RUHR-UNIVERSITÄT BOCHUM

Cross Platform Code Gadget Discovery for Code Reuse Attacks

Patrick Wollgast

Master's Thesis – May 26, 2015.
Chair for System Security.

1st Supervisor:	Prof. Dr. Thorsten Holz
2nd Supervisor:	Dipl.-Biol. Robert Gawlik
Advisor:	Dipl.-Biol. Robert Gawlik

Abstract

Due to the insufficiency of defenses like ASLR, DEP, and SSP to prevent exploitation of vulnerable programs, several defense mechanisms incorporating CFI and heuristic checks have emerged. An ideal CFI prevents both code-injection and code-reuse attacks. However, the emerged defenses have to make trade-offs between security and practicability. Because of this trade-off, multiple code-reuse bypasses for the CFI and heuristic checks have been published.

In this thesis we present some of these defense mechanisms and the published code-reuse bypasses. Based on the presented bypasses, we develop a system to automatically discover CFI resistant code snippets, called gadgets, for code-reuse attacks. To perform the discovery process as architecture independent as possible, we utilize the intermediate language VEX. As the gadgets to bypass the CFI and heuristic checks are rather complex, we have to analyze the gadgets. For this, we propose a translation from VEX to Z3, called Zex3, and a symbolic execution engine for the translation, Zolver3. We show that the analysis results of Zolver3 are suitable to apply semantic definitions to the gadgets. With the help of Zex3 and Zolver3, we can analyze and search gadgets on x86, x86-64/AMD64, and ARM. To evaluate our system, we analyze the gadget distribution on all three platforms. Additionally, we show that we find the same gadgets with our semantic search than the original publication found for their evaluation. As no prior work towards CFI resistant gadgets on ARM has been published, we evaluate our search on ARM on a fictional but probable scenario.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Datum

Unterschrift

Contents

1 Introduction

1.1 Motivation

The development of various defense mechanisms, such as the introduction of *data execution prevention* (DEP) [18], *stack smashing protection* (SSP) [13], and *address space layout randomization* (ASLR) [42], have raised the bar for reliable exploit development significantly. Nevertheless, a dedicated attacker is still able to achieve code execution [33, 25]. *Information leaks* are utilized to counter ASLR and reveal the layout of the address space or other valuable information [40, 33]. To circumvent DEP the attackers have added code-reuse attacks, such as *return-oriented programming* (ROP) [39, 9, 26], *jump-oriented programming* (JOP) [16, 11, 5], and *call-oriented programming* (COP) [10], to their repertoire. Code-reuse attacks do not inject new code but chain together small chunks of existing code, called *gadgets*, to achieve arbitrary code execution.

In response to this success, the defensive research was driven to find protection methods against code-reuse attacks. Some results of this research are *kBouncer* [32], *ROPecker* [12], *EMET* [19]/*ROPGuard* [21], *BinCFI* [53], and *CCFIR* [52]. These defenses incorporate two main ideas. The first is to enforce *control-flow integrity* (CFI) [1, 2]. With perfect CFI the control-flow can neither be hijacked by code-injection nor by code-reuse [22]. However, the overhead of perfect CFI is too high to be practical. Therefore the proposed defense methods try to find the sweet spot between security and tolerable overhead. The second idea is to detect code-reuse attacks by known characteristics of an attack like a certain amount of gadgets chained together. All of those schemes defend attacks on the x86/x86-64 architecture. For other architectures the research is lacking behind [15].

Several generic attack vectors have been published by the offensive side to highlight the limitations of the proposed defense methods. Although single implementations can be disabled with common code-reuse attacks by exploiting a vulnerability in the implementation [14, 7], the generic ones rely on longer and more complex gadgets [22, 17, 37, 10, 23]. Since the gadgets loose their simplicity by becoming longer it also becomes harder to find specific gadgets and chain them together. To our knowledge there is no gadget finder available to search CFI resistant gadgets.

1.2 Related Work

Related work is already partially covered by our Motivation. Chapter 2 also gives a detailed introduction to research on countermeasures against code-reuse attacks and ways to defeat these countermeasures. Therefore, we just focus on an overview of the development and history of code- reuse attacks in this chapter. Additionally, we introduce the features of Miasm which are related to our work.

The first public exploit utilizing code-reuse was published in 1997 by Solar Designer in an email to the Bugtraq mailing list [41]. It was a *return-into-libc* exploit that bypasses the non-executable stack by directly returning to the system function in libc. In 2001 Nergal published an article in Phrack Magazine called *Advanced return-into-lib(c) exploits (PaX case study)* [34]. Nergal describes several advancements of the original return-to-libc approach, such as shifting the stack pointer, chaining functions, inserting NULL bytes, and to call multiple functions with arguments. With the introduction of x86-64, a new application binary interface (ABI) was introduced for the new 64 bit processes. This ABI requires the parameters for a function to be passed in registers instead of the stack. This rendered the current return-into-libc technique useless for 64 bit processes. Sebastian Krahmer proposed a method called *borrowed code chunks* technique in 2005 [27] to get the parameters from the stack into the registers. He chains together code snippets ending with return instruction to perform the required operations. The approach to chain together small code snippets was extended by Shacham in 2007 [39]. He was the first to present a Turing-complete instruction set consisting of small code snippets ending in return instructions, which he called *gadgets*. In his work Shacham also introduces the name *return-oriented programming* (ROP). Since then, Shacham's approach for x86 has been applied to other architectures, for example to SPARC by Buchanan et al. [9], and to ARM by Kornau [26]. Kornau was also the first to utilize an intermediate language (IL) for an architecture independent gadget search. Modifications of the original ROP utilize jumps (JOP) [16, 11, 5], calls (COP) [10], and sigreturns (SROP) [8] to redirect the control-flow to the beginning of the next gadget.

For this thesis we develop a translation unit from the IL VEX to the syntax of the *satisfiability modulo theories* (SMT) solver Z3 [48]. Parallel to the development of our VEX to Z3 translation, the open source reverse engineering framework Miasm has also received Z3 bindings. While Miasm does not support as many architectures as VEX, the use of Miasm has other advantages. We discuss these advantages during the course of this thesis.

1.3 Contribution

The objective of this thesis is to develop a tool that assists the exploit developer at locating gadgets which circumvent CFI and heuristic checks. All changes to registers and memory space must be immediately evident to the exploit developer. For this reason, we analyze the gadgets and categorize every register and every memory write by a set of semantic definitions. We use an IL for the analysis to support different architectures without the effort of adjusting the algorithms to new architectures. Because of the high architecture coverage, *VEX* [44] is our choice for the IL. VEX is part of *Valgrind*, an instrumentation framework intended for dynamic use [43]. To use VEX statically *pyvex* [51] is utilized. We instrument the SMT solver Z3 [48] for the analysis itself. Unfortunately, there is no interface from VEX or pyvex to Z3 available. Hence, we develop a translation from VEX to Z3. A wrapper adds the architecture layouts to Z3, thus making Z3 suitable for symbolic execution and path constrain analysis of the gadgets. To sum up, our thesis makes the following contributions:

1. We develop a tool to discover CFI and heuristic check resistant gadgets in an architecture independent offline search.
2. We provide semantic definitions for a convenient search of the gadgets.
3. We provide a modular translation unit from VEX/pyvex to Z3, including a symbolic execution engine implemented for x86, x86-64, and ARM.

1.4 Organization of this Thesis

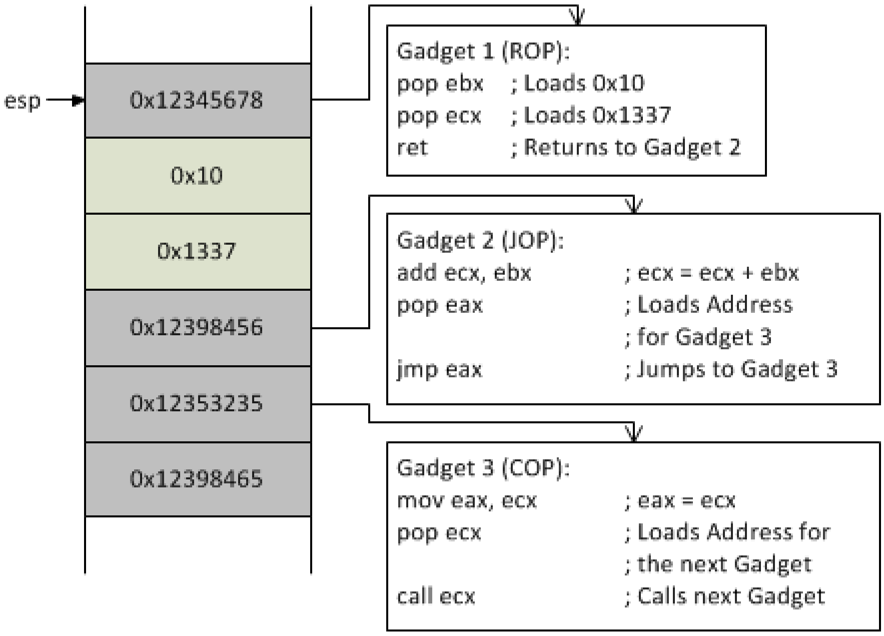
Our thesis is structured as follows. In Chapter 2 we provide background information on code-reuse attacks, proposed protections against code-reuse attacks, and methods to circumvent these protections. Additionally, we give an introduction to the IL VEX and the SMT solver Z3. Chapter 3 presents our algorithms that translate VEX instructions to Z3. Also, we present the general structure of our translation module. We conclude the chapter by explaining the algorithms and structure of our execution engine for the VEX to Z3 translation: Zolver3. We provide our gadget definitions and algorithms for the gadget search and analysis in Chapter 4. Chapter 5 begins with an overview of our implementation, to give a grasp of the coherences of the different components. Afterwards, we give a more detailed view of the implementation's components. In Chapter 6, we present an overview of the distribution of the gadgets, which are discovered by our gadget finder. We also show that we find the same gadgets as the ones presented in the papers that influenced our gadget definitions. Chapter 7 concludes our thesis and gives an outlook to future research.

2 Background

The following chapter starts with an introduction to code-reuse attacks. After the introduction to code-reuse attacks follows an introduction to CFI and an overview of the current protection mechanisms deploying CFI and heuristic checks. It is important to understand the concept of CFI and the presented heuristic checks, as we focus on CFI and heuristic check resistant gadgets in this thesis. Afterwards, we present countermeasures to the introduced defense mechanisms. Following this, we introduce the intermediate language VEX and the theorem prover Z3. We utilize both during the course of this thesis.

2.1 Code-reuse Attacks

The introduction of code and data separation, for example implemented by the *data execution prevention* (DEP) [18] on Windows, provides a useful protection against the injection of new code. This contradicts with the goal of an attacker to achieve arbitrary code execution by code-injection upon exploitation of a vulnerability. Therefore, attackers often resort to reusing code already provided by the vulnerable executable itself or one of its libraries. Vulnerabilities suitable for code-reuse attacks are memory corruptions like a stack, heap, or integer overflow, or a dangling pointer. The technique most commonly applied to reuse existing code is *return-oriented programming* (ROP) [39, 9]. The concept behind ROP is to combine small sequences of code, called *gadgets*, that end with a return instruction. All combined gadgets of an exploit are often referred to as a gadget *chain*. A gadget chain with a ROP gadget (Gadget 1) is given in Figure 2.1. To be able to combine these gadgets either a sequence of return addresses, each address pointing to the next gadget in the chain, has to be placed on the stack or the stack pointer has to be redirected to a buffer containing these addresses. The process of redirecting the stack pointer is called *stack pivoting*. For architectures with variable opcode length like x86/x86-64 the instructions used for the gadgets do not have to be aligned to the original, by the compiler intended, instructions. But even without the unintended instructions, for example at architectures with fixed opcode length, previous work has shown that enough gadgets for arbitrary computations can be located [26, 16, 9]. Automated tools to search gadgets and chain them together are also available [24, 31]. Variations of ROP are *jump-oriented programming* (JOP) [16, 11, 5] and *call-oriented programming* (COP) [10]. JOP uses jumps instead of returns to direct the control-flow to the



```
Gadget 1 (ROP):
pop ebx   ; Loads 0x10
pop ecx   ; Loads 0x1337
ret       ; Returns to Gadget 2
```

```
Gadget 2 (JOP):
add ecx, ebx        ; ecx = ecx + ebx
pop eax             ; Loads Address
                    ; for Gadget 3
jmp eax             ; Jumps to Gadget 3
```

```
Gadget 3 (COP):
mov eax, ecx        ; eax = ecx
pop ecx             ; Loads Address for
                    ; the next Gadget
call ecx            ; Calls next Gadget
```

Figure 2.1: A simple gadget chain for the x86 architecture containing a ROP, JOP, and a COP gadget. The gadgets perform the calculation 0x10 + 0x1337.

next gadget, and COP uses calls. Gadget 2 and Gadget 3 in Figure 2.1 are examples for JOP and COP gadgets.

Due to their complexity, code-reuse attacks are typically used to make injected code executable thus defeating protections like DEP and redirect the control-flow to the injected code [33, 25].

A well-established defense against code-reuse attacks is *address space layout randomization* (ASLR) [42]. ASLR randomizes regions like the base address of the executable and its libraries, the stack, and the heap. Therefore, an attacker does not know the addresses of the required gadgets. However, a single library without ASLR [25] or an information leak [33] is enough to defeat ASLR and enable code-reuse attacks again. Even *fine-grained* ASLR [46, 4] which randomizes for example the *basic block* (BB) order is vulnerable to *just-in-time code reuse* (JIT-ROP) [40]. JIT-ROP utilizes an information leak to scan the vulnerable process for gadgets during the attack and compiles a gadget chain on the fly.

2.1.1 Current Countermeasures

The following section gives an introduction to proposed countermeasures against code-reuse attacks. At first we give a short introduction to *control-flow integrity* (CFI) as it plays an important role for the presented defenses. Afterwards, we explain the features of five different defense mechanisms for the x86/x86-64 architecture.

Even though there are CFI systems for the ARM architecture [15] available, the research for the x86/x86-64 architecture is further developed and this thesis was written with just the presented defenses in mind.

Control-Flow Integrity

Abadi et al. [1, 2] introduced CFI on the *12th ACM conference on Computer and communications security* (CCS). A program maintains CFI if the control-flow remains in a pre-defined *control-flow graph* (CFG). This pre-defined CFG contains all intended execution paths of the program. The nodes of the CFG are BBs, a sequence of instructions, and edges are direct and indirect control-flow transfers, for example calls, jumps, or returns. If an attacker redirects the control-flow via code-injection or code-reuse to an unintended execution path the CFI is violated and the attack detected.
In an ideal CFG every indirect transfer has a list of valid unique identifiers (ID)s and every transfer target has an ID assigned [22]. This way it can be checked before every indirect transfer if the target is valid. In Figure 2.2 an example for an ideal CFG is given. It is just one check performed for every *indirect call* (IC) in function1(). This means that every IC has just one valid call target, unlike the return of function2(), which has two targets. Both invalid return addresses at function3() are ROP transfers.

If CFI is applied to proprietary software, it is problematic to cover such a detailed CFG. To construct the CFG, the program has to be disassembled and a pointer analysis performed. Every error made during this process may lead to false positives during runtime of the protected program. Therefore, implemented CFI solutions typically reduce the number of IDs by assigning the same ID to the same category of targets. However, this also decreases the security, for example if all returns share the same ID, the invalid return in function3() to ID12 can not be distinguished from the valid return to ID13. Both transfers maintain CFI.

kBouncer

The approach by Pappas et al. [32] to prevent code-reuse attacks is called kBouncer. It won Microsoft's *BlueHat Prize Contest* in 2012 [6]. To perform CFI checks kBouncer leverages the *Last Branch Record* (LBR). LBR is a recent feature of Intel processors which can only be enabled and disabled in kernel mode. Therefore, kBouncer consists of a user and kernel mode component. Like the name suggests, LBR records the last taken branches or a subset of the last branches. In the case of kBouncer, LBR is configured to just record indirect branches in user mode. 16 entries are available to store the recent branches to the LBR stack. Each entry contains the source and destination address of the taken branch. With these information kBouncer can enforce that every return address is preceded by an intended

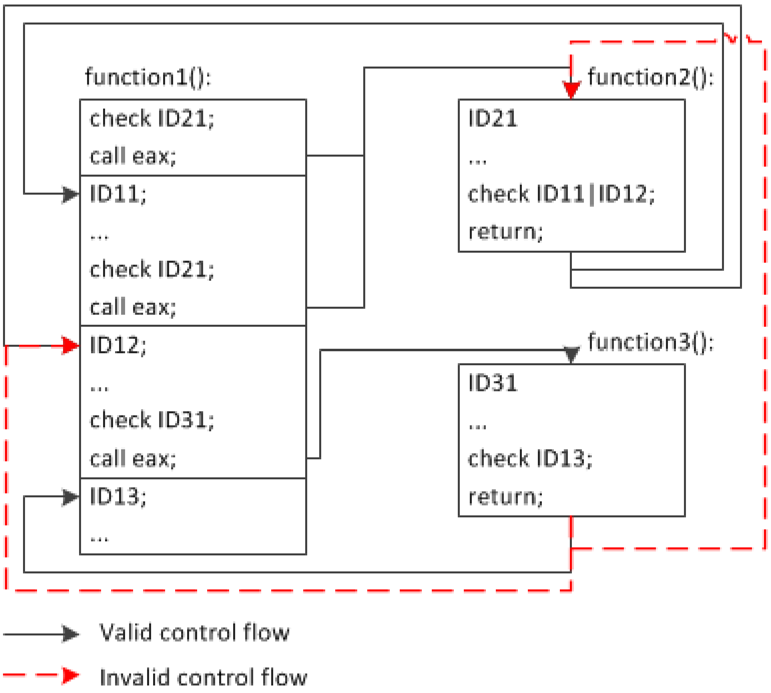


Figure 2.2: The CFG of an ideal CFI. In an ideal CFI all invalid control flow transfers are detected.

or unintended call instruction. Otherwise kBouncer reports a CFI violation. In the following we refer to the instruction after a call as call-site (CS).
Besides the CFI enforcement, a heuristic check is performed. For the heuristic check kBouncer inspects the 8 last indirect branches in the LBR stack. If all entries match kBouncers gadget definition, an attack is reported. An entry is considered a gadget if it contains up to 20 instructions and ends in an indirect control-flow. The instruction count is the length of the shortest possible path a gadget can take from the start to the endpoint. To reduce the runtime the gadgets are searched for and evaluated beforehand during an offline gadget search.
These checks are invoked whenever one out of 52 critical WinAPI functions, like *VirtualProtect* and *WinExec*, is called. The user mode component hooks these critical functions and triggers the checks in the kernel mode component. A checkpoint system prevents that an attacker skips the hooks by calling a function on a lower level than the corresponding WinAPI function.

ROPecker

ROPecker by Cheng et al. [12] also utilizes the LBR stack to look for gadgets in the past control-flow. Additionally the future control-flow is also checked. To check for gadgets in the future control-flow, ROPecker combines online emulation of the flow, stack inspection, and an offline gadget search. Since gadgets are already searched offline and stored to a database, ROPecker has also the possibility to detect unaligned gadgets.

To detect gadgets, ROPecker does not apply CFI enforcements, but merely relies on heuristics. During an offline analysis of the to-be protected program all possible aligned and unaligned gadgets are collected and stored to a database. A gadget in the context of ROPecker is a sequence of up to 6 instructions ending with an indirect control-flow transfer. Sequences containing direct branch instructions are excluded from the definition. ROPecker inspects the past control-flow first. For this, LBR just records indirect branch instructions in user mode. The first non-gadget encountered while walking the LBR backwards terminates the search for gadgets in the past control-flow. If the found gadgets do not exceed a certain threshold the future control-flow is inspected. The research of Cheng et al. suggests that a threshold between 11 and 16 gadgets is a suitable number. Afterwards, the future control-flow is also inspected for gadgets. Here, too, the first non-gadget terminates the search. If the combined number of encountered gadgets from the past and future control-flow is above the threshold, an attack is reported.

Two methods trigger the detection logic. The first method intercepts calls to *mprotect*, *mmap2* and *execve* by modifying the system call table. This way no binary rewriting has to be applied and potentially dangerous arguments can be rejected. Potentially dangerous arguments are for example requests to change the code section to writable. The second method is a sliding window mechanism implemented with the help of code page permissions. ROPecker sets just the most recent visited pages to executable. An instruction outside of the to executable set pages throws an access violation. At every access violation the detection logic is started. If no attack is detected the new page is set to executable and the oldest page in the window is set to non-executable. Cheng et al. suggest a window size of 8 to 16 KB, which corresponds to 2 to 4 pages.

EMET

The *Enhanced Mitigation Experience Toolkit* (EMET) [19] is an optional utility provided by Microsoft to enhance the security of Windows. At time of writing the most recent version is 5.2. Since *EMET 3.5 Technology Preview* EMET also incorporates ROPGuard by Fratic [21]. ROPGuard placed second at Mircosoft's BlueHat Prize Contest in 2012 [6]. In this paragraph the most relevant features of EMET to prevent code-reuse are described. For further information on its general exploit mitigation features refer to other research [35, 3] and the *EMET User Guide* [20].

In contrast to kBouncer and ROPecker, EMET does not use the LBR stack to inspect multiple indirect branches in the past, but merely ensures that the return address of the current function points to an intended or unintended CS. In addition, EMET simulates the future control-flow to detect return instructions which are not preceded by a call instruction. The threshold specifying how many return instructions should

be verified is 15 by default.
Furthermore, EMET applies several heuristic checks. To prevent stack pivoting it checks if the stack pointer is in the stack range of the corresponding thread. It also rejects calls to functions like *VirtualProtect* if they are trying to set stack memory to executable. The last check prevents loading of libraries from *Uniform Naming Convention* (UNC) paths, for example *\\attacksite\evil.dll*.
Similar to kBouncer, EMET also hooks functions to start its detection engine upon a call to a critical function call. The current version of EMET hooks a total of 56 functions.

BinCFI

BinCFI by Zhang and Sekar [53] is a pure CFI approach to counter code-injection and code-reuse attacks. It is applied to *commercial off-the-shelf* (COTS) binaries without the need for debug symbols or relocation information. To produce a correct disassembly of the binary, Zhang and Sekar add several error correction methods on top of *objdump*.
BinCFI uses 2 IDs to ensure the integrity of the CFG. The first ID defines rules for targets of *return* (RET) instructions and *indirect jumps* (IJ). For these transfers the following targets are allowed:

1. Intended CSs.
2. Exception handlers which are discovered by parsing the ELF sections *.eh_frame* and *.gcc_ except_ table*.
3. Constant code pointers. This also includes code pointers to shared libraries.
4. Computed code pointers, like the ones produced by switch statements in C/C++ programs.

The second ID combines rules for indirect control-transfers from the *procedure linkage table* (PLT) and ICs. PLT is a part of the *Executable and Linkable Format* (ELF), the standard format for executables on Unix systems. Whenever a dynamically linked function from another module is called, the PLT is involved in the process of linking and calling the function. The following targets are defined for these transfers:

1. Exported symbols found in the *.dynamic* section.
2. Constant code pointers.
3. Computed code pointers.

Every indirect transfer is instrumented to jump to one of two verification routines. Each ID has its own routine which resides inside the protected binary.

CCFIR

Similar to BinCFI, CCFIR [52] is also a pure CFI approach applied to binaries without source code. It was presented by Zhang et al. on the *Proceedings of the 2013 IEEE Symposium on Security and Privacy.* To produce a correct disassembly CCFIR utilizes the relocation information of the target binary. Every Windows binary and library contains a relocation table, if it supports dynamic linking or ASLR. Therefore, it is save to assume that the relocation information are available in the general case. With the generated disassembly all indirect control-transfers and all targets are collected. Each indirect transfer is redirected through a *Springboard.* The Springboard contains all valid control-flow targets and thereby prevents that the flow is redirected to invalid targets. Initial permutation of the Springboard at program startup raises the bar for attackers further.
CCFIR uses a 3 ID approach to ensure CFI. The first ID groups the targets of ICs with the targets of IJs. These transfers are just allowed to target *function entry points* (EP). A set of security sensitive functions are excluded from this ID. They have to be called with a direct function call. ID2 contains return addresses. Only return addresses at intended CSs are valid targets for return instructions. The third ID contains return addresses within the previously mentioned security sensitive functions. However, return instructions of security sensitive functions are allowed to target return addresses of ID2 and ID3, while normal return instructions are just allowed to target return addresses of ID2.

2.1.2 Defeating the Countermeasures

For all defenses against code-reuse attacks in Chapter 2.1.1, bypasses have been published. While some bypasses exploit vulnerabilities in a specific implementation to disable the checks all together [14, 7], we focus on generic bypasses to defeat the protections. We divide the defense policies in two categories, CFI policies posing limitations on indirect branch instructions and heuristic policies looking for typical characteristics of code-reuse attack vectors. In the following we present discovered bypasses of current publications for both categories.

CFI Policies

Publications focusing on kBouncer, ROPecker, and EMET/ROPGuard [23, 10, 37] just have to bypass the CS checks. However, publications that include BinCFI and CCFIR in their considerations [22, 17] also have to take into account that ICs and IJs are limited to certain control-flow targets like EPs. Göktas et al. [22] categorize the gadgets by their prefix (CS or EP), their payload (IC, *fixed function call* (F), other instructions), and their suffix (RET, IC, IJ). This categorization results in 18 $(2 \cdot 3 \cdot 3)$ different gadget types. They even use gadgets containing conditional jumps. With these gadget categories, they are able to bypass CCFIR, which they consider stricter than BinCFI. Another interesting gadget type is the *i-loop-gadget* [37]. Schuster et al. use a loop containing an IC to chain gadgets and invoke security sensitive functions. Gadgets from both publications, the one of Göktas et al. and the one of Schuster et al., are implemented by this thesis and further explained in Chapter 4.1.1.

Heuristic Policies

The heuristic policies check for chains of short instruction sequences. To evade those checks mostly long gadgets with minimal side effects are proposed [23, 10]. If the heuristic check encounters the long instruction sequence the evaluation is terminated and the chain is classified as benign. Another elegant method is to invoke a function call to an unsuspicious function like *lstrcmpiW* [37]. If the unsuspicious function does not alter the global state of the program and takes enough indirect branches, the attack can not be discovered by the heuristic checks.

2.2 VEX

Valgrind [43] is a tool collection incorporating tools to perform actions like memory debugging, memory leak detection, and dynamic profiling. One can also create own tools for dynamic profiling. Even though the list of supported platforms is long, for example x86, x86-64/AMD64, PPC, ARM, and MIPS [44], the tools just have to support one architecture. Valgrind executes all tested programs in a virtual machine (VM). This VM executes the byte-code of the intermediate language (IL) VEX, former called Ucode. To analyze a program, Valgrind translates the program's assembler code just-in-time (JIT) to VEX code. Therefore, all Valgrind-tools only have to support the architecture of Valgrind's VM and its VEX code. However, Valgrind is designed with dynamic analysis in mind. To perform static analysis based on VEX byte-code, pyvex was written [51]. Pyvex is a Python 2 tool to statically translate opcodes of an architecture supported by Valgrind to VEX. A big disadvantage of VEX and pyvex is that if an instruction is not supported, a segmentation fault is thrown which can not be caught by a try and except block in Python 2. This causes ones analysis script to abort. Nevertheless, we valued the high architecture coverage of VEX over this disadvantage. In the following chapter we give an introduction to the VEX architecture. It is important for this thesis to be familiar with VEX, because we use VEX in combination with pyvex and Z3 to introduce platform-independence to our analysis. At first we present some platform-specific details of the VEX architecture for x86/AMD64 and ARM32. Afterwards we introduce the instruction set of VEX.

2.2.1 Architecture-dependent Components

Even so VEX is an architecture-neutral language, every supported architecture has some architecture dependent parts embedded in VEX. The files implementing the architecture dependent components are located in the paths *VALGRIND\VEX\priv* and *VALGRIND\VEX\pub*. For x86 these files are *guest_x86_defs.h*, *guest_x86_helpers.c*, *guest_x86_toIR.c*, and *libvex_guest_x86.h*. These files implement the state of the x86 CPU and helper functions to perform calculations which are specific for the architecture, such as flag calculations and branch condition calculations. The CPU state, internally just a block of memory, contains all registers of the currently simulated CPU. Each register corresponds to an offset inside the CPU state's memory block. For example, the x86 register eax occupies 4 bytes beginning at the offset 8 of memory block. Table 2.1 shows all general purpose, flag, and instruction pointer registers of x86 and AMD64, and their corresponding VEX registers. *CC_OP*, *CC_DEP1*, *CC_DEP2*, and *CC_NDEP* are internally used by VEX to calculate flags and conditions.

ARM processors can have extensions to the normal ARM 32 bit instruction set, namely Jazelle (J), Thumb (T), and Tumb2 (T2). J extends the processor by a Java

Table 2.1: A list of the general purpose, flag, and instruction pointer registers on x86 and AMD64, and their VEX translation.

x86/AMD64 Registers	x86 VEX Registers	AMD64 VEX Registers
EAX (x86)/ RAX (AMD64)	8	16
ECX (x86)/ RCX (AMD64)	12	24
EDX (x86)/ RDX (AMD64)	16	32
EBX (x86)/ RBX (AMD64)	20	40
ESP (x86)/ RSP (AMD64)	24	48
EBP (x86)/ RBP (AMD64)	28	56
ESI (x86)/ RSI (AMD64)	32	64
EDI (x86)/ RDI (AMD64)	36	72
R8 (AMD64)		80
R9 (AMD64)		88
R10 (AMD64)		96
R11 (AMD64)		104
R12 (AMD64)		112
R13 (AMD64)		120
R14 (AMD64)		128
R15 (AMD64)		136
CC_OP	40	144
CC_DEP1	44	152
CC_DEP2	48	160
CC_NDEP	52	168
DFLAG	56	176
IDFLAG (Bit 21 of eflags)	60	200
ACFLAG (Bit 21 of eflags)	64	192
EIP (x86)/ RIP (AMD64)	68	184

byte-code implementation to allow native Java byte-code execution. The T mode adds an additional 16 bit instruction set to allow higher code density. T2 extends the original Thumb mode by some 32 bit instructions. The T/T2 extensions are

Table 2.2: A list of the general purpose and flag registers on ARM and their VEX translation.

ARM Registers	ARM VEX Registers
R0	8
R1	12
R2	16
R3	20
R4	24
R5	28
R6	32
R7	36
R8	40
R9	44
R10	48
R11	52
R12	56
R13/SP	60
R14/LR	64
R15/PC	68
CC_OP	72
CC_DEP1	76
CC_DEP2	80
CC_NDEP	84
QFLAG32	88

integrated in VEX. All instructions on ARM processors are aligned to four bytes, or two bytes if T/T2 mode is used. Therefore, the last bit of the instruction pointer is redundant to address instructions, as it is always zero. VEX uses this bit to encode if T/T2 is enabled. Another feature of ARM is that multiple forms of endianness are supported for data access, like little-endian and big-endian. However, in pyvex the endianness has to be set statically before the build process of pyvex is started. By default it is set to little-endian. Table 2.2 presents all general purpose and flag related ARM registers and their VEX translations. In contrast to x86/AMD64 the instruction pointer is also a general purpose register and addressable by the same instructions as all other general purpose registers.

For further information on the architecture-independent components of VEX we recommend the study of the source files [45], as they are the only available documentation.

2.2.2 VEX Instructions

In general IRs have a list of instructions and translate the original architecture dependent assembler to these instructions. The concept behind VEX differs slightly from this one. While VEX also has instructions, they can not be inspected separately but must be inspected in groups. Every translation group is called *super block* (IRSB) and relates to one or several original assembler instructions. A IRSB always has one entry point and at least one exit point, but multiple conditional exits are also possible. An example IRSB is given in Listing 2.1. The IRSB starts with a type environment, indicating the type of each temporary variable, line 2. The *I32* indicates that the variables are 32 bit integers. Following to the type environment are several statements and expressions, representing the translated instructions.

```
 1  IRSB {
 2     t0:I32  t1:I32  t2:I32  t3:I32
 3
 4     IR-NoOp
 5     ...
 6     IR-NoOp
 7     ------ IMark(0x0, 3, 0) ------
 8     t2 = GET:I32(24)
 9     t1 = 0x10:I32
10     t0 = Add32(t2,t1)
11     PUT(40) = 0x3:I32
12     PUT(44) = t2
13     PUT(48) = t1
14     PUT(52) = 0x0:I32
15     PUT(24) = t0
16     PUT(68) = 0x3:I32
17     t3 = GET:I32(68)
18     PUT(68) = t3; exit-Boring
19  }
```

Listing 2.1: IRSB of the x86 instruction *add esp, 10h* (Intel syntax).

Statements are operations with side-effects, like writing to a register of the CPU state. What follows are all statements supported by VEX:

- Ist_NoOp - No operation.
- Ist_IMark - Provides address and length of the original assembler instruction. In Listing 2.1 the Ist_IMark in line 7 indicates that the *add esp, 10h* instruction starts at address 0x0 and is 3 bytes long.

- Ist_AbiHint - Provides information about the platform's ABI.
- Ist_Put - Writes to a register in the CPU state. Line 11 in the given IRSB example writes the 32 bit integer value 0x3 to the register at offset 40 in the CPU state.
- Ist_PutI - Also writes to a register in the CPU state, but unlike Ist_Put the register to write to is not fixed. This statement is used to write to rotating register files, like the x87 floating-point unit (FPU) stack.
- Ist_WrTmp - Assigns the source value to a temporary variable. An example is given in Listing 2.1 line 9. Every temporary variable can just be assigned once, otherwise the IRSB is invalid.
- Ist_Store - Stores a value in the specified endianness to the supplied memory address.
- Ist_LoadG - A guarded load. If the guard evaluates to true, the value is loaded from memory to a temporary variable. If it evaluates to false the old value remains in the temporary variable.
- Ist_StoreG - Like Ist_LoadG, but executing a store operation instead of a load in case the guard evaluates to true.
- Ist_CAS - Compare and swap. It compares a provided value with the content of a memory address. If they match another data value is stored to the checked address. Otherwise no store is performed.
- Ist_LLSC - Can be a Load-Linked or Store-Conditional.
- Ist_Dirty - Calls a function, for example a architecture dependent function, with side effects on the CPU state.
- Ist_MBE - A memory bus event, like an acquisition or release of the hardware bus lock.
- Ist_Exit - Not the exit seen in line 18 of the example, but a conditional exit. If the guard value of the exit evaluates to true a jump is performed to the instruction at the specified address, otherwise the next instruction in the IRSB is executed.

The operations without side-effects are called *expressions*. VEX supports the following expressions:

- Iex_Binder - Only used internally by VEX.
- Iex_Get - Reads the content of a register in the CPU state at a fixed offset. An example is given in Listing 2.1 line 8.
- Iex_GetI - Is the counterpart to Ist_PutI. It is used to read from a non-fixed offset in the CPU state.

- Iex_RdTmp - Gets the value of a temporary variable.
- Iex_Qop - A operation with four arguments, for example *MAddF64r32(t1, t2, t3, t4)*.
- Iex_Triop - A operation with three arguments, like *MulF64(1, 2.0, 3.0)*
- Iex_Binop - A operation with two arguments, like *Add32(t2,t1)* in the given IRSB example.
- Iex_Unop - A operation with just one argument, for example *Not1(t1)*.
- Iex_Load - Loads a value from a specified memory address.
- Iex_Const - Represents a constant expression, like *0x3:I32* in line 11 of the given IRSB example.
- Iex_ITE - An if-then-else operation. If the condition evaluates to nonzero, the expression in the *iftrue* parameter is returned, otherwise the *iffalse* parameter is returned.
- Iex_CCall - A call to a side-effect free function. This expression is used to call for example the flag calculation functions.
- Iex_VECRET - Contains a pointer to a V128 or V256. Just used as an argument for Ist_Dirty.
- Iex_BBPTR - Also just used as an argument for Ist_Dirty.

We refer to the file *VALGRIND\VEX\pub\libvex_ir.h* [45] for further information on VEX's operations.

2.3 Z3 Theorem Prover

Z3 is an open source *satisfiability modulo theories* (SMT) solver initiated by Microsoft [48]. SMT solvers combine the *propositional satisfiability* (SAT) problem with *background theories* like arithmetic, array, and bit vectors [28]. This combination makes them a powerful tool to solve for example path constrains in a control-flow of a program.

The advantage of Z3 is its available for every common operating system, for example Windows, Mac OS, Linux, and FreeBSD, and that APIs for both Python 2 and C/C++ are available. The support for C/C++ is beneficial in case we port our work to C to improve the speed.

A small example is given in the Listings 2.2 and 2.3. The assembler snippet in Listing 2.2 sets *ebx* to 10, if *eax* contains 0, otherwise ebx is set to 5. Listing 2.3 expresses the semantical equivalent of Listing 2.2 in the Z3 Python API [29], Z3Py. Since a register on the x86 architecture is just a bit vector, eax is initiated as 32bit BitVec in Z3. The value is unknown and should be solved by Z3. The result of the previously stated if/else expression is assigned to ebx in line 3. In line 5 we add the constrain to the Solver that ebx is supposed to contain the value 10 at the end of the execution. The satisfiability check yields *sat*. The model of the Solver in line 9 reveals that eax has to be 0 for ebx to contain 10.

```
test eax, eax
jz @skip
mov ebx, 5
retn
@skip:
mov ebx, 10
retn
```

Listing 2.2: A simple x86 assembler (Intel syntax) snippet for a Z3 example.

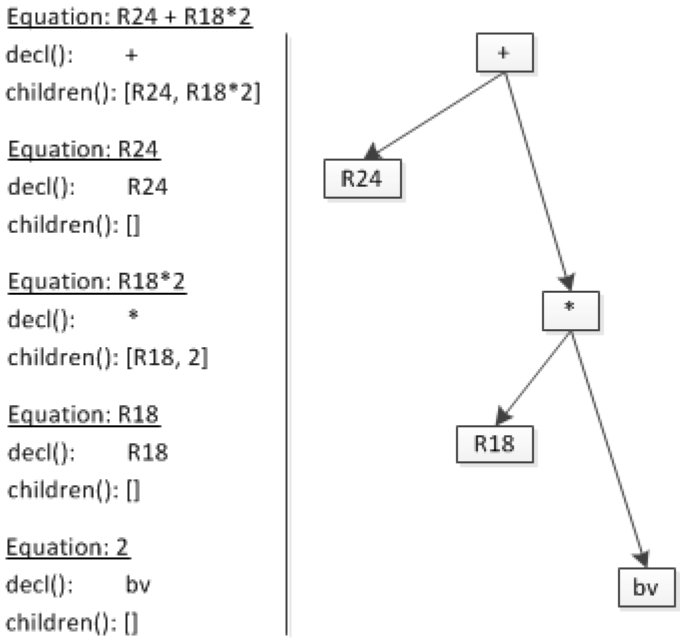
```
1 >>> from z3 import *
2 >>> eax = BitVec('eax', 32)
3 >>> ebx = If(eax & eax == 0, 10, 5)
4 >>> s = Solver()
5 >>> s.add(ebx == 10)
6 >>> s.check()
7 sat
8 >>> s.model()
9 [eax = 0]
```

Listing 2.3: The semantical Z3Py equivalent of Listing 2.2.

2.3.1 Structure of an Equation

We use Z3 equations to apply our semantic definitions to the gadgets, see Chapter 4.2.1. To understand the process of applying the semantic definitions to the equations, it is important to have basic knowledge of a Z3 equation structure.

Every Z3 equation is an expression tree, a mathematical expression represented in tree form. Every leaf node is a variable, for example a BitVec, a BitVecVal, an Integer, or an Array, and every non-leaf node is an operation like a multiplication or an addition. An exemplary equation with the corresponding expression tree is given in Figure 2.3. The value of the current node can be accessed in Z3Py with a call to *decl* from the current object. A call to *children* returns a list with all direct children of the current node. In the given example equation, *R24 + R18*2*, *R24* and *R18* are both BitVec variables. The *2* is a BitVecVal. One can observe that for BitVecs the name of the BitVec is displayed, while for BitVecVars just *bv* for BitVec is displayed.



```
Equation: R24 + R18*2
decl():     +
children(): [R24, R18*2]

Equation: R24
decl():     R24
children(): []

Equation: R18*2
decl():     *
children(): [R18, 2]

Equation: R18
decl():     R18
children(): []

Equation: 2
decl():     bv
children(): []
```

Figure 2.3: An illustration of the Z3 equation structure for the example equation *R24 + R18*2*.

2.3.2 Simplification of Equations

To simplify equations Z3 has the build in method *simplify.* simplify applies several algorithms to simplify the equation. At the time of writing 41 algorithms are available. However, by default not all are activated. A list of all available algorithms can be viewed with a call to *help_simplify.*
An example is given in Listing 2.4. One can see that without simplify just the variable *d* was exchanged for *b*. With simplify though, all multiplications and additions inside the braces are summarized. Additionally, the call to simplify in line 10 has additionally the algorithm *som* activated, which is turned off by default. An explanation of the algorithm is given in line 14.

```
>>> from z3 import *
>>> a = BitVec(’a’, 32)
>>> b = BitVec(’b’, 32)
>>> c = BitVec(’c’, 32)
>>> d = b
>>> (a*2 + b + d + c + a + d*2)*a
(a*2 + b + b + c + a + b*2)*a
>>> simplify((a*2 + b + d + c + a + d*2)*a)
a*(3*a + 4*b + c)
>>> simplify((a*2 + b + d + c + a + d*2)*a, som=True)
3*a*a + 4*a*b + a*c
>>> help_simplify()
...
som (bool) put polynomials in som-of-monomials form (default: false)
...
```

Listing 2.4: An example for Z3Py’s simplification.

3 Zex3 and Zolver3: Models and Algorithms

Our VEX to Z3 translation, called *Zex3*, is closely intertwined with our execution engine *Zolver3*. The following chapter gives an overview of the processes of Zex3 and Zolver3 and shows how they interact. We start with an introduction to the algorithms and design of Zex3. Afterwards we introduce Zolver3's algorithms.

3.1 Zex3

We want to have the ability to perform an analysis with our VEX to Z3 translation at any time. Even if the process originally performing the translation has already been terminated. Therefore, we must be able to store the output of the translation to a persistent storage. To fulfill this objective, Zex3 performs a translation from VEX to strings in Z3Py syntax. This way we can write the output strings to a file or to a database and access the translation at any time. The idea to utilize strings was developed in corporation with M. Osterloh.
We start this chapter by introducing a selection of the algorithms involved in the translation process from VEX to Z3. We continue with an overview of the design to highlight the modularity of Zex3, and conclude this chapter with the algorithm used to sort out unsupported instructions.

3.1.1 Translation Process

We take the approach to implement a handler for every statement and expression. For an explanation of statements and expressions refer to Chapter 2.2.2. These handlers take the values from the current statement or expression object and construct a string representing the functionality of the object in Z3Py syntax. To illustrate the translation process we show the algorithms involved during the translation of the x86 assembler instruction *inc ebx*. For simplicity the calculations of the flags and the instruction pointer are omitted.

```
1  t2 = GET:I32(20)
2  t1 = Add32(t2,0x1:I32)
3  t0 = t1
4  PUT(20) = t0
```

Listing 3.1: The x86 assember instruction *inc ebx* translated to VEX.

Statements

One can see upon inspection of Listing 3.1 that all assignments are statements. The statements performing the assignments in our example are Ist_WrTmp, lines 1 to 3, and Ist_Put in line 4. To cover all assignments, we initiate our translation by iterating over all statements of the IRSB object and calling their handlers, Algorithm 3.1.1.

Algorithm 3.1.1: VEX TO Z3 TRANSLATION INITIATION ALGORITHM

```
Input: irsb
begin
    forall the statements of irsb do
        CallHandlerByTag(statement)
```

The first three statements are of the type Ist_WrTmp. The structure of a Ist_WrTmp statement contains an *offset* specifying the name of the temporary variable, for example in line 1 the offset is 2, hence the name of the temporary variable is *t2*. We use this offset as an index to a dictionary containing all encountered temporary variable content. The content of the statement is stored in the *data* field of the structure and consists of an expression. To get the content as a string, the handler of the expression is called. This process is displayed in Algorithm 3.1.2.

Algorithm 3.1.2: HANDLER FUNCTION OF THE IST_WRTMP STATEMENT

```
Function Ist_ WrTmp(st)
    Data: st {the datastructure of the Ist_WrTmp statement}, tmp {dictionary of
          the Zex3 class containing the current values of VEX tmp variables}
    tmp(offset) ← CallHandlerByTag(st.data.tag, data)
```

The other statement that occurs in our example is Ist_Put to store the calculated value back to the CPU state register, Algorithm 3.1.3. Like Ist_WrTmp, Ist_Put also contains an offset as destination. This offset specifies the register of the CPU state, see Chapter 2.2.1. The source is constructed similarly to Ist_WrTmp. Ist_Put also has a data field containing the expression of the source value, therefore we call the handler of the associated expression to get the Z3Py string. Our writes to a register are always of the size of the register. Therefore, we have to consider a special feature of x86 and AMD64 at constructing the destination and source string. Both, x86 and AMD64, possess sub-registers, which may start at an offset of the original register. For example, the register eax has the size 4 bytes and contains the sub-registers *al* (1 byte), *ah* (1 byte), and *ax* (2 byte). Both, al and ax start at at the beginning of eax, thus we just have to extend the source to 4 byte and ensure that the rest of the register is not overwritten. However, the sub-register ah starts at an offset of 1 byte to the beginning of eax. To account for ah, we have to adjust the destination and source string. After the destination and source adjustments, an *Equation* object is created, Algorithm 3.1.3 line 11. The Equation object contains the destination, source, and potential condition string of the equation and further metadata. Conditions can be set by statements or expressions such as the conditional Ist_Exit statement. All assignments following Ist_Exit are just executed if the exit condition is invalid. One type of metadata are MemOP objects, which are created for every memory load and store. They are constructed by parsing the source string and creating a MemOP object for every encountered load and store.

Algorithm 3.1.3: Handler Function of the Ist_Put Statement

```
1  Function Ist_Put(st)
       Data: st {the datastructure of the Ist_Put statement}, condition {globally set
             for the Zex3 object}, equations {list containing all equations of the Zex3
             object}
       Result: equation {structure containing all relevant information about the
               Z3Py equation}
2      dst ← CreateString(st.offset)
3      src ← CallHandlerByTag(st.data.tag, data)
4      size ← GetDataSize(data)
5      if $size < arch\_reg\_size$ then
6          if IsAtOffsetToRegisterStart(st.offset) then
7              dst ← AdjustDst(dst)
8              src ← ExtendAndAdjustSrc(src)
9          else
10             src ← ExtendSrc(src)
11     equation ← Equation(dst, src)
12     AddRegUseToGlobalAndEqMetadata()
13     equation.AddMemOPs(src)
14     equation.AddCondition(condition)
15     equations.Add(equation)
```

Listing 3.2 shows the *pretty print* of the equation from our example. The appended _*o* indicates that the variable contains an output value of the IRSB. _*i* is by implication an input value of the IRSB. These appendixes also prevent the confusion of for example *R8* for *R88* while parsing and matching strings.

```
R20_o = (R20_i + BitVecVal(0x1, 32))
dst_registers: set(['R20_o'])
src_registers: set(['R20_i'])
```

Listing 3.2: *Pretty print* of example equation.

Expressions

Expression handlers can only be called from the initially called statement handlers, see Algorithm 3.1.1, or from other expression handlers. Compared to statements, expressions are rather simple. They just construct the Z3Py string of the current expression and return the string to the calling statement or expression. The expression handler containing the most checks is the one for Iex_Get, which is used in line 1 of Listing 3.1. Iex_Get has to account for the same special features of x86 and AMD64 that Ist_Put has to account for. But instead of extending the register, Iex_Get has to extract the loaded value, lines 6 to 10 in Algorithm 3.1.4. In the lines 3 to 5 we check if the returned register is supposed to be an input register. An example highlighting the importance of this check is given in Listing 3.3. In line 1 of Listing 3.3 the instruction output of *R24* is changed. If the already changed value is used in line 2 for the memory write, *R28* will be written to the wrong address.

```
1 R24_o = (R24_i - BitVecVal(0x4, 32))
2 [(R24_i - BitVecVal(0x4, 32)):32:<] = R28_i
```

Listing 3.3: The VEX to Z3 translation of the x86 assembler instruction *push ebp*. The instruction pointer changes are omitted. In line 1 the IRSB output value of the register *R24* is changed and in line 2 the input value is used again.

The Iex_RdTmp in the lines 3 and 4 of our example in Listing 3.1 just return the content for the temporary variable of the *tmp* dictionary, which was set by Algorithm 3.1.2.

Line 2 of Listing 3.1 is more complex and visualized in Figure 3.1. The expression Iex_Binop is initially called from the statement handler. However, the expression contains two arguments consisting of further expressions, which are resolved by calling their respective expression handlers.

The expressions Iex_Qop, Iex_Triop, Iex_Binop, and Iex_Unop are combined in a single algorithm, namely Algorithm 3.1.5. The string for the operation of the expression is taken from a global dictionary containing a format string for every supported operation. All operands are simply collected in a list and afterwards the formated string of the operation is returned.

Algorithm 3.1.4: Handler Function of the Iex_Get Expression

```
Function Iex_Get(data)
    Data: data {the datastructure of the Iex_Get expression}
    Result: reg {string containing the register}
    reg ← CreateOutRegString(data.offset)
    if IsInReg() then
        reg ← CreateInRegString(data.offset)
        AddRegUseToGlobalMetadata()
    if IsAtOffsetToRegisterStart(data.offset) then
        reg ← AdjustReg(reg)
    size ← GetDataSize(data)
    if $size < arch\_reg\_size$ then
        reg ← ExtractReg(reg)
    return reg
```

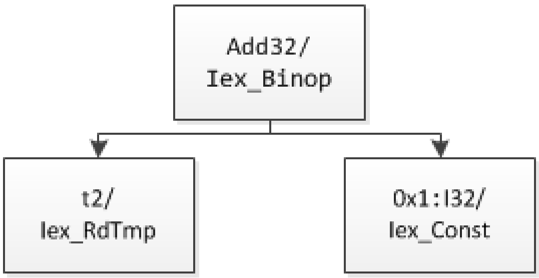


Figure 3.1: The structure of the Iex_Binop expression from our example in Listing 3.1.

Algorithm 3.1.5: Handler Function of the Iex_ALLop Expressions

```
Function Iex_ALLop(data)
    Data: data {the datastructure of the Iex_ALLop expression}
    Result: {string containing the operation with all operands}
    mne ← GetOperationFormatStringFromDict(data.op)
    forall the operands of data do
        ops.Add(CallHandlerByTag(operand.tag, operand))
    return FormatString(mne, ops)
```

3.1.2 Architecture Dependencies

One requirement for Zex3 is to keep the effort to add an additional architecture minimal. To comply with this requirement, we have to keep modularity of the design in mind. This is why we have the core component and all architecture dependent components in separate modules. Figure 3.2 shows the interaction between the components of Zex3.

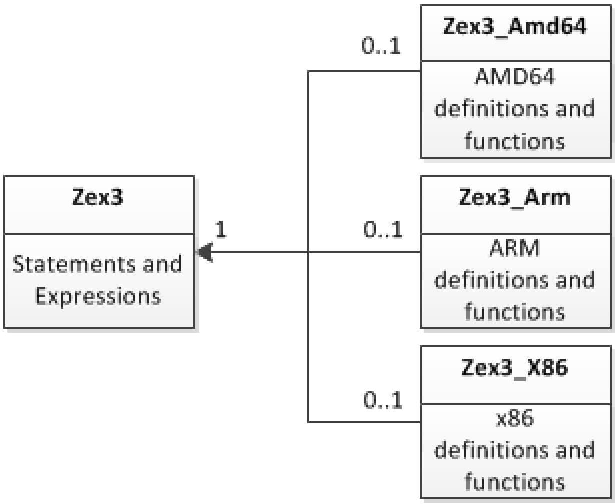

Zex3
Statements and Expressions
1
0..1
Zex3_Amd64
AMD64 definitions and functions
0..1
Zex3_Arm
ARM definitions and functions
0..1
Zex3_X86
x86 definitions and functions

Figure 3.2: An illustration of the interaction between the Zex3 modules.

This way we can create an object of the current architecture and direct all architecture dependent operations through this object. An example for this procedure is the expression Iex_CCall, which creates a Z3Py string containing architecture dependent calls, like shown in Listing 3.4. *zex3* in Listing 3.4 is the name of Zolver3's Zex3 object. The object containing the current architecture object is *arch*.

```
self.zex3.arch.x86g_calculate_condition(self.s, BitVecVal(0x2, 32), R40_i, R44_i, R48_i,
    R52_i)
```

Listing 3.4: Iex_CCall example output.

3.1.3 Unsupported Instructions

An disadvantage of VEX and pyvex is that an unsupported instruction causes a segmentation fault. Due to the segmentation fault, the whole Python 2 environment executing the analysis script crashes. As we can not prevent the crash with a try and except block in Python 2, we sort out the unsupported instructions by working with two processes. One process that deletes the unsupported instructions and one that tests the instructions. This way we can detect unsupported instructions by the crash of the testing process. Figure 3.3 gives an overview of the processes relationship. At first the initial or parent process obtains an ordered list of all opcodes without duplicates. The first element of this list is passed as an argument to a child process. The child process obtains the same opcodes list and starts testing the opcodes. The start point of the tests is always the opcode passed to the process. Before the child process creates a VEX object, it communicates the upcoming opcode to the parent process, so that the parent is aware of the current opcode. If the child processes crashes, an unsupported instruction is encountered. In this case the parent process deletes the opcode from the list. If the last opcode in the list has not been reached, the parent process restarts the child with the opcode after the deleted one as argument. Otherwise the cleansing is finished and all unsupported instructions are deleted.

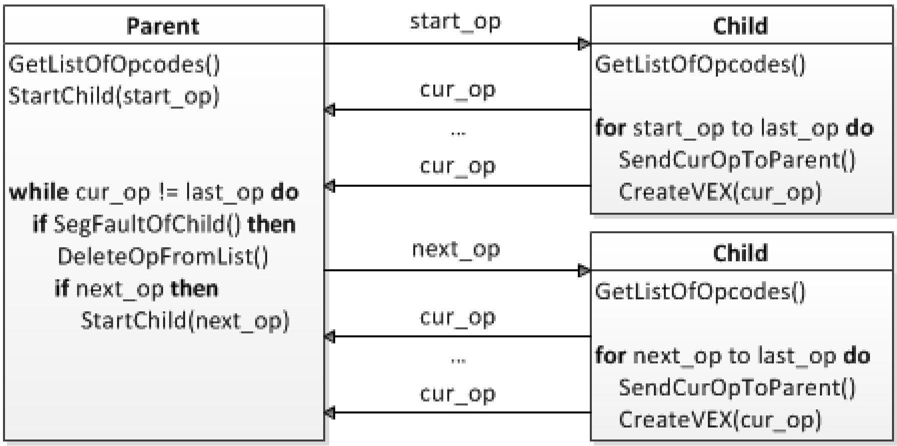


Figure 3.3: Schematic of the processes relationship during opcode deletion of unsupported instructions. The parent keeps track of the current opcode, restarts the child at crashes, and deletes unsupported opcodes. The child reports the upcoming opcode and creates VEX objects.

3.2 Zolver3

The output of Zex3 can not be fed to Z3Py directly. An additional component is needed to wrap Z3Py up in the properties from the corresponding architecture, such as the registers and the memory. We describe this component, called Zolver3, in the following chapter.

3.2.1 Adding an Zex3 Object

Every Zex3 object contains the Z3Py equations equivalent to one VEX IRSB. We use Algorithm 3.2.1 to adjust each equation to Zolver3 and to add restrictions to memory accesses. After all equations have been adjusted they are evaluated.

The algorithm adjusts the generic format of the memory accesses, the registers, and the final equation with the condition to the format of Zolver3. We adjust these properties here and not directly during translation by Zex3, because often information

Algorithm 3.2.1: Algorithm to Add Zex3 Objects to Zolver3

```
Function add_ zex3(zex3, addr)
    Data: zex3 {Zex3 object containing all equations of the IRSB}, addr {address
          of the instruction represented by the IRSB}, mem {Z3 array
          representing the memory space with all write accesses}, regs {structure
          addressing the Z3Py equations by destination register and instruction
          address}
    forall the equations of zex3 do
        dst ← equation.dst
        src ← equation.src
        cond ← equation.cond
        if ContainsMemLoad(equation) then
            adjust the memory load of the src string with string replaces to the
            Zolver3 architecture
        if ContainsCondMemLoad(equation) then
            adjust the memory load of the cond string with string replaces to the
            Zolver3 architecture
        forall the regs of equation.src_regs do
            set the source registers to the last or the one before the last assigned
            register value, depending on if it is a input or output value
        if ContainsMemLoad(equation) then
            set user's memory read constrains on memory address
        forall the regs of equation.cond_regs do
            set the registers of the condition equation to the last or the one before
            the last assigned register value, depending on if it is a input or output
            value
        if ContainsMemStore(equation) then
            forall the regs of equation.cond_regs do
                set the registers of the memory store equation to the last or the one
                before the last assigned register value, depending on if it is a input or
                output value
            set user's memory store constrains on memory address
            set up the memory store out of the memory store equation, src, and
            condition string
            mem ← eval(mem_store)
        else
            if ContainsCond(equation) then
                set up the source string with the condition
            regs[dst][addr] ← eval(src)
```

about the address and values of the previous instruction in Zolver3 are needed. However, these information are not available during the translation process. We work with functions for string-manipulation such as *replace* and format strings to change the equation strings to the desired format. The adjusted equations are evaluated as Python 2 expressions and assigned to either a Z3 array representing the memory space, or to a structure containing all register contents.

3.2.2 Function Calls

To support calls to external functions, like WinAPI functions, Zolver3 has handlers for external functions. Currently we perform the checks, if an instruction contains a call to a external function with information not included in Zolver3 itself. The handler functions receive all important structures, for example the Z3 solver, the memory, and register structures, so that one can implement the functionality of the function in Z3.

3.2.3 Adding a Gadget

The functionality to add gadgets to Zolver3 does not belong to a generic version of a symbolic execution engine, but it is included in the Zolver3 version of this thesis. Besides that, the following algorithm to add gadgets is a good sample algorithm for Zolver3.

Algorithm 3.2.2: Algorithm to Add Gadgets to Zolver3

```
Function add_gadget(gadget)
    Data: gadget {structure containing all information of the gadget, including the
          Z3Py strings for every instruction of every BB}, solver {Z3 solver}
    solver.add(last_set_next_ip == start_addr_of_gadget)
    forall the bbs of gadget do
        forall the instructions of bb do
            if IsFixedFuncCall(instruction) then
                FixedFuncHandler(solver, regs, mem, addresses, func_args)
            else
                add_zex3(instruction.zex3, current_addr)
        if IsNotLastBBInPath(bb) then
            solver.add(last_set_next_ip == start_addr_of_next_bb)
```

We use Algorithm 3.2.2 to add gadgets to Zolver3. At first we connect the previous gadget with the current one by adding the condition to the Z3 solver that the last set instruction pointer must be the first address of the new gadget. If there is no previous gadget, the last set instruction pointer is a variable symbolic value. After the gadgets have been combined, we start adding the current gadget. A gadget can contain multiple BBs, therefore we iterate over all BBs and check for every instruction of each BB, if a fixed call to an external function is performed. If a call to an external function is performed we call the handler of the corresponding function. Otherwise we add the zex3 object of the instruction to Zolver3. The BB themselves are combined like the gadgets, by adding constrains to the solver.

3.2.4 Architecture dependencies

The design of Zolver3 is similar to the one of Zex3 displayed in Figure 3.2. The main module wraps the functionality of Z3Py, for example *push*, *pop*, *check*, and *reset*, and the architecture modules contain memory and register related functionalities. However, the most important architecture dependent functions are also wrapped by the main module, like *init_ regs*.

4 Gadget Search: Models and Algorithms

Simple gadgets are usually found by searching for return opcodes in a given binary. If the gadget finder encounters a return opcode, it disassembles backwards and tries to locate valid instructions. In case a useful operation such as *inc register* is found, the search is terminated. Like stated previously, our gadget definitions are supposed to circumvent CFI and heuristic checks. Therefore, the search has to be different than the search for common gadgets. The following chapter provides an overview of the algorithms used to locate the gadgets. Since our gadgets are more complex, an extra analysis process is performed on each gadget, which is described afterwards. We conclude the chapter with an introduction to the algorithms utilized during the search for specific gadgets.

4.1 Gadget Discovery

Before we can describe the process of the gadget discovery, we have to define the gadgets first. The definition of the gadgets is important insofar as they define the bounds and specify the content of the gadgets. After the definition of the gadgets is given, we introduce the algorithms to locate all points of interest for the gadget discovery and the algorithm to discover the gadgets themselves.

4.1.1 Gadget Categories

Like mentioned in Chapter 2.1.2, our gadgets are a selection with partially small modifications of the gadgets defined by Göktas et al. [22] and Schuster et al. [37]. In the following we describe the gadgets discovered by our gadget search and how they differ from the gadgets of Göktas et al. and Schuster et al..

Our outer gadget bounds are defined like the ones by Göktas et al.. Every gadget has to start with an EP or at a CS and end with an IC, IJ, or RET. These bounds are illustrated by exemplary gadgets in Figure 4.1.
Göktas et al. define the content of a gadget as either an IC, a *fixed function call* (F), or other arbitrary instructions. We decided against an extra category for gadgets

containing an IC. With the ulterior motive in mind to chain the gadgets automatically, we concluded that simplicity of the already complicated gadgets is preferred. Besides that, one can still query if another gadget starts at the CS of the IC from the current gadget or the other way around. Gadget c) in Figure 4.1 shows how a EP-IC-Ret gadget of Göktas et al. gets stored by our definition. We incorporate gadgets containing a fixed function call in our definition. Fixed function calls are beneficial in two ways. Instead of reading the address of the function from ones own *import address table* (IAT) and preparing the call, one can simply use the gadget with the fixed function call. However, this just works if all parameters of the function can be set to the desired values. Furthermore, defenses preventing calls to security sensitive functions [52] can be circumvented by using gadgets containing a legitimate call to the function. A list of all implemented fixed function calls is given in Appendix A.1.

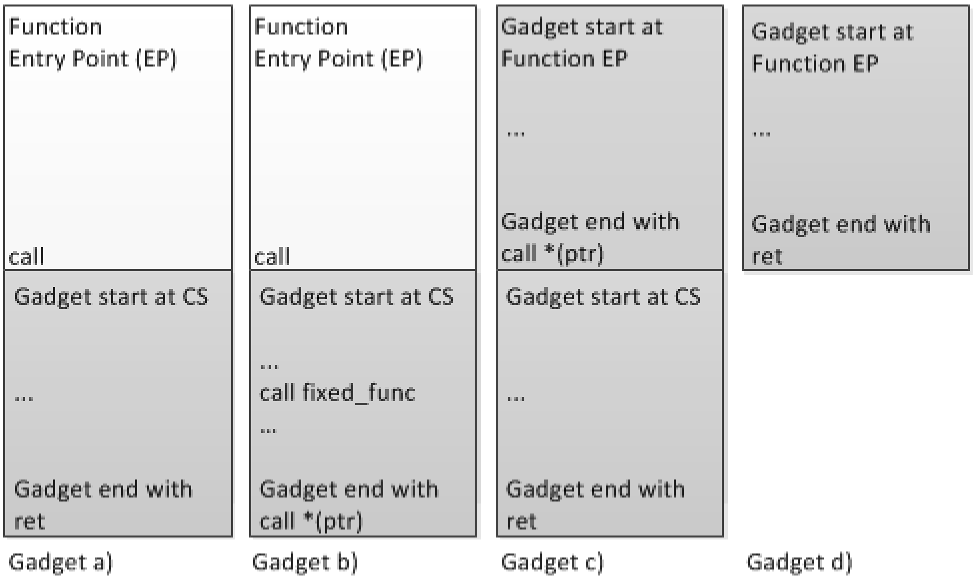
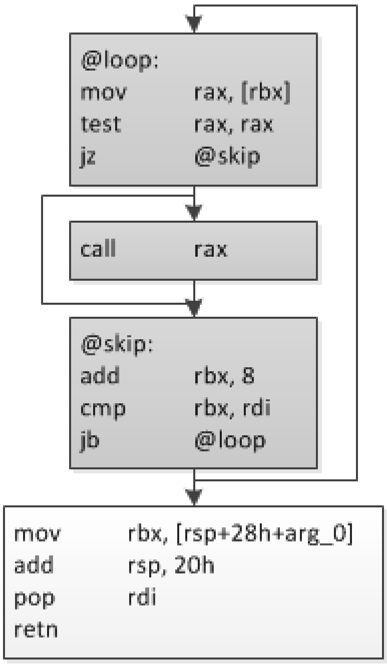


Figure 4.1: These are exemplary gadgets that our gadget finder is able to discover.
Gadget a) is a CS-RET gadget with arbitrary instructions.
Gadget b) is CS-IC gadget and contains a fixed function call.
Gadget c) actually consists of two gadgets. One EP-IC and one CS-RET gadget.
Gadget d) is a EP-RET gadget. The whole function is the gadget.

Another useful gadget is the loop gadget. Loops can be used as a *dispatch gadget* [37, 5] to invoke other gadgets. Figure 4.2 shows a gadget proposed by Schuster et al.. During the first iteration of the loop, rbx points to the beginning of a list with the addresses of the to-be dispatched gadgets. rdi points to the end of this list during all iterations of the loop. If the end of the loop is reached the RET of the gadget is taken. The difference between the proposed gadget and the gadget defined for our search is that just the gray BBs in Figure 4.2 belong to our loop gadget. For simplicity, loop gadgets must end with an IC and start at the CS of its own IC. Hence, the BB beginning with the label *@skip* and the BB containing the RET are an own CS-RET gadget. This has the advantage that also loop gadgets in big functions without a tailing gadget (CS-RET) are found. One can continue from the last gadget by manipulating the stack in the last gadget of the loop to return to another gadget. Another possibility is to invoke a last gadget containing another suffix than RET. Additionally one can query if another gadget starts at the beginning of the loop gadget.



```
@loop:
mov     rax, [rbx]
test    rax, rax
jz      @skip

call    rax

@skip:
add     rbx, 8
cmp     rbx, rdi
jb      @loop

mov     rbx, [rsp+28h+arg_0]
add     rsp, 20h
pop     rdi
retn
```

Figure 4.2: The instructions of an example loop gadget by Schuster et al. transfered to a graph view. Just the gray BBs belong to a loop gadget by our definition.

Since we analyze the path constrains of the gadgets, we can also include gadgets containing conditional jumps. During the search process the path constrains of the gadgets are verified for the current state of the process, again, see Chapter 4.3. However, to keep the gadgets simple every path is a single gadget. To prevent too complicated gadgets, and to reduce the analysis time, the user defined size of the gadgets is limited to 30 instructions by default. Göktas et al. has shown that a gadget size of 30 instructions is sufficient to find enough gadgets for an attack.

All supported gadget definitions from this section are summarized in Tabel 4.1.

Table 4.1: A list of all gadget definitions implemented by our gadget finder.

Prefix	Content	Suffix
EP	Arbitrary Instructions	IC
EP	Arbitrary Instructions	IJ
EP	Arbitrary Instructions	RET
EP	F	IC
EP	F	IJ
EP	F	RET
CS	Arbitrary Instructions	IC
CS	Arbitrary Instructions	IJ
CS	Arbitrary Instructions	RET
CS	F	IC
CS	F	IJ
CS	F	RET
CS	Loop	IC

4.1.2 Discovering Points of Interest

To locate the gadgets, the search algorithm follows the paths of the CFG. But so far we have not collected information such as, starting points in the graph and addresses of fixed function call locations. The starting points for the search algorithm are IC, IJ, and RET instructions. Our algorithm to locate these points of interest, Algorithm 4.1.1, works in two phases. In the first phase the algorithm checks all imported modules for fixed function calls. If a fixed function call is encountered, the address in the import section is cross referenced to find all calls to the fixed function call. All addresses containing a call are stored with the corresponding name of the fixed function call.

During the second phase, the algorithm iterates over every instruction belonging to a function. If an instruction is a RET, IC, IJ, or a call the address of the instruction is added to the corresponding list of addresses. ICs and fixed function calls are excluded from the list of calls, since they are stored in an extra list.

Algorithm 4.1.1: INFORMATION EXTRACTION ALGORITHM

Input: *import_section_modules* {list of all modules in the import section}, *functions* {list of all functions of the binary}, *fixed_func_modules* {list of all modules containing fixed functions}, *fixed_funcs* {list of all fixed functions}

Output: fixed_func_addrs {dictionary with the address of the call as index and the function name as value}, ICs {list of addresses containing a indirect call}, IJs {list of addresses containing a indirect jump}, calls {list of addresses containing a call excluding ICs}, RETs {list of addresses containing a RET}

```
begin
    forall the modules of import_section_modules do
        if module in fixed_func_modules then
            forall the imported_funcs of module do
                if imported_func in fixed_funcs then
                    add all address containing calls to the imported function to the
                    dictionary fixed_func_addrs with the corresponding function
                    name
    forall the funcs of functions do
        forall the instr of func do
            if IsRET(instr) then
                add the address of instr to RETs
            else if IsCall(instr) then
                if IsIndirectCall(instr) then
                    add the address of instr to ICs
                else if addr of instr is in fixed_func_addrs then
                    continue
                else
                    add the address of instr to calls
            else if IsIndirectJmp(instr) then
                add the address of instr to IJs
```

4.1.3 Gadget Extraction with Depth-first Search

To construct the gadgets from Chapter 4.1.1 we have to traverse the CFG of every function in the binary. Because our gadgets are limited to a single path, we can walk every path separately. We start our traversal from the discovered gadget endpoints, namely ICs, IJs, and RETs. We walk every possible path backwards until we discover a gadget starting point (EP and CS), or until we exceed the maximum instruction length of the gadgets. The algorithms we use are a modification of the depth-first search (DFS). DFS traverses a graph by exploring a path as long as possible before visiting other branches.

Algorithm 4.1.2 is used to initiate the search. At first, it tries to find the BB containing the gadget *end_address* passed to the function. Afterwards, we check if there are any calls or fixed function calls between the end_address and the beginning of the BB. If we encounter a call, a CS gadget is created and the algorithm aborts. Before a gadget is added to the gadget list, we always check if a gadget with the same opcode sequence is already in that list. In case there is already a gadget, the current gadget is discarded. Otherwise, we just increase our analysis time later on to perform the same tests on several identical gadgets. If a fixed function call is encountered we store the information of the fixed function call and split the current BB. The first BB for our path starts at the beginning of the original BB and ends at the fixed function call. The second BB starts at the CS of the fixed function call and ends with the gadget end_address. Additionally, we create a CS gadget at the CS of the fixed function call. The next encountered fixed function call is treated as a normal call, a gadget containing a fixed function call gets created, and the search aborted. Afterwards, we check if the current BB contains the EP. In that case, we create a EP gadget and abort the search. We check the number of instructions of the current path in the lines 19 to 21. To traverse all possible paths backwards, we create path information and call Algorithm 4.1.3 in a loop iterating over all direct preceding BBs. To each call of Algorithm 4.1.3 we pass the index of a preceding BB. Algorithm 4.1.3 first checks if the passed BB has already been visited before. If that is the case, a loop gadget is only added, if the current path starts at a CS and ends at a IC. In any case, the algorithm returns if the BB has already been visited. Afterwards, the checks for a call, fixed function call, and EP from Algorithm 4.1.2 are repeated. The number of encountered instructions is updated and checked. Finally, we loop over all directly preceding BBs of the current BB. In each loop we update the path information and recursively call Algorithm 4.1.3.

Algorithm 4.1.2: BFS Gadget Extraction Init Algorithm

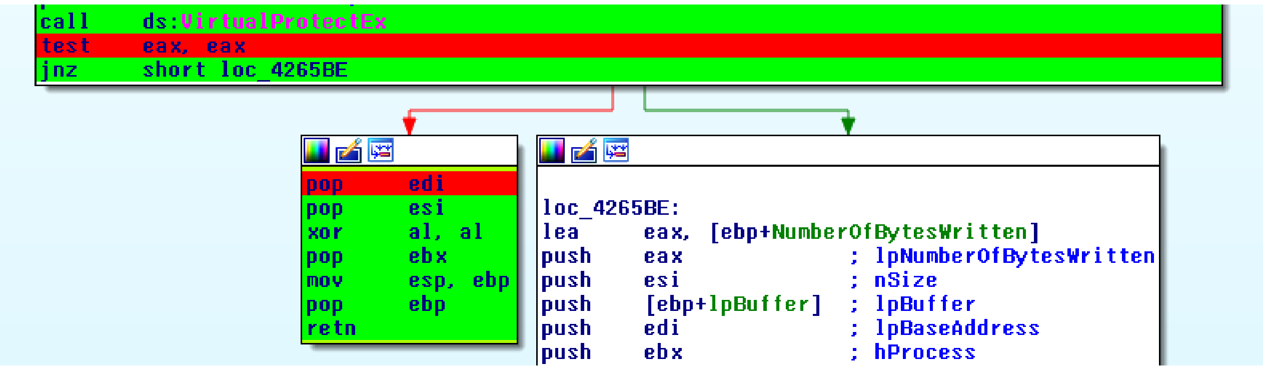
```
Function BFSInit(end_addr, func_graph, gadgets)
    Data: end_addr {IC, IJ, and RET}, func_graph {graph of the function}
    Result: gadgets {structure to store gadgets into, passed to function by
            reference}
    n ← GetBBIndexOfAddr(end_addr, func_graph)
    if n == -1 then
        failed to find the BB index
        return
    for i ∈ end_addr .. bb_start_addr do
        if IsCall(i) then
            store CS gadget to gadgets if no gadget same instructions in gadgets
            return
        if IsF(i) then
            if NoPrevF() then
                create bb object from F CS to end_addr and from bb start to F
                store CS gadget to gadgets if no gadget same instructions in gadgets
            else
                store CS-F gadget to gadgets if no gadget same instructions in
                gadgets
                return
    if BBContainsEP() then
        store EP gadget to gadgets if no gadget same instructions in gadgets return
    instr_depth ← NrOfInstrInBB()
    if CheckIfMaxInstrDepthExceeded(instr_depth) then
        return
    forall the bb_indexes of direct_prev_bbs do
        create path information
        BFSRec(bb_index, path_info, func_graph, gadgets)
```

Algorithm 4.1.3: BFS Gadget Extraction Recursive Algorithm

```
Function BFSRec(bb_index, path_info, func_graph, gadgets)
    Data: bb_index {index of the current BB in func_graph}, path_info
          {information of the taken path}, func_graph {graph of the function}
    Result: gadgets {structure to store gadgets into, passed to function by
            reference}
    if BBAlreadyVisited() then
        if PathStartsAtCSEndsAtIC() then
            store CS IC loop gadget to gadgets if no gadget same instructions in
            gadgets
        return
    for $i \in end\_addr .. bb\_start\_addr$ do
        if IsCall(i) then
            store CS gadget to gadgets if no gadget same instructions in gadgets
            return
        if IsF(i) then
            if NoPrevF() then
                create bb object from F CS to end_addr and from bb start to F
                store CS gadget to gadgets if no gadget same instructions in gadgets
            else
                store CS-F gadget to gadgets if no gadget same instructions in
                gadgets
                return
    if BBContainsEP() then
        store EP gadget to gadgets if no gadget same instructions in gadgets return
    instr_depth ← instr_depth + NrOfInstrInBB()
    if CheckIfMaxInstrDepthExceeded(instr_depth) then
        return
    forall the bb_indexes of direct_prev_bbs do
        update path information
        BFSRec(bb_index, path_info, func_graph, gadgets)
```

4.2 Gadget Analysis

We accomplish two objectives with the gadget analysis. First, we sort out unsatisfiable path constrains, second we match every register output and every memory write to a semantic definition.
At the time of compilation it is unknown to the compiler if a function call succeeds. Therefore, checks for the return value are normally inserted in the calling function. Depending on the return value a different path in the control-flow is taken. We might encounter such checks in gadgets containing a fixed function call. During exploitation we expect the fixed function call to succeed, hence, a gadget depending on a failed fixed function call poses unsatisfiable path constrains. An example for such a unsatisfiable path constrain is given in Figure 4.3. The colored BBs belong to the tested gadget. In case the fixed call to *VirtualProtectEx* succeeds, the return value in eax is non-zero. However, if the return value is non-zero the jump to *loc_4265BE* is taken and the BB belonging to the gadget is not reached. We use the Z3 wrapper Zolver3 to symbolically execute the gadget. If the return value is implemented for the tested fixed function call, a simple call to *check* is sufficient to check if the tested gadget is satisfiable. In the case of an unsatisfiable gadget, like in Figure 4.3, the analysis process is aborted.



```
call    ds:VirtualProtectEx
test    eax, eax
jnz     short loc_4265BE
```

```
pop     edi
pop     esi
xor     al, al
pop     ebx
mov     esp, ebp
pop     ebp
retn
```

```
loc_4265BE:
lea     eax, [ebp+NumberOfBytesWritten]
push    eax                 ; lpNumberOfBytesWritten
push    esi                 ; nSize
push    [ebp+lpBuffer]      ; lpBuffer
push    edi                 ; lpBaseAddress
push    ebx                 ; hProcess
```

Figure 4.3: An example for a fixed function call gadget with unsatisfiable path constrains.

With the current level of information one is only able to search through the discovered gadgets based on their boundaries, without any knowledge of their effects on the state of the process. This makes an efficient search to chain the gadgets impossible. Therefore, the second objective is to match every register output and every memory write to a semantic definition. An high level overview of this process is given in Figure 4.4. We use the results of the symbolic execution of the gadget from the first objective. Zolver3 provides the state of every register and every memory write based on the, in our case symbolic/variable, input values of the registers and memory. We do not have to trace every instruction of the gadget ourself, but we can treat the gadget as a black box. We send symbolic input values in and get all modifications to the global state of the process by the gadget based on these symbolic input values. This means that all register and memory store output values are expressions of the register input values. We can use these expressions to apply our semantic definitions to the gadgets. The process of applying the semantic definitions to the output equations is explained in Chapter 4.2.1.

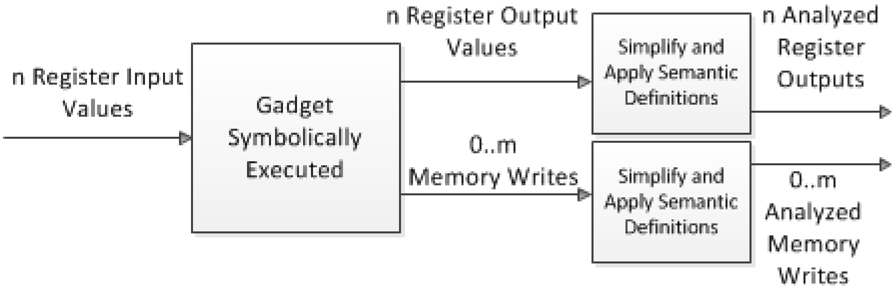

n Register Input Values
Gadget Symbolically Executed
n Register Output Values
0..m Memory Writes
Simplify and Apply Semantic Definitions
Simplify and Apply Semantic Definitions
n Analyzed Register Outputs
0..m Analyzed Memory Writes

Figure 4.4: High level overview of the gadget analysis process. n is dependent of the number of registers of the current architecture. m is the number of performed memory writes by the gadgets. There can be no memory writes at all during the execution of the gadget.

4.2.1 Semantic Definitions

In the following chapter, we present our semantic definitions. These definitions allow the exploit developer, combined with the search presented in Chapter 4.3, to search gadgets with specific operations performed on a specific register or memory address. Afterwards we explain the algorithm to apply the definitions to the gadgets.
At code-reuse attacks, the defined gadget types are the available instruction set for the attacker. Therefore, the gadget definitions must cover all necessary instructions to perform arbitrary computations. The following gadget types are necessary to accomplish this:

1. MovReg: A gadget to move the content of one register to another.
2. LoadReg: A gadget to load a specific content into a register.
3. Arithmetic: A gadget to perform arithmetic operations between registers.
4. LoadMem: A gadget to load the content of a specified memory area into a register.
5. StoreMem: A gadget to store the content of a register to a specified memory area.

With these gadgets even jumps and conditional jumps are possible. ROP uses the stack pointer to load the next instruction. Hence, an addition to or subtraction from the stack pointer changes the next instruction. This way the exploit writer can jump through his ROP-code. JOP and COP often use a dispatcher gadget, like the loop gadget, to invoke the gadgets of the chain. During the loop iterations, one register holds a pointer into the buffer with the next gadgets. Instead of the stack pointer, like at ROP, the register holding the pointer to the buffer has to be modified for jumps. Conditional jumps, however, are more complicated. They have to be accomplished by chaining several arithmetic operations [17]. But the study of exploits [33] reveals, that jumping by manipulating the stack pointer is rarely used. Normally the chains just set the shellcode to executable and redirect the control-flow to the beginning of the shellcode. Snow et al. [40] come to a similar conclusion regarding the gadget definitions in their research.

We expand the list of necessary definitions by four more semantic definitions. We add the four semantic definitions because they represent operations which are commonly found in our gadgets. Alternatives to extending the gadget definitions are discussed in Chapter 7.5, but they increase the complexity of the algorithms which match the definitions to the output equations and of the search itself. In the following paragraphs the semantic definitions are introduced. The algorithm which applies the definitions to the gadgets gets explained in the last paragraph.

MovReg

The MovReg simply moves the content of the source register to the destination register. It is useful if an operation must be performed on a register, but no gadget with the desired operation is found for this register. But, if a gadget with the operation is found for another register, the content of the original register can be moved the new register by a MovReg gadget.

The Z3 expression tree of the source, see Figure 4.5, contains just an input register, with the condition that this source and destination register differ.

Figure 4.5: Z3 expression tree of the semantic definition MovReg.

LoadReg

The LoadReg gadget loads a constant value to the destination register. Three destinations are possible. The first possibility is that the value gets loaded from the stack. This is normally achieved by a *pop* instruction, see Listing 4.1 line 3. A pop instruction loads the value from the top of the stack to the destination register and decreases the stack pointer. The second and third method are in regards to the Z3's expression tree the same, after the source equation has been simplified by Z3. For example the instruction in Listing 4.1 line 1 loads the value 0 into the register eax by a direct *mov* instruction. The instruction in line 2 performs the *xor* operation with the same register as both operands, hence the result of the arithmetic operation is 0. The simplify method of Z3 reduces the calculation directly to the result and sets ebx to 0.

```
1 mov eax, 0
2 xor ebx, ebx
3 pop ecx
4 retn
```

Listing 4.1: A x86 assembler (Intel syntax) snippet with three examples to load constant values to registers.

For the LoadReg definition two expression trees are applied. The first expression tree, displayed in Figure 4.6, contains just a BitVecVal. BitVecVals are Z3's representation of BitVecs initialized to constant values. This tree is used for equations, simplified to a constant BitVec, and direct assignments of BitVecVals.

Figure 4.6: One of the two Z3 expression trees of the semantic definition LoadReg.

The second expression tree is the same as the one used for LoadMem, Figure 4.8, with the additional condition that the selected address has to be the content of the stack pointer at one point during the execution of the gadget.

Arithmetic

The Arithmetic definition is limited to those equations that can not be solved during the analysis. If the result is known and therefore constant, the equation is a LoadReg. Usually, the operands of operations with an unknown result are two different registers or a register and a constant value, see Figure 4.7. Table 4.2 contains all arithmetic operations of the Arithmetic definition.

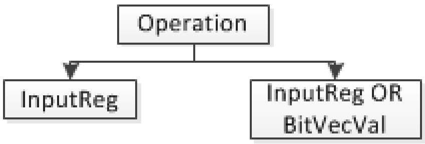


Figure 4.7: Z3 expression tree of the semantic definition Arithmetic.

LoadMem

LoadMem is used to load the value from a specific memory address into a register. For simplicity, the size of the loaded value must be the same as the bit size of the register. Otherwise various expressions for the extension and extraction of the selected

Table 4.2: A list of all arithmetic operations.

a + b	addition of a and b
a − b	subtraction of b from a
a ∗ b	multiplication of a and b
a / b	division of a through b
a | b	bitwise or of a and b
a & b	bitwise and of a and b
a ˆ b	bitwise xor of a and b
a << b	left shift of a by b bits
a >> b	right shift of a by b bits

values also have to be considered. Values loaded from the stack are excluded by this definition, as they are already defined by LoadReg.

Figure 4.8 shows the expression tree of a LoadMem. To read a value from an array in Z3, the *SELECT* declaration is used. The address can either consist of a single register, a BitVecVal, or an arithmetic operation. If the address consists of a BitVecVal the LoadMem can just statically read from one address. However, if there exists a LoadMem with a BitVecVal for the desired address, we do not have to prepare a register with the loading address ourself.

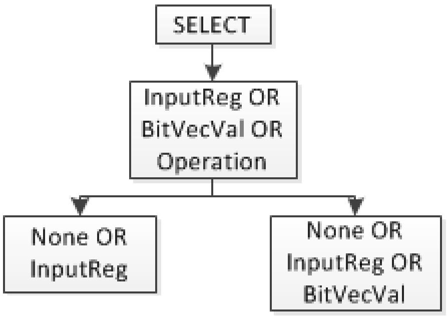

SELECT
InputReg OR BitVecVal OR Operation
None OR InputReg
None OR InputReg OR BitVecVal

Figure 4.8: Z3 expression tree of the semantic definition LoadMem.

ArithmeticLoad

The ArithmeticLoad loads the value from a specified memory address, performs an arithmetic operation on it, and stores the result to the destination register. To represent this definition in a Z3 expression tree, we expand the LoadMem tree by an arithmetic operation, see Figure 4.9. This operation is useful on architectures like x86 without distinct operations for memory loads. For example, on x86 the source register of an arithmetic operation can be a memory location. Listing 4.2 shows a possible ArithmeticLoad on x86 with an addition. The value at the address *esi+4* is loaded from memory, added to the value contained in eax, and stored to eax. But this operation has also a use-case on architectures with distinct memory load/store operations. On these architectures this definition is equivalent to a distinct memory load followed by an arithmetic operation on the loaded value.

```
add     eax, [esi+4]
retn
```

Listing 4.2: An ArithmeticLoad example in x86 assembler (Intel syntax).

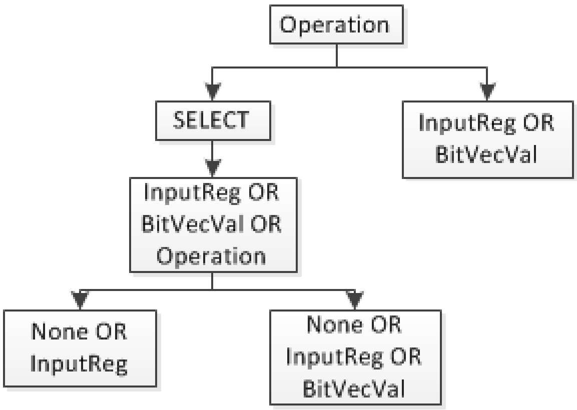

Operation
SELECT
InputReg OR BitVecVal
InputReg OR BitVecVal OR Operation
None OR InputReg
None OR InputReg OR BitVecVal

Figure 4.9: Z3 expression tree of the semantic definition ArithmeticLoad.

StoreMem

StoreMem is one of two definitions with not a register, but a memory address as destination. It stores either the content of a register, a BitVecVal, or the value of a LoadMem to the specified memory address. The definition for the address is the same as for the LoadMem. Figure 4.10 shows the Z3 expression tree of StoreMems. The top *STORE* operation has two branches, one for the address equation and one for the stored value.

ArithmeticStore

ArithmeticStore adds an arithmetic operation to StoreMem. The use-case in mind is similar to the one of ArithmeticLoad. To illustrate this, the example from ArithmeticLoad, Listing 4.2, is changed to Listing 4.3. The instruction in the example is the same, except that the destination is the memory location, now. At first the

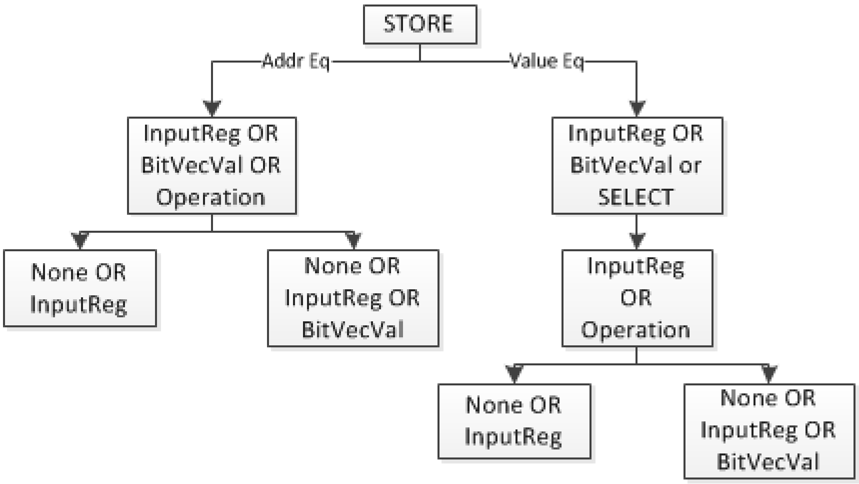

STORE
Addr Eq
Value Eq
InputReg OR BitVecVal OR Operation
InputReg OR BitVecVal or SELECT
None OR InputReg
None OR InputReg OR BitVecVal
InputReg OR Operation
None OR InputReg
None OR InputReg OR BitVecVal

Figure 4.10: Z3 expression tree of the semantic definition StoreMem.

instruction loads the value at *esi+4*, adds eax to it, and writes the result back to *esi+4*.
Due to the current implementation of the algorithm, the load of the address equation and value equation are evaluated separately, see Figure 4.11. However, this just broadens the definition of the ArithmeticStore operation. Additionally, input registers are also allowed as operands, instead of the memory select.

```
add     [esi+4], eax
retn
```

Listing 4.3: An ArithmeticStore example in x86 assembler (Intel syntax).

NOP - No Operation

If the input value of the register is also the output value after the execution of the gadget, the gadget performs no operation to this register. This is very useful during the gadget search, because registers that are already filled with a specific value can be marked as static.

The expression tree of the NOP definition is the same as the one for MovReg, Figure 4.5. However, the condition for the source input register is different. For MovReg the source register has to be different than the currently checked destination register. For NOP the source and currently checked destination register have to be the same.

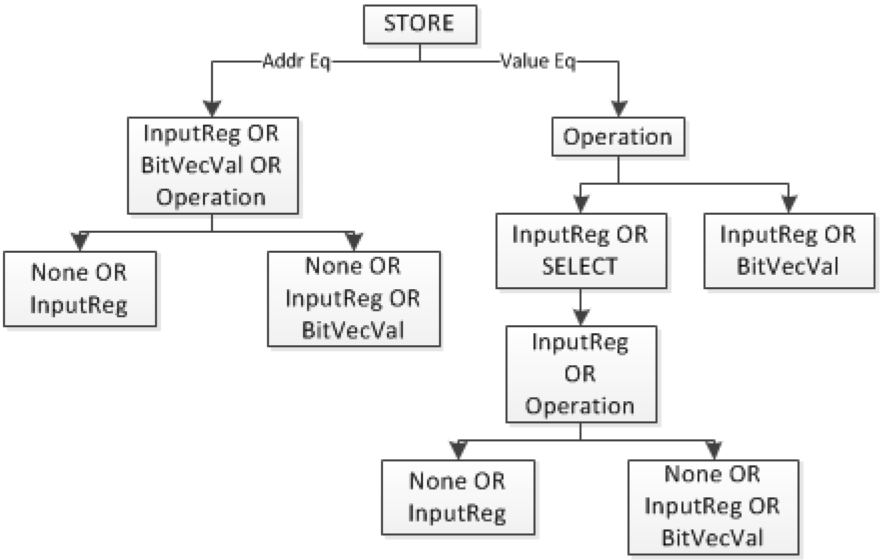


Figure 4.11: Z3 expression tree of the semantic definition ArithmeticStore.

Undefined

If none of the previous semantic definitions match the equation of the register, the register gets marked as undefined. One is advised to inspect the operations of registers marked as Undefined manually.

Applying The Definitions

At the end of the symbolic execution, we have an output equation for every register and memory write. These equations consists of expression trees, like our definitions. We use two algorithms to apply the definitions to the output equations. Algorithm 4.2.1 matches the definitions with a register as destination to the output equations and Algorithm 4.2.2 matches the memory write equations.

Our gadgets vary from one instruction to 30 or more instructions. Therefore, it is hard to limit the whole gadget to one atomic operation like commonly done for classical gadgets. We take the approach to apply our definitions to every register and get as many operations for every gadget, as the architecture has registers. To apply the definitions to every register we loop over all equations belonging to classifiable registers and perform checks if the definitions match. Classifiable registers are the general purpose registers of the architecture plus the instruction pointer. These are the registers one can usually access. The flags are not classified or watched, since other algorithms take care of them, see Chapter 4.3. The first step is to get the simplified equation with a call to Z3's *simplify* method, see Chapter 2.3.2. Afterwards we compare the input and output equations of the register. If they match the register performs no operation and is classified as NOP. The following checks are split into two groups, depending of the number of children of the top node. This way we can reduce the number of performed checks during the classification. We use the methods *children* and *decl* to compare the expression tree of the output equation and the definition. The use of these two functions is explained in Chapter 2.3.1.

Algorithm 4.2.1: Register Classification Algorithm

Input: *classifiable_regs* {equations of all classifiable registers of the current architecture}

Output: {list of register classifications}

```
begin
    forall the regs of classifiable_regs do
        reg_output_simpl ← simplify(reg_output)
        if reg_output_simpl == reg_input then
            save the register as NOP
        else if reg_output_simpl.children() == 0 then
            if IsMovReg(reg_output_simpl) then
                save the register as MovReg
            else if IsLoadReg(reg_output_simpl) then
                save the register as LoadReg
            else
                save the register as Undefined
        else if reg_output_simpl.children() == 2 then
            if ContainsArithmetic(reg_output_simpl) then
                if IsArithmeticLoad(reg_output_simpl) then
                    save the register as ArithmeticLoad
                else if IsArithmetic(reg_output_simpl) then
                    save the register as Arithmetic
                else
                    save the register as Undefined
            else if ContainsMemLoad(reg_output_simpl) then
                if IsMemLoadFromStack(reg_output_simpl) then
                    save the register as LoadReg
                else if IsLoadMem(reg_output_simpl) then
                    save the register as LoadMem
                else
                    save the register as Undefined
            else
                save the register as Undefined
        else
            save the register as Undefined
```

We use the recursive Algorithm 4.2.2 to match the definitions to the performed memory writes. Every new memory store added to the original memory array adds a new layer consisting of a Z3 store operations. This algorithm peels each layer off. To classify the current store operation, the memory address is checked first. Only if the address equation matches the definition of a StoreMem/ArithmeticStore, the value equation is checked. At the end *MemStoreClassification* calls itself without the current store operation to peel the next layer off the memory array.

Algorithm 4.2.2: Memory Store Classification Algorithm

```
Function MemStoreClassification(stores, memory)
    Data: memory {simplified Z3 array representing the memory space with all
          write accesses}
    Result: stores {dictionary of all memory store addresses with the
            corresponding classification, passed to function by reference}
    if IsStore(memory) then
        if IsValidStoreAddr(memory) then
            if IsValidStoreValue(memory) then
                add memory store as StoreMem to stores
            else if IsValidArithmeticStoreValue(memory) then
                add memory store as ArithmeticStore to stores
        MemStoreClassification(stores, content of memory without current store)
```

4.3 Semantic Search

In the previous steps the gadgets have been discovered by their bounds and we have analyzed every effect of the gadgets on the global state of the process. As we want the search for the gadgets to be flexible, we perform the search on register and memory write basis. One can specify only the type of a single register or the types, operations, and operands of many registers. Naturally a search with just the type of a single register specified yields a lot of potential gadget candidates. In the following chapter we define algorithms to order the gadget candidates and sort unsatisfiable gadgets out.

4.3.1 Complexity Ordering

We have to present the simplest gadgets first upon a search to speed up the process of the gadget chaining. To provide the gadgets in a decreasing complexity order, we apply four criteria.
The first criteria is that the gadgets with the lowest instruction count are presented first. Gadgets with a low instruction count are usually simple, as they do not perform many operations to increase the complexity. Our next priority is for the gadgets to contain the least amount of memory writes. For every unnecessary memory write one has to ensure that the write address is inside a writable memory area. This requirement can interfere with already set up registers. Following to the memory writes comes the priority to contain the least amount of memory reads in the gadgets. The reason is the same as for the memory writes, however, readable memory areas are typically encountered more often and therefore easier to set up. Our last ordering criteria is that as many registers as possible should contain NOP definitions. The algorithm implementing the described ordering of the search is shown in Algorithm 4.3.1

Algorithm 4.3.1: Gadget Complexity Ordering Algorithm

```
Input: search_criteria {user defined search criteria}
Output: gadgets {list of ordered gadgets}
begin
    gadgets ← GetGadgetsWith(search_criteria)
    gadgets ← OrderByInstrCountAsc(gadgets)
    gadgets ← SubOrderByMemStoresAsc(gadgets)
    gadgets ← SubOrderByMemLoadsAsc(gadgets)
    gadgets ← SubOrderByNOPCountDesc(gadgets)
```

4.3.2 Gadget Verification

Our gadgets support paths containing conditional branches. The exact analysis of the conditions can be tricky. For example, one wants to load the value 0x12345678 from a specific memory address into a register. Algorithm 4.3.1 returns a gadget list with a LoadMem operation at the first position that contains a conditional jump. The pitfall is that the jump is only taken, if the LoadMem loads a NULL value. This renders the gadget useless to load the value 0x12345678. Therefore, invalid gadgets like the described one have to be sorted out. We use Algorithm 4.3.2 to check the constrains of the gadget list with Zolver3 until a satisfiable gadget is encountered.

Algorithm 4.3.2: Verify Gadget Algorithm

```
Input: reg_constrains {user defined register constrains}, gadgets {list of ordered
       gadgets}
Output: valid_gadget {first gadget fulfilling reg_constrains}
begin
    foreach gadget in gadgets do
        zolver3 ← new Zolver3()
        zolver3.AddGadget(gadget)
        zolver3.SetConstrains(reg_constrains)
        if zolver3.IsSat() then
            valid_gadget ← gadget
            break
```

5 Implementation

Our implementation consists of 6537 lines of Python 2 code. One interacts with an IDA Pro plugin for the gadget extraction, a Python 2 script to analyze the gadgets, and finally with a Python 2 class to search for specific gadgets by semantic definitions. We start this chapter with an overview of the components and how they are connected, before we look at each component in greater depth.

5.1 Overview

In this section we give a rough overview of the files involved in our implementation and how they interact. This way one has a grasp of the big picture of our implementation before we present the details of the different components in the following sections of this chapter. What follows is a short summary of the files contained in our implementation:

- db.py - Contains a class for all database interactions of our implementation.
- gf.py - Contains a class for the gadget discovery and classes to store the elements of the gadgets, such as a class for the gadget itself, for BBs, and for the instructions contained in the BBs.
- gf_crit_func.py - Contains everything related to support critical function calls, like WinAPI functions, with our gadget search and Zolver3.
- gfTestPlugin.py - The IDA Pro plugin to discover and store the gadgets.
- pygadb.py - Is used to search the gadgets by semantic definitions.
- pyzex3.py - The file containing the architecture independent parts of our VEX to Z3Py translation.
- pyzex3_amd64.py - The AMD64 related definitions and functions of our VEX to Z3Py translation.
- pyzex3_arm.py - All ARM related definitions and functions of our VEX to Z3Py translation.
- pyzex3_dumper.py - A script to sort out invalid instructions and initiate the analysis of the discovered gadgets.

- pyzex3_dumper2.py - The script to analyze the discovered gadgets.
- pyzex3_x86.py - x86 related functions and definitions of our VEX to Z3Py translation.
- pyzolver3.py - The architecture independent parts of our execution engine.
- pyzolver3_amd64.py - The AMD64 related parts of Zolver3.
- pyzolver3_arm.py - ARM related parts of our execution engine.
- pyzolver3_x86.py - The x86 related parts of our execution engine.

In Figure 5.1 we illustrate the relationship between these files. One can see upon inspection of Figure 5.1 that *db.py* plays a central role. The plugin uses it to store the discovered gadgets, the analysis script uses it to load the gadgets and store the analysis results back to the database, and the search class uses it to perform the semantic search on the gadgets. Furthermore, the analysis script also uses it to maintain a list of all encountered unsupported instructions.

For the gadget discovery itself, the plugin uses *gf.py*. We separated the gadget discovery classes from the plugin-file so that the classes can be utilized by either a IDA Pro plugin, or by a script from the command line outside of IDA with idalink [50]. idalink is a Python 2 tool to use the IDA API outside of the IDA interface. However, our current implementation contains only a IDA Pro plugin. To check if modules and functions of the import section are considered critical, gf.py checks if they are included in the critical functions and modules lists of *gf_ crit_func.py*, see Algorithm 4.1.1.

After *pyzex3_ dumper.py* sorted out all already encountered unsupported instructions from the current gadget database, it starts itself as a child process to sort out all new unsupported instructions, refer to Figure 3.3 for an illustration of this procedure. Afterwards, pyzex3_dumper.py starts the script *pyzex3_ dumper2.py* as a child process to analyze the remaining gadgets. We perform the analysis of the gadgets also in a child process to evade the effects of memory leaks emerging from the statical use of VEX with pyvex. For the analysis pyzex3_dumper2.py creates Zex3 objects and passes them to a Zolver3 object for symbolic execution. *pyzolver3.py* utilizes the critical function handler from gf_crit_func.py to account for calls to external functions and *pyzex3.py* for register sizes and calls to architecture dependent functions.

To search gadgets a function for each semantic definition is implemented in *pygadb.py*. With these functions one can search for specific gadgets and receives a list of sorted gadget candidates. To verify that the gadgets are valid for the currently set register and memory state we use pyzolver3.py. Additionally, one can set conditions for registers which are also considered during the verification of the gadgets by Zolver3.

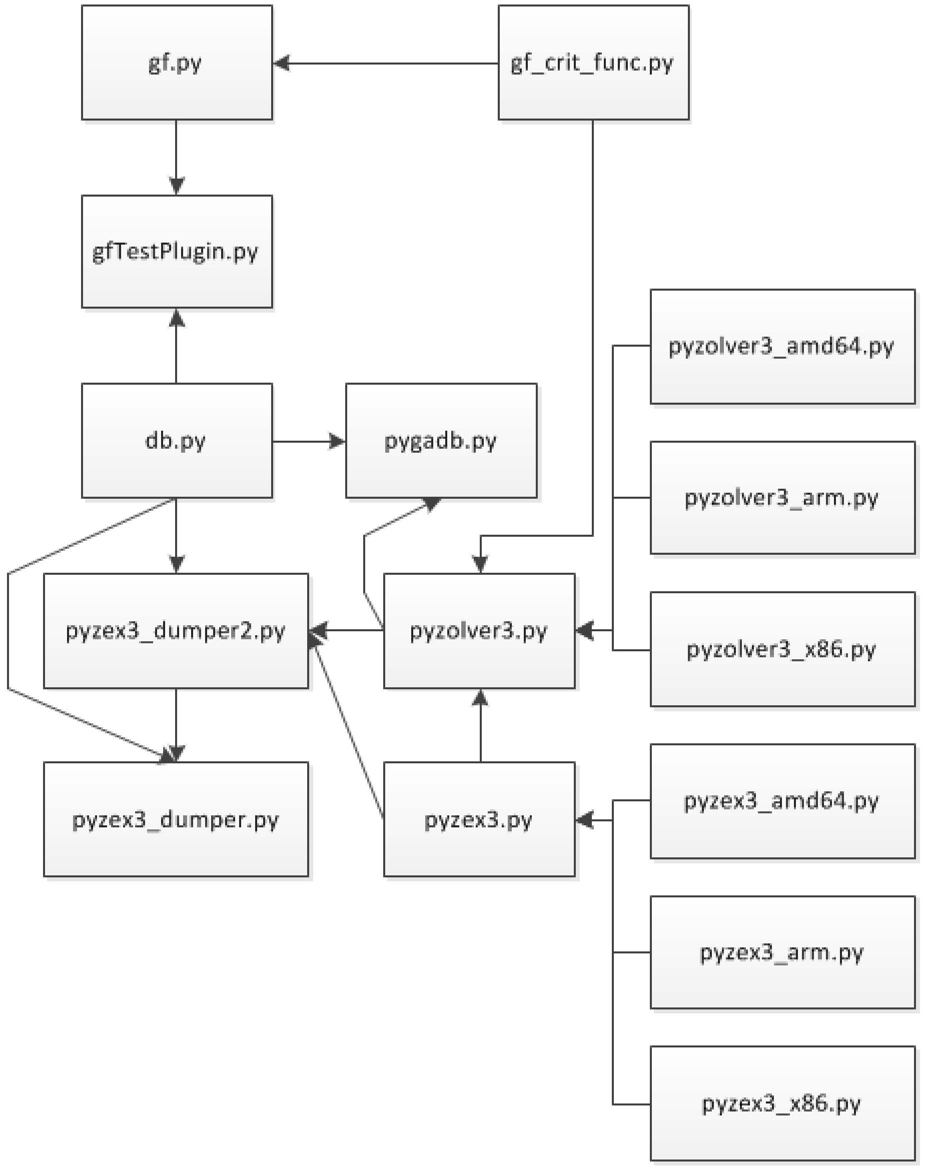


Figure 5.1: Overview of the relationship between the components of our implementation. Each arrow indicates an inclusion of a file. To be exact, the file the arrow head points to includes the file the arrow head points away from.

5.2 Gadget Discovery

To discover the gadgets we use an IDA Pro plugin. The use of IDA Pro has several advantages, such as easy detection of the target architecture of the binary, no need to parse the binary format ourself, and a supplied CFG for aligned instructions. In the following, we describe the implementation of our plugin from architecture detection, to the gadget search, and finally how the gadgets are stored.

5.2.1 Architecture Detection

During the initialization of the plugin we detect the architecture of the binary with the code in Listing 5.1. We just load the plugin if a supported architecture is detected. The detected architecture is passed as an argument to the class *gfTestPlugin*, which provides the functionalities of our plugin. The architecture is important during the gadget search to determine which critical functions and modules lists should be used during the search.

```
if idaapi.ph_get_id() == idaapi.PLFM_386:
    # Check if already initialized
    if not ’_gfTestPlugin’ in globals():
        if idaapi.ph.flag & idaapi.PR_USE64:
            _gfTestPlugin = gfTestPlugin("AMD64")
        else:
            _gfTestPlugin = gfTestPlugin("X86")
        return idaapi.PLUGIN_KEEP
elif idaapi.ph_get_id() == idaapi.PLFM_ARM:
    if not ’_gfTestPlugin’ in globals():
        _gfTestPlugin = gfTestPlugin("ARM")
    return idaapi.PLUGIN_KEEP
else:
    return idaapi.PLUGIN_SKIP
```

Listing 5.1: Architecture detection of our IDA Pro plugin. We just load the plugin if a supported architecture is detected.

5.2.2 Searching the Gadgets

When our plugin initiates the gadget search, the first step is to gather information, like indirect call, indirect jump, and return addresses. For this purpose we introduced Algorithm 4.1.1. However, during the introduction we neglected the collection of information on the CFG of the functions. The reason is that IDA Pro has already created graphs for its discovered functions. And since we only iterate over functions which are discovered by IDA Pro, we expect the graph to be created. Listing 5.2 shows the code we use to receive IDA Pro’s graph of a function.

```
f = idaapi.get_func(func)

if not f:
    continue

fc = idaapi.qflow_chart_t("", f, idaapi.BADADDR, idaapi.BADADDR, idaapi.FC_PREDS)
```

Listing 5.2: IDAPython code to receive information on a function and its CFG by supplying a address contained in the function. The code snipet is from a loop iterating over all functions of the binary.

The checks for call, IJ, and return instructions from Algorithm 4.1.1 can be performed architecture independently by IDA API functions such as *is_ call_ insn*. Unfortunately, there are no IDA API functions to check for indirect calls. To avoid architecture dependencies by checking for specific mnemonics, we resort to check if IDA Pro detects a call instruction and then check for specific operand type flags. The code to check for indirect calls can be viewed in Listing 5.3.

```
def __is_indirect_call_insn(self, addr):
    """ Checks if it's a
        call and
        o_phrase (call [reg]) or
        o_displ (call [reg+offset]) or
        o_reg (call reg).
    """

    if idaapi.is_call_insn(addr):

        cmd = idautils.DecodeInstruction(addr)

        if cmd == None:
            return False

        if (cmd.Op1.type == idaapi.o_phrase or cmd.Op1.type == idaapi.o_displ
            or cmd.Op1.type == idaapi.o_reg):
            return True

    return False
```

Listing 5.3: IDAPython code to detect if an instruction contains an indirect call.

To initiate the actual search process, we extend the loop which iterates over all functions in Algorithm 4.1.1 by a call to a gadget extraction function after all relevant information on the currently parsed function have been extracted. This function calls Algorithm 4.1.2 for all IC, IJ, and RET gadget endpoints.

5.2.3 Storing the Gadgets

To store the gadgets, our plugin creates a SQLite database with the same name as the IDB file of the binary. Therefore, a IDB file has to be created before our plugin is started from the IDA Pro interface. If a database file is already present, our plugin performs no operation. We open the connection to the database in gfTestPlugin.py and pass the connection to the gadget search function in gf.py. The search function then stores every discovered gadget directly to the database. An alternative approach is to keep a list of all gadgets and store them to the database after a certain threshold of discovered gadgets has been reached or after all gadgets have been discovered. However, the disadvantage of the alternative approach is that Python 2 consumes more memory to keep all gadgets in a list than storing the gadgets directly to the database and committing them to the database after a certain threshold of stored gadgets is reached. Since IDA Pro is bound to a 32 bit process space there is a fair chance that we run out of memory with the alternative approach if we discover a few hundred-thousand gadgets.

5.3 Zex3 and Zolver3

In the following section we discuss the implementation of our handler functions incorporated in Zex3 and Zolver3. We also discuss the coverage of VEX's statements, expressions, and architecture dependent functions implemented in Zex3.

5.3.1 Handler Functions

Our handler functions are based on the implementation of vexWrapper.py by R. Gawlik. The idea of vexWrapper.py is to use a *new style class* for classes requiring handler functions. A new style class in Python 2 is a class which inherits from the *object* class. These classes store their methods internally as functions in a dictionary. To invoke entries of this dictionary we can simply call the method *__getattribute__* and provide the desired function name as an argument to __getattribute__. The function *handle_Ist* in Listing 5.4 corresponds to the function *CallHandlerByTag* in Algorithm 3.1.1. One can see that Zex3 inherits from the object class and therefore is a new style class. st is a VEX statement object. The tag field of statements consists of a string containing the statement's name, for example *Ist_Put*. By naming the handler methods identical to the string of the tag fields we can invoke the statement handler methods by supplying the tag as argument to a call to __getattribute__. We also apply this procedure to expressions and their tags. Furthermore, we use __getattribute__ to invoke the handler functions of Zolver3's external function calls. For a more detailed explanation of __getattribute__ refer to the *Descriptor HowTo Guide* [30].

```
class Zex3(object):

   def handle_Ist(self, st):
      self.__getattribute__(st.tag)(st)
```

Listing 5.4: Implementation of Zex3's statement handler function.

5.3.2 Zex3 Functions

Currently we have not implemented all statements and expressions of VEX for our Zex3 implementation. We give an overview of the statements' and expressions' implementation status in Table 5.1. Handler functions without an implementation just raise an exception, stating that the statement or expression is not implemented. Two handlers need further explanation, namely the one for Ist_CAS and Iex_ALLop. The Ist_CAS statement is separated in a single-element and double-element case.

Table 5.1: All statements and expressions of VEX and their implementation status of Zex3.

Statements and Expressions	Is Implemented
Ist_Put	✓
Ist_PutI	✗
Ist_WrTmp	✓
Ist_Store	✓
Ist_LoadG	✓
Ist_StoreG	✓
Ist_CAS	✓[a]
Ist_LLSC	✗
Ist_Dirty	✗
Ist_MBE	✗
Ist_Exit	✓
Iex_Get	✓
Iex_GetI	✗
Iex_RdTmp	✓
Iex_ALLop	✓[a]
Iex_Load	✓
Iex_Const	✓
Iex_ITE	✓
Iex_CCall	✓
Iex_VECRET	✗
Iex_BBPTR	✗

[a] Just partially implemented.

The difference is that the double-element case compares and swaps a value which is separated in a high and low element and the single-element case compares and swaps just a single element. Our current implementation only supports the single-element case. However, we have not encountered a double-element case during our evaluation. The other handler, Iex_ALLop, implements Algorithm 3.1.5. Several operations are incorporated in this handler, such as Iex_Binop and Iex_Unop. The completeness of the implementation, however, does not depend on Algorithm 3.1.5 itself, but on the dictionary which is used to retrieve the format strings of the operations. Currently, our handlers to load and store values from registers and memory just cover integer values. Therefore, our format string dictionary leaves strings for instructions which cover for example floating point operations out.

Our implementation of the x86 and AMD64 dependent functions currently comprises the functions to calculate flags and conditions. Due to the complexity of the two architectures, these are just 3 out of 15 (x86) or 3 out of 21 (AMD64) functions, which we can call from Iex_CCall. Additionally, 15 underlying functions for x86 and 19 for AMD64 are implemented to calculate the flags and conditions. All functions are a direct translation from Valgrind's C implementation to a mixture of Python 2 and Z3Py. Even assertions are implemented by adding constrains to Z3Py's solver. During evaluation of our translation, we call the architecture dependent functions from the Z3Py context. Therefore, we have to stay in the Z3Py context for all non-constant variables. Otherwise Z3 sets the variables to a fixed value which most likely leads to wrong results and therefore unsatisfiable constrains.

For the ARM architecture, VEX contains just 5 helper functions which are callable from Iex_CCall. All 5 functions are implemented by Zex3.

5.4 Gadget Analysis

For the gadget analysis we use the script pyzex3_dumper.py outside of IDA Pro in a 64 bit Python 2 environment. We have to use a 64 bit environment to properly support binaries for 64 bit architectures, such as AMD64, with Valgrind. One supplies the database filename and the VEX architecture to the script to initiate the analysis. Then, the script sorts out all unsupported instructions and analyzes the remaining gadgets.

5.4.1 Unsupported Instructions

We divide the deletion of unsupported instructions in two phases. The first phase deletes all known unsupported instructions and the second phase deletes all unsupported instructions which have not been encountered before.
To delete all opcodes of known unsupported instructions, we use the code in Listing 5.5. At first we load all known unsupported opcodes for the current architecture from the database *skip.db*. Afterwards we load all opcodes from the collected gadgets and intersect them with the known unsupported opcodes. The gadgets containing opcodes which are inside the intersection of the two opcode groups are deleted from the gadget database.

```
with database("skip.db") as skip:
    skip.create_skip_tables()
    skips = skip.read_skip_opcodes(arch)

with database(db_file) as db:
    gadget_ops = db.read_opcodes_all()

    for op in filter(set(gadget_ops).__contains__, skips):
        db.delete_gadgets_by_opcodes(op[0], op[1])

    db.commit()
```

Listing 5.5: The code used to read all known unsupported opcodes from the skip database. Afterwards the intersection of the current opcodes and known unsupported opcodes is deleted from the current database.

To delete the unsupported instructions which have not been encountered before, we use the procedure presented in Figure 3.3. The child process shown in Figure 3.3 is another instance of pyzex3_dumper.py with a special argument. Depending on if the argument is set the script executes the behavior of the parent or the child. The child prints the upcoming opcode and information on if it is a Thumb or Thumb2 instruction to *stdout*. In case the child crashes, the parent deletes the gadgets containing the last announced opcode. Additionally to the deletion of the gadgets and opcodes in this phase, the parent also adds the encountered unsupported opcodes to the skip.db database.

5.4.2 Analysis

The GitHub page of pyvex states “when used statically, memory is never freed” [51]. Therefore, binaries with a high gadget count can cause our analysis process to run out of memory. This is why we use a child process to perform the analysis, similar to the procedure for undiscovered, unsupported instructions. After the child process terminates, the operating system takes care of the unfree memory by freeing the memory of the process. To analyze the gadgets, the parent process starts a child process in an infinite loop and passes a start and end value as arguments. The child process then loads the gadgets whose IDs are bigger than the start value and less or equal the end value to analyze them. If no gadgets in this range are available, it is assumed that there are no gadgets with a higher ID than the start value. In this case the child process returns 0 to the parent and the loop is aborted. Otherwise, the child process returns a 1 and a new child process is started to analyze the next range. The memory usage of the analysis process can be adjusted by increasing or decreasing the analysis range of the child. The code of the described infinite loop is shown in Listing 5.6.

```
i = 0

while 1:
    p = subprocess.Popen(["python2", "pyzex3_dumper2.py",
                              "-a", arch,
                              "-d", db_file,
                              "-s", str(i),
                              "-e", str(i+10000)])

    p_status = p.poll()

    while p_status == None:
        p_status = p.poll()

    if p_status == 0:
        break

    i += 10000
```

Listing 5.6: The loop starting a child process to analyze the gadgets. The gadget ID range which gets analyze is set by start and end value arguments to the child process.

To analyze a gadget, the child process creates Zex3 objects for every instruction and updates the database and gadget object accordingly. If Zex3 does not support the translation of an opcode, this opcode also gets added to skip.db and the gadget deleted. Afterwards, we add the updated gadget to a Zolver3 object with Algorithm 3.2.2. To fulfill our first objective for the gadget analysis from Chapter 4.2, namely to sort out unsatisfiable path constrains, we check the satisfiability of the gadget. We set a timeout for the satisfiability check by Z3 to 100 milliseconds to keep the analysis efficient. If the gadget is unsatisfiable, either by invalid path constrains or by exceeding the timeout, the gadget gets deleted from the database. Our results show that a sufficient amount of gadgets remain with a timeout of 100 milliseconds. After the satisfiability check, we use Algorithm 4.2.1 to classify the gadget's registers and Algorithm 4.2.2 to classify its memory stores. At the end the respective database fields are updated and set.

5.5 Gadget Search

We perform our gadget search solely on the SQLite database. This way the search can be conducted on any system containing a Python 2 environment. Nevertheless, a search in IDA Pro has the advantage over a simple command line environment that the graph view can be utilized to highlight the instructions and the flow of the gadget. The following section explains the initialization process of the search and how the search itself is implemented.

5.5.1 Initialization

During the initialization phase of our gadget search class *GaDB*, we set the database path, database name, and the architecture of the binary. In case one loads the class in IDA Pro, we are able to detect the architecture and the database file automatically. To detect the architecture, we perform checks similar to the ones Listing 5.1 performs during the initialization of the gadget detection. Since we create the database files by a certain rule, see Chapter 5.2.3, we can apply the same rule to detect the current database file. If GaDB is loaded outside of IDA Pro, one has to supply information on the architecture and the database manually. To consider more than one database file for the gadget search, one has the possibility to add additional database files.

5.5.2 The Search

To conduct the search, we have implemented a function for every semantic definition. These functions set the options of the operation performed by the semantic definition. Additionally, there are functions for fixed function calls and loops. We list all functions and the associated options in Appendix A.2. An example for a search function is *MovReg* in Listing 5.7. Each search function sets up a dictionary with all specific options for its operation. Our exemplary function MovReg sets up a *reg* dictionary with the destination and source register. Other possible dictionaries are *args* for a function call and its arguments and *mem* for operations performing a memory store. We pass the stored information to the function *__search*. The __search function parses kwargs for all general options, for example *EP=True* to indicate that the gadget must start at an EP. __search also looks for specified conditions on output registers. General options and conditions can be set by every search function. If all general options are set and bulk search is not activated, we search in the specified databases for suitable gadgets. We cover bulk search, which we use to set multiple operations for a single gadget, at the end of the section. The list of gadgets returned by the database query are already sorted, as we use the query itself to implement Algorithm 4.3.1. We abort the search after the first database that returns a list of potential gadgets, in case multiple database files have been specified. We proceed this way due to our implementation of Algorithm 4.3.1. Afterwards the potential gadgets of the returned list are verified until a gadget is discovered that satisfies the specified conditions. If a gadget is found, we use *auto_inspect* to either jump to the gadget and highlight it in IDA Pro's graph, print information on the gadget, print information and highlight the gadget, or perform no operation at all.

```
def MovReg(self, **kwargs):
    reg = {}

    if ’dst’ in kwargs:
        reg[’dst’] = self.zolver3.arch.reg_map[kwargs[’dst’].lower()]
    reg[’type’] = 0
    if ’src’ in kwargs:
        reg[’src1’] = self.zolver3.arch.reg_map[kwargs[’src’].lower()]+"init"

    regs = [reg]
    self.__search(({}, regs, [], kwargs))

    if not self.bulk_switch:
        self.verify_gadget()

        if self.cur_gadget:
            self.auto_inspect()
        else:
            print "No gadgets were found."
```

Listing 5.7: Implementation of the search function for the semantic definition MovReg.

Bulk Search

To define multiple operations for a single gadget, one can call the function *bulk_ start*. A call to this function causes all operations from every called search function to be added to the *bulk* dictionary. After all operations have been added, we use the function *bulk_ exec* to execute the search. If no gadgets are found, one has to repeat the search with fewer options manually, as we do not reduce the options automatically until a gadget is found.

6 Evaluation

In this chapter, we show the distribution of the different gadget types, of the different fixed function calls, and the distribution of the gadgets' instruction count. We also show that it is possible to discover enough gadgets with our gadget finder for successful exploitation. For this purpose, we provide the search queries to find the same or similar gadgets used by Göktas' et al. [22] exemplary exploit.

6.1 Testing Environment

We conduct all tests for our evaluation on a 64 bit Linux system, running on a Intel XEON processor E3 with 3.3 GHZ. As we use the Windows version of IDA Pro to run our plugin for the gadget discovery, we have to utilize Wine [47]. Inside of Wine is a 32 bit Python 2.7 environment to run the plugin. On the Linux system itself is a 64 bit Python 2.7 environment installed. We used the 64 bit environment for the translation from VEX to Z3 and the analysis of the gadgets. For Zex3's translation process we use Valgrind 3.9.0 and pyvex's latest commit at the time of testing [49].

6.2 Statistics

We present the distribution of the gadget types and gadget lengths in this chapter. For our evaluation, we analyzed the x86 and AMD64 version of *ieframe.dll* and *mshtml.dll* of Internet Explorer (IE) 8.0.7601.17514. We selected these libraries, as they are often used during exploitation of IE [33], and because they are used by Göktas et al. [22] to write an exemplary exploit with CFI resistant gadgets. To evaluate our gadget finder on ARM, we analyzed Debian's (little-endian) libc-2.19.so, as we expect libc to always be loaded during exploitation of a Linux system on ARM. All gadgets contained in libc-2.19.so are in ARM mode. The gadget counts presented in this section are the total number of gadgets, including gadgets with and without conditional branches.

Table 6.1 contains information on how many gadgets are available and how long it takes to have a database ready for search queries. During the analysis, we delete the gadgets containing instructions that are unsupported by either VEX or Zex3. We also delete the gadgets with unsatisfiable path constrains. The row *Remaining*

Table 6.1: This table shows the number of available gadgets and the time it takes to discover and analyze them. All times are displayed in seconds (s).

	ieframe.dll	mshtml.dll	ieframe.dll	mshtml.dll	libc-2.19.so
Architecture	x86	x86	AMD64	AMD64	ARM
Discovered Gadgets	99355	160266	108010	181749	12401
Remaining After Analysis	91584	147695	95062	163827	10450
Discovery Phase (s)	51.9	91.6	66.7	122.0	11.0
Deletion Phase 1st Run (s)	5.2	8.9	1.3	2.0	1.4
Deletion Phase 2nd Run (s)	37.9	60.9	244.2	393.1	0.57
Analysis Phase 1st Run (s)	6434.8	14540.6	8219.9	25560.0	2195.0
Analysis Phase 2nd Run (s)	6395.4	14356.7	7777.3	25182.7	1871.0

After Analysis shows how many gadgets remain after the mentioned gadgets have been deleted. The *Discovery Phase* shows how long our IDA Pro plugin takes to discover the gadgets and to write them to the database. For the *Deletion Phase* and *Analysis Phase* we show two runs, because during the first run all instructions unsupported by either VEX or Zex3 are stored in skip.db. During the second run, phase 1 of our deletion process, Chapter 5.4.1, takes effect. Therefore, the *Deletion Phase* takes longer but the *Analysis Phase* finishes faster, as fewer gadgets have to be analyzed. However, the success of this methods differers, depending on the number of instructions in skip.db and in the current gadget database. The most unsupported instructions are detected for AMD64 and the fewest for ARM. Consequently the time to intersect the instructions from mshtml's (AMD64) database and skip.db takes the longest. The time needed for libc's second Deletion Phase is even shorter than for its first. A possible explanation is that it is faster to intersect these small instruction pools, than to restart the child process upon crashes because of unsupported instructions. Another beneficial factor for ARM are the frequent conditional instructions of the architecture. These instructions result in complex Z3Py equations and, therefore, take longer to translate and to symbolically execute than x86 and AMD64 instructions. Hence, the time benefit for ARM is higher if the instructions do not have to be translated and analyzed.

6.2.1 Gadget Type Distribution

This section provides information on the distribution of the different gadget start and end types, on how many loops are discovered, and which fixed function calls are discovered.

The information on the gadget start and end type distribution are presented in Table 6.2. We refrain from including the content of gadgets, loops and fixed function calls, as their distribution is shown in Table 6.3 and Table 6.4. On inspection of Table 6.2, we notice that the combination with the highest amount of gadgets is CS-RET. With CS-RET gadgets one can execute common ROP exploits, without triggering CFI checks. Due to the high proportion of CS-RET gadgets, the highest possibility to find suitable gadgets for a gadget chain is using a ROP chain.

Table 6.2: The number of available gadgets categorized by gadget start and end type.

	ieframe.dll	mshtml.dll	ieframe.dll	mshtml.dll	libc-2.19.so
Architecture	x86	x86	AMD64	AMD64	ARM
EP-IC	4255	4245	4354	3947	261
EP-IJ	59	370	172	1009	79
EP-RET	11521	16723	10950	16517	2615
CS-IC	36300	55225	38679	68791	1226
CS-IJ	67	28	76	1365	240
CS-RET	39382	71104	40831	72198	6029

Our loop counts, presented in Table 6.3, are just based on our loop definition. This means that all listed loops end with an IC and start at the CS of the IC. The number of discovered loops can still be further increased by implementing loops for JOP or allowing looser loop definitions.

Table 6.3: The number of gadgets containing loop for each analyzed library.

	ieframe.dll	mshtml.dll	ieframe.dll	mshtml.dll	libc-2.19.so
Architecture	x86	x86	AMD64	AMD64	ARM
Loops	348	443	335	464	55

In Table 6.4 we list all fixed function calls that remain after the analysis phase. We have not included libc-2.19.so, because we have not implemented any fixed function calls for ARM, yet. We notice that all functions typically used by attackers for malicious behavior are available in each address space, like VirtualProtect to set memory to executable or writable, LoadLibrary to load a library to the address space, and CreateProcess to create a process. Gadgets containing fixed function calls are not restricted to some gadget start and end types, but are interspersed throughout all start and end type combinations.

Table 6.4: All fixed function calls remaining after the analysis and their count per analyzed library.

	ieframe.dll	mshtml.dll	ieframe.dll	mshtml.dll
Architecture	x86	x86	AMD64	AMD64
msvcrt_memcpy	105	187	162	424
KERNEL32_VirtualProtect	1	0	1	0
KERNEL32_VirtualAlloc	0	0	1	0
KERNEL32_MapViewOfFile	7	1	4	0
KERNEL32_LoadLibraryW	22	10	28	10
KERNEL32_LoadLibraryExW	4	0	0	0
KERNEL32_LoadLibraryA	1	0	2	6
KERNEL32_CreateProcessW	5	0	1	0
KERNEL32_CreateFileW	1	0	2	0
KERNEL32_CreateFileMappingW	1	1	0	0

6.2.2 Gadget Length Distribution

We analyze the distribution of the gadgets' path length in this section. Common code-reuse attacks usually utilize rather short gadgets, which end after a certain operation is performed. Our gadgets on the other hand are limited by a user defined instruction limit and a few start point and endpoint instructions. We use the default instruction limit of 30 instructions for our evaluation.

Both charts, the one for x86 in Figure 6.1 and the one for AMD64 in Figure 6.2, look rather similar. They both peak at 8 to 9 instructions per gadget. Even gadgets containing just one instruction are included. On first sight, multiple gadgets with just a single instruction may seem suspicious. Especially as we just store a single gadget upon encountering multiple gadgets with the same instruction sequence. However, they can be easily explained. Gadgets with a single instruction consists of different *retn* instructions and ICs. These instructions are mostly preceded by calls, such as *__SEH_epilog4* in case of retn instructions. On AMD64 the amount of gadgets containing just one instruction is lower compared to x86, while more gadgets with a higher instruction count are available. This can be explained due to differences in their ABIs. For example, the first arguments on AMD64 are passed in registers instead of the stack. Therefore, fewer retn instructions with different parameters to clean up the stack are necessary.

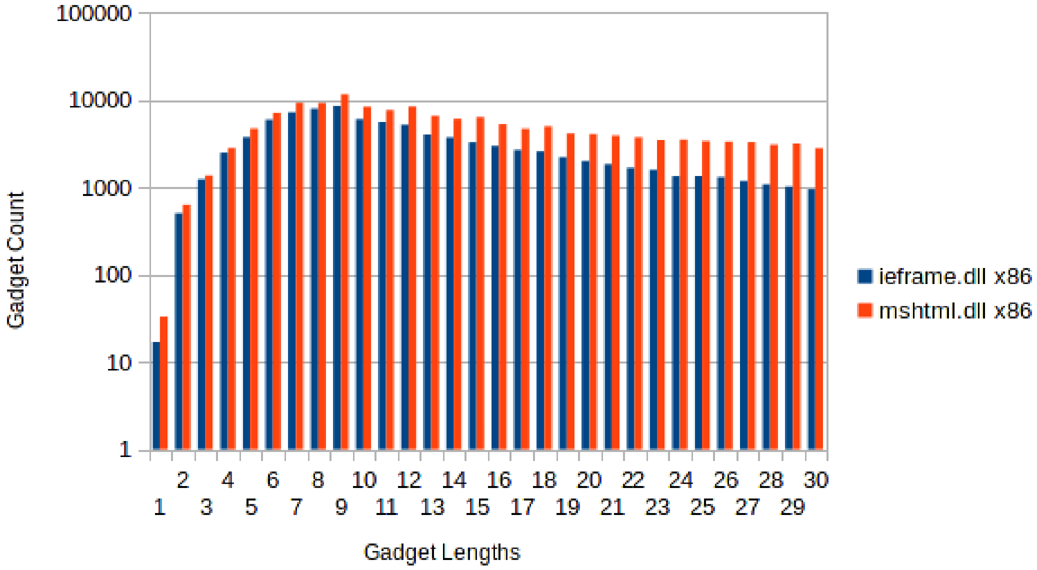


Figure 6.1: The distribution of ieframe's and mshtml's gadgets on the x86 architecture based on their instruction count.

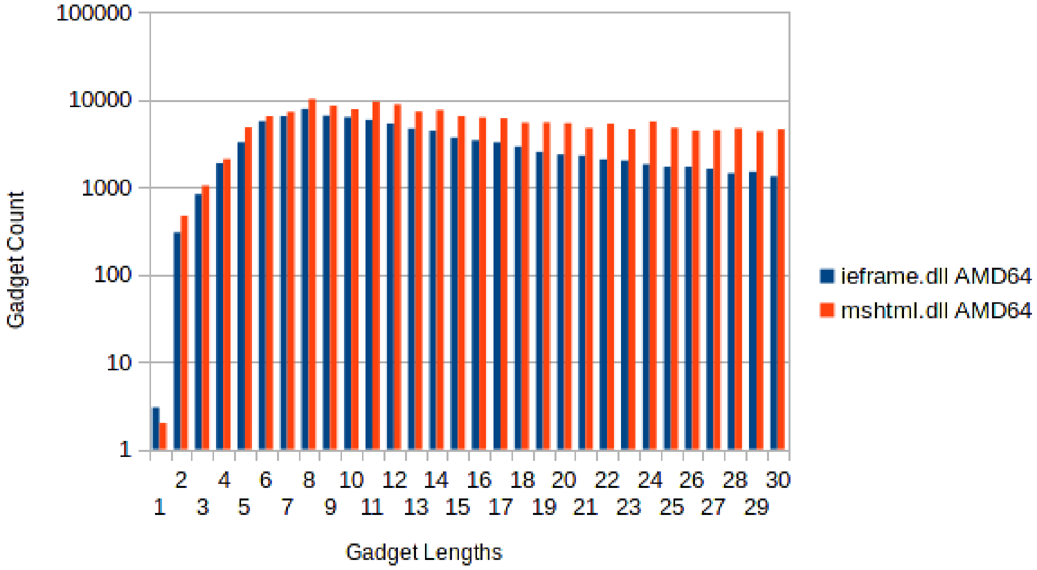


Figure 6.2: The distribution of ieframe's and mshtml's gadgets on the AMD64 architecture based on their instruction count.

The gadget lengths in the chart for ARM, Figure 6.3, are more evenly distributed compared to the ones for x86 and AMD64. One can observe peaks around the gadget lengths 16 and 20. Upon investigating, we determine that the peaks result from a high gadget density in several switch statements.

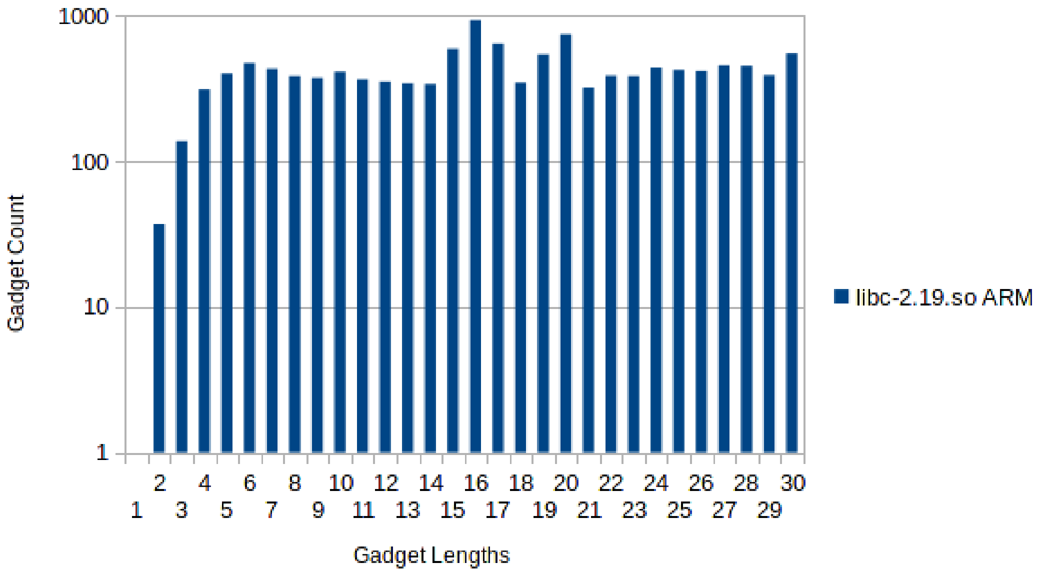


Figure 6.3: The distribution of libc's gadgets on the ARM architecture based on their instruction count.

6.3 CVE-2012-1876

CVE-2012-1876 describes a heap overflow vulnerability. It was initially used by VUPEN at Pwn2own 2012 to compromise an IE 9 installation [33]. Göktas et al. developed an exemplary exploit for this vulnerability to demonstrate the effectiveness of their gadget bound definitions against CCFIR [22]. To evaluate our gadget search for x86, we demonstrate that we find the same or equivalent gadgets as they have found. Unfortunately, our IE version differs slightly from their version. Because of this difference some gadgets are not present in our libraries. If this is the case, we verify that the gadgets are definitely not available and offer equivalent gadgets.

Gadget 1-1

The initial control transfer of their exploit directs the control-flow to the first gadget by an IJ. Consequently, the first gadget must start at an EP. Göktas et al. use an EP-IC-R gadget, Listing 6.1, to switch from EP gadgets to CS gadgets. They achieve this by first pushing the return address as an argument and then breaking the caller's assumption with Gadget 1-2. However, Gadgets 1-3 and 1-4 have to set up the state for Gadget 1-1, as a pointer to the buffer is required as the argument to the gadget, line 4.

```
1  mov   edi, edi
2  push  ebp
3  mov   ebp, esp
4  mov   eax, [ebp+8]
5  push  dword ptr [ebp+0Ch]
6  mov   ecx, [eax]
7  push  dword ptr [eax+0Ch]  ; push return address
8  push  eax
9  call  dword ptr [ecx+10h]
10 test  eax, eax
11 jge   short loc_76836DA5  ; if eax >= 0:---+
12 mov   eax, 80004005h      ;                |
13 pop   ebp                 ; <--------------+
14 retn  8
```

Listing 6.1: Gadget of Göktas et al. to switch from EP to CS gadgets. The gadget is contained in ieframe.dll.

Our current implementation does not contain a function to connect EP-IC gadgets with the sequent CS-RET gadgets, yet. That's why we have to look for the gadgets in two phases. First we search the EP-IC gadget and then the CS-RET.
We know that our gadget has to push an argument to the stack, so we look for a StoreMem gadget with a write to esp. Listing 6.2 shows our search query. The minimal instruction length and indirect call value is used to find the gadget faster. Under normal circumstances we do not even know the register containing the buffer, as we would expect a dereferencing of ecx to another register followed by a push of register+offset.

```
Python>g.StoreMem(EP=True, IC=True, dreg1="esp", dop="+", Min_inst=9, IC_value="dword ptr [
    ecx+10h]")
...
Python>g.next()
```

Listing 6.2: Search query one of two to replicate the gadget in Listing 6.1.

For the second gadget, we can replicate the operations of the expected gadgets with a bulk search, see Listing 6.3. An implementation to search directly for EP-IC gadgets followed by CS-RET gadgets can simply query the database for CS-RET gadgets that start at the CS of EP-IC gadgets.

```
Python>g.bulk_start()
Python>g.LoadReg(dst="ebp", CS=True, RET=True, Min_inst=4)
Python>g.Arithmetic(dst="esp", sreg1="esp", op="+", sbv=16)
Python>g.bulk_exec()
...
Python>g.next()
```

Listing 6.3: Search query two of two to replicate the gadget in Listing 6.1.

Gadget 1-2

Gadget 1-2 in Listing 6.4 is just supposed to break the caller's assumption by not removing the passed arguments from the stack. Hence, the only purpose is to return to the EP-CS-RET gadget.

```
1 mov   eax, [ecx+74h]
2 retn
```

Listing 6.4: Göktas' et al. gadget in ieframe.dll to break the caller's assumption.

Due to the differences in our library versions, our search for exactly this gadget yields no results, Listing 6.5 lines 1 and 2. We verified that the gadget is not available by conducting a search in IDA Pro for the instruction contained in line 1 of Listing 6.4. But, as the purpose of the gadget is just to return, we can search for a NOP gadget that starts at an EP and ends with an RET. This search results in a gadget containing only a retn instruction.

```
1 Python>g.LoadMem(dst="eax", sreg1_mem="ecx", op_mem="+", sbv_mem=0x74, EP=True, RET=True)
2 No gadgets were found.
3 Python>g.NOP(EP=True, RET=True)
```

Listing 6.5: Our search queries to find a equivalent gadget to the one in Listing 6.4.

Gadget 1-3

Gadget 1-1 receives the pointer to the buffer via the first argument to the gadget. To push the argument and to call Gadget 1-1, Göktas et al. use the gadget from Listing 6.6. However, the gadget pushes edi as first argument. Therefore, they have to set up edi with a pointer to the buffer, Listing 6.8, before they can call Gadget 1-3.

```
mov   edi, edi
push  esi
mov   esi, ecx
mov   eax, [esi]
push  edi                    ; callee's argument
call  dword ptr [eax+0C4h]
...
```

Listing 6.6: Gadget to push a pointer to the buffer and to call Gadget 1-1.

We specify the source register in our query, because we receive too many useless gadgets otherwise. An alternative search query is in Listing 6.10.

```
Python>g.StoreMem(dreg1="esp", dop="+", sreg="edi", EP=True, IC=True)
```

Listing 6.7: Our search query to find the gadget from Listing 6.6.

Gadget 1-4

Göktas et al. use Gadget 1-4 to dereference the button object from ecx to edi. After the dereference, edi holds a pointer to the buffer.

```
mov   edi, edi
push  ebp
mov   ebp, esp
push  ebx
push  esi
mov   esi, ecx
push  edi
mov   edi, [esi]          ; edi pointer to buffer
call  dword ptr [edi+14h]
...
```

Listing 6.8: Gadget to set up edi with a pointer to the buffer.

We find the gadget immediately with the search query in Listing 6.9.

```
Python>g.LoadMem(dst="edi", sreg1_mem="ecx", EP=True, IC=True)
```

Listing 6.9: Our search query to find Gadget 1-4.

Alternative to the Gadgets 1-3 and 1-4

We find an alternative gadget to replace the Gadgets 1-3 and 1-4 by the query in Listing 6.10. The query searches for a gadget which directly dereferences the button object and pushes the address of the buffer to the stack.

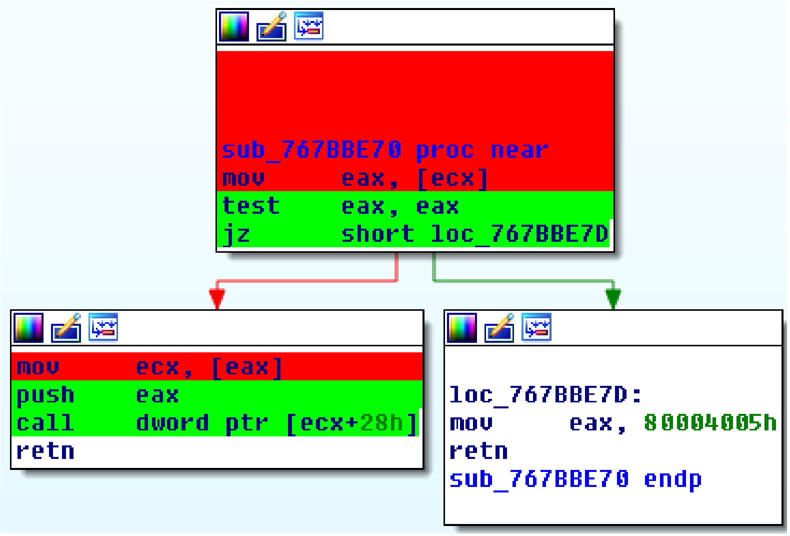
```
Python>g.StoreMem(dreg1="esp", dop="+", sreg1_mem="ecx", EP=True, IC=True)
```

Listing 6.10: Search query to find a gadget which directly pushes the address of the buffer.

Figure 6.4 shows the first result of this search. Unfortunately, we can't use this gadget to replace Gadget 1-1, as Gadget 1-1 pushes its return address to the stack and loads ebp with the address of the buffer.

Gadget 2-1

So far Göktas et al. have switched from EP to CS gadgets and have loaded ebp with a pointer to their buffer. To stack pivot, they use the gadget in Listing 6.11. Additionally, the newly loaded esp gets increased by 0x28 during the executing of Gadget 2-1.



```
sub_767BBE70 proc near
mov     eax, [ecx]
test    eax, eax
jz      short loc_767BBE7D
```

```
mov     ecx, [eax]
push    eax
call    dword ptr [ecx+28h]
retn
```

```
loc_767BBE7D:
mov     eax, 80004005h
retn
sub_767BBE70 endp
```

Figure 6.4: Alternative gadget to replace the Gadgets 1-3 and 1-4.

```
1 pop   edi
2 pop   esi
3 mov   esp, ebp
4 pop   ebp
5 retn  20h
```

Listing 6.11: Gadget for stack pivoting. It is located in mshtml.dll

To find Gadget 2-1, we first try to search for an Arithmetic gadget. This results in a gadget consisting of the instructions of Listing 6.11 lines 3 to 5. However, as Göktas et al. do not describe a purpose for loading edi and esi, this gadget should be sufficient. The exact gadget from Listing 6.11 can be found by the bulk search in Listing 6.12.

```
Python>g.Arithmetic(dst="esp", sreg1="ebp", op="+", sbv=0x28)
...
Python>g.bulk_start()
Python>g.Airthmetic(dst="esp", sreg1="ebp", op="+", sbv=0x28)
Python>g.LoadReg(dst="edi")
Python>g.LoadReg(dst="esi")
Python>g.bulk_exec()
```

Listing 6.12: The search queries to find Gadget 2-1.

Gadget 2-2

According to Göktas et al., their stack pointer needs to be increased further. They use the gadget in Listing 6.13 to increase the stack pointer by 0x18 bytes.

```
retn 14h
```

Listing 6.13: Gadget in mshtml.dll to increase the stack pointer.

The gadget in Listing 6.13 simply adds 0x18 to esp. Therefore, we search for an Arithmetic gadget containing this operation, Listing 6.14.

```
Python>g.Airthmetic(dst="esp", sreg1="esp", op="+", sbv=0x18)
```

Listing 6.14: The query we use to locate Gadget 2-2.

Gadget 3-1

To switch from a code-reuse attack to a code-injection attack, Göktas et al. first make existing program code writable and then write their shellcode there. They use Gadget 3-1, Listing 6.15, to write the shellcode to the program code via a call to memcpy.

```
push eax                 ; destination
call memcpy
add  esp, 0Ch
xor  eax, eax
jmp  short loc_7672DCE7
pop  ebx
pop  edi
pop  esi
pop  ebp
retn 8
```

Listing 6.15: The gadget in ieframe.dll to copy shellcode to existing program code.

Our gadget finder also deems the gadget in Listing 6.15 to be the best gadget for a memcpy call. One can view our search query in Listing 6.16.

```
Python>g.Call(func="msvcrt_memcpy", CS=True, RET=True)
```

Listing 6.16: The search query to find Gadget 3-1.

Gadget 3-2

Listing 6.17 shows the gadget Göktas et al. use to make existing program code writable.

```
and  dword ptr [ebp-0Ch], 0
lea  eax, [ebp-0Ch]
push eax               ; address to save old protection
push 40h               ; new protection
push ebx               ; size
mov  ebx, [ebp-8]
push ebx               ; address
call ds:VirtualProtect
test eax, eax
jz   loc_766E9531      ; if eax==0: goto error handler
mov  eax, [ebp+8]
and  dword ptr [edi+4], 0
mov  [edi+8], eax
mov  [edi+10h], esi
mov  [edi+0Ch], ebx
mov  eax, [ebp-0Ch]
mov  [edi+14h], eax
mov  eax, dword_768E2CCC
mov  [edi], eax
mov  dword_768E2CCC, edi
xor  eax, eax
pop  edi
pop  ebx
pop  esi
leave                  ; == mov esp, ebp and pop ebp
retn 14h
```

Listing 6.17: The gadget which is used by Göktas et al. to make existing code writable. It is contained in ieframe.dll.

We use the search query in Listing 6.18 to find Gadget 3-2.

```
Python>g.Call(func="KERNEL32_VirtualProtect", CS=True, RET=True)
```

Listing 6.18: Our search query to find the gadget in Listing 6.17.

6.4 Simple ARM Example

To evaluate our gadget finder on ARM, we exploit a fictional use-after-free vulnerability. The instruction initiating our chain is an IC in ARM mode and the first argument, stored in R0, contains a pointer to our prepared buffer. The protection in place is similar to CCFIR. This means, IC and IJ can just transfer the control-flow to EPs and RETs are just allowed to return to legitimate CS. We assume that an information leak is available, which is usually the case for real-world exploits. Our gadget pool is derived from Debian's libc-2.19.so. All discovered gadgets are in ARM mode. The goal of the exploit is to execute *system("/bin/sh")*.

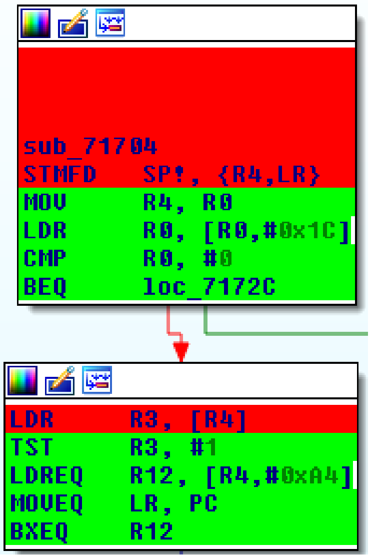
On ARM the first argument to a function is not passed on the stack, but in the register R0. Therefore, to execute system("/bin/sh") we have to load the address of a string containing "/bin/sh" into R0. We do not have to load the string to memory ourself, as it is already present in libc-2.19.so. We use the information leak to get the address of libc-2.19.so. The address of libc-2.19.so is also required to get the address of system. We place the addresses later on in our buffer. But at first, we have to find the gadgets to load the addresses from the buffer and call the system function. A pointer to the buffer is passed to our gadgets in R0. Due to the protection scheme in place, the gadget has to start at an EP. The end of the gadget is not defined, yet. Listing 6.19 shows the search to find a gadget that loads an address from the supplied buffer into R0. While the first gadget does not suite our needs, the next one does. The discovered gadget is displayed in Figure 6.5.

```
Python>g.LoadMem(dst="R0", sreg1_mem="R0", EP=True)
...
Python>g.next()
Start Addr: 0x71704 End Addr: 0x71728
Starts at the function entry point.
Ends with an indirect call (R12).
Nr of instructions: 10
Nr of memory loads: 3
Nr of memory stores: 1
Reg Types:
  R48_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R64_: [8, ’’, ’’, ’’, ’’, ’’, ’’]
  R56_: [8, ’’, ’’, ’’, ’’, ’’, ’’]
  R28_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R52_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R60_: [2, ’4294967288’, ’+’, ’R60_init’, ’’, ’’, ’’]
  R32_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R44_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R12_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R20_: [3, ’’, ’’, ’’, ’R8_init’, ’’, ’’]
  R36_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R16_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
  R24_: [0, ’R8_init’, ’’, ’’, ’’, ’’, ’’]
  R68_: [8, ’’, ’’, ’’, ’’, ’’, ’’]
  R8_: [3, ’’, ’’, ’’, ’28’, ’+’, ’R8_init’]
  R40_: [7, ’’, ’’, ’’, ’’, ’’, ’’]
Memory Stores:
  (’4294967288’, ’+’, ’R60_init’): [5, ’R24_init’, ’’, ’’, ’’, ’’, ’’]
  (’4294967292’, ’+’, ’R60_init’): [5, ’R64_init’, ’’, ’’, ’’, ’’, ’’]
```

Listing 6.19: A gadget search for a LoadMem gadget. The first result is replaced by ... in the output and the information of the second gadget are displayed.

The gadget fulfills two objectives. First, it loads the address of “/bin/sh” from our buffer to R0, *LDR R0, [R0,#0x1C]*. And second, it loads the address of system to R12 and calls R12 at the end. This way the objective to execute system(“/bin/sh”) is achieved with a single gadget.

The buffer that we use during the exploit is shown in Listing 6.20. At offset 0 the buffer must contain 0x00000001 to satisfy *TST R3,#1*. Just if this check is valid, the address of system gets loaded and called. Normally, the address of the first gadget for the initial control-flow transfer is also supplied from this buffer. However, we have not specified at which offset the address has to be for our fictional vulnerability.



```
sub_71704
STMFD   SP!, {R4,LR}
MOV     R4, R0
LDR     R0, [R0,#0x1C]
CMP     R0, #0
BEQ     loc_7172C
```

```
LDR     R3, [R4]
TST     R3, #1
LDREQ   R12, [R4,#0xA4]
MOVEQ   LR, PC
BXEQ    R12
```

Figure 6.5: An ARM gadget which loads the address of "/bin/sh" from the supplied buffer in R0, loads the address of system from the buffer to R12, and ends with an IC of R12.

```
Buffer+0x00 => 0x00000001  # Must contain 0x00000001.
Buffer+0xXX => 0x00071704  # Address of the first gadget.
Buffer+0x04 => 0x41414141
...
Buffer+0x18 => 0x41414141
Buffer+0x1C => 0x00122F58   # .rodata:00122F58 aBinSh     DCB "/bin/sh",0
Buffer+0x20 => 0x41414141
...
Buffer+0xA0 => 0x41414141
Buffer+0xA4 => 0x0003B190   # .text:0003B190 system
```

Listing 6.20: Buffer containing the data for our exploit. Everything, but the addresses at the offsets 0x1C and 0xA4, the address for the initial control-flow transfer, and the 0x00000001 at offset 0, is filled with random data.

7 Conclusion and Future Work

During the course of this thesis we have presented an architecture independent gadget finder for CFI and heuristic check resistant gadgets, a translation module from VEX to Z3Py called Zex3, and a symbolic execution engine utilizing our translation called Zolver3. The CFI protections that we have taken into account for in this thesis are supposed to prevent exploitation of vulnerabilities or make exploitation impractical by increasing the effort to write an exploit. However, we have shown that with reasonable effort an automated gadget discovery of CFI resistant gadgets is possible. We have also shown that Z3 is capable to analyze complex CFI resistant gadgets. The results of the analysis are suitable to categorize the gadgets by applying semantic definitions to each modification of the gadget to the program's state. Afterwards, the semantic definitions can be used for a convenient search for specific gadgets. This reduces the effort to break the CFI checks significantly. Furthermore, our work shows that the approach to discover and analyze CFI resistant gadgets can be transfered to other architectures by utilizing an IL. The effort to support another architecture for the gadget search just consists of the effort to support another architecture by Zex3 and Zolver3. To our knowledge we are the first to apply the concept of CFI resistant gadgets on architectures other than x86 and x86-64/AMD64.

The following sections present several possibilities for future research and ways to improve our implementation of the gadget finder, Zex3, and Zolver3.

7.1 Switching to Miasm

The reverse engineering framework Miasm also supports the translation from its IL representation to Z3. Additionally, it has emulation capabilities and can highlight side effects of instructions. However, the disadvantage is that Miasm does not support as many architectures as Valgrind does. But our gadget finder would still benefit a lot from a switch from VEX to Miasm. For VEX we have to store the opcodes of all instructions and additional information, which are not available to us otherwise, to the database, as we have to run our analysis outside of IDA Pro. Miasm on the other side can be directly used from IDA Pro. Therefore, we do not have to store the additional information, which reduces the size of our database. We also expect a significant increase in speed of the search as a whole, as we do not have to make the efforts we currently do because of some peculiarities of VEX, such as sorting out invalid instructions with an extra process, perform the analysis in an extra process, and load and store additional information to the database.

7.2 Switching to SMT2

SMT2 is the language of the 2nd version of *SMT-LIB*. The advantage of SMT2 is that Z3Py has a dedicated function to parse a string containing SMT2 expressions. Therefore, we can discard Python 2's eval function from Zolver3 to evaluate the equation string, which poses a security risk if the string comes from an untrusted source. Nevertheless, we still have to parse the strings ourself to insert the SMT2 code of fixed function calls.

7.3 Enhancing the Database Support

Currently, we only support SQLite databases. With the broadening of the database support to other databases, such as PostgreSQL, we can also support parallel analysis of the gadgets by multiple processes. Due to the nature of SQLite, this is currently not possible. However, SQLite databases have the advantage that the database file can easily be copied to another device.

7.4 Enhancing the Discovery Phase

Because of our modular design, the gadget discovery phase is independent from the gadget analysis phase. Hence, it is easy for us to extend our gadget repertoire, as we just have to add functionality to the discovery phase and to the database. This way we can easily include gadgets containing unintended instructions or gadgets consisting of virtual functions [36].

7.5 Feed Back of Definition Matching Output

Our analysis tries to match our definitions to the output equations of the gadgets once. If they do not match 100 percent, the equation is undefined. To render our additional definitions redundant, see Chapter 4.2.1, partially matching definitions can be fed to the analysis function again. A partially match is for example a Arithmetic operation with a MemLoad as one of its operands. By feeding the operand containing the MemLoad to the analysis function again, the definitions can be chained, even multiple times, and a more precise output analysis is achieved. A schematic of this procedure is displayed in Figure 7.1.

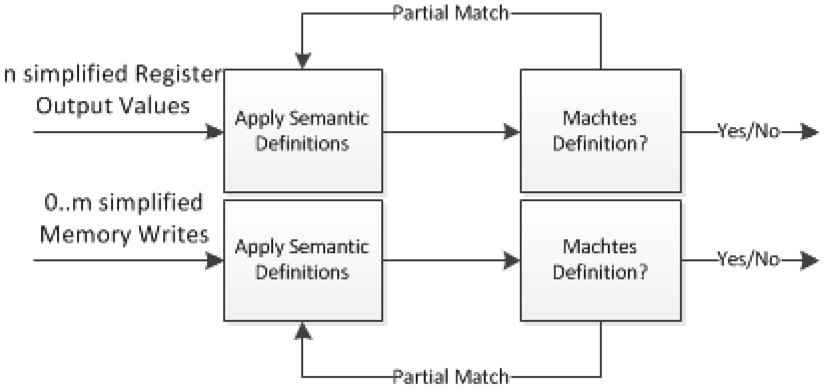

Partial Match
n simplified Register Output Values
Apply Semantic Definitions
Machtes Definition?
Yes/No
0..m simplified Memory Writes
Apply Semantic Definitions
Machtes Definition?
Yes/No
Partial Match

Figure 7.1: Semantic definitions could be applied again if they just match partially after they have been applied once. With a partial match we mean if for example one operand does not match Arithmetic because it is a MemLoad. The definitions would then be applied just to this operand.

Another possibility to achieve a more precise analysis is to feed the complete output register equation back as a valid register. Currently, we only define symbolic input registers as valid register matches for our semantic definitions. Adding the output equation as a valid match would reduce the number of undefined equations significantly.

However, both proposed changes to our current algorithms increase the analysis time and require a more complex database scheme. In future work it has to be evaluated if the time tradeoff is worth the improved results.

7.6 Compiler

The final goal is to use our gadget finder for automated gadget chaining. Upon every search a list of gadget candidates is returned. Hence, one way to achieve automated gadget chaining is by blindly trying to chain the gadgets from the different gadget lists until the result of Zolver3 is sat. But a better way is to engineer a logic that chains the gadgets according to the desired goal and to their side effects. Due to the complexity of the gadgets, it is probably not possible to find a side effect free, atomic instruction set consisting of CFI and heuristic check resistant gadgets. Therefore, an intelligent solution always have to take into account the side effects of the gadgets.

A Appendix

A.1 Fixed Functions

The following two tables contain all implemented calls to external functions of the x86 and AMD64 architectures and their implementation status. If the return value is implemented, the eax/rax register is set to the value of a successful function call result. In case the parameters are implemented, we set Z3 variables to a memory read of the respective stack address or to a register read of the respective register, depending on the calling convention. One can use the Z3 variables to set parameters and to check if certain parameters are possible with the current gadget.
There are currently no external functions implemented for the ARM architecture.

Table A.1: A list of all fixed functions for the x86 architecture.

	Return Value	Parameters
KERNEL32_LoadLibraryA	✓	✓
KERNEL32_LoadLibraryW	✓	✓
KERNEL32_LoadLibraryExA	✓	✓
KERNEL32_LoadLibraryExW	✓	✓
KERNELBASE_LoadLibraryExA	✓	✗
KERNELBASE_LoadLibraryExW	✓	✗
KERNEL32_LoadPackagedLibrary	✓	✓
ntdll_LdrLoadDll	✓	✓
KERNEL32_VirtualAlloc	✓	✓
KERNEL32_VirtualAllocEx	✓	✓
KERNELBASE_VirtualAlloc	✓	✗
KERNELBASE_VirtualAllocEx	✓	✗
ntdll_NtAllocateVirtualMemory	✓	✓
KERNEL32_VirtualProtect	✓	✓
KERNEL32_VirtualProtectEx	✓	✓

Table A.1: A list of all fixed functions for the x86 architecture.

	Return Value	Parameters
KERNELBASE_VirtualProtect	✓	✗
KERNELBASE_VirtualProtectEx	✓	✗
ntdll_NtProtectVirtualMemory	✓	✓
KERNEL32_HeapCreate	✓	✓
KERNELBASE_HeapCreate	✓	✗
ntdll_RtlCreateHeap	✓	✓
KERNEL32_CreateProcessA	✓	✓
KERNEL32_CreateProcessW	✓	✓
KERNEL32_CreateProcessInternalA	✓	✗
KERNEL32_CreateProcessInternalW	✓	✗
ntdll_NtCreateUserProcess	✓	✓
ntdll_NtCreateProcess	✓	✓
ntdll_NtCreateProcessEx	✓	✓
KERNEL32_CreateRemoteThread	✓	✓
KERNEL32_CreateRemoteThreadEx	✓	✓
KERNELBASE_CreateRemoteThreadEx	✓	✗
ntdll_NtCreateThreadEx	✓	✓
KERNEL32_WriteProcessMemory	✓	✓
KERNELBASE_WriteProcessMemory	✓	✗
ntdll_NtWriteVirtualMemory	✓	✓
KERNEL32_WinExec	✓	✓
KERNEL32_CreateFileA	✓	✓
KERNEL32_CreateFileW	✓	✓
KERNELBASE_CreateFileW	✓	✗
ntdll_NtCreateFile	✓	✓
KERNEL32_CreateFileMappingA	✓	✓
KERNEL32_CreateFileMappingW	✓	✓
KERNELBASE_CreateFileMappingW	✓	✗
KERNELBASE_CreateFileMappingNumaW	✓	✗

Table A.1: A list of all fixed functions for the x86 architecture.

	Return Value	Parameters
ntdll_NtCreateSection	✓	✓
KERNEL32_MapViewOfFile	✓	✓
KERNEL32_MapViewOfFileEx	✓	✓
KERNELBASE_MapViewOfFile	✓	✗
KERNELBASE_MapViewOfFileEx	✓	✗
KERNEL32_MapViewOfFileFromApp	✓	✗
ntdll_NtUnmapViewOfSection	✓	✓
ntdll_NtMapViewOfSection	✓	✓
ntdll_NtContinue	✓	✓
ntdll_NtSetContextThread	✓	✓
KERNEL32_SetProcessDEPPolicy	✓	✓
KERNEL32_GetProcessDEPPolicy	✓	✓
msvcrt_memcpy	✓	✓

Table A.2: A list of all fixed functions for the AMD64 architecture.

	Return Value	Parameters
KERNEL32_LoadLibraryA	✓	✓
KERNEL32_LoadLibraryW	✓	✓
KERNEL32_LoadLibraryExA	✓	✓
KERNEL32_LoadLibraryExW	✓	✓
KERNELBASE_LoadLibraryExA	✓	✗
KERNELBASE_LoadLibraryExW	✓	✗
KERNEL32_LoadPackagedLibrary	✓	✓
ntdll_LdrLoadDll	✓	✓
KERNEL32_VirtualAlloc	✓	✓
KERNEL32_VirtualAllocEx	✓	✗
KERNELBASE_VirtualAlloc	✓	✗
KERNELBASE_VirtualAllocEx	✓	✗

Table A.2: A list of all fixed functions for the AMD64 architecture.

	Return Value	Parameters
ntdll_NtAllocateVirtualMemory	✓	✗
KERNEL32_VirtualProtect	✓	✓
KERNEL32_VirtualProtectEx	✓	✗
KERNELBASE_VirtualProtect	✓	✗
KERNELBASE_VirtualProtectEx	✓	✗
ntdll_NtProtectVirtualMemory	✓	✗
KERNEL32_HeapCreate	✓	✓
KERNELBASE_HeapCreate	✓	✗
ntdll_RtlCreateHeap	✓	✗
KERNEL32_CreateProcessA	✓	✗
KERNEL32_CreateProcessW	✓	✗
KERNEL32_CreateProcessInternalA	✓	✗
KERNEL32_CreateProcessInternalW	✓	✗
ntdll_NtCreateUserProcess	✓	✗
ntdll_NtCreateProcess	✓	✗
ntdll_NtCreateProcessEx	✓	✗
KERNEL32_CreateRemoteThread	✓	✗
KERNEL32_CreateRemoteThreadEx	✓	✗
KERNELBASE_CreateRemoteThreadEx	✓	✗
ntdll_NtCreateThreadEx	✓	✗
KERNEL32_WriteProcessMemory	✓	✗
KERNELBASE_WriteProcessMemory	✓	✗
ntdll_NtWriteVirtualMemory	✓	✗
KERNEL32_WinExec	✓	✓
KERNEL32_CreateFileA	✓	✗
KERNEL32_CreateFileW	✓	✗
KERNELBASE_CreateFileW	✓	✗
ntdll_NtCreateFile	✓	✗
KERNEL32_CreateFileMappingA	✓	✗

Table A.2: A list of all fixed functions for the AMD64 architecture.

	Return Value	Parameters
KERNEL32_CreateFileMappingW	✓	✗
KERNELBASE_CreateFileMappingW	✓	✗
KERNELBASE_CreateFileMappingNumaW	✓	✗
ntdll_NtCreateSection	✓	✗
KERNEL32_MapViewOfFile	✓	✗
KERNEL32_MapViewOfFileEx	✓	✗
KERNELBASE_MapViewOfFile	✓	✗
KERNELBASE_MapViewOfFileEx	✓	✗
KERNEL32_MapViewOfFileFromApp	✓	✗
ntdll_NtUnmapViewOfSection	✓	✓
ntdll_NtMapViewOfSection	✓	✗
ntdll_NtContinue	✓	✓
ntdll_NtSetContextThread	✓	✓
KERNEL32_SetProcessDEPPolicy	✓	✓
KERNEL32_GetProcessDEPPolicy	✓	✓
msvcrt_memcpy	✓	✓

A.2 Search Options

In this section we present all available options for our gadget search. The options are divided in general options, which can be defined for any search function, and options which are specific for each search function.

General Options

db_file Specifies a database name to conduct the search on. If specified just the database from this parameter is used for the search.

new_zolver3 Specifies if a new, blank Zolver3 or the standard Zolver3 of GaDB should be used for all following gadget verifications.

cond Specifies a condition for an output register of the discovered gadgets. Cond must be a tuple containing the compare operand (==, !=, >, >=, <, <=) and a integer value, for example cond=('==', 0) to set a condition for the destination register or cond=('eax', '==', 0) to set a condition for another register than the destination register.

imm Specifies a list of registers which must be unchanged by the gadget, for example imm=['edx','ecx'].

CS Specifies that the gadget must start at a CS, CS=True.

EP Specifies that the gadget must start at a EP.

IJ Specifies that the gadget must start at a IJ.

IJ_value Specifies the operand of the jump. This is just a string compare and must match exactly the syntax of the jump operand, for example
IJ_value='qword ptr [rax+8]'.

IC Specifies that the gadget must start at a IC.

IC_value Specifies the operand of the call.

RET Specifies that the gadget must start at a RET.

Min_inst Specifies the minimum number of instructions contained in the gadget.

MovReg

dst Specifies the destination register, for example dst='eax'.

src Specifies the source register.

LoadReg

dst Specifies the destination register.

value Specifies the value which should be loaded to the register. If this option is specified just a load with this specific value is search for. Otherwise a generic load from the stack is searched. The value can be an integer or a string and in decimal or in hexadecimal.

Arithmetic

dst Specifies the destination register.

op Specifies the performed arithmetic operation.

sbv Specifies a BitVec as one operand. op must also be set if sbv is set.

sreg1 Specifies one register as operand.

sreg2 Specifies another register as operand. It can just sbv and one register, two registers, or a single register be set.

LoadMem

dst Specifies the destination register.

op_mem Specifies the arithmetic operation which is performed on the address.

sbv_mem Specifies a BitVec as one address operand. op_mem must also be set if sbv_mem is set.

sreg1_mem Specifies one register as address operand.

sreg2_mem Specifies another register as address operand. It can just sbv_mem and one register, two registers, or a single register be set.

ArithmeticLoad

dst Specifies the destination register.

op Specifies the performed arithmetic operation.

sbv Specifies a BitVec as one operand. op must also be set if sbv is set.

sreg1 Specifies one register as operand.

sreg2 Specifies another register as operand. It can just sbv or a single register be set.

op_mem Specifies the arithmetic operation which is performed on the address.

sbv_mem Specifies a BitVec as one address operand. op_mem must also be set if sbv_mem is set.

sreg1_mem Specifies one register as address operand.

sreg2_mem Specifies another register as address operand. It can just sbv_mem and one register, two registers, or a single register be set.

StoreMem

dop Specifies the arithmetic operation which is performed on the destination address.

dbv Specifies a BitVec as one destination address operand. dop must also be set if dbv is set.

dreg1 Specifies one register as destination address operand.

dreg2 Specifies another register as destination address operand. It can just dbv and one register, two registers, or a single register be set.

sbv Specifies a BitVec as the source.

sreg Specifies a register as the source.

op_mem Specifies the arithmetic operation which is performed on the source address.

sbv_mem Specifies a BitVec as one source address operand. op_mem must also be set if sbv_mem is set.

sreg1_mem Specifies one register as source address operand.

sreg2_mem Specifies another register as source address operand. It can just sbv_mem and one register, two registers, or a single register be set.

ArithmeticStore

dop Specifies the arithmetic operation which is performed on the destination address.

dbv Specifies a BitVec as one destination address operand. dop must also be set if dbv is set.

dreg1 Specifies one register as destination address operand.

dreg2 Specifies another register as destination address operand. It can just dbv and one register, two registers, or a single register be set.

op Specifies the performed arithmetic operation.

sbv Specifies a BitVec as one operand. op must also be set if sbv is set.

sreg1 Specifies one register as operand.

sreg2 Specifies another register as operand. It can just sbv and one register, two registers, or a single register be set.

op_mem Specifies the arithmetic operation which is performed on the source address.

sbv_mem Specifies a BitVec as one source address operand. op_mem must also be set if sbv_mem is set.

sreg1_mem Specifies one register as source address operand.

sreg2_mem Specifies another register as source address operand. It can just sbv_mem and one register, two registers, or a single register be set.

NOP

reg Specifies the register.

Call

func Specifies the fixed function call, for example func='msvcrt_memcpy'.

argx Specifies the argument of the function call, where x is the number of the argument starting at 1, for example arg1=0x1337.

Loop

The Loop operation does not have any specific options.

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

[1] Abadi, Martín ; Budiu, Mihai ; Erlingsson, Úlfar ; Ligatti, Jay: Control-flow Integrity. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2005 (CCS '05), S. 340–353. – URL http://doi.acm.org/10.1145/1102120.1102165. – ISBN 1-59593-226-7

[2] Abadi, Martín ; Budiu, Mihai ; Erlingsson, Úlfar ; Ligatti, Jay: Control-flow Integrity Principles, Implementations, and Applications. In: *ACM Trans. Inf. Syst. Secur.* 13 (2009), November, Nr. 1, S. 4:1–4:40. – URL http://doi.acm.org/10.1145/1609956.1609960. – ISSN 1094-9224

[3] Bachaalany, Elias: *Inside EMET 4.0.* 2013. – URL http://recon.cx/2013/slides/Recon2013-Elias%20Bachaalany-Inside%20EMET%204.pdf. – REcon

[4] Bhatkar, Sandeep ; Sekar, R. ; DuVarney, Daniel C.: Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. Berkeley, CA, USA : USENIX Association, 2005 (SSYM'05), S. 17–17. – URL http://dl.acm.org/citation.cfm?id=1251398.1251415

[5] Bletsch, Tyler ; Jiang, Xuxian ; Freeh, Vince W. ; Liang, Zhenkai: Jump-oriented Programming: A New Class of Code-reuse Attack. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. New York, NY, USA : ACM, 2011 (ASIACCS '11), S. 30–40. – URL http://doi.acm.org/10.1145/1966913.1966919. – ISBN 978-1-4503-0564-8

[6] *Microsoft BlueHat Prize Contest.* – http://www.microsoft.com/security/bluehatprize/, as of May 26, 2015

[7] *Bypassing Microsoft EMET 5.1 - yet again.* – http://blog.sec-consult.com/2014/11/bypassing-microsoft-emet-51-yet-again.html, as of May 26, 2015

[8] Bosman, E. ; Bos, H.: Framing Signals - A Return to Portable Shellcode. In: *Security and Privacy (SP), 2014 IEEE Symposium on*, May 2014, S. 243–258. – ISSN 1081-6011

[9] Buchanan, Erik ; Roemer, Ryan ; Shacham, Hovav ; Savage, Stefan: When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2008 (CCS '08), S. 27–38. – URL http://doi.acm.org/10.1145/1455770.1455776. – ISBN 978-1-59593-810-7

[10] Carlini, Nicholas ; Wagner, David: ROP is Still Dangerous: Breaking Modern Defenses. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA : USENIX Association, August 2014, S. 385–399. – URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini. – ISBN 978-1-931971-15-7

[11] Checkoway, Stephen ; Davi, Lucas ; Dmitrienko, Alexandra ; Sadeghi, Ahmad-Reza ; Shacham, Hovav ; Winandy, Marcel: Return-oriented Programming Without Returns. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2010 (CCS '10), S. 559–572. – URL http://doi.acm.org/10.1145/1866307.1866370. – ISBN 978-1-4503-0245-6

[12] Cheng, Yueqiang ; Zhou, Zongwei ; Yu, Miao ; Ding, Xuhua ; Deng, Robert H.: ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, The Internet Society, 2014. – URL http://www.internetsociety.org/doc/ropecker-generic-and-practical-approach-defending-against-rop-attacks

[13] Cowan, Crispin ; Pu, Calton ; Maier, Dave ; Hintony, Heather ; Walpole, Jonathan ; Bakke, Peat ; Beattie, Steve ; Grier, Aaron ; Wagle, Perry ; Zhang, Qian: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. Berkeley, CA, USA : USENIX Association, 1998 (SSYM'98), S. 5–5. – URL http://dl.acm.org/citation.cfm?id=1267549.1267554

[14] *Disarming and Bypassing EMET 5.1*. – https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/, as of May 26, 2015

[15] Davi, Lucas ; Dmitrienko, Alexandra ; Egele, Manuel ; Fischer, Thomas ; Holz, Thorsten ; Hund, Ralf ; Nürnberger, Stefan ; Sadeghi, Ahmad-Reza: MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In: *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, The Internet Society, 2012. – URL http://www.internetsociety.org/mocfi-framework-mitigate-control-flow-attacks-smartphones#overlay-context=mocfi-framework-mitigate-control-flow-attacks-smartphones

[16] Davi, Lucas ; Dmitrienko, Ra ; Sadeghi, Ahmad reza ; Bochum, Ruhr universität ; Davi, Lucas ; Dmitrienko, Ra ; Sadeghi, Ahmad reza ; Win, Marcel: Return-oriented programming without returns on ARM. 2010. – Forschungsbericht

[17] Davi, Lucas ; Sadeghi, Ahmad-Reza ; Lehmann, Daniel ; Monrose, Fabian: Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA : USENIX Association, August 2014, S. 401–416. – URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi`. – ISBN 978-1-931971-15-7

[18] *Changes to Functionality in Microsoft Windows XP Service Pack 2.* – `https://technet.microsoft.com/en-us/library/bb457151.aspx`, as of May 26, 2015

[19] *Enhanced Mitigation Experience Toolkit - EMET - TechNet Security.* – `https://technet.microsoft.com/en-us/security/jj653751`, as of May 26, 2015

[20] : *EMET User Guide.* – URL `http://download.microsoft.com/download/7/A/A/7AA570E7-92DF-4C28-BE12-E72831797666/EMET%20User%27s%20Guide.pdf`

[21] *Microsoft Security Toolkit Delivers New BlueHat Proze Defensive Technology | News Center.* – `http://news.microsoft.com/2012/07/25/microsoft-security-toolkit-delivers-new-bluehat-prize-defensive-technology/`, as of May 26, 2015

[22] Göktas, Enes ; Athanasopoulos, Elias ; Bos, Herbert ; Portokalidis, Georgios: Out of Control: Overcoming Control-Flow Integrity. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. Washington, DC, USA : IEEE Computer Society, 2014 (SP '14), S. 575–589. – URL `http://dx.doi.org/10.1109/SP.2014.43`. – ISBN 978-1-4799-4686-0

[23] Göktaş, Enes ; Athanasopoulos, Elias ; Polychronakis, Michalis ; Bos, Herbert ; Portokalidis, Georgios: Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA : USENIX Association, August 2014, S. 417–432. – URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas`. – ISBN 978-1-931971-15-7

[24] Hund, Ralf ; Holz, Thorsten ; Freiling, Felix C.: Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: *Proceedings of the 18th Conference on USENIX Security Symposium*. Berkeley, CA, USA : USENIX Association, 2009 (SSYM'09), S. 383–398. – URL `http://dl.acm.org/citation.cfm?id=1855768.1855792`

[25] Joly, N.: *Criminals Are Getting Smarter: Analysis of the Adobe Acrobat / Reader 0-Day Exploit.* September 2009. – URL http://www.vupen.com/blog/20100909.Adobe_Acrobat_Reader_0_Day_Exploit_CVE-2010-2883_Technical_Analysis.php

[26] Kornau, T.: *Return Oriented Programming for the ARM Architecture.* 2009. – URL http://www.zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf

[27] Krahmer, S.: *Return Oriented Programming for the ARM Architecture.* 2005. – URL users.suse.com/~krahmer/no-nx.pdf

[28] Moura, Leonardo ; Bjørner, Nikolaj: Formal Methods: Foundations and Applications. Berlin, Heidelberg : Springer-Verlag, 2009, Kap. Satisfiability Modulo Theories: An Appetizer, S. 23–36. – URL http://dx.doi.org/10.1007/978-3-642-10452-7_3. – ISBN 978-3-642-10451-0

[29] Moura, Leonardo de: *Z3Py Guide.* – URL http://cpl0.net/~argp/papers/z3py-guide.pdf

[30] *Descriptor HowTo Guide – Python 2.7.10rc1 documentation.* – https://docs.python.org/2/howto/descriptor.html, as of May 26, 2015

[31] Pakt: *ROPC - A Turing complete ROP compiler.* – URL https://github.com/pakt/ropc

[32] Pappas, Vasilis ; Polychronakis, Michalis ; Keromytis, Angelos D.: Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13).* Washington, D.C. : USENIX, 2013, S. 447–462. – URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas. – ISBN 978-1-931971-03-4

[33] Pelletier, A.: *Advanced Exploitation of Internet Explorer Heap Overflow (Pwn2Own 2012 Exploit).* July 2012. – URL http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php

[34] *Advanced return-into-lib(c) exploits (PaX case study).* – http://phrack.org/issues/58/4.html, as of May 26, 2015

[35] Piper, Scott: *EMET 4.1 Uncovered.* – URL http://0xdabbad00.com/wp-content/uploads/2013/11/emet_4_1_uncovered.pdf

[36] Schuster, Felix ; Tendyck, Thomas ; Liebchen, Christopher ; Davi, Lucas ; Sadeghi, Ahmad-Reza ; Holz, Thorsten: Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In: *36th IEEE Symposium on Security and Privacy (Oakland)*, Mai 2015

[37] Schuster, Felix ; Tendyck, Thomas ; Pewny, Jannik ; Maass, Andreas ; Steegmanns, Martin ; Contag, Moritz ; Holz, Thorsten: Evaluating the Effectiveness of Current Anti-ROP Defenses. In: Stavrou, Angelos (Hrsg.) ; Bos, Herbert (Hrsg.) ; Portokalidis, Georgios (Hrsg.): *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings* Bd. 8688, Springer, 2014, S. 88–108. – URL `http://dx.doi.org/10.1007/978-3-319-11379-1_5`. – ISBN 978-3-319-11378-4

[38] Security, CEA I.: *cea-sec/miasm - GitHub*. – URL `https://github.com/cea-sec/miasm`

[39] Shacham, Hovav: The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2007 (CCS '07), S. 552–561. – URL `http://doi.acm.org/10.1145/1315245.1315313`. – ISBN 978-1-59593-703-2

[40] Snow, Kevin Z. ; Monrose, Fabian ; Davi, Lucas ; Dmitrienko, Alexandra ; Liebchen, Christopher ; Sadeghi, Ahmad-Reza: Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. Washington, DC, USA : IEEE Computer Society, 2013 (SP '13), S. 574–588. – URL `http://dx.doi.org/10.1109/SP.2013.45`. – ISBN 978-0-7695-4977-4

[41] *Bugtraq: Getting around non-executable stack (and fix)*. – `http://seclists.org/bugtraq/1997/Aug/63`, as of May 26, 2015

[42] Team, PaX: *Address Space Layout Randomization (ASLR)*. 2003. – URL `https://pax.grsecurity.net/docs/aslr.txt`

[43] *Valgrind Home*. – `http://valgrind.org/`, as of May 26, 2015

[44] *Valgrind: Supported Platforms*. – `http://valgrind.org/info/platforms.html`, as of May 26, 2015

[45] *Valgrind Current Release*. – `http://valgrind.org/downloads/`, as of May 26, 2015

[46] Wartell, Richard ; Mohan, Vishwath ; Hamlen, Kevin W. ; Lin, Zhiqiang: Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2012 (CCS '12), S. 157–168. – URL `http://doi.acm.org/10.1145/2382196.2382216`. – ISBN 978-1-4503-1651-4

[47] *WineHQ - Run Windows applications on LInux, BSD, Solaris and Mac OS X*. – `https://www.winehq.org`, as of May 26, 2015

[48] *Home - Z3Prover/z3 Wiki - GitHub.* – `https://github.com/Z3Prover/z3/wiki`, as of May 26, 2015

[49] ZARDUS, Yan: *little endian arm by default - zardus/pyvex@d81bfe0 - GitHub.* – URL `https://github.com/zardus/pyvex/commit/d81bfe0ee7583d599bdd6d6c8cc091a61a42e01e`

[50] ZARDUS, Yan: *zardus/idalink - GitHub.* – URL `https://github.com/zardus/idalink`

[51] ZARDUS, Yan: *zardus/pyvex - GitHub.* – URL `https://github.com/zardus/pyvex`

[52] ZHANG, Chao ; WEI, Tao ; CHEN, Zhaofeng ; DUAN, Lei ; SZEKERES, Laszlo ; MCCAMANT, Stephen ; SONG, Dawn ; ZOU, Wei: Practical Control Flow Integrity and Randomization for Binary Executables. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy.* Washington, DC, USA : IEEE Computer Society, 2013 (SP '13), S. 559–573. – URL `http://dx.doi.org/10.1109/SP.2013.44`. – ISBN 978-0-7695-4977-4

[53] ZHANG, Mingwei ; SEKAR, R.: Control Flow Integrity for COTS Binaries. In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13).* Washington, D.C. : USENIX, 2013, S. 337–352. – URL `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang`. – ISBN 978-1-931971-03-4