the functions present in the target binary sequentially, translate them into REIL and scan for instructions involving the **PC** register. The approach was abolished after initial runtime tests revealed that it would take too long and there are better methods to get the right information. The second idea and the used approach is to use SQL queries to fetch all possible **PC** altering instructions from the database. The module id which is used in the query has been fetched from the database as well, with the query in Listing 5.2.

LISTING 5.1: SQL QUERY TO GET **PC** ALTERING INSTRUCTIONS

```sql
SELECT DISTINCT address FROM bn_instructions b
JOIN bn_operands ON b.id = bn_operands.instruction
JOIN bn_operand_expressions ON bn_operands.id = bn_operand_expressions.operand_id
JOIN bn_expression_tree ON bn_operand_expressions.expression_id = bn_expression_tree
    .id
WHERE b.module_id = 'moduleID'
AND ( symbol = 'PC'
AND ( ( mnemonic like 'LDM%' AND bn_operands.position != '0')
OR ( bn_operands.position = '0' AND mnemonic not like 'LDM%') )
OR ( mnemonic = 'BX' AND symbol != 'PC' AND symbol != 'b4' ) )
ORDER BY address
```

LISTING 5.2: SQL QUERY TO GET THE MODULE ID FROM A MODULE NAME

```sql
SELECT id from bn_modules b where name = 'moduleName'
```

With the query in Listing 5.1 the addresses of all **PC** altering instructions are fetched from the database and stored in a list within the program. As the initial query does not return to which function an instruction belongs to, this information must be fetched with an additional SQL query.

LISTING 5.3: SQL QUERY TO GET A FUNCTION ID BASED ON ADDRESS AND MODULE

```sql
SELECT parent_function from bn_instructions bi
JOIN bn_codenode_instructions bci ON bci.instruction = bi.id
JOIN bn_code_nodes bcn ON bcn.node_id = bci.node_id
where bi.address = 'address' AND bi.module_id = 'moduleID');
```

With the query in Listing 5.3 the function id for loading the function in BinNavi is fetched from the database. With the extracted information it is now possible to load exactly the function of any given **PC** altering instruction.

5.3. Extracting information

In the implementation the step to extract the information from the available disassembly is performed for each **PC** altering instruction. In a loop the extracted list of **PC** altering instructions is traversed and the source function is loaded. Depending on the configured depth threshold the function is traversed, starting from the **PC** altering instruction upwards where upwards means that in a single basic block the instructions are traversed from high to low addresses. At the last instruction of a basic block, given the threshold has not yet been reached, all incoming edges for the current basic block of the function's control flow graph are included into the traversal routine.

The retrieval of information from the function is implemented with callbacks to the class BasicBlockReverseWalker. Within this class the function walkReverse performs the described upwards traversal of the function. The callbacks implement the extraction of the instruction information as well as the path information.

5.3.1. Extracting instruction information callback

For each of the encountered instructions within the upwards traversal of the functions control flow graph, the instruction is translated to REIL. This translation leads to a structure called *ReilGraph* consisting of *ReilBlocks* which are connected with edges to map the translated native instruction into the REIL space. This graph is analysed with the algorithms from Section 4.1.2. These algorithms are called from the callback function in Listing 5.4.

LISTING 5.4: INSTRUCTION INFORMATION CALLBACK

```java
@Override
public boolean call(final List<BasicBlock> currentPath, final BasicBlock currentBasicBlock,
      final Instruction currentInstruction)
{
 final long currentAddress = currentInstruction.getAddress().toLong();
 if (m_addressToForests.hasTree(currentAddress))
 {
  return true;
 }
 try
 {
  final ReilGraph reilGraph = currentInstruction.getReilCode();
  final OperandTreeMap operandTreeMap = ExpressionTreeExtractor.extractor(reilGraph,
       currentInstruction.getAddress().toHexString());
  m_addressToForests.put(currentAddress, operandTreeMap);
 }
 catch (final InternalTranslationException e1)
 {
  e1.printStackTrace();
 }
 return true;
}
```

The extraction and translation of the native instruction to an expression tree is always complete, meaning that for each native instruction where a REIL translator exists the expression tree will map the native instruction with all operations into the REIL space and no information about the original instruction is lost.

As it is perfectly possible in a graph to visit a location twice during iteration, the algorithm makes sure that a previously processed instruction will not be processed again.

5.3.2. Extracting path information callback

As the information about the native instructions translated to expression trees can not be used by itself, the algorithms described in Section 4.1.3 perform path extraction with the same callback functionality described in Section 5.3.1. The callback is much simpler than the callback for extracting information from a native instruction. The path is also terminated if the threshold is reached or, if no more native instructions exist upwards from the current instruction.

5.4. Merging of extracted information

Both the information about the extracted paths and the information about the extracted REIL mapping of native instructions are not useful if not properly combined. Listing 5.5 shows the function which controls this part of the process also explained in Section 4.2.

LISTING 5.5: MERGING THE EXTRACTED INFORMATION

```
public static void mergePathOperandTrees(final AddressBlocksPathMap addressToPath, final
     AddressOperandTreeMap addressToForests, final PathOperandTreeMap pathToOperandTreeMap
     )
{
 for (final Entry<Pair<Long, List<BasicBlock>>, List<Long>> addressToPathElement :
      addressToPath.entrySet())
 {
  final OperandTreeMap currentPathOperandTreeMap = new OperandTreeMap();
  boolean updateFlag = true;
  final List<Long> addressToPathElementPath = addressToPathElement.getValue();
  for ( final Long currentAddressToPathElement : addressToPathElementPath)
  {
   final OperandTreeMap currentAddressOperandTreeMap = AddressOperandTreeMapHelper.
        jumpConditionDeterminator(addressToForests, addressToPathElementPath,
        currentAddressToPathElement);

   traverseAndUpdateAddressOperandTreeMap(currentAddressOperandTreeMap,
        currentPathOperandTreeMap );
   final OperandTreeMap tempOperandMap = buildTemporaryOperandTreeMap(
        currentPathOperandTreeMap);
   fixAddressSuffix(tempOperandMap, currentPathOperandTreeMap);
   final Pair<Long, List<Long>> pathToOperandTreeMapKey = new Pair<Long, List<Long>>(
        addressToPathElement.getKey().first(), addressToPathElement.getValue());
   pathToOperandTreeMap.put(pathToOperandTreeMapKey, currentPathOperandTreeMap);
  }
 }
}
```

In the actual implementation there is also a check implemented which controls the current tree size of the merged tree and kills the complete operand tree map if the configured threshold is reached. This can happen if a group of conditional execution instructions are in a certain path and the conditions are combined resulting in a quadratic increase of size.

5.4.1. Updating the expression tree in a path

Updating the expression tree for a single path is performed through the iterative walk of all addresses which are sequentially present in the path and updating the resulting operand tree map with the information encountered in the next instructions operand tree map.

The implementation 5.6 of the merging process proved to be a harder problem than expected and is still considered to be changed in the future. This is primarily due to one aspect which is very unlikely but still possible. In the function which updates the expression trees of the current instruction with the already processed instructions earlier in the path an update of a certain case in the expression tree is incorrect if the condition 5.4.1 holds.

Listing 5.6: Update expression tree function

```java
public static void updateExpressionTree(final LinkedBinaryTree<ComparableReilOperand>
     currentExpressionTree, final OperandTreeMap currentPathOperandTreeMap, final
     ComparableReilOperand operandTreeKey)
{
 final ArrayList<ComparableReilOperand> elementsInTreeToBeUpdated = new ArrayList<
     ComparableReilOperand>();

 for (final Position<ComparableReilOperand> position : currentExpressionTree.positions())
 {
  if (position.element().getType() == OperandType.REGISTER)
  {
   if ( currentPathOperandTreeMap.keySet().contains(position.element()))
   {
    elementsInTreeToBeUpdated.add(position.element());
   }
  }
 }

 for (final ComparableReilOperand currentUpdateElement : elementsInTreeToBeUpdated)
 {
  traverseFoundOperandPositions(currentExpressionTree, currentPathOperandTreeMap,
      currentUpdateElement);
 }
 currentPathOperandTreeMap.storeTree(operandTreeKey, currentExpressionTree);
}
```

Condition *Given a tree that references the two registers **R1** and **R2**. The tree for **R1** also references **R2** but with an older state, then the update will update the **R1** reference correctly and the **R2** reference in the new tree correctly but the **R2** reference in the attached **R1** tree incorrectly.*

5.4.2. Simplification of expression trees

The implementation of the expression trees is only a very simple function that does school math to reduce the size of the expression trees. Initially the idea was to use a SAT solver Wikipedia [2009b] or a library for boolean arithmetic to simplify the expressions extracted from the instructions. But it proved to be efficient enough to just simplify according to the rules specified in Figure 4.11 for the tree size to be reduced significantly in the cases that mattered.

5.5. Using the extracted information

The implementation details presented so far are all part of information gathering while the following part can be summarized under the name information analysing and sieving. The information that is now presented after the gathering process needs to be filtered to receive the actual instruction sequences useful for chaining.

5.5.1. Finding suitable sequences

Even though most of the gadget location algorithms are freely interchangeable between different platforms, this is not the case for control flow altering instruction sequences or system call instruction sequences.

One always has to keep in mind that certain implementations that are valid for one architecture are not valid for another one. Therefore, even though this thesis only covers the ARM architecture specifically, the implementation was designed to support all architectures where REIL translators exist. The following limitations about all gadgets always have to be kept in mind. All instruction sequence locator algorithms are very dependent on the structure of the REIL translator. Even if simple arithmetic operations like **ADD** can be found platform-independently it can still be a challenge to locate shifts.

LISTING 5.7: REGISTER RIGHT SHIFT LOCATOR

```java
private static Triple<ComparableReilOperand, ComparableReilOperand, ComparableReilOperand>
     locateRegisterRightShiftRegisterGadget(final OperandTreeMap operandTreeMap)
{
 for (final ComparableReilOperand operandTreeMapKey : operandTreeMap.keySet())
 {
  if ( ComparableReilOperandHelper.isNativeRegister(operandTreeMapKey))
  {
   final LinkedBinaryTree<ComparableReilOperand> treeToBeCheckedForMatch = operandTreeMap.
        getTree(operandTreeMapKey);
   final Position<ComparableReilOperand> rootNodePosition = GadgetLocatorHelper.
        checkExactRootNodeValue(treeToBeCheckedForMatch,      );
   if ( rootNodePosition != null )
   {
    final Position<ComparableReilOperand> rightNodePosition = GadgetLocatorHelper.
         checkExactRightOfNodeValue(treeToBeCheckedForMatch, rootNodePosition,      );
    if ( rightNodePosition != null )
    {
     final Position<ComparableReilOperand> zeroNodePosition = GadgetLocatorHelper.
          checkExactLeftOfNodeValue(treeToBeCheckedForMatch, rightNodePosition,    );
     final Position<ComparableReilOperand> rightRightNodePosition = GadgetLocatorHelper.
          checkExactRightOfNodeValue(treeToBeCheckedForMatch, rightNodePosition,      );
     if ( (rightRightNodePosition != null) && (zeroNodePosition != null) )
     {
      final Position<ComparableReilOperand> shifterAndMaskPosition = GadgetLocatorHelper.
           checkExactRightOfNodeValue(treeToBeCheckedForMatch, rightRightNodePosition,
            );

      final ComparableReilOperand shifterOperand = locateValidOperands(
           treeToBeCheckedForMatch.buildSubtree(treeToBeCheckedForMatch.left(
           rightRightNodePosition)));
      final ComparableReilOperand shiftedOperand = locateValidOperands(
           treeToBeCheckedForMatch.buildSubtree(treeToBeCheckedForMatch.left(
           rootNodePosition)));
      if ( (shifterAndMaskPosition != null) && (shifterOperand != null) && (shiftedOperand
           != null)  )
      {
       return new Triple<ComparableReilOperand, ComparableReilOperand,
            ComparableReilOperand>(operandTreeMapKey, shiftedOperand, shifterOperand);
      }
     }
    }
   }
  }
 }
 return null;
}
```

While the complexity of cross platform support is mainly due to the way REIL translators handle shifts, other cases also exist. ABIs [1] of different architectures have different calling conventions. This makes it very difficult to port branch and call instruction sequences to another architecture. In case of branches and calls an architecture-specific implementation of the search process is inevitable. The most special case is the system call. Even though architectures define a standard way to perform this operation, every operating system implements this in a different way. Therefore one can see the first major limitation of the current implementation. System calls are only supported in Windows Mobile.

5.5.2. Evaluating suitable sequences

The scarcity of stack space in an exploitation scenario is often underestimated. If the available stack space is too small some bugs can not even be exploited with the technique of return ori-

[1] Application Binary Interface

ented programming. Therefore the need to find the smallest sequence for a specific operation is mandatory. In the current implementation (Listing 5.8) the tree size has been used as the main indicator for the complexity of the instruction sequence. Using the tree size as a metric is a two-sided coin. On the one hand it is easy to calculate and it provides a good match for side effect freeness and general register use.

LISTING 5.8: COMPLEXITY CALCULATION ROUTINE

```java
public HashMap<String, Triple<GadgetType, String, Address>>
     getLeastComplexGadgetForSpecificGadgetType(final GadgetType type)
{
 final HashMap<String, Triple<GadgetType, String, Address>> leastComplexGadget = new
      HashMap<String, Triple<GadgetType, String, Address>>();
 final HashMap<String, Integer> complexityMap = new HashMap<String, Integer>();

 for (final Triple<GadgetType, String, Address> key : m_hashmap.keySet())
 {
  if ( !complexityMap.containsKey(key.second()))
  {
   complexityMap.put(key.second(), Integer.MAX_VALUE);
  }

  if ( key.first().equals(type) )
  {
   if ( complexity(key) < complexityMap.get(key.second()) )
   {
    leastComplexGadget.put(key.second(), key);
    complexityMap.put(key.second(), complexity(key));
   }
  }
 }

 return leastComplexGadget;
}
```

On the other hand it does not consider stack usage. Even though the stack usage is reflected in the general register use, as all registers get popped in the function epilogue, there might be a need for further refinement in the future. Another issue is memory usage with writes and reads which must be taken into consideration. If an algorithm should be able to build the shellcode for an algorithm provided automatically from a library, then the memory accesses need to be modelled in more detail.

6. Experimental results

In Chapter 1 the thesis was presented. In Chapter 2 the basics about the architecture and the operating system as well as a general introduction into the topic was presented. In the following chapters the algorithms and the implementation were described. In this chapter examples for the analysis results of a set of libraries are given and compared. Also, a simple proof of concept exploit is presented and explained as there exist some conditions which impact the reliability of exploits on the target platform in general.

All the tests regarding the analysis of a library begin with the following provided data:

1. A dynamically linked library which has been compiled for the ARM architecture.
2. The correct offset addresses for the library.

The main objective in this chapter is to show that the developed algorithms yield sufficient results for a given library and are able to extract the specified instruction sequences automatically. As a side objective the general usability of the located instruction sequences shall be shown with an example exploit for a simple buffer overflow.

6.1. Testing environment

All the tests were carried out on machines with a Java (JDK 1.6.0) installation and the BinNavi (version 2.2) software suite installed. The requirements for BinNavi to work on a machine are a MySQL (version 5.1) database on the local machine or accessible to the machine as well as the correct Python plug-ins for importing data into the database. Microsoft Visual Studio 2008 was used to develop the vulnerable target server and the debugger used in this thesis. Eclipse 3.5 was used to develop the BinNavi plug-in which contains the algorithms for gadget extraction from the libraries.

To test the exploit, the vulnerable server was compiled without stack cookie protection. As no other exploit mitigating techniques exist on Windows Mobile this was the only change to the default compiler settings of Visual Studio.

6.2. Library comparison

Unlike Windows versions for desktop machines, Windows Mobile images are not provided to the customer by Microsoft but by the device OEM enabling him to change certain settings of the device to fit his needs. Therefore all images and libraries are slightly different. Initially a set of Windows Mobile 6.1 libraries are compared and their differences are shown. The tests were carried out on a machine with a Core 2 Duo 2.4 Ghz and 4 Gigabytes of RAM of whom 3 Gigabytes were available to the 64 Bit Java VM. The Operating System for all tests is Windows 7 64 Bit Enterprise.

The results provided in table 6.1 show the analysis of a selected set of libraries. There are some aspects which must be explained: The matches are always compiler dependant and use information about the compiler to match emitted code. The trees to match instruction sequences where initially developed for Windows Mobile and then tested against other operating systems with other compilers. Therefore the analysis of the IPhone library did not yield all of the gadgets which it did for the Windows Mobile library (indicated by the brackets around the yes). But a

LIBRARY ORIGIN	# OF FUNCTIONS	ANALYSIS TIME (DB/DISK)	# OF GADGETS	COMPLETE
Emulator	2748	(4.97/2.98)minutes	30340	yes
Device dump	2757	(4.74/1.68)minutes	29952	yes
IPhone libSystem.B.dylib	6111	(16.27/6.39)minutes	76634	(yes)

FIGURE 6.1.: LIBRARY COMPARISON

manual analysis then confirmed that the missing "conditional branch" gadget is present but the matching trees need to be adjusted to find them for the IPhone libraries as well.

As numbers by themselves do not provide any information the description about the fields in the table 6.1 is the following.

LIBRARY ORIGIN describes where the library comes from.

# OF FUNCTIONS How many functions in this library could be analysed.

ANALYSIS TIME (DB/DISK) The time it took for a complete analysis of the library where DB indicates an initial analysis and disk an analysis that already has all the information extraction performed and only the gadget location process must be done.

# OF GADGETS The total number of gadgets that were located in the library.

COMPLETE Indicates if the complete gadget suite has been located in the library or if any of the gadgets for Turing-completeness are missing.

The number of located gadgets indicate that a lot of the function epilogues are usable but to be able to understand that this large number can be misleading the distribution of the gadgets in Figure 6.2 needs to be taken into account. Because most of the simpler gadgets can be located in various ways and might even be present in a gadget that itself performs a lot more work.

GADGET TYPE	#	GADGET TYPE	#
REGISTER_SET_TO_REGISTER	11644	REGISTER_SET_TO_MEMORY_DEREFERENCE	58
COMPARE	5047	MEMORY_ADDITION	51
MEMORY_SET_TO_REGISTER	3239	MEMORY_DECREMENT	46
REGISTER_SET_TO_ZERO	2526	MEMORY_SUBTRACTION	44
REGISTERS_SET_TO_REGISTERS	2203	MEMORY_DEREFERENCE_SET_TO_MEMORY	34
REGISTER_SET_TO_CONSTANT	2162	REGISTER_LEFT_SHIFT_REGISTER	30
REGISTER_SET_TO_MEMORY	825	REGISTER_AND	29
MEMORY_SET_TO_MEMORY	533	MEMORY_DEREFERENCE_SET_TO_REGISTER	20
REGISTER_INCREMENT	413	REGISTER_RIGHT_SHIFT_REGISTER	14
REGISTER_DECREMENT	380	REGISTER_XOR	12
REGISTER_SUBTRACTION	336	MEMORY_OR	9
REGISTER_ADDITION	264	MEMORY_XOR	7
MEMORY_INCREMENT	150	MEMORY_SET_TO_MEMORY_DEREFERENCE	4
REGISTER_OR	111	MEMORY_AND	1
CONDITIONAL_BRANCH	82	REGISTER_NOT	0
FUNCTION_CALL	66	REGISTER_NEGATION	0

FIGURE 6.2.: COREDLL.DLL GADGETS IN NUMBERS

Like presented in Figure 6.2 there are gadgets which have not been located by the automated search. These missing gadgets can be constructed from other gadgets present. [1]

[1] All the numbers presented are the result of analysing the Windows Mobile 6.1 emulator image.

6.3. Exploit example

6.3.1. Vulnerable service

The vulnerable service is a basic TCP server which has been compiled for Windows Mobile using Visual Studio 2008. The stack protection with cookies has been disabled for the project as well as the optional optimizations. [2] Listing 6.1 shows the vulnerable function of the server application.

LISTING 6.1: VULNERABLE FUNCTION

```
1  char buf[1024];
2  wchar_t tb[64];
3
4  void svr_run(int ssock)
5  {
6   char string[1024];
7   int csock, i, pos;
8   struct sockaddr_in csa;
9   socklen_t csalen;
10
11  swprintf(tb, L"string: %p", string);
12  MessageBox(0, tb, 0, 0);
13
14  csalen=sizeof(csa);
15  csock = accept(ssock, (struct sockaddr*)&csa, &csalen);
16  if (SOCKET_ERROR == csock)
17  {
18   ERR(WSAGetLastError());
19   return;
20  }
21
22  for(pos = 0; (i=recv(csock, buf, 1024, 0)) > 0;)
23  {
24   memcpy(string+pos, buf, i);
25   pos += i;
26  }
27 }
```

Line 1 specifies a static buffer ”buf” which is 1024 bytes in size. This buffer is the source buffer used in the overflow. In line 6 the function local buffer ”string” of size 1024 bytes is defined. This buffer is the target for the overflow. The overflow happens in the lines 22–26. In line 22 the recv function writes 1024 bytes it receives from csock into the global buffer ”buf” and returns the number of written bytes which is stored in the variable i. For each of the received 1024 bytes the memcpy in line 24 copies the contents of the buffer ”buf” into the buffer ”string” and updates the offset position where it places the contents in the next iteration of the loop. After the first 1024 byte are copied into the buffer ”string” there is no more space available and due to the improper bounds checking of the copy function the received data is copied into the adjacent stack frame. If the input is carefully crafted this can be used to gain control over the program.

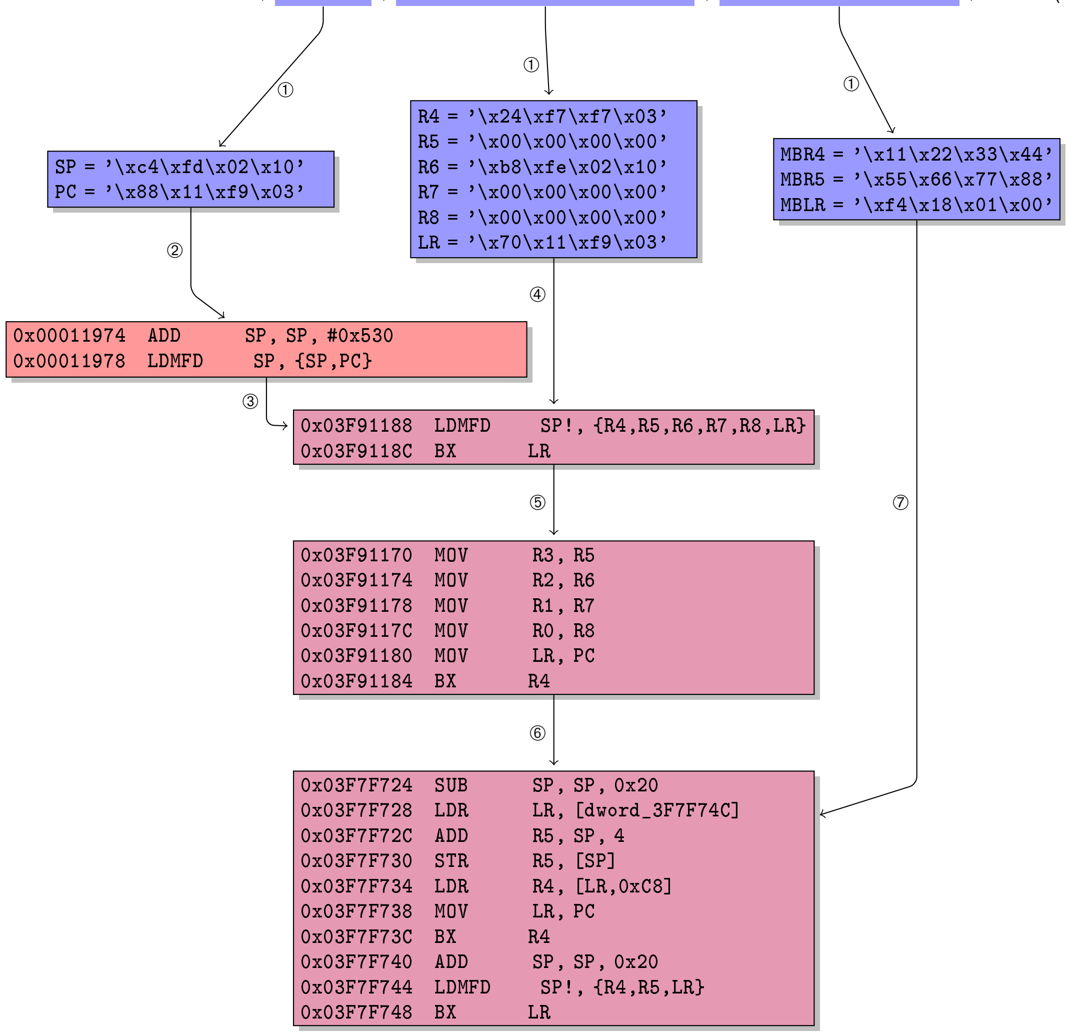
6.3.2. Shellcode

This section shows a small example shellcode which can be used to exploit the vulnerable function in Listing 6.1 with the use of return oriented programming.

Figure 6.3 only uses a limited set of the available gadgets to better show the greater concept of return oriented programming. At the top of the figure the actual shellcode layout on the stack is presented. The numbers used in the shellcode in Figure 6.3 are specific offsets for one compiled version of the vulnerable server and should not be the center of attention. The light blue fields hold the input for the selected set of gadgets. The light red block is the epilogue of the exploited function. The light purple blocks are gadgets from the analysed library. During testing the behaviour of the emulator showed some unaccountable stack corruptions upon exploitation of the function in debug mode. The origin of the corruptions could not be determined but is believed to

[2] Optimizations might reorder variables on the stack or even put them in a different context. Therefore to keep the layout of variables as defined in the code, optimizations are disabled.

$$SHELLCODE = X * 1296 + \boxed{\text{SP + PC}} + \boxed{\text{R4 + R5 + R6 + R7 + R8 + LR}} + \boxed{\text{MBR4 + MBR5 + MBLR}} + A * 60 \quad (6.1)$$



```
SP = '\xc4\xfd\x02\x10'
PC = '\x88\x11\xf9\x03'
```

```
R4 = '\x24\xf7\xf7\x03'
R5 = '\x00\x00\x00\x00'
R6 = '\xb8\xfe\x02\x10'
R7 = '\x00\x00\x00\x00'
R8 = '\x00\x00\x00\x00'
LR = '\x70\x11\xf9\x03'
```

```
MBR4 = '\x11\x22\x33\x44'
MBR5 = '\x55\x66\x77\x88'
MBLR = '\xf4\x18\x01\x00'
```

```
0x00011974  ADD       SP, SP, #0x530
0x00011978  LDMFD      SP, {SP,PC}
```

```
0x03F91188  LDMFD     SP!, {R4,R5,R6,R7,R8,LR}
0x03F9118C  BX       LR
```

```
0x03F91170  MOV       R3, R5
0x03F91174  MOV       R2, R6
0x03F91178  MOV       R1, R7
0x03F9117C  MOV       R0, R8
0x03F91180  MOV       LR, PC
0x03F91184  BX       R4
```

```
0x03F7F724  SUB       SP, SP, 0x20
0x03F7F728  LDR       LR, [dword_3F7F74C]
0x03F7F72C  ADD       R5, SP, 4
0x03F7F730  STR       R5, [SP]
0x03F7F734  LDR       R4, [LR,0xC8]
0x03F7F738  MOV       LR, PC
0x03F7F73C  BX       R4
0x03F7F740  ADD       SP, SP, 0x20
0x03F7F744  LDMFD      SP!, {R4,R5,LR}
0x03F7F748  BX       LR
```

① The shellcode which is used to exploit the function is split into three logical parts which provide input for the library instruction sequences.

② The two 32 bit words which are located in the shellcode at positions 1296 and 1270 are the arguments for the LDMFD instruction.

③ The control flow is now passed to the first library instruction sequence.

④ The arguments which are needed for the LDMFD instruction present in the first instruction sequence are passed through the stack. The execution is continued at the address specified by the last argument popped from the stack.

⑤ No arguments from the stack are needed for the function call gadget as they are already present in the right registers due to the earlier LDMFD instruction.

⑥ Control flow continues at the address which is present in register **R4**. The register points to the address of the function MessageBoxW + 4 bytes. Adding 4 is necessary to avoid that the function destroys the carefully chained gadget frames on the stack with an STMFD instruction. The arguments of the MessageBoxW function have been prepared in the previous instruction sequence where register **R2** holds a pointer to a string.

⑦ Upon termination of the MessageBoxW function the control flow continues at the address specified in the gadget frame on the stack.

FIGURE 6.3.: SHELLCODE EXAMPLE

lie in the way the emulator works in debug mode. It can not be verified if this behaviour is present in release mode as well because the stack is not observable. Also, for an exploit developed for Windows Mobile the attacker must always conform to the restrictions Windows Mobile puts on the stack pointer and the frame pointer. Otherwise the exploit will not work. Another important issue with exploitation is that the slot of the process can not be guessed with 100% reliability. Therefore exploiting can be difficult if no information leakage can be used to guess the used slot. Even though this is not a problem for the offsets in the library, the stack pointer and frame pointer must be set correctly inside the stack frame area of the current slot.

6.3.3. Conclusion

With the proof of concept exploit accompanied by the shellcode it has been shown that return oriented programming on the ARM architecture is possible. Even though the exploit does not use a larger part of the possible gadgets, it shows one important point: exploitation with return oriented programming on ARM works, and if done with an automated search algorithm for gadget selection, it is also efficient.

7. Conclusion and further work

The previous chapters have shown that return oriented programming on the ARM architecture is possible. Also, it was shown that the algorithms developed in this thesis can automatically extract the described gadgets from a library. As discussed, platform independent algorithms for the extraction of instruction sequences are important to be able to handle the growing architecture diversity encountered.

In this final chapter the further areas of research in the topic of return oriented programming are introduced. These areas are not part of the thesis but are believed to be important to the further development of tools for return oriented programming.

7.1. Automatic gadget compiler

One of the logical next steps in the area of return oriented programming is that not only the gadget search algorithms are able to find instruction sequences automatically but that also the combination of gadgets will be automated. Compiling of gadgets into a return oriented program automatically has some prerequisites that must be met within the initial gadget search process. There must be a clear definition about the pre- and post-conditions of an instruction sequence. The complexity of the instruction sequence must be deterministic, the side effects of the gadget must be known, and they must be avoided by the implemented compiler.

7.2. Gadget description language

Even if the pre- and post-condition calculations performed in the algorithms have already contributed to defining a description language for gadgets and their possible combinations, to be able the efficiently program a compiler, a description language is needed. The language could abstract the process even further from the analyst and could be easily adaptable even by people with little or no assembly background.

7.3. Live system scanning

Another area of research which would be a vital addition to the field of return oriented programming would be to integrate a live system scanning like described in Pablo Soles work on DEPlib Sole. The system should be able to stop the execution of the currently running process and save its state. When the state has been saved it should scan all available linked libraries and extract the possible gadgets from them. This would enable an attacker to not rely on static analysis of the libraries which might be present when the bug is triggered but to analyse the state of the program exactly when the bug is triggered. This also solves the problem of offsets within the targeted program and allows the attack to avoid offset errors.

7.4. Partial function reconstruction

The algorithms which have been developed in this thesis have the potential to be useful within other fields of static analysis as well. The path analysis functionality presented could be used to understand single paths in the program. This would for example be useful for input crafting or general understanding of the function in question. Even though the algorithms right now do not enable an analyst to ask these questions the change in program logic is only marginal. Even though the depicted idea is not generally solvable in all cases, the cases which can be solved provide great benefit to the analyst.

7.5. Attack vector broadening

All presented work in this thesis is also applicable to heap overflows and possibly other bug classes as well, but there is no publicly available documentation where return oriented programming has been used for any other attack vector but stack based buffer overflows. The broadening of the use of return oriented programming in combination with other exploit techniques is believed to provide good results.

7.6. Polymorphism

For each of the searched gadgets the algorithms can find multiple instruction sequences which perform the same operation. Therefore the following scenario is part of further research: For each attack a payload is used which specifies what the exploited process should do after control has been hijacked. In most cases this payload is identical for each attack carried out. The algorithms in this thesis enable the attacker to perform any of the operations using a different instruction sequence for each attack. Therefore the attacker can use a unique attack pattern for each attempt rendering some of the defensive mechanisms useless and make incident response more complicated.

7.7. Conclusion

In this thesis novel algorithms for return oriented programming on ARM and the problems during their implementation have been discussed. The implementation of the resulting system is the first tool which uses a platform independent intermediate language to perform the gadget search and it is the first tool for return oriented programming on ARM. The necessary background about the ARM architecture was presented and the differences to other architectures in regard to return oriented programming where shown. The Windows mobile operating system was also described to be able to understand what differentiates a mobile operating system from a desktop system, and why some limitations apply to the mobile world only. As all presented algorithms use the meta-language REIL as their basis, the meta-language, its instructions, and the used REIL VM where introduced. The presented proof of concept exploit showed that return oriented programming is possible on the ARM architecture. The thesis made is therefore proven. In this final chapter some of the future areas of work have been outlined which are believed to be of importance for return oriented programming and its further automation. It is hoped that the described research areas will receive sufficient attention in the close future and lead to further results and techniques that are applicable to modern real world operating systems.

A. Bibliography

Openbsd. http://www.openbsd.org. [Online; accessed 11-December-2009].

James P. Anderson. Computer security technology planning study. Technical report, HQ Electronic Systems Division, 1972.

M. Becher, F.C. Freiling, and B. Leider. On the effort to create smartphone worms in windows mobile. In *Information Assurance and Security Workshop, 2007. IAW '07. IEEE SMC*, pages 199–206. IEEE, 2007. ISBN 1-4244-1304-4. doi: 10.1109/IAW.2007.381933.

Michael Becher and Ralf Hund. Kernel-level interception and applications on mobile devices. http://pi1.informatik.uni-mannheim.de/filepool/publications/TR-2008-003.pdf, May 2008. [Online; accessed 11-December-2009].

Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In Paul Syverson and Somesh Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.

Tim Burrell. Gs cookie protection effectiveness and limitations. http://blogs.technet.com/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx, 2009. [Online; accessed 11-December-2009].

CERT/CC. Cert advisory ca-2001-19. Technical report, CERT, July 2001. [Online; accessed 11-December-2009].

CERT/CC. Cert advisory ca-2003-04. Technical report, CERT, January 2003. [Online; accessed 11-December-2009].

S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In *Proc. 2009 USENIX/ACCURATE/IAVoSS Electronic Voting Technology Workshop (EVT/WOTE '09)*, 2009. [Online; accessed 13-December-2009].

Microsoft Corporation. Arm stack frame layout. http://msdn.microsoft.com/en-us/library/ms881427.aspx, April 2004. [Online; accessed 11-December-2009].

Cortulla. http://forum.xda-developers.com/showthread.php?p$\bar{1}$571715, October 2007.

Solar Designer. Getting around non-executable stack (and fix). Technical report, false.com, 1997a. [Online; accessed 11-December-2009].

Solar Designer. Non-executable stack patch. http://lkml.indiana.edu/hypermail/linux/kernel/9706.0/0341.html, 1997b. [Online; accessed 11-December-2009].

XDA developer members. xda-developers. http://xda-developers.com/. [Online; accessed 13-December-2009].

Suelette Dreyfus and Julian Assange. *Underground : tales of hacking, madness obsession on the electronic frontier / Suelette Dreyfus ; with research by Julian Assange*. Mandarin, Kew, Vic. :, 1997. ISBN 1863305955 1863305955. [Online; accessed 11-December-2009].

Thomas Dullien. Automated reverse engineering. http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-halvar.pdf, 2003. [Online; accessed 11-December-2009].

Thomas Dullien and Sebastian Porst. Reil the reverse engineering intermediate language. http://www.zynamics.com/BinNavi/manual/html/reil.htm, 2008. "[Online; accessed 11-December-2009]".

Nicolas Economou and Alfredo Ortega. Smartphone (in) security. In *Proceedings of Ekoparty 2008*. EKOPARTY, October 2008. [Online; accessed 11-December-2009].

Security Focus. Bugtraq mailing list. [Online; accessed 11-December-2009].

Willem Jan Hengeveld. about windows ce 3.0. http://www.xs4all.nl/ itsme/projects/xda/wince-kernelinfo.html. [Online; accessed 11-December-2009].

Hex-Rays. Windows ce arm debugger primer. http://www.hex-rays.com/idapro/wince/index.htm. [Online; accessed 11-December-2009].

Tim Hurman. Armed combat: The fight for personal securtiy. http://www.pentest.co.uk/documents/armed_combat.pdf. [Online; accessed 11-December-2009].

Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/ krahmer/no-nx.pdf, September 2005. [Online; accessed 11-December-2009].

Boris Michael Leidner. Vorraussetzungen für die entwicklung von malware unter windows mobile 5. Master's thesis, RWTH Aachen, February 2007.

Sue Loh. Inside windows ce api calls. http://blogs.msdn.com/ce_base/archive/2006/02/02/Inside-Windows-CE-API-Calls.aspx, February 2006. [Online; accessed 11-December-2009].

ARM Ltd. Arm architecture reference manual. Pdf, ARM Ltd., 2005. [Online; accessed 11-December-2009].

ARM Ltd. Procedure call standard for the arm architecture. Pdf, ARM Ltd., 2008. [Online; accessed 11-December-2009].

Collin Mulliner. Exploiting pocketpc. In *Proceedings of What The Hack! 2005*, July 2005. [Online; accessed 13-December-2009].

Collin Mulliner. Advanced attacks against pocketpc phones. In *Proceedings of DEFCON 2006*, August 2006. [Online; accessed 13-December-2009].

Collin Mulliner. Symbian exploitation and shellcode development. In *Proceedings of BHJP 2008*, 2008. [Online; accessed 11-December-2009].

Collin Mulliner and Charlie Miller. Injecting sms messages into smart phones for security analysis. http://www.usenix.org/events/woot09/tech/full_papers/mulliner.pdf, 2009a. [Online; accessed 11-December-2009].

Collin Mulliner and Charlie Miller. Fuzzing the phone in your phone. In *Proceedings of BHUSA 2009*, 2009b. [Online; accessed 11-December-2009].

Grant Nieporte. Seven pounds. Columbia Pictures Movie, 2009.

Ryan Glenn Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, University of California, San Diego, 2009. [Online; accessed 11-December-2009].

San. Hacking windows ce (pocketpcs & others). Technical Report 63, Phrack Staff, aug 2005. [Online; accessed 11-December-2009].

Bruce Sanderson. Ram, virtual memory, pagefile and all that stuff. http://support.microsoft.com/?scid=kb%3Ben-us%3B555223&x=8&y=12. "[Online; accessed 13-December-2009]".

Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

Pablo Sole. Defeating dep, the immunity debugger way. http://www.immunitysec.com/downloads/DEPLIB.pdf, 2008. [Online; accessed 11-December-2009].

Wikipedia. Nx-bit — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?titleN̄XBit&oldid=66438075, 2009a. [Online; accessed 13-December-2009].

Wikipedia. Erfüllbarkeitsproblem der aussagenlogik — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Erfüllbarkeitsproblem_der_Aussagenlogik&oldid6̄7761000, 2009b. [Online; accessed 11-December-2009].

Wikipedia. Source lines of code — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Source_lines_of_code&oldid=329192361, 2009c. [Online; accessed 11-December-2009].

Rafal Wojtczuk. The advanced return-into-lib(c) exploits. Technical report, www.phrack.org, 2001. [Online; accessed 11-December-2009].

zynamics GmbH. http://www.zynamics.com/, 2004. [Online; accessed 11-December-2009].

B. List of Figures

Listings

List of Algorithms