Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets

Patrick Wollgast, Robert Gawlik[(✉)], Behrad Garmany, Benjamin Kollenda, and Thorsten Holz

Horst Görtz Institute for IT-Security (HGI), Ruhr-Universität Bochum, Bochum, Germany
robert.gawlik@rub.de

Abstract. Memory corruption vulnerabilities are still a severe threat for software systems. To thwart the exploitation of such vulnerabilities, many different kinds of defenses have been proposed in the past. Most prominently, *Control-Flow Integrity* (CFI) has received a lot of attention recently. Several proposals were published that apply coarse-grained policies with a low performance overhead. However, their security remains questionable as recent attacks have shown.

To ease the assessment of a given CFI implementation, we introduce a framework to discover code gadgets for code-reuse attacks that conform to coarse-grained CFI policies. For this purpose, binary code is extracted and transformed to a symbolic representation in an architecture-independent manner. Additionally, code gadgets are verified to provide the needed functionality for a security researcher. We show that our framework finds more CFI-compatible gadgets compared to other code gadget discovery tools. Furthermore, we demonstrate that code gadgets needed to bypass CFI solutions on the ARM architecture can be discovered by our framework as well.

1 Introduction

Memory corruption vulnerabilities have threatened software systems for decades. The deployment of various defense mechanisms, such as *data execution prevention* (DEP) [15], *stack smashing protection* (SSP) [10], and *address space layout randomization* (ASLR) [30] have raised the bar for reliable memory corruption exploitation significantly. Nevertheless, a dedicated attacker is still able to achieve code execution [24,31]. *Information leaks* are utilized to counter ASLR and reveal the layout of the address space, or to harvest code to build a payload just-in-time [31,41]. To circumvent DEP, attackers have added code-reuse attacks to their repertoire, such as *return-oriented programming* (ROP) [5,25,37], *jump-oriented programming* (JOP) [3,8,13], and *call-oriented programming* (COP) [7]. 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

I. Askoxylakis et al. (Eds.): ESORICS 2016, Part I, LNCS 9878, pp. 602–620, 2016.
DOI: 10.1007/978-3-319-45744-4_30

are *kBouncer* [29], *ROPecker* [9], *EMET* [17] including *ROPGuard* [18], *BinCFI* [45], and *CCFIR* [44]. 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 [20]. However, the overhead of perfect CFI is too high to be practical. Therefore, the proposed defense methods try to strike a balance 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 [12,32]. 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 bypassed with common code-reuse attacks by exploiting a vulnerability in the implementation [4,11], generic circumventions rely on longer and more complex gadgets [7,14,20,21,35] or complete functions [34]. Since the gadgets loose their simplicity by becoming longer, it also becomes harder to find specific gadgets and chain them together. To the best of our knowledge there is no gadget discovery framework available to search for CFI resistant gadgets. To be able to assess a CFI solution, it is necessary to discover code gadgets which could execute within the boundaries of the solution's CFI policies or detection heuristics. We provide a framework which is able to discover CFI resistant code gadgets or complete functions across different architectures, an increasingly important property as CFI starts to evolve on non-x86 systems as well. Notably, no search for CFI resistant code gadgets has been performed for ARM, while defenses for this architecture have already been developed [12,32]. The information provided by our framework helps security researchers to quickly prototype exploit examples to test a given CFI solution.

We opted to use an *intermediate language* (IL) for the analysis of extracted code to support different architectures without the effort to adjust the algorithms to new architectures. Because of the high architecture coverage, *VEX* is our choice for the IL. VEX is part of *Valgrind*, an instrumentation framework intended for dynamic use [42]. We harness VEX in static analysis manner [38,40] and utilize the SMT solver *Z3* [27] to translate code gadgets into a symbolic representation to enable symbolic execution and path constraint analysis. Our evaluations shows that our framework discovers 1.2 to 154.3 times more CFI-resistant gadgets across different architectures and operating systems than other gadget discovery tools. Additionally, we show that CFI-resistant gadgets are available in binary code for the ARM architecture as well, which should be taken into account by future CFI solutions.

In summary, we make the following contributions:

- We develop a framework to discover CFI and heuristic-check resistant gadgets in an architecture-independent, offline search.
- Our framework delivers semantic definitions of extracted code gadgets and classifies them based on these definitions for convenient search and utilization by a security researcher.

– To the best of our knowledge, we are the first to provide a code gadget discovery framework which reveals CFI resistant gadgets across different processor architectures, and show that CFI-compatible gadgets are also prevalent on the ARM architecture.

2 Technical Background

We begin by briefly describing code-reuse attacks, CFI approaches, and heuristic techniques proposed by recent research to defend against runtime attacks. It is important to understand the concept of CFI and the heuristic checks, as we focus on gadgets that are resistant against these approaches. Architecture independence is another issue that is tackled by our framework.

2.1 Code-Reuse Attacks

The introduction of *data execution prevention* (DEP) [15] on modern operating systems provided a useful protection against the injection of new code. To bypass DEP, 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 such as stack, heap or integer overflows, or a dangling pointer. The technique most commonly applied to reuse existing code is *return-oriented programming* (ROP) [5,37]. 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*. To be able to combine these gadgets, either a sequence of return addresses has to be placed on the stack where each address points to the next gadget, 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 as intended by the compiler. Previous work has shown that enough gadgets for arbitrary computations can be located [5,13,25] even without those unintended instructions. This is an interesting observation that especially concerns architectures with fixed opcode length. Automated tools that search for gadgets and chain them together have also been developed by past research [22,28].

Over the years, research on code-reuse attacks has proposed different variations of ROP such as *jump-oriented programming* (JOP) [3,8,13] and *call-oriented programming* (COP) [7]. JOP uses jumps instead of returns to direct the control-flow to the next gadget, and COP uses calls. 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 [24,31].

2.2 Control-Flow Integrity (CFI)

The concept of CFI was first introduced by Abadi et al. [1,2]. A program maintains the CFI property, if the control flow remains in a predefined *control-flow graph* (CFG). This predefined CFG contains all intended execution paths of the program. If an attacker redirects the control flow via code injection or code-reuse attacks to an unintended execution path, the CFI property is violated and the attack is detected. In an ideal CFG, every indirect transfer has a list of valid unique identifiers (IDs) and every transfer target has an ID assigned to it [20]. These IDs are checked before indirect transfers occur to ensure that the target is valid.

If CFI is applied to proprietary software, it becomes problematic to generate 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. Another issue with the classical CFI approach as proposed by Abadi et al. is performance. Therefore, implemented CFI solutions—also called coarse-grained approaches—typically reduce the number of IDs by assigning the same ID to the same category of targets. Examples of coarse-grained approaches are BinCFI [45] and CCFIR [44]. BinCFI uses two IDs to ensure the integrity of the CFG. The first ID defines rules for targets of *return* (RET) instructions and *indirect jumps* (IJ). The second ID combines rules for indirect control-transfers from the *procedure linkage table* (PLT) and indirect calls (ICs). Each ID has its own routine which resides inside the protected binary. Every indirect transfer is instrumented to jump to one of the two verification routines. Similar to BinCFI, CCFIR is also a coarse-grained CFI approach applied to binaries without source code. 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. An initial permutation of the Springboard at program startup additionally raises the bar for attackers.

2.3 Heuristic Approaches

In 2013 Pappas et al. [29] introduced kBouncer, an heuristic-aided approach that leverages modern hardware features to prevent code-reuse attacks. To perform CFI checks, kBouncer utilizes the *Last Branch Record* (LBR). LBR is a feature of contemporary Intel and AMD 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. Each entry in the LBR contains the source and destination address of the taken branch. By fetching some bytes just before the destination address, kBouncer can examine and enforce that every return address is preceded by a call instruction. Otherwise kBouncer reports a CFI violation. Besides the CFI enforcement, a heuristic check is performed by inspecting the last 8 indirect branches. If all entries match kBouncers gadget definition, an attack is reported. A gadget is considered as an entry if it contains up to 20 instructions and ends in an indirect control flow. The checks are invoked whenever one out of 52 critical WinAPI functions such as *VirtualProtect* or *WinExec* is called. The user-mode component hooks these critical functions and triggers the checks in the kernel mode component.

Another heuristic-aided approach is ROPecker by Cheng et al. [9], which also utilizes the LBR stack to look for gadgets in the past control flow. Additionally, the future control flow is also examined. 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. 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 by utilizing the LBR to record indirect branch instructions. The first non-gadget encountered while walking the LBR backwards terminates the search for gadgets in the past control flow. Afterwards, the future control flow is inspected for gadgets. If the combined number of encountered gadgets from the past and future control flow is above a predefined threshold, an attack is reported. The research of Cheng et al. suggests that a threshold between 11 and 16 gadgets is a suitable number.

2.4 Defeating the Countermeasures

All presented defenses against code-reuse attacks have been bypassed in recent years. While some attacks exploit vulnerabilities in a specific implementation to disable the checks [4,11], 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.

Attacks focusing on kBouncer, ROPecker, and EMET/ROPGuard [7,21,35] just have to bypass the call site (CS) checks. However, attacks against BinCFI and CCFIR [14,20] also have to take into account that ICs and IJs are limited to certain control-flow targets like function entry points (EPs). Göktaş et al. [20] 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* [35]. In their work, Schuster et al. use a loop containing an IC to chain gadgets and invoke security sensitive functions.

The heuristic policies explained in Sect. 2.3 check for chains of short instruction sequences. To evade these checks, long gadgets with minimal side effects were proposed [7,21]. If the heuristic check encounters a 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* [35].

If the unsuspicious function does not alter the global state of the program and takes enough indirect branches, the attack cannot be discovered by the heuristic checks.

3 Design and Implementation

The process of discovering suitable code gadgets which fulfill certain CFI policies consist of broadly two phases: first, appropriate code has to be discovered and extracted. Second, it is translated into the symbolic representation and can then be classified according to semantic definitions.

3.1 Gadget Discovery

Before we can describe the process of the gadget discovery, we have to define the gadgets' properties first. The definition of the gadgets is important 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.

Gadget Categories. Except minor modifications, our gadgets conform to the specifications defined by by Göktaş et al. [20] and Schuster et al. [35] as explained in Sect. 2.4. Their definitions provide sufficient properties to, for example, find complete functions for code-reuse and other CFI resistant gadgets. We used their definitions to restrict the gadget discovery, but definitions can be extended and added in modular fashion to our framework to support additional gadget types. The bounds of our gadgets have to conform to legitimate control-flow targets. Thus, they have to start at an EP or at a CS and end with an IC, IJ, or RET. The content of a gadget is defined as either an IC, a *fixed function call* (F), or other arbitrary instructions. We opted to drop IC as gadget content definition, because we can connect a gadget ending with an IC with the gadget it follows starting at the CS. Fixed function calls are beneficial in two ways. Instead of reading the address of the function from the *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 [44] can be circumvented by using gadgets containing a legitimate call to the function. As we show in Sect. 4.1, many hardcoded function calls inside of gadgets exist.

Another useful gadget is the loop gadget. Loops can be used as a *dispatch gadget* [3,35] to invoke other gadgets. Figure 1 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 gadget returns. The difference between the proposed gadget and the gadget defined for our search is that just the gray basic blocks in Fig. 1 belong to our loop gadget definition. For simplicity, loop gadgets end with an IC and start either at the CS of its IC or at an EP. Hence, the basic block beginning with the label *@skip* and

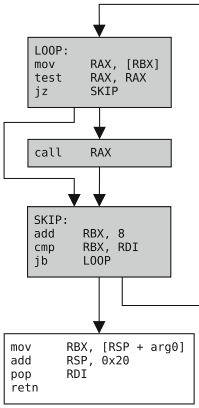

Fig. 1. Instructions of an example loop gadget. Just the gray basic blocks belong to a loop gadget by our definition.

Table 1. Gadget types supported by our framework.

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

the last basic block comprise a separate, overlapping CS-RET gadget. This has the advantage that also loop gadgets in big functions without a tailing gadget (CS-RET) are found. Additionally, one can query if another gadget starts at the end of the loop gadget. This way, when searching for *tailless* loop gadgets, we can query if code which overlaps, comprises a gadget containing another suffix than RET. All supported gadget definitions are summarized in Table 1. These definitions allow us to extract code with conditional jumps such that each single code path represents a single gadget in a path-insensitive way. As each of them is verified with symbolic execution later on, path-sensitive code gadgets arise and path-insensitive gadgets are dropped (see Sect. 3.2).

Discovering Points of Interest. To locate gadgets, our search algorithm follows the paths of the CFG. The starting points for the search algorithm are IC, IJ, and RET instructions. The algorithm to locate these points of interest works in two phases. In the first phase, addresses of all calls to fixed functions in all modules of a program of interest are extracted and kept. The set of fixed functions comprises critical imported functions which handle memory management, process and thread creation, and file I/O. These are typically very valuable for an attacker. 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 starting points.

Gadget Extraction with Depth-First Search. To retrieve the gadgets shown in Table 1, we have to traverse the CFG of every function in the binary. As we limit gadgets to single paths at first and can merge them into conditional gadgets later on in Sect. 3.2, we walk *each* 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 an adjustable maximum instruction length of the gadgets. The algorithms we use are a modification of depth-first search (DFS).

First, the basic block is located containing the gadget endpoint. Afterwards, we check if there are any calls or fixed function calls between the endpoint and the basic block's beginning. If we encounter a call, a CS gadget is created and the path traversal stops. Before a gadget is added to the gadget list, we check if a gadget with the same opcode sequence is already in that list to optionally discard or keep it for later analysis. If a fixed function call is encountered, we store the information of the fixed function call and split the current basic block. The resulting first block starts at the beginning of the original basic block and ends at the fixed function call. The resulting second block starts at the CS of the fixed function call and ends with the gadget endpoint. Thus, a CS prefixed gadget is created. Path traversal continues and on a hit of a call, the traversal stops. We check if the current basic block contains the EP. In that case, we create a EP prefixed gadget. To traverse all possible paths backwards, we keep path information and iterate over all direct preceding basic blocks.

Then, for each block, we check if the basic block has been visited before. If that is the case, a loop gadget is only added, if the traversed path starts at a CS and ends at a IC. In any case, the traversal returns if the basic block has already been visited. Afterwards, the checks for a call, fixed function call, and EP are repeated. Finally, the instruction length of the gadget is checked and updated.

3.2 Gadget Analysis

Two objectives are accomplished with the gadget analysis: first, we sort out gadgets with unsatisfiable path constraints, and second, gadgets are matched to semantic definitions and classified accordingly. This simplifies the utilization by a security researcher to find wanted functionality. To make a simplified search possible, code gadgets are transformed to a symbolic representation, executed symbolically to determine its execution contexts and clustered into semantics due to their execution effects.

Lifting Code Gadgets with Zex3 *to Raw Symbolic Representations.* Code gadgets are first translated to instructions of the VEX IL. These are mapped to Z3 expression as evaluable strings and stored offline. Thereby, most architecture-dependent peculiarities, such as stack and flags usage, are abstracted away and implicit execution effects are made explicit. The goal of this part of the framework, which we named *Zex3*, is to gather raw symbolic expression which are closely related to the structure of VEX IL instructions. Thus, registers and memory accesses are still architecture dependent.

Unification of Raw Symbolics with Zolver3. Unification of architecture-dependent registers and memory handling is done by a developed Z3 wrapper which we named *Zolver3*. The goal is to gather symbolic expressions for each gadget to be symbolically evaluable by *one* component only, namely Z3. Therefore, symbolic equations created by Zex3 are transformed into a generic format, such that register usage, memory reads and writes are adjusted. This produces a single base usable to separate symbolic representations into semantic bins and to verify satisfiability of each code gadget. As mentioned in Sect. 3.1, each gadget is a single path. Thus, symbolic execution of overlapping gadgets can yield conditional gadgets as well.

Symbolic Analysis of Code Gadgets. It is necessary for a security researcher during exploit development to rule out code gadgets which do not fulfill a desired functionality. We illustrate what we name *unsatisfiability* on a gadget with a fixed function call: at the time of compilation, it is unknown if a function call during runtime will succeed. Therefore, checks for the return value are normally inserted in the calling function by the developer. 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 constraints.

With the current level of information, a researcher is only able to search through the discovered gadgets based on their boundaries. There is no knowledge about the gadget's effects on the state of the to-be-exploited process during runtime. This makes an efficient search to chain gadgets cumbersome. Therefore, the second objective is to match every register output and every memory effect of the symbolic representation to a semantic definition. Zolver3 provides the state of every register and every memory effect based on the symbolic variables and 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 symbolic expressions of the 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 as follows.

Semantic Definitions. In the following, we present our semantic definitions. These definitions allow the researcher, combined with the search presented in Sect. 3.3, to search gadgets with specific operations performed on a specific register or memory address. One or more definitions are assigned to each gadget, based on the operations the gadget performs. When a security researcher develops a code-reuse attack, the defined gadget types are the available instruction set. Therefore, the gadget definitions must cover all necessary instructions to perform arbitrary computations. The following gadget types are necessary to accomplish this:

- MovReg: A gadget to move the content of one register to another.
- LoadReg: A gadget to load a specific content into a register.

– Arithmetic: A gadget to perform arithmetic operations between registers.
– LoadMem: A gadget to load the content of a specified memory area into a register.
– StoreMem: A gadget to store the content of a register to a specified memory area.

We add following four semantic definitions, because they represent operations which are commonly found in gadgets. Alternatives to extend the gadget definitions are discussed in Sect. 6.

– ArithmeticLoad: A gadget that loads the value from a specified memory address, performs an arithmetic operation on it, and stores the result to the destination register.
– ArithmeticStore: A gadget that extends a StoreMem gadget with an arithmetic operation
– NOP - No Operation: A gadget that keeps certain registers untouched. This is very useful during a gadget search, because untouched registers can be marked as static.
– Undefined: If none of the previous semantic definitions match the equation of the register, the register gets marked as undefined.

These gadget types are enough to create functionality containing jumps and conditional jumps. 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 developer can jump through her ROP chain. JOP and COP often use a dispatcher gadget, like the loop gadget, to invoke the gadgets of the chain. During the loop iteration one register holds a pointer into the buffer containing subsequent gadgets. Instead of the stack pointer (like in ROP), the register holding the pointer to the buffer has to be modified for jumps. Conditional jumps, however, are more complicated as they have to be accomplished by chaining several arithmetic operations [14]. But a study of exploits [31] 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. [41] come to a similar conclusion regarding the gadget definitions in their research.

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 Z3 *expression trees*, which represent the AST of Z3 expressions. Our definitions are stored as Z3 expression trees as well. Thus, we can match each symbolic operation a gadget performs against our definition and tag the gadget with one or more definitions.

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 and the instruction pointer. These are the registers that are usually accessible. We try to match every memory write to definitions recursively, because memory accesses can be nested and every new memory store adds a new layer consisting of Z3 store operations.

3.3 Semantic Search

In the previous steps, the gadgets have been discovered by their bounds and we have analyzed every effect the gadgets may have on the global state of a running process. As we want the search for the gadgets to be flexible, we perform the search on a register and memory write basis. One can specify 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 yields a lot of potential gadget candidates. In the following section, we explain methods to order the gadget candidates and to eliminate unsatisfiable gadgets.

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 typically do not perform many operations. The second criterion is to sort by the least amount of memory writes. For every unnecessary memory write, it has to be ensured that the write address is inside a writable memory area. Then the priority comes 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 criterion requires as many registers as possible to contain NOP definitions, as this limits unwanted side-effects such as overwriting a register which is set up by a previous gadget.

Gadget Verification. Our gadgets support paths containing conditional branches. The exact analysis of the conditions can be tricky. For example, a gadget is needed to load the value 0x12345678 from a specific memory address into a register. The complexity ordering algorithm may return a gadget list with a LoadMem gadget ranked first that contains a conditional jump. The pitfall is that the jump is only taken, if the LoadMem operation loads a NULL value. This renders the gadget useless to load the value 0x12345678. Therefore, invalid gadgets similar to the one described above have to be sorted out. We automatically check the constraints of the gadget list with Zolver3 until a satisfiable gadget is encountered. A search query is specified by a researcher in the language *Python*. Thereby the start/end type and the content definition of the gadget is normally specified, as well as the semantics and operations which the gadget has to fulfill.

4 Evaluation

In the following, we evaluate our prototype. More specifically, we analyze the distribution of the different gadget types across different processor architectures,

Table 2. Number of available gadgets listed by gadget start and end type, and their corresponding discovery and analysis runtime.

	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
Loops	348	443	335	464	55
Runtime (s)	12925.2	29058.7	16309.4	51259.8	4079.0

demonstrate that we can discover enough gadgets for successful exploitation, and compare our framework to existing tools. We conducted all tests for our evaluation on a 64 bit Linux system running on an Intel Xeon processor E3 with 3.3 GHz. For CFG and disassembly creation, we use IDA Pro, and VEX of Valgrind 3.9.0 is used for Zex3's translation process. Furthermore, we use pyvex's latest commit at the time of testing [39].

4.1 Gadget Type Distribution

For our evaluation, we analyzed the x86/AMD64 version of *ieframe.dll* and *mshtml.dll* of Microsoft's Internet Explorer (IE) 8.0.7601.17514. We selected these libraries as they are often used during exploitation of IE [31]. To evaluate our gadget finder on ARM, we analyzed Debian's (little-endian) libc-2.19.so, because we expect *libc* to always be loaded during exploitation of a Linux system on ARM. All gadgets residing in libc-2.19.so are in ARM mode. The gadget numbers presented in this section are the total number of gadgets, including gadgets with and without conditional branches.

Table 2 summarizes the gadget start and end type distribution. Note that the combination with the highest number 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 searching for a ROP chain. Our loop counts, also presented in Table 2, are 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 relaxed loop definitions.

It is worth noting that all functions typically used by attackers for malicious behavior are available, such as *VirtualProtect* to set memory to executable or writable, *LoadLibrary* to load a library into 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. For the x86 and AMD64 DLLs mentioned in Table 2, we found 982 gadgets with hardcoded calls to functions which allocate memory, change memory permissions, load DLLs, or perform file I/O operations.

4.2 Exploiting ARM with One CFI-Resistant Gadget

To evaluate our gadget finder on ARM, we exploit an artificial 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 only 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 write the string to memory ourselves, as it is already present in libc-2.19.so. We use the information leak to get the base address of libc-2.19.so. The address of libc-2.19.so is also required to get the address of `system()`. But at first, we have to find the gadgets to load the address of `system()` and the string `"/bin/sh"` from the buffer and call the `system()` function. These addresses are placed later on in our buffer. 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. An automatically discovered gadget that exhibits the required actions is displayed in Fig. 2. First, it loads the address of `"/bin/sh"` from our buffer to R0 via `LDR R0, [R0,#0x1C]`. And second, it loads the address of `system()` to R12 and calls R12 at the end. This way,

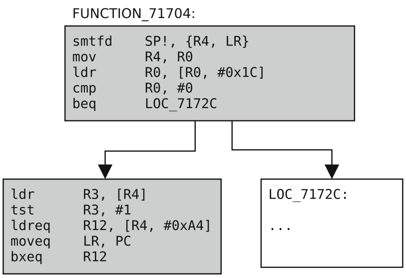

Fig. 2. An ARM gadget which loads the address of `"/bin/sh"` from the supplied buffer in R0, loads the address of `system()` from the buffer to R12, and ends with an IC of R12.

```
# Must contain 0x00000001.
Buf+0x00 => 0x00000001
...
# .rodata:00122F58 aBinSh DCB "/bin/sh",0
Buf+0x1C => 0x00122F58
...
# .text:0003B190 system
Buf+0xA4 => 0x0003B190
...
# Address of the first gadget
# Offset in buffer is dependent on freed object
Buf+0xXX => 0x00071704
```

Fig. 3. Buffer exploit data. Only addresses at the offsets `0x1C` and `0xA4`, the address for the initial control-flow transfer (`0x71704`), and `0x1` at offset `0x00` have to be set.

the objective to execute `system("/bin/sh")` is achieved with a single gadget. The buffer that we use during the exploit is shown in Fig. 3. At offset `0x00` the buffer must contain `0x1` to satisfy `TST R3,#1`. Just if this check is valid, the address of `system()` gets loaded and called.

4.3 Comparison to Other Gadget Discovery Tools

To investigate how our framework performs compared to other tools, we used ROPgadget [33], XROP [43], and IDA sploiter [23] to search for unique gadgets in *mshtml.dll*, *ieframe.dll*, and *libc-2.19.so*. ROPgadget performs a semantic search based on the disassembly of Capstone [6], while XROP and IDA sploiter perform a standard instruction search. Thereby, IDA sploiter uses IDA Pro. Hence, we can compare our framework to a tool which uses the same disassembly as input. We searched gadgets with a length of max. 30 instructions with ROPgadget and IDA sploiter, and with a max. length of five instructions in XROP, because the length cannot be adjusted. Then we dropped unaligned gadgets which these tools

Table 3. Number of unique EP and CS gadgets found by other tools in comparison to our framework. Improvement factor states the factor of more gadgets found by our tool.

Tool		CFI-resistant gadgets	Improvement factor
IDA sploiter:	libc (ARM):	0	ARM not supported
	ieframe.dll (x86):	11721	7.8
	mshtml.dll (x86):	14762	10.0
	ieframe.dll (x86_64):	14192	6.7
	mshtml.dll (x86_64):	19984	8.2
ROPgadget:	libc (ARM):	8677	1.2
	ieframe.dll (x86):	28747	3.2
	mshtml.dll (x86):	30631	4.8
	ieframe.dll (x86_64):	10479	9.1
	mshtml.dll (x86_64):	14283	11.5
XROP:	libc (ARM):	1107	9.4
	ieframe.dll (x86):	660	138.8
	mshtml.dll (x86):	957	154.3
	ieframe.dll (x86_64):	1531	62.1
	mshtml.dll (x86_64):	2479	66.1
Our framework:	libc (ARM):	10450	-
	ieframe.dll (x86):	91584	-
	mshtml.dll (x86):	147695	-
	ieframe.dll (x86_64):	95062	-
	mshtml.dll (x86_64):	163827	-

delivered, as well as non CFI-resistant gadgets. Overall, it is shown in Table 3 that our tool found 1.2 times to 154.3 times more gadgets than other tools.

5 Related Work

Code-reuse attacks have evolved from a simple *return-into-libc* [16] into a highly sophisticated attack vector. In times of DEP, Krahmer was the first to propose a method called *borrowed code chunks* technique [26]. By chaining code snippets together that end with return instructions, Krahmer showed how to perform specific operations and as a consequence bypass DEP. His work was extended by Shacham in 2007 [37], who showed that Turing-completeness can be achieved by reusing instruction sequences that end in return opcodes, thus leading to the name *Return-Oriented-Programming*. He called those sequences *gadgets*. Large code bases typically provide enough gadgets to achieve Turing-completeness.

While the first attacks targeted the x86 architecture, the concepts have been shown to be applicable on ARM [25] or SPARC [5] systems as well. ASLR [30] has been successful in stopping static ROP chains. However, its ineffectiveness has also been shown in the presence of *information leaks*. Even fine-grained re-randomization can be circumvented by the means of *just-in-time ROP* as demonstrated by Snow et al. [41]. During the attack, they harvest gadgets based on the *Galileo* algorithm introduced by Shacham et al [37]. The algorithm starts at *return* instructions and iterates backwards over a code section to retrieve gadgets that end with the return instruction. A table lookup matches their gadgets against semantic definitions. This differs from our approach as we lift only CFI-permitted code paths to an intermediate representation (VEX) having a high ISA coverage, and symbolically evaluate the gadgets to achieve a semantic binning. Schwartz et al. developed a gadget search and compiler framework to automatically generate ROP chains. They apply program verification techniques to categorize gadgets into semantic definitions [36]. However, they do not take CFI policies into account.

To aid in both the development of ROP attacks and CFI defenses, toolkits to locate suitable gadgets have emerged. Frameworks such as the one introduced by Kornau [25] or ROPgadget [33] utilize an intermediate language to abstract the underlying architecture. However, these do not locate gadgets conforming to the constraints introduced by CFI solutions. Our framework fills this gap and enables researchers to test their CFI policies on multiple architectures with only one toolkit. Closely related to our work is research which tries to measure the *gadget quality* by introducing several metrics [19]. However, these metrics are bound to an architecture, while our approach is architecture independent.

6 Discussion

The core property of our framework is the ability to quickly test CFI policies on multiple architectures. With the possibility to locate gadgets conforming to the same constraints in multiple environments, we enable researches to gain a fast overview on the security of policies. This is applicable not only to one architecture, but to all systems supported by our toolkit. As such, it speeds up evaluation allowing more time to be invested into the design of the policies. The multi-platform approach also enables to determine differences between architectures, each of which have an impact on the availability of certain gadget classes. One specific gadget class can commonly occur on one architecture, while it is nearly non-existent on another architecture, consequently not posing a risk. Allowing researchers to focus on the most relevant gadget classes for each architecture may lead to defenses that fit more to the environment. While there are other toolkits that are able to locate gadgets on ARM, our framework differs in that it allows to apply the same CFI policies to different architectures.

Limitations. At the current state, we did not include a compiler that is able to generate complete chains from the found gadgets. While we simplify the task by providing a query interface, the last step is still manual. The simplest approach would be to blindly combine chains of gadgets until one of them satisfies the constraints. However, a better solution is to combine gadgets based on a logic that translates an intermediate language written by a developer to a series of gadgets. However, this is no easy task as avoding CFI detections requires longer and more complex gadgets, which are not side-effect free. The compiler would need to account for both, the intended effects and the compensation of any side effect of the gadget. Due to the modular design, we can support additional gadget types and architectures. For instance, it is possible to extend the discovery phase to locate unintended instructions or whole virtual functions needed for a COOP-attack [34]. Another option is extending the definitions by a limit of targets for an IC of a gadget. This allows assessing fine-grained CFI defenses.

7 Conclusion

We presented a framework that not only discovers code-reuse gadgets across multiple architectures, but also locates gadgets that can be used with deployed CFI defenses. While our framework can be used in an offensive way, we deem its value for defensive research to be higher. By quickly testing CFI constraints on multiple architectures, it is possible to focus on the most relevant attack vectors and improve both the defensive capabilities and the performance. In this process, we also showed that it is possible to locate CFI-compatible gadgets not only on x86, but also on ARM. CFI research is lacking behind on mobile platform and we hope that by providing an effective evaluation tool, further work on this topic can be simplified.

Acknowledgment. This work was supported by ERC Starting Grant No. 640110 (BASTION).

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: ACM Conference on Computer and Communications Security (CCS) (2005)
2. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inform. Syst. Secur. (TISSEC) (2009)
3. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2011)
4. Bypassing Microsoft EMET 5.1 - Yet Again. http://blog.sec-consult.com/2014/11/bypassing-microsoft-emet-51-yet-again.html
5. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: ACM Conference on Computer and Communications Security (CCS) (2008)
6. Capstone - The Ultimate Disassembly Framework. http://www.capstone-engine.org/
7. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses. In: USENIX Security Symposium (2014)
8. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: ACM Conference on Computer and Communications Security (CCS) (2010)
9. Cheng, Y., Zhou, Z., Yu, M., Ding, X., Deng, R.H.: ROPecker: a generic and practical approach for defending against ROP attacks. In: Symposium on Network and Distributed System Security (NDSS) (2014)
10. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium (1998)
11. Disarming and Bypassing EMET 5.1. https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/
12. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., Sadeghi, A.: MoCFI: a framework to mitigate control-flow attacks on smartphones. In: Symposium on Network and Distributed System Security (NDSS) (2012)
13. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Return-oriented programming without returns on ARM. Technical report, HGI-TR-2010-002, Ruhr-University Bochum (2010)
14. Davi, L., Lehmann, D., Sadeghi, A.-R., Monrose, F.: Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: USENIX Security Symposium (2014)
15. Changes to Functionality in Microsoft Windows XP Service Pack 2. https://technet.microsoft.com/en-us/library/bb457151.aspx
16. Designer, S.: Return-to-Libc. Attack (1997)
17. Enhanced Mitigation Experience Toolkit - EMET - TechNet Security. https://technet.microsoft.com/en-us/security/jj653751
18. Microsoft Security Toolkit Delivers New BlueHat Prize Defensive Technology — News Center. http://news.microsoft.com/2012/07/25/microsoft-security-toolkit-delivers-new-bluehat-prize-defensive-technology/
19. Follner, A., Bartel, A., Bodden, E.: Analyzing the gadgets. In: Caballero, J., Bodden, E., Athanasopoulos, E. (eds.) ESSoS 2016. LNCS, vol. 9639, pp. 155–172. Springer, Heidelberg (2016). doi:10.1007/978-3-319-30806-7_10

20. Göktaş, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: IEEE Symposium on Security and Privacy (2014)
21. Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G.: Size does matter: why using gadget-chain length to prevent code-reuse attacks is hard. In: USENIX Security Symposium (2014)
22. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: USENIX Security Symposium (2009)
23. IDA Sploiter. https://thesprawl.org/projects/ida-sploiter/
24. Joly, N.: Criminals are getting smarter: analysis of the adobe acrobat/reader 0-day exploit, September 2009. http://web.archive.org/web/20141018060115/, http://www.vupen.com/blog/20100909.Adobe_Acrobat_Reader_0_Day_Exploit_CVE-2010-2883_Technical_Analysis.php
25. Kornau, T.: Return-oriented programming for the ARM architecture (2009). http://www.zynamics.com/downloads/kornau-tim-diplomarbeit-rop.pdf
26. Krahmer, S.: x86–64 buffer overflow exploits and the borrowed code chunks exploitation technique (2005). http://users.suse.com/~krahmer/no-nx.pdf
27. Microsoft-Research. Z3: Theorem Prover (2014). http://z3.codeplex.com/
28. Pakt. ROPC - A Turing Complete ROP Compiler. https://github.com/pakt/ropc
29. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: USENIX Security Symposium (2013)
30. PaX Team. Address Space Layout Randomization (2001). https://pax.grsecurity.net/docs/aslr.txt
31. Pelletier, A.: Advanced Exploitation of Internet Explorer Heap Overflow (Pwn2Own 2012 Exploit), July 2012. http://web.archive.org/web/20141005134545/, http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php
32. Pewny, J., Holz, T.: Control-flow restrictor: compiler-based CFI for iOS. In: Annual Computer Security Applications Conference (ACSAC) (2013)
33. ROPgadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/
34. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., Holz, T.: Counterfeit object-oriented programming: on the difficulty of preventing code-reuse attacks in C++ applications. In: IEEE Symposium on Security and Privacy (2015)
35. Schuster, F., Tendyck, T., Pewny, J., Maaß, A., Steegmanns, M., Contag, M., Holz, T.: Evaluating the effectiveness of current Anti-ROP defenses. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 88–108. Springer, Heidelberg (2014)
36. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: exploit hardening made easy. In: USENIX Security Symposium (2011)
37. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM Conference on Computer and Communications Security (CCS) (2007)
38. Shoshitaishvili, Y.: Pyvex - GitHub. https://github.com/zardus/pyvex
39. Shoshitaishvili, Y.: Pyvex@d81bfe0 - GitHub. https://github.com/zardus/pyvex/commit/d81bfe0ee7583d599bdd6d6c8cc091a61a42e01e
40. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In: Symposium on Network and Distributed System Security (NDSS) (2015)
41. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.-R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: IEEE Symposium on Security and Privacy (2013)

42. Valgrind Home. http://valgrind.org/
43. XROP - Tool to generate ROP gadgets for ARM, x86, MIPS and PPC. https://github.com/acama/xrop
44. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control-flow integrity and randomization for binary executables. In: IEEE Symposium on Security and Privacy (2013)
45. Zhang, M., Sekar, R.: Control-flow integrity for COTS binaries. In: USENIX Security Symposium (2013)