Master's Thesis

# Automatic Stub Generation for Dynamic Symbolic Execution of ARM binary

1910407 Nguyen Thi Van Anh

| | |
|---|---|
| Supervisor | Prof. Mizuhito Ogawa |
| Main Examiner | Prof. Mizuhito Ogawa |
| Examiners | Prof. Satoshi Tojo |
| | Associate Prof. Nao Hirokawa |
| | Associate Prof. Daisuke Ishii |

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

September, 2021

## Abstract

In a recent survey, it is reported that the number of IoT attacks in 2020 has been rose to 66% compared to the previous year. Due to the vast number of IoT devices with weak authentication and lack of security protection ability, IoT devices have become easy targets for exploitation. With the rapidly increasing number of IoT devices, even when the computing power of each IoT device is low, they can collaborate for making large-scale attacks (e.g., DDoS, and crypto-jacking). Therefore dealing with IoT malware has become more and more urgent and necessary. There are many existing approaches for analyzing malware including static analysis and dynamic analysis. However, disassemblers can be easily cheated by obfuscation techniques, and dynamic analyses can be detected and prevented by VM awareness, anti-debugging, and trigger-based behavior. To bypass obfuscation techniques, especially for indirect jump resolution, it is necessary to apply Dynamic Symbolic Execution to reconstruct malware execution trace.

ARM is a processor family, which is most popularly used for IoT devices. In previous work, a DSE tool for ARM Cortex-M - CORANA was preliminarily built by extracting ARM formal semantics from natural language descriptions. However, malware frequently runs in both user mode and kernel mode, and they also connect with other external systems (e.g, C&C servers, or other end-user devices). The result of the external calls might affect deeply the analysis, especially in the presence of anti-debugging or trigger-based code. An approach is to prepare API Stub of system calls to interact with the external environment. Automatically generation of API Stub can help the symbolic execution engine produce meaningful execution traces while reducing the cost of manual API implementation.

We target ARM on Linux, where its API specification is available in the Linux Manual Page. This thesis proposed an approach to systematically generate Linux API Stub from the C library function interface description. For each library function, first, we apply pattern matching to retrieve the information on its name, parameters, and return type. After that, we using predefined type conversion rules to statically decided on the types of parameters. Then, three kinds of Java classes of the target function which are structure definition class, interface-mapped class, and API Stub class are generated. By 1659 collected API descriptions, we are able to produce 1129 API Stubs and 267 structure definition classes. We also proposed using serialization to handle the execution of multiple processes. To demonstrate the ability of CORANA after adding the generated API Stubs to support external calls (CORANA/API), we performed a detailed analysis on a Mirai sample using the tool. The result shows that CORANA/API is able to trace real-world IoT malware samples and is resilient against several obfuscation techniques, which overcomes the existing DSE tools, e.g., angr.

***Keywords***— IoT malware, malware analysis, Dynamic Symbolic Execution, ARM Cortex-M

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Symbolic execution [1] is a classical method in software engineering, especially aimed at test data generation for the control flow coverage. There are lots of tools for high-level programming languages, such as C/C++ and Java are developed. For instance, KLEE [2], CUTE [3], and JPF-SE [4] are just examples from a long list.

Recently, the tools for symbolic execution on binary code gradually increases, e.g., McVeto[5] is an early static symbolic execution example, and from around 2015, several dynamic symbolic execution tools, miasm[6], Mayhem[7], Klee-MC[8], CoDisasm[9], S2E[10], angr[11], and BEPUM[12], become available. They mostly set the first target as x86.

Different from high-level programming languages, binary code has

- no syntax, i.e., no grammar constraints on the order of instructions,
- no distinction on data and code, i.e., everything is a binary sequence, and
- often rigid specification (even described by natural languages).

and its control flow graph is not directly obtained, where the control flow graph of a high-level programming language is obtained for free during the parsing.[1] The need to directly analyze binary code may be on *legacy code*, which will have no available source code, and *malware*. The control flow graph construction, and equivalently the disassembly become a challenge when malware adopts the obfuscation techniques. The syntactic disassembler, e.g., IDApro[15] and CAPSTONE[16], are easily cheated by the obfuscation techniques, especially combined with indirect jumps and self-modification to confuse the next control point. This is also difficult to solve by dynamic analyze when VM awareness, anti-debugging, and/or trigger-based behavior exist. Dynamic symbolic execution on binary code is especially useful for such a challenge [17], [18].

Note that most binary code is not self-contained. The execution environment of binary is on an OS, which has the *privillege hierarchy* and the *API (system library functions)* to request higher privileged tasks and/or common system tasks. We can regrad three levels of binary code actions.

[1] For object-oriented languages, an inter-procedural control flow like a call graph requires a *points-to* analysis [13], [14].

1. *Instruction level*, the same level execution, which can be rearded as the *state transition.*

2. *System level*, the higher privilleged tasks by external function calls.

3. *Inter-system level*, communication with external systems.

Typically, malware consists of three levels of techniques.

1. *Obfuscation*, e.g., indirect jump, self-modification, opaque predicates, antidebugging and trigger-based behavior.

2. *Infection*, e.g., heap/stack overrun, and brute-force attack.

3. *Malicious action*, e.g., information leakage, and remote control.

Among them, the obfuscations without using API calls are at the instruction level execution. The obfuscation techniques with using API calls (e.g., anti-debugging and trigger-based behavior) [19] and the typical infection techniques are at the system level execution. The malicious actions are mostly at the inter-system level via the internet. Some infection techniques to scan surrounding devices are also at the inter-system level.

Another dimension of malware is, either PC malware or IoT malware.

- *PC malware* focuses on mostly x86 with typical OSs, e.g., Windows, Linux, and macOS. To bypass anti-virus protection, it often uses heavy obfuscations, typically introduced by various *packers*.

- *IoT malware* is often naive compared to PC malware. However, the target platforms vary a lot. e.g.,

  - MPU (Micro Processor Unit) is mostly 32/64 bits, e.g., x86, ARM Cortex-A, MIPS32/64, MC68000, Sparc, and PowerPC.
  - MCU (Micro Controller Unit) is either 4/8/16/32 bits, e.g., ARM Cortex-M, Z80, PIC, AVR, MSP430 (TI), and various own MCUs (like Runesas, NEC, Hitachi, and Mitsubishi).

Thus, there are demands to develop binary DSE tools not only on x86, but on various platforms.

For the instruction level, there are two approaches to cover various platforms.

- Use of intermediate assembly language (IAL), e.g., VEX, LLVM, and BAP (used in angr, Klee-MC, and MAYHEM, respectively). Pro is that one can apply a popular disassembler like CAPSTONE to translate into an IAL, but as the opposite side of a coin, Con is that a syntactical disassembler can be easily cheated since the translation is equivalent to disassembly.

- Automatic extraction of formal semantics of each instruction set and automatically generate an individual binary DSE tool.

Our approach is the latter, e.g., BEPUM for x86 [20], CORANA for ARM [21], and SyMIPS [22] for MIPS, based on the observations that

- Malware rarely uses the floating-point instructions. Therefore, we can avoid FPU.
- Malware is often a user-level process and even if it forks processes, mostly they do not communicate with each other, i.e., no synchronization. Therefore, we can simplify the model as a sequential state transition system on the environment model consisting of *memory, stack, registers, and flags*.
- Each instruction set often has a rigid specification (though in natural language) as a developer's manual. It describes the action of instruction and the flag updates. The former is sometimes associated with the pseudo-code description and the latter is easily processed by the similarity analysis to decide binary actions (i.e., update a flag or not).

For the system level, since DSE cannot trace all the system behavior, we need to model the system environment or the stubs to simulate the system function calls. Of course, we can try manually to prepare a more accurate simulation, but OSs are not only Windows, Linux, and macOS, especially small IoT devices. Even if we restrict ourselves to typical OSs, they already have lots of library functions. For instance, we have found more than thousands of APIs for Windows [23] and Linux (Chapter 7.1). There are three approaches to cover various platforms.

- Klee-MC abstracts the environment as a model [2]. However, this is a quite rough approximation and rarely achieves enough accuracy.
- MAYHEM [7] and angr [11] fuse the concrete execution and symbolic execution of the program by allowing switching between GDB debugger and their symbolic engine.
- BEPUM [12] and Syman [24] prepare the *API Stub* of system calls to execute it in the external environment to obtain an exact snapshot of the environment update. An API Stub requires an interface as a proxy to invoke the native function, retrieves the return value, and updates the environment after the system/library function call.

Our approach is the last. To handle more than thousands of API functions, we follow an automatic generation of API stubs of BEPUM for Windows API [23], but applying to Linux to extend CORANA. It is based on the observation that

- For API stubs, it is enough to collect the interface information (e.g., the type information to avoid errors) instead of the full semantics.
- Similar to the instruction set, OS library functions have rigid specifications (even in natural language) and often follow to the naming convention.

This approach helps the symbolic execution engine produce meaningful execution traces while reducing the cost of manual API implementation.

This thesis proposes an extension *CORANA/API* of DSE tool *CORANA* with the API stubs of Linux system functions, i.e.,

- A method for automatic generation of API Stubs for ARM on Linux systems. The same approach can be extended to multiple platforms that run on Linux systems.
- The API generation system is able to produce 1129 API stubs from 1659 collected API descriptions, in addition, are 267 structure definition classes.

With CORANA/API, we analyze IoT malware on ARM/Linux, e.g., Mirai. The result shows that CORANA/API is able to trace real-world IoT malware samples and is resilient against several obfuscation techniques, which overcomes the existing DSE tools, e.g., angr.

**DSE tool CORANA for ARM-Cortex M.** ARM is a processor family, which is most popularly used for IoT devices. It has many variations call Cortex (e.g., Cortex-A, Cortex-M, Cortex-R). we focus on the Cortex-M series since it was popular for IoT devices. In previous work, a DSE tool for ARM CortexM - CORANA was preliminarily built by extracting ARM formal semantics from natural language descriptions [21]. It considers the program execution as a sequential user-mode process. However, this is rarely the case in malware when malware frequently runs in both user mode and kernel mode, and they also connect with other external systems (e.g, C&C servers, or other end-user devices). To fully symbolically execute a program, we have to consider three different labels: instruction, external call, and external system. Our prior work in [21] only focused on the dynamic symbolic execution of ARM binaries on the instruction label, making the DSE tool - CORANA encounter problems when the binary contains library function calls and system calls.

**Handling fork.** Malware is also frequently used `fork()` to create several processes. Therefore, a method to generate API Stubs for external calls automatically and handle multiple concurrent processes is much needed to complete the framework for the DSE tool. Our ultimate goal is to propose a method to semi-automatically generate API Stub for ARM on Linux from the description. Moreover, we expect to handle the concurrent processes by serialization with the assumption that processes are executed without overlapping and do not interact with each other.

**Linux API specification.** We target ARM on Linux, where its API specification is available in the Linux Manual Page. Manually preparing API Stub for each external call to the Linux system is a tedious and time-consuming task. The good news is that we can automatically generate API Stubs because of the following reasons:

- We can invoke a native function in Java by defining the JNA interface that is equivalent to the C library function interface.

- Only information on the C library function interface is sufficient enough to generate API Stub.
- The data transferred on demand between the emulated environment and the actual environment complies with certain conventions (e.g., passing parameters rule, memory allocation rule).

The systematic generation of the Linux API Stub from the C library function interface description proceeds as follows. For each library function, first, we apply pattern matching to retrieve the information on its name, parameters, return type, and other related details. After that, we using predefined type conversion rules to statically decided on the types of function parameters. Then, three kinds of Java classes of the target function which are structure definition class, interface-mapped class, and API Stub class are generated.

# Related work

Although symbolic execution was introduced in the 70s, it has regained interest since the 2000's due to the advances in constraint solving and the availability of high-performance computing systems. Early works on symbolic execution tools mostly focus on the source code level ([2], [3], [25]). Recently, the target of many symbolic execution tools have been extended to binary code (e.g., McVeto[5], CoDisasm[9], MAYHEM[7], KLEE-MC[8], BE-PUM[12], angr[11], CORANA[21]). Most of them using existing disassemblers or binary lifters to translate binary code to an intermediate binary representation such as LLVM in KLEE-MC, VEX in angr, and BAP in MAYHEM. This approach ensures the symbolic execution tools can analyze binaries from multiple architectures (e.g., x86-64, x86, ARM, MIPS) without preparing execution engines for specific architectures. However, this method does not perform well in the presence of obfuscated code such as indirect jumps, self-modifying code, and over-lapping instructions. To overcome this limitation, some works have directly interpreted binary using a one-step disassembler. McVeto[5] and BE-PUM[12] are both directly apply symbolic execution on x86 binaries. However, McVeto resolves indirect jumps by analyzing the possible candidates statically and use an SMT solver to solve the path constraints to decide the jump destination, whereas BE-PUM decides the destination of indirect jumps by performing concolic testing. The strategy employed by McVeto mostly only works for compiled binary code. On the other hand, BE-PUM execution engine requires a huge effort to implement the binary emulator. Therefore a method to automatically extract the formal semantics of binary instructions is much needed.

Previously, the semantic extraction of the x86 specifications has been investigated to extend BE-PUM[20]. Later, the semantics of ARM and MIPS are automatically extracted to built CORANA[21] and SyMIPS[22], respectively. The extraction of ARM instructions' semantics is more challenging since there exists only natural language descriptions for ARM, without the pseudocode description like x86 and MIPS. Across 6 variants of ARM

Cortex-M, 692 instructions among 1039 collected specifications have been successfully extracted. However, the work in [21] focus on the execution of ARM Cortex-M binaries on the instruction label, without considering the external call label. Therefore in this thesis, we investigate the method to automatically generate API Stub for external calls to complete the DSE tools for ARM binaries run on Linux. Our idea of using API Stub to handle external calls follows the previous work that automatically generates Windows API Stubs for x86 binaries in BE-PUM [23]. There are some fundamental differences between our work and the automatic generation of Windows API Stubs in [23]:

- We focus on the function library calls of ARM binaries run on Linux, instead of Windows API on x86.
- The Windows API descriptions from Microsoft Developer Network (MSDN) are well documented and contains description in natural language for each parameter, while for Linux APIs we can only obtain the function interface and its source code from the GNU C Library.
- To decide variable types in the API Stub, we based on some observed conventions to statically determine the type, instead of using Machine earning techniques since there is no detailed description of the parameters.
- We proposed serialization to deal with multi-processes since multi-processing is a common property of Linux's programs.

# Thesis Outline

This thesis is composed of 8 chapters. Chapter 1 is the introduction, the next chapters are summarized as follows:

- **Chapter 2** presents some obfuscation and infection techniques that are often employed by malware to bypass analysis and detection. It also presents Dynamic Symbolic Execution (concolic testing) for binary analysis and some notable tools for analyzing ARM binaries.
- **Chapter 3** introduces the formal semantics of ARM and their extraction process from the natural language description to preliminarily built the Dynamic Symbolic Execution tool for ARM binary on the instruction label - CORANA.
- **Chapter 4** discusses the interaction between the symbolic execution engine of a program's user process and the OS environment, which is Linux/Unix OS in this case. This chapter presents our choice of approach to handle external calls by implementing API Stubs running the calls in the actual environment. We also handle the execution of multiple processes by serialization.
- **Chapter 5** mentions the information extraction of the API description. After extracting API elements from the description, we present a statistical method to

transform and represent those elements (e.g., function name, parameter names, types) in JNA.

- **Chapter 6** explains the automated API Stub generation processes.
- **Chapter 7** shows the result of the API Stub generation. We analyze the execution trace of CORANA after adding the support for handling external calls and multiple processes against some obfuscated code. Finally, a detailed analysis of Mirai, a variant of IoT malware, is represented.
- **Chapter 8** summarize the main contributions of the thesis and its current limitations. After that, several future works are described to show potential directions to improve and extend our study.

# Chapter 2

# DSE for Malware Analysis

## 2.1 Malware analysis difficulties

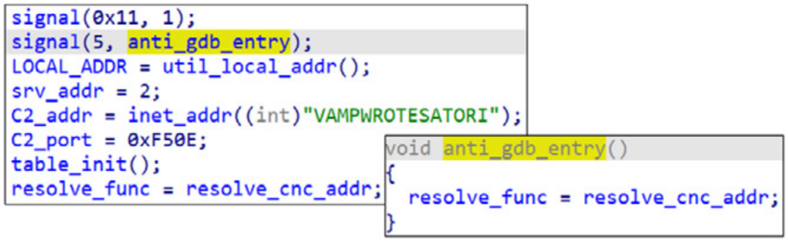
In this section, we discussed typical obfuscation (e.g., anti-debugging, opaque predicate, self-modification, indirect jump) and infection techniques (e.g., stack overflow, brute-force attack) which are widely employed by malicious software.

### 2.1.1 Obfuscation technique

Malware often includes various obfuscation techniques to bypass anti-virus software and resit the reverse engineering e.g., indirect jump, opaque predicate, and self-modification for cheating disassemblers and static analysis, and VM awareness, anti-debugging and trigger-based behavior to hide malicious intention from virtual machine or sandbox.

**Anti-debugging**

Anti-debugging techniques are used to check whether the program is running on a debugger and change its behavior if in a debugger. There are several anti-debugging methods such as breakpoint detection which set false breakpoints or checking `ptrace`.



```
signal(0x11, 1);
signal(5, anti_gdb_entry);
LOCAL_ADDR = util_local_addr();
srv_addr = 2;
C2_addr = inet_addr((int)"VAMPWROTESATORI");
C2_port = 0xF50E;
table_init();
resolve_func = resolve_cnc_addr;
```

```
void anti_gdb_entry()
{
  resolve_func = resolve_cnc_addr;
}
```

Figure 2.1: Example of anti-debugging techniques

Figure 2.1 show the anti-debugging technique employed in Mirai, a popular IoT malware. The malware registers function that returns the real C&C address as a handler for `SIGTRAP(5)` signal. If debugging is not involved, then the malware normally executes the handler function that was previously registered to obtain the real C&C address. If running in the debugging environment, fake C&C address will eventually acquired. Another popular technique is debugger detection by using library functions (e.g., `isDebuggerPresent()` in *Windows* and `ptrace()` in *Linux*). `ptrace()` can be used since a program can be traced by only one process at a time and almost all debuggers use `ptrace()` (including *GDB*).

### Opaque-predicate

Opaque predicates are constant predicates that always true or always false. Opaque predicates can be used to make static analysis more difficult by creating extra control flow, or to add deadcode. Figure 2.3 shows an simple example of opaque predicates, the instructions at `label_1` are never jumped to, but modern disassemblers like IDA Pro or Ghidra cannot determine that property. This leads to inaccurate disassembly.

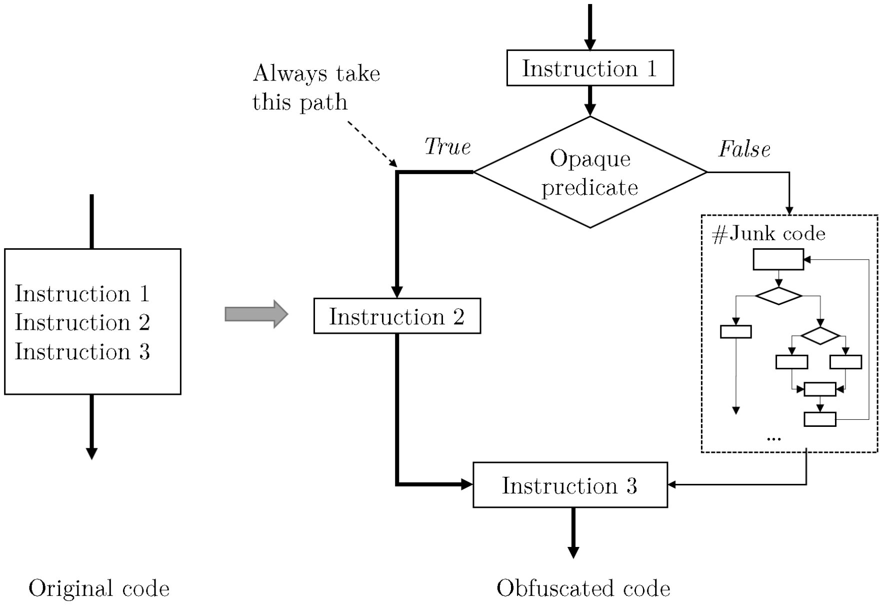
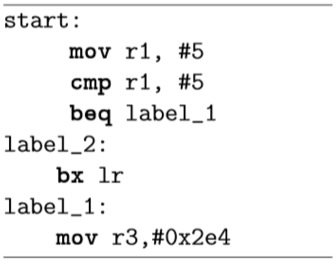


Figure 2.2: Obfuscated code with opaque predicates



```
start:
     mov r1, #5
     cmp r1, #5
     beq label_1
label_2:
    bx lr
label_1:
    mov r3,#0x2e4
```

Figure 2.3: Example of opaque predicate code

### Self-modification

Self-modifying code [26] is an effective method to hide the real behaviour of the program since only at run-time that the true code is exposed through transformation. Self-modifying code can be used for protecting property right or concealing malicious behaviour to bypass detection. The key mechanism is that code instructions will be replaced during the program execution. Self-modifying code poses a great challenge for *static analysis*. Two main implementations of self-modifying codes are *packing* and using *self-modifying instruction*.

Originally, *packing* applies compression to make executable size smaller and now it is used to hide the executable content. A packed binary contains a unpacking code section, which decrypts the packed file back to the original file at runtime.

*Self-modifying instructions* are used in the case where code is considered as data that can be read and written. The program below shows an example of self-modifying code in the C language [1].

```c
#include <stdlib.h>
char code[]= "\xe8\x1f\x00\x00\x00\x58\xc6\x40\x0e\xeb\x43\xc6\x40\x06\xeb"
  "\xc6\x40\x07\x08\x50\xf5\x42\x51\x4b\xb8\x01\x00\x00\x00\xbb"
  "\x00\x00\x00\x00\xcd\x80\xeb\xdf";
int main() {
  int (*func)() = (int (*)()) code;
  (int)(*func)();
  return EXIT_SUCCESS;}
```

After disassembling this program, the actual opcodes are not shown and hidden in the data section.

Assembled Code

```
401f85  ENDBR64
401f89  push rbp
401f8a  mov rbp, rsp
401f8d  sub rsp,0x10
401f91  lea rax,[code]
401f98  mov qword ptr[RBP+local_10],rax
401f9c  mov rdx,qword ptr[RBP+local_10]
401fa0  mov eax,0x0
401fa5  call rdx=>code
401fa7  mov eax,0x0
```

Actual Code

```
call    0x24
pop     eax
mov     BYTE PTR [eax+0xe],0xeb
inc     ebx
mov     BYTE PTR [eax+0x6],0xeb
mov     BYTE PTR [eax+0x7],0x8
push    eax
cmc
inc     edx
push    ecx
dec     ebx
mov     eax,0x1
mov     ebx,0x0
int     0x80
jmp     0x5
```

[1] https://github.com/JonathanSalwan/binary-samples/blob/master/anti-disassembler/Linux/self-modifying_code/self-modifying_code-i386.c

### Indirect jump

Indirect jump stores the jump target in a register or memory instead of a value of the target address. In contrast to a direct jump, which statically points to a location in the program, the target of an indirect jump is dynamically decided. Since static analyses are easily confused by arithmetic operations, it is difficult to resolve indirect jumps using static analyses. At 0x9a50 in the example, the jump destination of the branching instruction bx lr is not directly specified but depends on the information stored at register lr.

```
0x9a48      sub  sp, fp, #12
0x9a4c      ldm  sp,fp, sp, lr
0x9a50      bx   lr
0x9a54      mov  ip, sp
0x9a58      push {fp, ip, lr, pc}
```

Figure 2.4: Example of indirect jumps

We have discussed some popular obfuscation techniques that exist and are frequently used by malicious software. Dynamic symbolic execution can be used to overcome the limitation of both dynamic and static analysis [17], [27].

## 2.1.2 Infection technique

Many malware aim to exploit software security holes or operating system vulnerabilities to spread and simultaneously infect million of computers. Characterize infection techniques of malware can play an important role in detecting and preventing malicious programs.

### Stack (heap) overflow exploitation

Buffer overflow [28] is probably one of the most well-known form of security vulnerability. Overflows take advantage of data stored in the stack or the heap. The goal is to inject instructions into a buffer and overwrite the return address to take the control of the program's execution flow and gain root privileges.

The stack locates a memory area that used as temporary data storage for executing functions and local variables. There are several ways attackers can manipulate the system by exploiting stack buffer overflow. A common technique is overwriting the stack frame return address to continue execution at a location contains injected malicious shell code. This scenario is described in Figure 2.5. When the func() function finishes running, the current function returns to the caller, which is the main() function. To exploit the stack-based vulnerability, the attacker sends to the program some malicious contents consists of: (i) a chain of NOP instructions, (ii) a new return address that points to a specified location and (iii) some malicious code (usually shellcode) in the middle of the NOP chain. The buffer overflow occurs will cause the program jump to the sequence of NOP bytes. The system ignores the `NOP` instructions and read the next bytes until encounters the injected shellcode. The shellcode now is executed in the operating system shell, give the attacker full access to the system.

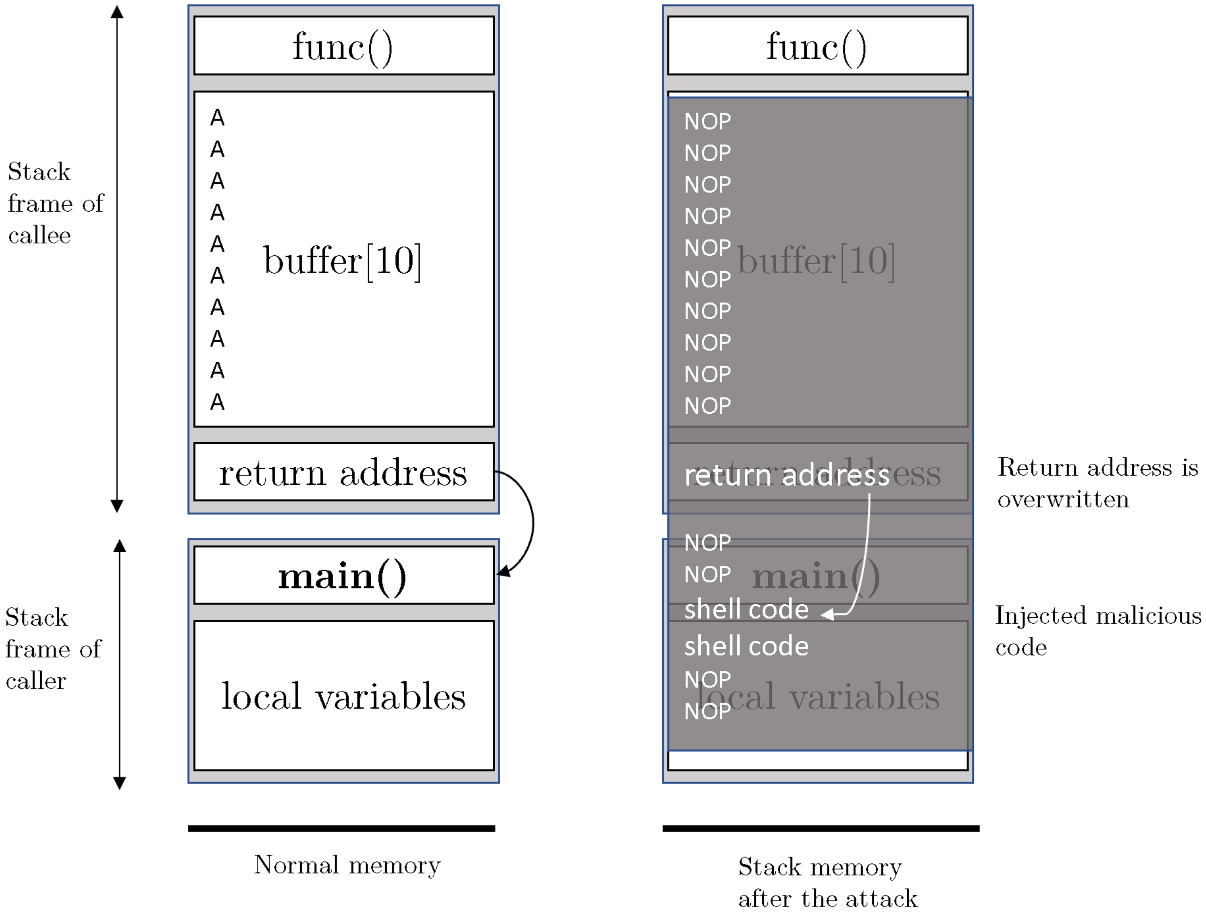


Figure 2.5: Stack overflows attack

In addition to the previously described technique, attackers can also manipulate the system by other techniques such as overwriting and executing an exception handler or more advanced methods.

### Brute-force attack

Despite of its simplicity, brute-forcing is still one of the most common methods used for attempting to gain access to systems and execute malware especially on IoT devices. Brute-force attacks are trial and error methods guessing the credential of the target system to gain the system's root privilege. Some attacks will loop through large combinations of usernames and passwords until finding the correct one, others will try a predefined set of usernames and passwords on as many systems as possible.

Many malware have used brute-force to attack Windows machines through guessing SMB (Server Message Block) or RDP (Remote Desktop Protocol) passwords, which are connection protocols of Windows computers. One of the most popular ransomware WanaCry exploited the vulnerability of SMB to infect millions of computers.

A brute-force way is simple but incredibly effective to infect IoT devices since most IoT devices still use their default credentials for authentication, or do not use any credentials. IoT devices, such as routers, printers, televisions, and cameras can be found everywhere and have been continuously growing for years. With the weak protection, millions of highly vulnerable devices connecting to the Internet have become easy targets for attackers to be recruited as a part of a botnet to create DDoS attacks or cryptocurrency mining. IoT devices are mostly attacked with the default credentials on SSH, Telnet or HTTP ports. The code below is from the leaked source code of Mirai, a popular IoT malware variant, which shows a part of the predefined username and password set that the malware used to try to gain access control of the target devices.

```
add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x41\x11\x17\x13\x13", 10); // root xc3511
add_auth_entry("\x50\x4D\x4D\x56", "\x54\x4B\x58\x5A\x54", 9);      // root vizxv
add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x4B\x4C", 8);      // root admin
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7);  // admin admin
add_auth_entry("\x50\x4D\x4D\x56", "\x1A\x1A\x1A\x1A\x1A\x1A", 6);  // root 888888
add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x4F\x4A\x46\x4B\x52", 5);  // root xmhdipc
add_auth_entry("\x50\x4D\x4D\x56", "\x46\x47\x44\x43\x57", 5);      // root default
    ...
```

Listing 1: The default credentials Mirai used for brute-forcing [29]

## 2.2 Dynamic Symbolic Execution

Symbolic execution [1] is a technique to explore possible states of a program. Instead of running on a specific input as in concrete execution, symbolic execution can take in symbolic values and explore multiple possible paths of a program. For each path, the *execution engine* maintains a first-order formula that describes the condition satisfied the path.

The interence rule of Hoare Logic is the Hoare triple:

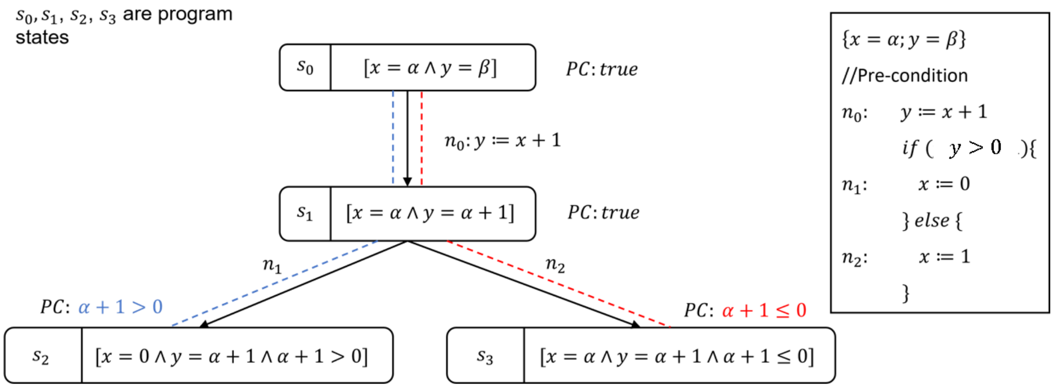
$$\{Precondition\} \quad Command \quad \{Postcondition\}$$

From the Hoare Logic viewpoint, symbolic execution computes new states of the program by applying Hoare logic inference rules on the formulae that describe the states. Consider the following program:

```
$\{x = \alpha \wedge y = \beta\}$
y := x + 1;

if (y > 0) {
      x := 0;

      } else {
      x := 1;

}
```

```
$\{x = \alpha \wedge y = \beta\}$
y := x + 1;
$\{x = \alpha \wedge y = \alpha + 1\}$
if (y > 0) {
      x := 0;
      $\{x = 0 \wedge y = \alpha + 1 \wedge \alpha + 1 > 0\}$ (1)
} else {
      x := 1;
      $\{x = 0 \wedge y = \alpha + 1 \wedge \alpha + 1 \leq 0\}$ (2)
}
Post−Cond: $\{(x = 0 \wedge y = \alpha + 1 \wedge \alpha + 1 > 0)$
           $\vee(x = 0 \wedge y = \alpha + 1 \wedge \alpha + 1 \leq 0)\}$
```

By applying appropriate inference rule at each execution step, the formulae that describe the states are generated. Symbolic state and path condition are two properties that corresponding to those formulae. A path condition describes the pre-condition of the execution path to the current statement, starting from the program entry. For example, symbolic state $[x \Rightarrow \alpha, y \Rightarrow \alpha+1]$ and path condition $\alpha+1 > 0$ correspond to the formula that is true at (1). Similarly, formula at (2) have symbolic state $[x \Rightarrow \alpha, y \Rightarrow \alpha + 1]$ and path condition $\alpha + 1 \leq 0$.


$s_0, s_1, s_2, s_3$ are program states
$s_0$
$[x = \alpha \wedge y = \beta]$
$PC: true$
$n_0: y \coloneqq x + 1$
$s_1$
$[x = \alpha \wedge y = \alpha + 1]$
$PC: true$
$n_1$
$n_2$
$PC$: $\alpha + 1 > 0$
$PC$: $\alpha + 1 \leq 0$
$s_2$
$[x = 0 \wedge y = \alpha + 1 \wedge \alpha + 1 > 0]$
$s_3$
$[x = \alpha \wedge y = \alpha + 1 \wedge \alpha + 1 \leq 0]$

```
$\{x = \alpha; y = \beta\}$
//Pre-condition
$n_0$:    $y \coloneqq x + 1$
          if (  $y > 0$  ){
$n_1$:       $x \coloneqq 0$
          } else {
$n_2$:       $x \coloneqq 1$
          }
```

Figure 2.6: Examples of symbolic execution

An SMT solver (e.g., Z3[30]) is often used to verify whether the explored path is *feasible*, i.e, check if the path condition is *satisfiable*. Classical (static) symbolic execution can face difficulty when dealing with external code not traceable by the executor, complex constraints or indirect jumps in binary code. A fundamental idea to make symbolic execution more feasible in practice is using dynamic symbolic execution (DSE).

- **Static symbolic execution.** The next instruction candidates are statically detected. Then each destination is checked by using a SMT solver to decide the feasibility of each newly create path condition.

- **Dynamic symbolic execution (or concolic testing).** The feasibility of next instruction is checked by testing with an instance from the precondition, which requires a binary emulator.

When dealing with indirect jump, it is insufficient to use a constraint solver to find all possible targets of the branching instruction, so the next instruction needs to be dynamically decided. For example, at the time of the indirect jump instruction *bx lr*, the state of the register *lr* is an expression of symbolic values. Therefore, an satisfiable instance of precondition are used to get the concrete value of *lr*.

## 2.3 SE tools for binary

This section introduces some notable symbolic execution tools for analysis of binary codes.

### 2.3.1 MAYHEM

MAYHEM [7] proposed the technique of hybrid symbolic execution, which is a combination of online and offline (concolic) execution. MAYHEM combines both techniques by swapping between symbolic execution engines when memory is under pressure. MAYHEM focus on taint analysis for exploitable bug finding.

#### Binary Representation

MAYHEM leverage BAP [31] to convert each x86 assembly instruction to BAP IL. The symbolic execution rules are based on the defined operational semantics for IL statement types [32].

#### Hybrid Symbolic Execution

Hybrid symbolic execution is introduced to maximize the effectiveness between ofline and online executor. First, the analysis is started in online mode. When the system reaches a certain memory usage, the analysis is switch to offline mode. In offline mode, no more new executors are forked, but the system produces checkpoints for online executions later. The idea behind this technique is distributing the online executor tasks into subtasks without losing potential paths.

#### External system interaction

MAYHEM prepares models for system and library calls (about 30 system call in Linux and 12 library calls in Windows). For multiple threads handling, it does not deal with multiple threads/multiple processes program when there are interaction between threads.

### 2.3.2 KLEE-MC

KLEE-MC [8] is a symbolic execution engine for binary, which is built on KLEE.

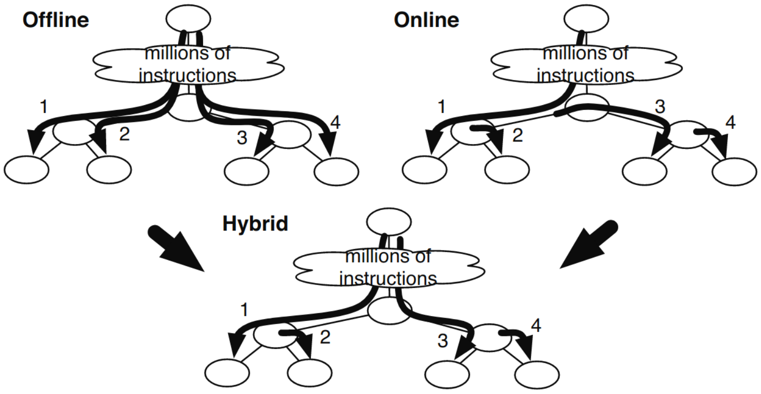


Figure 2.7: Hybrid symbolic execution in MAYHEM

### Intermediate Representation

KLEE-MC [8] uses a binary translation to translate from machine code into IR (e.g., x86-64 to LLVM). KLEE first converts machine code to VEX IR using Valgrind, then from VEX IR into LLVM IR. The binary codes are loaded into VEX IR as basic blocks.

### Symbolic Execution Model

In the original version of KLEE, it expects that all code is loaded without concerning about self-modification and dynamically loaded libraries. KLEE-MC use a new strategy, it loaded each basic block. Then it analyse the return value of a basic block to decide the next executed block.

### System Models

KLEE-MC models the system calls to interact between the symbolic engine and the underlying system. The system model in C supports both Linux (x86, x86-64, and 32-bit ARM) and Windows (x86). The return value of the modeled system calls are symbolic values, which means the model "*over-approximate*" the system interaction. KLEE-MC also provide a framework for users to use system hooks to modify program's control flow.

## 2.3.3 angr

`angr` [11] is a widely used open source binary analysis framework that support many analyses such that dynamic symbolic execution, backward slicing, or data-dependency analysis.

### Intermediate Representation

`angr` translates native binary code (e.g., ARM, MIPS, PPC, x86, and amd64) to an intermediate representation - VEX IR. Beside VEX IR, `angr` is also support several different representations (e.g., Capstone [16]). angr loads the binary by basic blocks of VEX IR statements.

### Control flow graph (CFG) and Indirect jumps

Angr contains several strategies to construct a control flow graph from binary samples. In angr, there are two types of CFG that can be generated: a static CFG (CFGFast) and a dynamic symbolic execution CFG (CFGEmulated).

- CFGFast uses static analysis to generate a CFG. It is significantly faster, but does not contain the control flow information. This is the same analysis performed by other popular reverse-engineering tools, such as IDA Pro, Ghidra.
- CFGEmulated uses symbolic execution to capture the CFG. While it is theoretically more accurate, it is much more slower and less complete due to missing system calls, missing hardware features, and so on.
  CFGEmulated first analyses a list of basic blocks and adds corresponding direct jumps for each block. Indirect jumps can not be resolved this way. For indirect jump, it traverses backwards until encounter a *merge point* (where multiple paths converged) or a threshold number of blocks. From that point, angr perform symbolic execution and use a constraint solver to resolve the indirect jump value.

### External call

One of the biggest limitations of angr when analyzing real-world software is the environment model. angr models some external calls by their own implementation in Python, called SimProcedure. Since having a SimProcedure for every library function is unrealistic, angr ends up executing statically loaded binary code and encounters errors with unsupported system calls. Although angr provides the framework for users to hook to a place and directly return any desired value at any given call, this is not an easy task for a large binary file, especially when malware usually contains more than 100 external function calls.

## 2.3.4 BINSEC/SE

BINSEC/SE [33] is a binary analysis engine focusing on the interaction between a tracer and the symbolic execution component.

### Binary Representation and PINSEC

BINSEC/SE is built on the platform of BINSEC, which transforms the trace of x86 instructions to the intermediate representation in DBA [34] (Figure 2.9). The tracer, which is called PINSEC[35] is used to generate execution trace of Linux and Windows binaries. After retrieving the result from PINSEC, the data is passed as the input of BINSEC/SE.

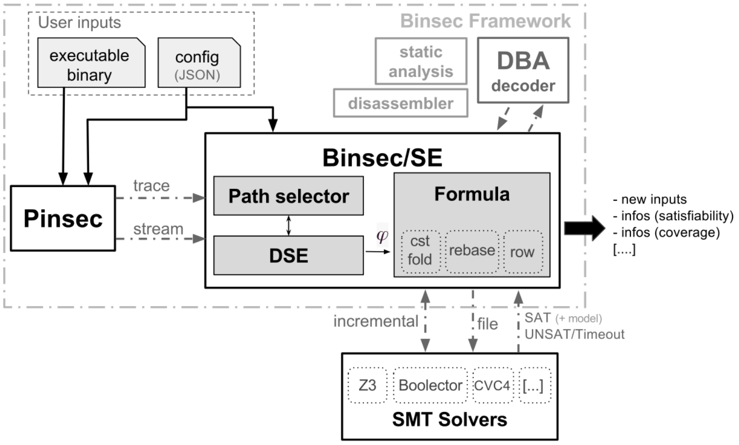
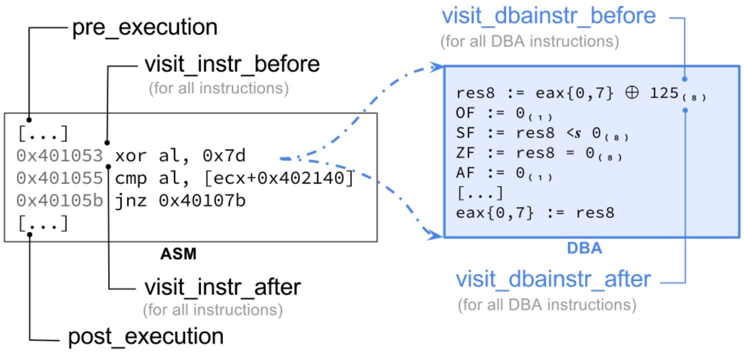


Figure 2.8: Architecture of BINSEC/SE



```
[...]
0x401053 xor al, 0x7d
0x401055 cmp al, [ecx+0x402140]
0x40105b jnz 0x40107b
[...]
```

```
res8 := eax{0,7} ⊕ 125(8)
OF := 0(1)
SF := res8 <s 0(8)
ZF := res8 = 0(8)
AF := 0(1)
[...]
eax{0,7} := res8
```

Figure 2.9: Corresponding DBA instructions of a x86 instruction

Two most notable differences between PINSEC and other pintools [35] are: (i) allowing injection of concrete or symbolic values in any location during the execution steps, and (ii) a remote C&C system for interaction between the DSE core and PINSEC (Figure 2.8).

### DSE engine

The path predicates that computed from the PINSEC trace are passed to the *Path Selection* module. The criteria to choose the best path is based on a user-defined *score* function of some properties such as trace length, call-depth, etc. Some strategies (e.g., DFS, BFS, and random path) are already provided.

# Chapter 3

# DSE for ARM Instructions: CORANA

This chapter discussed the formal semantics of ARM and the dynamic symbolic execution of ARM binaries on the instruction label. The semantic extraction process of ARM instructions from the natural language description to preliminarily built a DSE tool for ARM Cortex-M called CORANA (Cortex Analyser) is also described in this chapter.

## 3.1 Formal semantics of ARM

To build the emulator for the verification tools, the modeling of the instruction semantics [36] is required (CoDisam[9], BE-PUM[12]). Therefore we need to define the operational semantics of ARM in order to build the binary emulator for DSE. The main difficulties in defining formal semantics are weak memory model and synchronization. Luckily, our target is IoT malware, particularly ARM binary, which is mostly sequentially consistent. This implies that operations must appear to execute one at a time and in program order. In this study, the execution of each ARM instruction is regarded as a transition on the environment model made by *registers, memory, stack,* and *flags*. Based on the architecture design of ARM Cortex-M [37], we can define the environment model.

**Definition 3.1.1.** For the environment elements, $Values$ is a domain for an interpretation, the environment model of ARM Cortex-M is defined as a mapping:

$$Env : (R \to Values) \times (F \to Values) \times (M \to Values) \times (S \to Values) \tag{3.1}$$

- referred by $\langle R, F, M, S \rangle$ where the tuple $\langle R, F, M, S \rangle$ consists of:
  - R: a set of registers $R = \{r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, sp, lr, pc, apsr\}$ where *apsr* keeps the control code flags N,Z,C,V (and Q, GE in some ARM versions).
  - F: a set of flags $F = \{N, Z, C, V, Q, GE\}$
  - M: a set of n memory locations $M = \{m_0, m_1, m_2, ...\}$
  - S: a stack $(S \subset M)$

- $Values$ represent the domain of the environment elements (e.g, `32-bit BitSet` in concrete execution, or $Expression(Sym)$ in symbolic execution).

For a shorter notation, we denote $Env = \langle R, F, M, S \rangle$ to represent the mapping in Equation 3.1 as the environment model.

The description of the environment elements of ARM Cortex-M is written in the ARM Architecture Reference Manual [37] as follows:

1. **Registers:** The ARM registers are divided into two kinds: general purpose and special purpose registers.
   - R0-R12: General purpose registers
   - R13: SP (Stack Pointer). The stack pointer points to the top of the stack, which is a specially area in the memory. The stack pointer is used to allocate space on the stack.
   - R14: LR (Link Register). When a function call is made, the link register is updated as the memory address of the next instruction after the call.
   - R15: PC (Program Counter). The program counter tracks the current instruction location by storing the address of the current instruction plus 8.
2. **Flags:** The ARM flags are updated by execution of instructions. They might be used for branch decisions, or as input of the next instruction.
   - N: Set when the last operation result in a Negative value.
   - Z: Set when the result of the operation is Zero.
   - C: Set when the last operation resulted in a Carry.
   - V: Set when operation caused oVerflow, which means the result is greater than or equal to $2^{31}$, or less than $-2^{31}$.
     The combinations of these four flags (N, Z, C, and V) can create 15 branch conditions (Table 3.1).
3. **Memory:** A set of contiguous memory locations. Before the execution, the binary program is loaded on a memory area M.
4. **Stack:** Stack is a subset of memory where data is pushed or pop in a last-in-first-out (LIFO) manner.

The execution of an instruction $i$ is regarded as a transition $t_i$ on the environment model. In CORANA, each register in $R$, memory location in $M$ and $S$ are represented by a 32-bit vector, while flags in $F$ are boolean variables. The values of the registers and flags are initialized by symbolic values. Those values are updated throughout the execution of each instruction. The register PC points to the address of the next instruction anh the register SP points to the top of *Stack*.

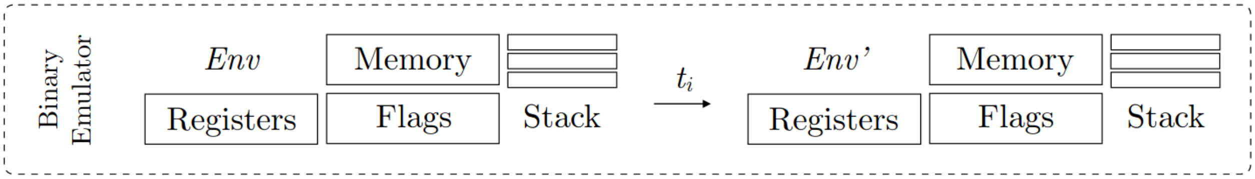


Figure 3.1: The semantics transition $t_i$ of the instruction $i$

The figure 3.2 shows some rules of the operational semantics of ARM instructions that are implemented in CORANA. Each rule describes a transition of the environment model caused by the corresponding ARM instruction. For example, the *mov* instruction copies the data item referred into the register and updates *pc* register.

$$\frac{R_{pc} = k;\ instr(k) = mov\ i\ j; R_j = m}{\langle F, R, M, S\rangle \to \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow m], M, S\rangle}\ [\text{MOV}]$$

$$\frac{\begin{array}{c} R_{pc} = k;\ instr(k) = ands\ i\ j\ h;\ R_j = m; R_h = n; a = m\ \&\ n; \\ N' = (a < 0); Z' = (a = 0); C' = (a \geq 2^{32} - 1); V' = (a \leq -2^{31}) \vee (a \geq 2^{31} - 1) \end{array}}{\langle F, R, M, S\rangle \to \langle F[N \leftarrow N'; Z \leftarrow Z'; C \leftarrow C'; V \leftarrow V'], R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow a], M, S\rangle}\ [\text{ANDS}]$$

$$\frac{R_{pc} = k;\ instr(k) = neg\ i\ j;\ R_j = n}{\langle F, R, M, S\rangle \to \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow !n], M, S\rangle}\ [\text{NEG}]$$

$$\frac{\begin{array}{c} R_{pc} = k;\ instr(k) = cmp\ i\ j;\ R_i = m; R_j = n; a = m - n; \\ N' = (a < 0); Z' = (a = 0); C' = (a \geq 2^{32} - 1); V' = (a \leq -2^{31}) \vee (a \geq 2^{31} - 1) \end{array}}{\langle F, R, M, S\rangle \to \langle F[N \leftarrow N'; Z \leftarrow Z'; C \leftarrow C'; V \leftarrow V'], R[pc \leftarrow k + \mid instr(k) \mid], M, S\rangle}\ [\text{CMP}]$$

$$\frac{R_{pc} = k;\ instr(k) = ldr\ i\ j;\ M_j = m}{\langle F, R, M, S\rangle \to \langle F, R[pc \leftarrow k + \mid instr(k) \mid; R_i \leftarrow m], M, S\rangle}\ [\text{LDR}]$$

Figure 3.2: Some examples of the operational semantics

In CORANA, the formal semantics of each ARM instruction is represented as a Java method build on top of a customized **BitVec** class, which is a pair $\langle bs, s\rangle$ where $bs$ is a **BitSet** 32-bit vector and $s$ is a string. Corresponding to the BitVector theory of SMT solvers, 35 basic methods had been manually implemented as basic functions for the binary emulator. All automatically generated Java methods are operated based on them.

### Example 1

According to the operation instruction, the UMAAL (Unsigned Multiply Accumulate Accumulate Long)[1] *"multiplies the 32-bit values in Rn and Rm, adds the two 32-bit values in RdHi and RdLo, and stores the 64-bit result to RdLo, RdHi"*. Its operation can be

[1] https://developer.arm.com/documentation/arm-and-thumb-instructions/multiply-instructions/umaal

described in the operational semantic and a corresponding Java method as the followings[38].

$$\frac{\begin{array}{c} R_{pc} = k; inst(k) = umaall\ rdlo\ rdhi\ rn\ rm; R_{rdlo} = lo; R_{rdhi} = hi; \\ R_{rn} = n; R_{rm} = m; a = m * n + lo + hi; hi' = a \gg 32; lo' = (a \ll 32) \gg 32; \end{array}}{\langle F, R, M, S\rangle \rightarrow \langle F, R[pc \leftarrow k + |inst(k)|; R_{rdlo} \leftarrow lo'; R_{rdhi} \leftarrow hi'], M, S\rangle} \text{[UMAAL]}$$

```
    public void UMAAL(Character l, Character h, Character n, Character m,
    Character suffix, Character cond) {
        if (cond == null || env.checkCond(cond)) {
            char [] flags = new char []{};
            BitVec result = null;
            result = mul(val(n), val(m));
            result = add(result, val(h));
            result = add(result, val(l));
            write(h, shift(result, Mode.RIGHT, 32));
            write(l, shift(shift(result, Mode.LEFT,32),Mode.RIGHT,32));
            if (suffix != null && suffix == 's') {
                if (result != null) {
                    env.updateFlags(flags, result);
    }}}}
```

Listing 2: The semantic of UMAAL instruction written in Java [21]

## 3.2 Path condition generation

### 3.2.1 Conditional branching in ARM

Every computing architecture has a mechanism for checking condition (e.g., if-else in C). In ARM, conditional execution are implemented using a set of *flags* that store the information of previous operations. In other words, the branch condition is checked based on the status of four flags *(N, Z, C, and V)* in the APSR (Application Process Status Register).

The flags is updated with the execution of instructions. The most common way to set condition flags is using the comparison operation (*e.g., CMP, CMN, TST, TEQ*). In addition, many arithmetic and logical instructions such as *SUBS, ADDS* can update the condition flags based on the results of the operations. The example below shows how the *ADDS* instruction updates the flags.

**Example 2**

The following code fragment shows how the flags are set in ARM:

```
ldr     r1, 0xffffffff
ldr     r2, 0x00000001
adds    r0, r1, r2
```

The result of this operation should be r0=0x100000000. However, since the 32-bit destination register cannot fit the result, the actual result saved in r0 is 0x00000000 and the flags are also updated accordingly.

- N = 0. The result is 0x0, which is positive, so N bit is 0.
- Z = 1. The Z bit is set to 1 since the result is 0x0.
- C = 1. C bit is 1.
- V = 0. The operation (which equivalent to $-1 + 1 = 0$) did not caused overflow, so V (oVerflow) is 0.

By using those four flags, we can control the program's flow by making jumps or executing certain instructions when a condition is satisfied. With combinations of the four conditional flags, 15 branch conditions are defined Table 3.1.

| **Symbol** | **Meaning** | **Condition** |
|---|---|---|
| EQ | Equal | Z==1 |
| NE | Not equal | Z==0 |
| CS/HS | Carry set | C==1 |
| CC/LO | Carry clear | C==0 |
| MI | Minus V1 | N==1 |
| PL | Plus | N==0 |
| VS | Overflow | V==1 |
| VC | No overflow | V==0 |
| HI | Unsigned higher | C==1 && Z==0 |
| LS | Unsigned lower or same | C==0 \|\| Z==1 |
| GE | Signed greater than or equal | N==V |
| LT | Signed less than | N!=V |
| GT | Signed greater than | Z==0 && N==V |
| LE | Signed less than or equal | Z==1 \|\| N!=V |
| AL | Always | |

Table 3.1: ARM Conditional Suffix [39]

### 3.2.2 Path condition in ARM

The path condition is a formula that describes the precondition of the execution path from the entry point to the current nodes. At current assembly instruction $i$, the path condition $\phi_i$ from initial node to $i$ is updated through each step of the computation as:

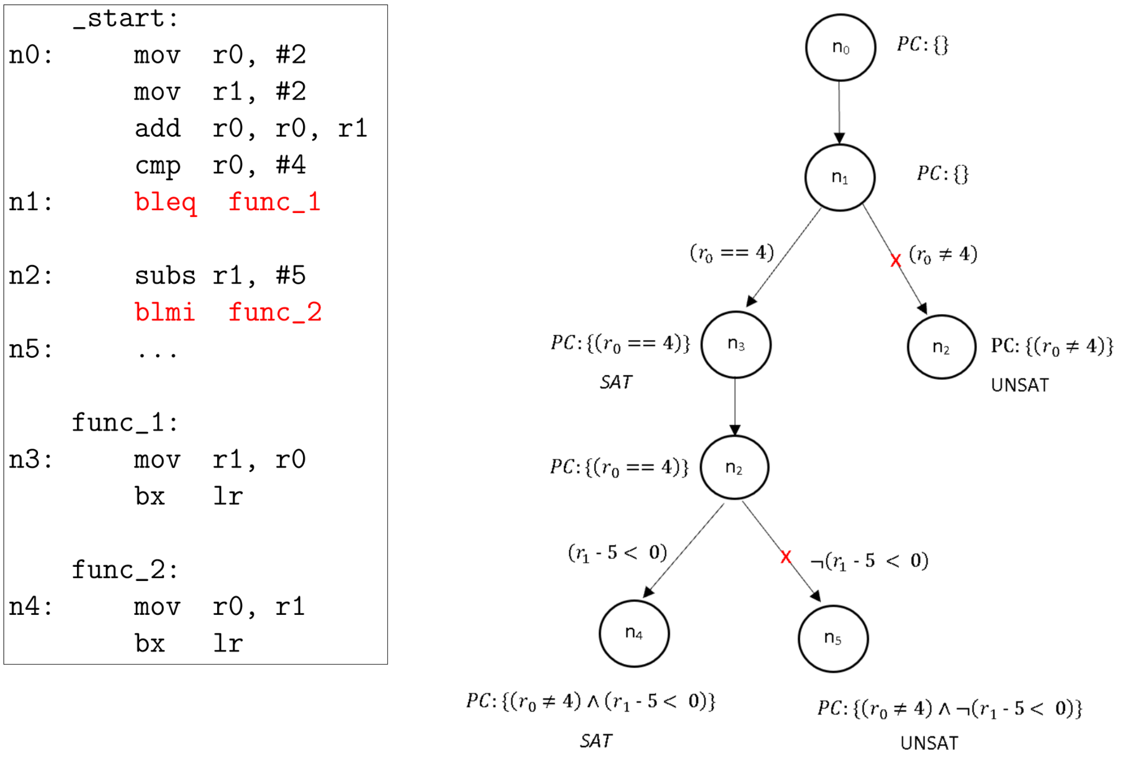
$$\begin{cases} \phi_0 = true \\ \phi_i = \phi_{i-1} \wedge c_i \end{cases} \tag{3.2}$$

with $c_i$ is the constraint generated by instruction $i$.

As discussed previously, ARM instructions can be made to execute conditionally by postfixing with certain condition codes. During the execution, the path condition is updated based on branching mechanism of ARM assembly.

**Example 3**

This figure show how path conditions are generated in CORANA.



```
    _start:
n0:     mov  r0, #2
        mov  r1, #2
        add  r0, r0, r1
        cmp  r0, #4
n1:     bleq  func_1

n2:     subs r1, #5
        blmi  func_2
n5:     ...

    func_1:
n3:     mov  r1, r0
        bx   lr

    func_2:
n4:     mov  r0, r1
        bx   lr
```

Figure 3.3: Path condition in the execution of ARM instructions

The control flow graph in the right hand describes the represented assembly codes. The instructions from location $n_0$ to $n_1$ performs two assignments ($r_0 = 2$ and $r_1 = 2$), then adds $r_0 = r_0 + r_1$ to make the states of $r_0$ and $r_1$ become 4 and 2, respectively. The *cmp* instruction compares the current value of $r_0$ to 4, makes $Z = 1$ (Zero flag is set).

At $n_1$, a conditional branch *bleq* occur and the prefix *eq* checks for $Z$ flag. The condition *eq* returns *true* when Z $==$ 1, which means $r_0 == 4$ and returns *false* when $r_0 \neq 4$. Using the SMT Solver to check the satisfiability of each path constraint, the negation branch returns UNSAT.

Continuing on the feasible path, at $n_2$ the register $r_2$ becomes -1 and the N flag (Negation) is set because of the negative result. The branching condition *blmi* checks if $N == 1$ i.e., $r_1 - 5 < 0$. The path condition for two new branches are updated by conjunction as $(r_0 \neq 4) \wedge (r_1 - 5 < 0)$ and $(r_0 \neq 4) \wedge \neg(r_1 - 5 < 0)$. Same as the previous branching condition, each path is checked whether it is SAT or UNSAT.

### Path condition in Hoare Logic

As previous mentioned in Section 2.2, the triple of Hoare Logic defined as:

$$\frac{Precondition}{Postcondition}\ \text{[Instruction]} \tag{3.3}$$

shows how the execution of a *Instruction* transform the system state.

**Definition 3.2.1.** A *symbolic state* (*pre/post-condition*) at location $i$ is a tuple $\langle \alpha_i, Env \rangle$ where:

- $\alpha_i$ is the path condition formula at $i$.
  $Var(\alpha_i) \subseteq Sym$, $Sym$ is the set of symbolic symbols.
- $Env = \langle F, R, M, S \rangle$ is the environment model. The value stored at a register, a flag or a cell in the memory can be a constant (e.g., specific address, arithmetic constant), a symbolic value, or a formula.

In previous section, we represent each ARM instruction as an operation semantic rule. By applying the formal semantics, we can almost immediately obtain the post-condition.

$$\frac{\langle \alpha, Env \rangle}{\langle \alpha', Env' \rangle}\ \text{[Instruction]}$$

$$\frac{\langle \alpha, \langle F, R, M, S \rangle \rangle}{\langle \alpha, \langle F, R[pc \leftarrow pc'; R_i \leftarrow m], M, S \rangle \rangle}\ \text{[MOV Ri, m]}$$

$$\frac{\langle \alpha, \langle F, R, M, S \rangle \rangle}{\langle \alpha, \langle F[N \leftarrow N'; Z \leftarrow Z'; C \leftarrow C'; V \leftarrow V'], R[pc \leftarrow pc'; R_i \leftarrow a], M, S \rangle \rangle}\ \text{[CMP m, n]}$$

with $a = m - n$,
and $N' = (a < 0), Z' = (a == 0), C' = (a \geq 2^{32} - 1), V = (a \leq -2^{31}) \vee (a \geq 2^31 - 1)$

$$\frac{\langle \alpha, \langle F, R, M, S \rangle \rangle}{\langle \alpha \wedge (Z_{flag} == 0), \langle F, R[pc \leftarrow k], M, S \rangle \rangle}\ \text{[BNE k]} \quad \text{if } Z_{flag} \text{ is not set}$$

$$\frac{\langle \alpha, \langle F, R, M, S \rangle \rangle}{\langle \alpha \wedge \neg(Z_{flag} == 0), \langle F, R[pc \leftarrow pc + 4], M, S \rangle \rangle}\ \text{[BNE k]} \quad \text{if } Z_{flag} \text{ is set}$$

## 3.3 Formal semantics extraction of ARM instructions

### 3.3.1 Semantic extraction from natural language specifications

Each ARM architecture has a relatively small number of instructions (around 60-300), but various variants exist (e.g., Cortex-A, Cortex-M, Cortex-R). Our study focuses on ARM Cortex-M, which is mostly used for micro-controllers such as IoT devices. To built the binary emulator for the dynamic execution tool, the operational semantic of ARM instructions need to be extracted from its natural language specification. The specification of ARM instruction from the official ARM documentation [39] contains five sections: *mnemonic, description, syntax, operation*, and *flags-update.*

| | |
|---|---|
| **Mnemonic** | UASX |
| **Description** | Unsigned Add and Subtract with Exchange |
| **Syntax** | UASX{cond} {Rd,} Rn, Rm |
| **Operation** | Subtracts the top halfword of the second operand from the bottom halfword of the first operand. Writes the unsigned result to the bottom halfword of the destination register. Adds the top halfword of the first operand with the bottom halfword of the second operand. Writes the unsigned result to the top halfword of the destination register. |
| **Flags-update** | This instruction set the APSR_GE bits according to the results. |

Figure 3.4: The specification of UASX in ARM Cortex-M7 [39]

In the previous work [21], the formal semantics of ARM have been semi-automatically extracted from the instruction specifications. Each ARM instruction is interpreted and represented by a Java method to build the binary emulator for the DSE tool (as shown in Example 1). Figure 3.5 illustrates an overview of the extraction process, with the dashes boxes representing manually prepared tasks.

The first step in the process, each sentence in the ARM instruction specification is normalized by parsing, lemmatization and word refinement. After normalizing, two parts of the instruction needs to be analyzed are operational semantic (Figure 3.5 - II) and flag modification information (Figure 3.5 - III).

- **Semantics Interpretation by Translation Rules**
  The interpretation follows a rule-based method by utilizing popular *NP-Phrases.* We manually prepared around 200 predefined rewriting rules to interpret the the formal semantics of instructions in a bottom-up manner.
  For example, a set $\{c_1, c_2, c_3\}$ of NP-Phrases is obtained through syntax normalization, where $c_1$ is "*first and second operand*", $c_2$ is "*two unsigned 32-bit integer*", and $c_3$ is "*multiply* $\Box_2$ *in* $\Box_1$". Based on the set of NP-Phrases, the following rules are repaired [38]:

  1. Rule *r1*: *first and second operand* → *rn, rm*

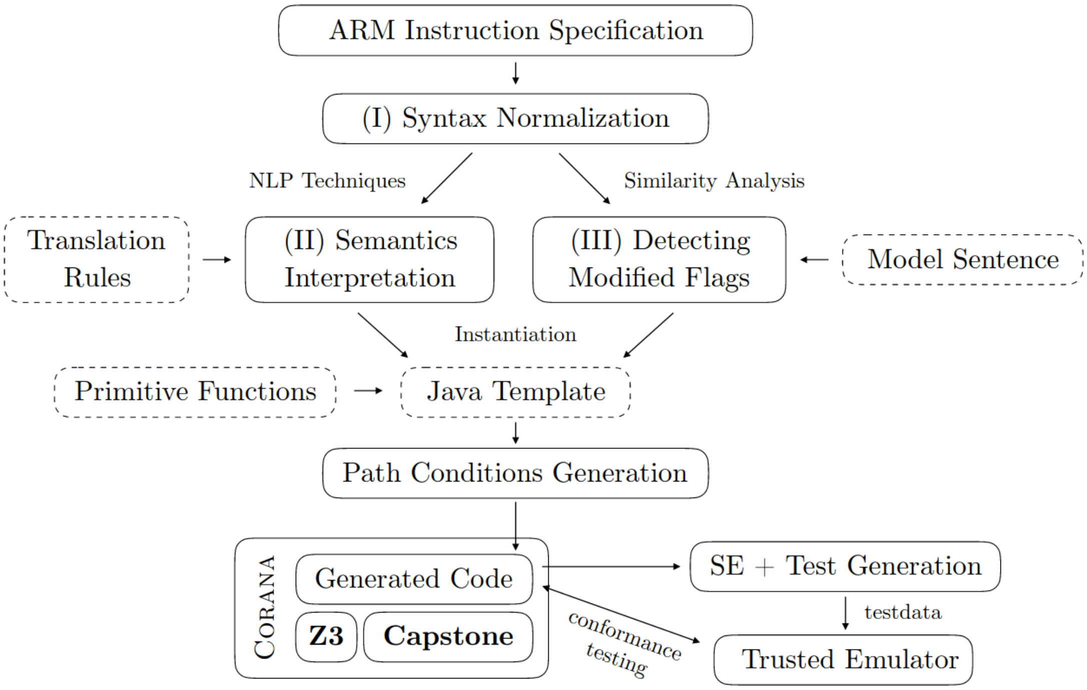


Figure 3.5: Semantic extraction from NLP specification of ARM [38]

2. Rule *r2*: *two unsigned 32-bit integer* → $val(\Box_3)$, $val(\Box_4)$
3. Rule *r3*: *multiply* $val(\Box_3)$, $val(\Box_4)$ *in rn, rm* → *mul(val(rn),val(rm))*

The instruction sentence is then transformed to a Java statement by applying the above rules in a bottom-up manner.

- **Modified Flag Detection**
  The descriptions about flags-update are written in natural language. To decide whether the flags information is changed or not, a topic modelling method - Latent Dirichlet Allocation (LDA)[40] is used. After training the LDA model, the simility between the topic distribution of the target sentence and the model sentence *"update effect set change modify"* are calculated. The flag is consider as *modified* or *unmodified* base on whether the result exceed a predefined threshold t.

After obtaining the interpretation of the instruction, the method represents for its operational semantic is built based on a Java template. The generated methods combining with predefined primitive functions are used to construct a binary symbolic execution engine for the DSE tool CORANA.

### 3.3.2 CORANA architecture

CORANA[21] consits of two main components: (I) A kernel including the emulated environments and the generated path conditions, and (II) A symbolic executor which built

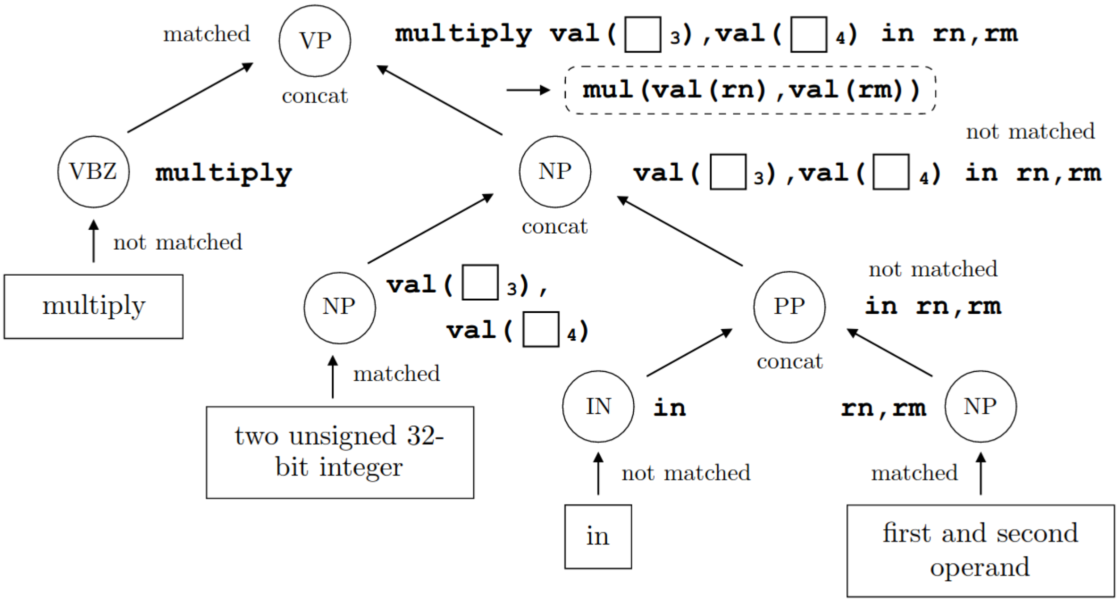


Figure 3.6: Semantic interpretation of ARM instruction [21]

from the generated Java methods and predefined primitive functions.

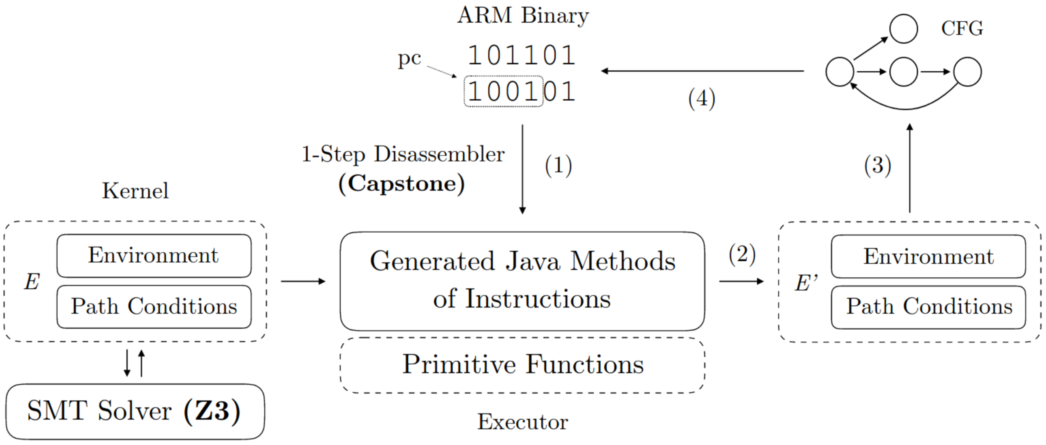


Figure 3.7: CORANA Architecture

Figure 3.7 shows the execution flow of CORANA to generate a control flow graph from an input binary file.
(1) First, the ARM binary file is single-step disassembled from the *_start* location of the program to obtain instruction $i$.
(2) The symbolic executor executes instruction $i$ and updates the kernel (including the environment and the path condition).
(3) If the next instruction is an indirect jump, SMT solver solves the path condition to obtain possible initial environment to reach the point. Then concolic testing will decide

the next location. A new node and its connected edge are then added to the control flow graph.
(4) Disassembles the next instruction and repeats the process until reaching the end of the program.

CORANA has covered 66.6% of ARM instructions over 6 variants. However, programs are usually not self-contained, they need to interact with the surrounding environment. Particularly in the case of malware binaries, system calls and library function calls are largely used. External calls that occur lead to an exception in CORANA and make the execution process be interrupted. To make symbolic execution more tractable, in the next chapter, we propose a solution in handling external calls of the program to the Linux/Unix system using API Stubs.

# Chapter 4

# DSE for Library Functions of Linux: CORANA/API

## 4.1 C Standard Library

Most programs are not self-contained but frequently interact with the underlying operating system and external systems. Typical examples are the interaction with the resource of the operating system (e.g., system variables, file system, and network) and the interaction with other devices that are beyond the current system over the network (e.g., Command & Control Server, and peer devices).

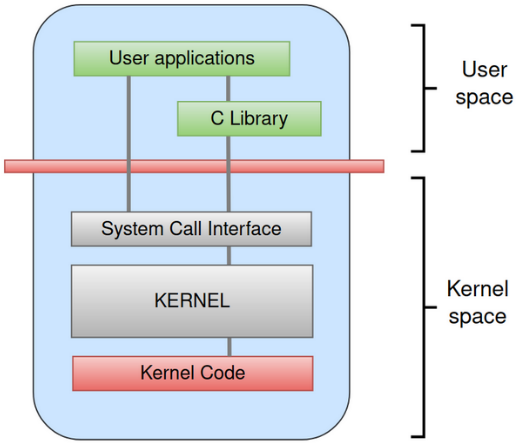


Figure 4.1: C Standard Library[1]

**Linux/POSIX API.** An user-level programs communicate with the kernel is by using system calls. When the program requires resources, it sends a request as a system call to the kernel and the mode changes to the kernel mode. Application programs do not usually contain direct system calls. In Linux system, the API allows user processes to access system resources and services of the kernel. Linux APIs mostly follow the POSIX standard.

**C Standard Library.** A C standard library is a wrapper for the Linux kernel system calls (e.g., GlibC, UClibc). A C standard library such as The GNU C Library[2] (GlibC) provides a cross-platform to execute functions that would otherwise require system-specific system calls. It provides the APIs for many systems that use Linux or Unix as the kernel. We collected source codes and descriptions of 1659 APIs from *glibc*.

## 4.2 External call template in JNA

Java Native Access (JNA) is a Java library that provides access to native shared libraries. JNA allows Java program to invoke native code via JNI.

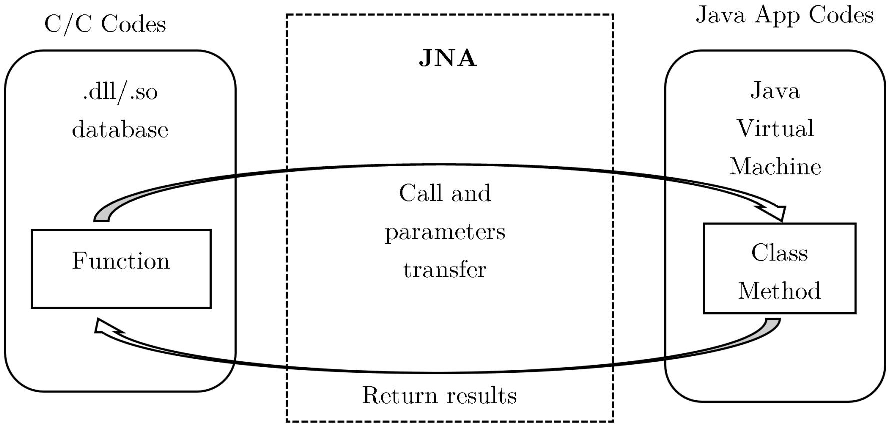


Figure 4.2: Java Native Access

**Example 4**

The following program loads the native C standard library implementation and invoke native *gettimeofday* function.

```
public interface CLibrary extends Library {
    CLibrary INSTANCE = Native.load("c", CLibrary.class);
    int gettimeofday(CStruct.timeval tv, CStruct.timezone tz);
}
public static void main(String[] args) {
    CStruct.timeval tv = new CStruct.timeval();
    CStruct.timezone tz = new CStruct.timezone();

    int ret = CLibrary.INSTANCE.gettimeofday(tv,tz);

    System.out.println("Result: " + ret);
```

[2] https://www.gnu.org/software/libc/

```
    System.out.println("Seconds since 1/1/1978: " + tv.tv_sec);
    System.out.println("Microseconds: " + tv.tv_usec);
}
```

### Library Mapping

To access to the methods of a shared library, a class corresponding to that library needs to be created. For example, a mapping for the C library itself can be defined in two ways: interface-mapped class and direct-mapped class.

```
// Alternative 1: interface-mapped class, dynamically load the C library
public interface CLibrary extends Library {
    CLibrary INSTANCE = (CLibrary)Native.loadLibrary("c", CLibrary.class);
}

// Alternative 2: direct-mapped class
public class CLibrary {
    static {
        Native.register("c");
}}
```

Listing 3: C library loaded class in JNA [41]

When we instantiate a native library interface via Native.load() to dynamically load the C library, JNA creates a proxy which routes all method invocations through a single handler. This handler has the responsibility to look up and invoke the appropriate function object which represents the corresponding function in the native library.

### Function Mapping

With interface-mapping, function names are mapped directly from their Java interface to the native library. Native libraries, such as C standard library contains many functions in the source code, but only a set of functions are used by the actual programs. JNA handles the run-time mapping of the function in the interface class to the method in the native library. We only need to call the method in the interface to call the native corresponding method. For instance, the *atoi* function in C can be called via JNA by the following.

```
public interface CLibrary extends Library {
    int atoi(String s);
}
```

Listing 4: The function *atoi* in JNA

### Type Mapping

Types in C need to be mapped to type with same size in JNA[3]. In C, types can generally divided into two kinds: primitive types and pointers. Parameters with primitive types are directly passed by value, while pointers are passed by reference to a cell or a block of cells in the memory. To map to JNA, all other types must be converted to one of the types in the Table 4.1.

Table 4.1: Types supported by JNA library [41]

| C Type | Native Representation | Java Type |
|---|---|---|
| char | 8-bit integer | byte |
| wchar_t | platform-dependent | char |
| short | 16-bit integer | short |
| int | 32-bit integer | int |
| int | boolean flag | boolean |
| enum | enumeration type | int |
| long long, __int64 | 64-bit integer | long |
| float | 32-bit floating point | float |
| double | 64-bit floating point | double |
| pointer (e.g. void*) | platform-dependent (32- or 64-bit pointer to memory) | Buffer |
| long | platform-dependent (32- or 64-bit integer) | NativeLong |
| const char* | NUL-terminated array (native encoding or jna.encoding) | String |
| char** | NULL-terminated array of C strings | String[] |
| void** | NULL-terminated array of pointers | Pointer[] |
| struct* | pointer to struct (argument or return) | Structure |
| struct | struct by value (member of struct) (or explicitly) | |
| union | same as Structure | Union |
| struct[] | array of structs, contiguous in memory | Structure[] |
| other | integer type | IntegerType |
| void (*FP)() | function pointer (Java or native) | Callback |

## 4.3 External call handling

As discussed in Chapter 1, it is preferable that the programs interact directly with their real environment while performing symbolic execution. Our approach follows to previous work such as Syman[24] and BEPUM[12] and prepares the API Stub of system calls to interact with external environment. An API Stub requires a interface as a proxy to invoke the native function, retrieves the return value, and updates the environment after the system/library function call. JNA is used to invoke the native C function in the API Stub.

[3] https://java-native-access.github.io/jna/4.2.0/overview-summary.html

For the symbolic execution with external function calls, two updates are matters, i.e., the path condition update and the environment update. Note that external function calls are mostly OS library function calls or API calls, which are executed in the kernel level. Thus, a user-level process cannot observe how it is computed. We show the extension of Hoare logic and the soundness of the API stub approach.

## Hoare Logic Revisited

Definition 3.2.1 presents the instruction level Hoare logic

$$\frac{Precondition}{Postcondition}\ [\text{Instruction}]$$

on symbolic states. For reasoning Hoare logic with extenal function calls, we introduce

- Each component of the path condition is associated the location where the component is introduced.
- Control flow associated to the environment, which induces the logical equivalence between the *path condition* and the pair of the control flow and the environment.
- Agents as processes and the awareness $A_a$ of each agent $a$. Processes are ordered by their priviledge levels.

**Definition 4.3.1.** An *extended symbolic state* at the location $i$ is a tuple $\langle \alpha_i, (CFlow, Env)\rangle$ where:

- $\alpha_i = \psi_{i_1} \wedge \cdots \wedge \psi_{i_j}$ is the path condition formula at $i$, where the locations $i_1, \cdots, i_j$ appear in $CFlow$ and $\psi_{i_j}$ is added at location $i_j$ with $Var(\psi_{i_j}) \subseteq Sym$, where $Sym$ is the set of symbolic symbols.
- $CFlow$ is the control flow reaching to $i$. $CFlow \in (Inst \times Loc)^*$ where $Inst$ is the set of ARM instructions in the code and $loc$ is the set of locations in a code section.
- $Env = \langle F, R, M, S\rangle$ is the environment model. The value stored at a register, a flag or a cell in the memory can be a constant (e.g., specific address, arithmetic constant), a symbolic value, or a formula.

In an extended symbolic state $\langle \alpha_i, (CFlow, Env)\rangle$ at the location $i$, $(CFlow, Env)$ is regarded as the collection of equational formulas such that $flow(i) = w \in (Inst \times Loc)^*$ as the value of $CFlow$, and $x = exp \in Exp(Sym)$ for $x \in F \cup R \cup M \cup S$ as the individual values of $\langle F, R, M, S\rangle$. With this view, the next lemma is immediate.

**Lemma 4.3.1**
Let $\langle \alpha_i, (CFlow, Env)\rangle$ be an extended symbolic state at location $i$. Then, $\alpha_i$ implies $(CFlow, Env)$, and vice versa.

For each process $a$ of the system, the *awareness* $A_a$ of $a$ restricts

- $\mathit{Sym}$ to $\mathit{Sym}_a$, which is the set of symbolic values visible from $a$.
- $\mathit{Loc}$ to $\mathit{Loc}_a$, which is the set of locations of the code section of $a$.

and we extend $A_a$ on the extended symbolc states such that

- $A_a(\alpha_i) = A_a(\psi_{i_1}) \wedge \cdots \wedge A_a(\psi_{i_j})$ filters $\psi_{i_j}$ in $\alpha_i$ as $\mathit{true}$ if $i_j \notin \mathit{Loc}_a$. Therefore each $A_a(\alpha_i)$ contains variable only from $\mathit{Sym}_a$.
- $A_a(\mathit{CFlow})$ of the control flow $\mathit{CFlow}$ is the sequence dropping a pair $(\mathit{inst}, \mathit{loc})$ with $\mathit{loc} \notin \mathit{Loc}_a$.

$A_a(\psi)$ is interpreted as $\psi_{\downarrow} \cap \Gamma$ under the view of *Information system* [42], where $\psi_{\downarrow} = \{\varphi \mid \psi \Rightarrow \varphi\}$ and $\Gamma$ is the set of formulas that contains variable only from $\mathit{Sym}_a$. In the context of the epistemic logic, $A_a$ is interpreted as the awareness operator [43]–[45].

**Lemma 4.3.2**

Let $\langle \alpha_i, (\mathit{CFlow}, \mathit{Env}) \rangle$ be an extended symbolic state at location $i$. For a process $a$, $A_a(\mathit{CFlow}, \mathit{Env})$ implies $A_a(\alpha_i)$, but not vice versa.

The statement of Lemma 4.3.2 can be regarded as the soundness of the path condition and is suggested by the facts that

- $A_a(\mathit{CFlow}, \mathit{Env})$ may contain the return value of an external function call, and the API stub updates $\mathit{Env}$ with the output as a constant. Additionally, $a$ cannot observe the system-level conditional branch and the control flow.
- $A_a(\mathit{CFlow}, \mathit{Env})$ contains the control flow of the execution in the code of $a$. That behavior requires $A_a(\alpha_i)$.

**Definition 4.3.2.** For an *extended symbolic state* $\langle \alpha_i, (\mathit{CFlow}, \mathit{Env}) \rangle$ at location $i$ in a process $a$, assume that an external function $f$ is called and $f$ updates $\mathit{Env}$ to $\mathit{Update}(\mathit{Env}, f)$. Then, the Hoare logic rule under the awareness $A_a$ of $a$ is

$$\frac{\langle \alpha_i, (\mathit{CFlow}, \mathit{Env}) \rangle}{\langle \alpha_i, (\mathit{CFlow}.(\texttt{call } f, i+1), \quad \mathit{Update}(\mathit{Env}, f) \rangle} \text{[External function } f \text{ call]}$$

## API Stub for Environment Update

Each API Stub needs to perform these operations (Figure 4.3):

- Load parameters from the memory of the DSE tool
- Invoke native function, retrieve return and updated values
- Write return and updated values into the memory of the DSE tool

**Example 5**

This example describes how the API stub of the function *gettimeofday* pass parameters from the emulated environment into JNA method, then update the memory after the call.

```
int gettimeofday (struct timeval *tv, struct timezone *tz);
```

The function have two *struct* parameters.

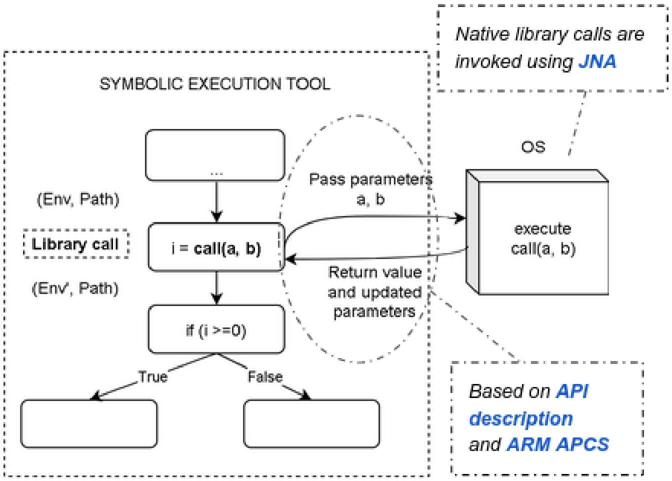


Figure 4.3: Solution for external call handling

```
struct timeval {
    time_t tv_sec;
    suseconds_t tv_usec;
}
```

```
struct timezone {
    int tz_minuteswest;
    int tz_dsttime;
}
```

Before the call, parameters are set up in the memory and loaded to the registers as shown in Figure 4.4. Two parameters in the first two registers are structure pointers that pointing to the structure contents in the memory. After running the native code in the actual environment, the return value is obtained. Finally, the return value and update parameters are written to the memory in the emulated environment.

### Passing parameters to the external call

The standard system for passing parameters and returning values between functions (or subroutines) is called a calling convention. For the ARM processor, how a subroutine interact with memory are defined in ARM Architecture Procedure Call Standard (AAPCS)[4]. According to the standard (Table 4.2), parameters are passed by placing the parameter values into registers R0 through R3, and the return value is placed into R0.

### Update the environment after the external call

In ARM system, after a subroutine, the following rule is applied:

- After the API call, the return value is always widened to 32 bits and stored in the R0 register. The return type can be int, long, boolean, a structure, pointer to a memory address, or void (no return value, the value of R0 register is kept as before API call).

[4] https://developer.arm.com/documentation/ihi0042/latest/

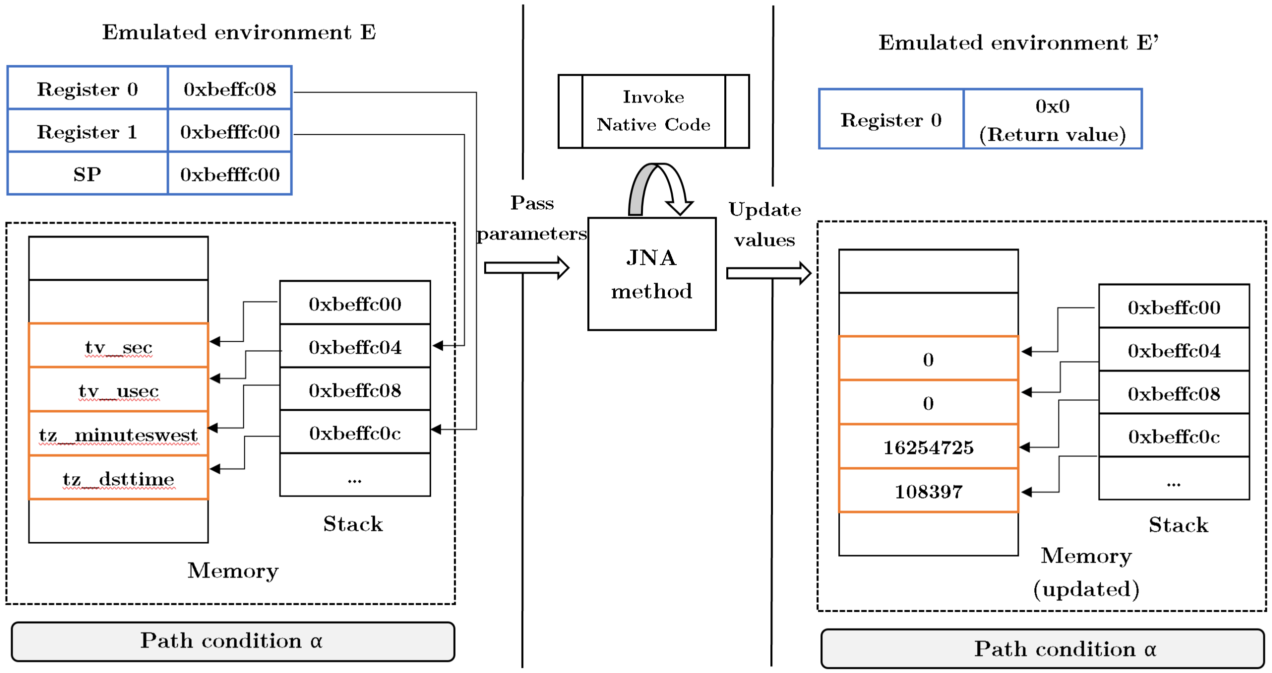


Figure 4.4: Memory handling in API Stub

- The value of memory area pointed by a pointer parameter may be updated after the function call. Parameters are passed by value to the external function. Even when the value of pointer parameter doesn't change, the memory area are pointed by a pointer parameter can be modified.
- Some functions, such as dynamic memory allocation functions, may create new memory area. We assume the return value of API doesn't point to a new memory area.

## 4.4 Multiple process handling

In a set of 5000 IoT malware samples, we observed that only nearly 2% that do not contain any `fork()` call to create multiple processes (Table 4.3). Malware spawn new processes largely to daemonize a program, connect to its command and control server or create parallel DDos attack processes.

Forking is the basic mechanism to create a process in Unix and Linux. When a `fork()` occurs, the system suspends and creates a replica of the current (parent) process, then it resumes both parent and the child processes concurrently. `fork()` call returns 0 to the child process and return a process-identifier PID ($PID > 0$) of the child process to the parent process. `wait()` is a another Unix/Linux call that are used with `fork()` to synchronize two or more related processes. When the parent process execute `wait()`, it

| Register | Usage | Function preserved | Description |
|---|---|---|---|
| R0 | Argument 1 and return value | No | If the return value or the first argument has 64 bits, both R0:R1 hold it. |
| R1 | Argument 2 | No | Argument 2 |
| R2 | Argument 3 | No | If the return value has 123 bits, R0:R3 hold it. |
| R3 | Argument 4 | No | If more than 4 arguments, the stack is used. |
| R4-11 | General-purpose V1-8 | Yes | Hold a local variable. |
| R12 | Intra-procedure-call register | No | R12 holds intermediate values between a procedure and its sub-procedure. |
| R13 | Stack Pointer | Yes | Top of the stack address. |
| R14 | Link Register | No | LR does not have to contain the same value after the function call. |
| R15 | Program Counter | No | |

Table 4.2: ARM Architecture Procedure Call Standard [39]

| Number of fork() | Number of examples | Percentage of examples |
|---|---|---|
| > 50 | 15 | 0.3 |
| 10-50 | 1278 | 25.56 |
| 0-10 | 3648 | 72.96 |
| 0 | 59 | 1.18 |

Table 4.3: Multiple process in IoT malware

is suspended until any of its child finishes. According to the POSIX documentation[5], `fork` requires cloning of the calling thread and its entire address space, while non-Unix OSes do not support this efficiently. Moreover, `fork` requires the child process may only execute POSIX async-signal-safe operations. JVM is a multiple thread processes and Java calls are not POSIX async-signal-safe, therefore, the child process will not work when calling `fork()` in JVM. We handle the concurrent processes by serialization with the assumption that processes are executed without overlapping and do not interact with each other. A concurrent system is said to be *serializable*[46] if any concurrent execution of a number of procedures is equivalent to some sequential execution of those procedures by one after another. One way of ensuring serializability is to make the execution follow locking protocols.

**Definition 4.4.1.** An execution is *well-locked* if a process never accesses a global variable or a dynamically allocated object without holding its protecting lock (i.e., obeying the

[5] https://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html

access rule).

According to a survey on IoT malware, forking is mostly used to create separate processes to carry different purposes with no interactions between processes[47]. Based on this observation, if we set the assumption for the malware execution that there is no communication between concurrent processes, then the execution is adhered to the locking protocol. This means that the execution is serializable, implying a sequential reduction can be applied. Instead of analyzing two processes concurrently, we sequentially perform symbolic execution on the parent and child process (Figure 4.5).

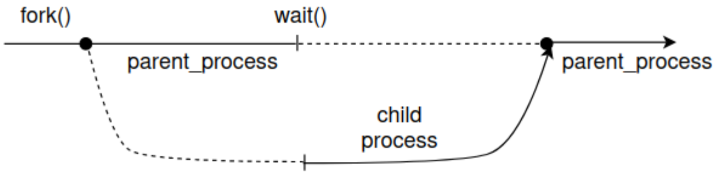


Figure 4.5: Execution order of two processes

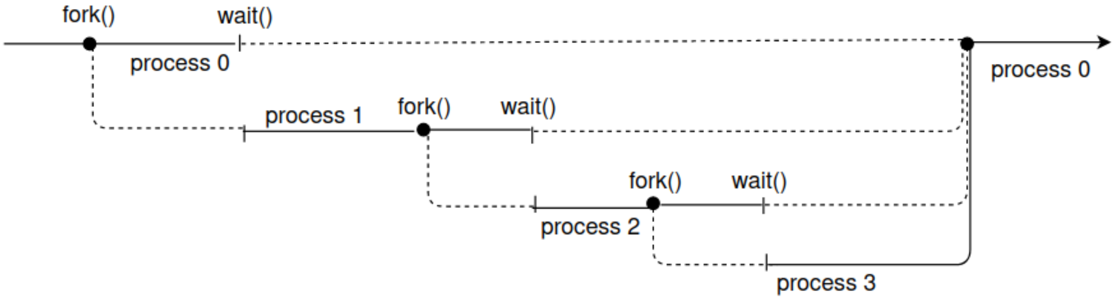


Figure 4.6: Execution order of multiple processes

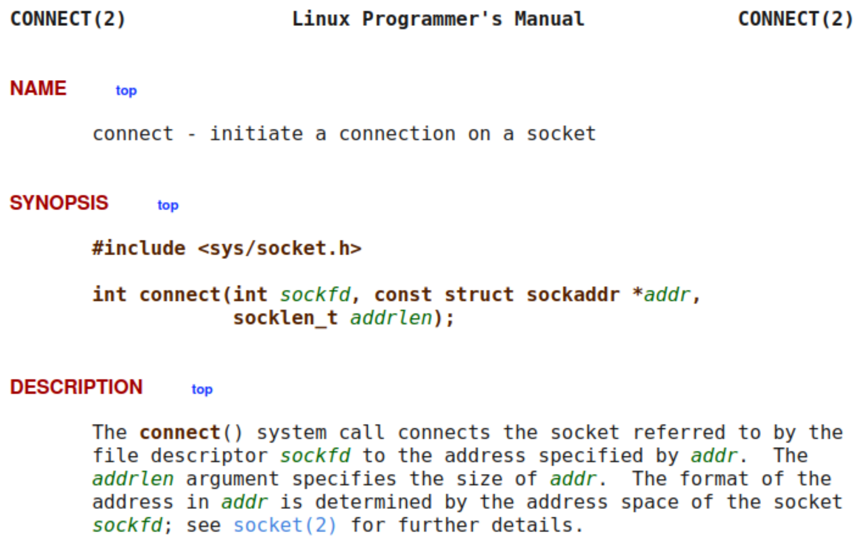
At the `fork()` point, the environment and the path condition of the main process are cloned. Then the parent process is symbolic executed first until `wait()` or `exit()` occur. Since in real system, the `wait()` function is called in case the parent process waits for a child process to end, the symbolic executed process of parent is halted then the cloned child process is started. At the convergence point of the parent and child processes, the environment and the path condition of multiple processes are joined by disjunction.

# Chapter 5

# External call specification extraction

## 5.1 API elements extraction

To correctly map from a C library function to an API Stub, it is necessary to identify the following information from the function description: library name, function name, function return type, number of parameters and type of each parameter.



```
CONNECT(2)              Linux Programmer's Manual              CONNECT(2)

NAME       top

       connect - initiate a connection on a socket

SYNOPSIS       top

       #include <sys/socket.h>

       int connect(int sockfd, const struct sockaddr *addr,
                   socklen_t addrlen);

DESCRIPTION       top

       The connect() system call connects the socket referred to by the
       file descriptor sockfd to the address specified by addr.  The
       addrlen argument specifies the size of addr.  The format of the
       address in addr is determined by the address space of the socket
       sockfd; see socket(2) for further details.
```

Figure 5.1: Description of *connect()* API from the manpage [48]

The Linux Programmer's Manual (manpage) documents the Linux kernel and C library interfaces. For C library interfaces, GNU C library (*glibc*) is the main focus, although other C library variations are also included. The manpage contains 8 sections as described by [48] as follows:

- User commands: Section 1 pages includes the command that supplied by the GNU C library.

- System calls: Section 2 contains the Linux kernel's system calls
- Library functions: Section 3 documents the functions provided by the standard C library
- Devices: This section documents details of devices in */dev*
- Files: Files section shows file formats and file systems
- Miscellaneous: Overviews, conventions, and other information
- System administration tools and daemons: Superuser and system administration commands are showed in this section, it also includes a small number of programs that are supplied by *glibc*.

To determine the information of a C library function, we need to extract the API synopsis of that function from the section 3 of the manpage. The synopsis contains the required *include* statements and the function declaration (Figure 5.1). For example, the following information will be extracted from the manpage [48] about the function *connect()*:

- Function name: **connect**
- Parameter names and types:
  - sockfd (**int**)
  - addr (**struct sockaddr**)
  - addrlen (**socklen_t**)
- Return type: **int**
- The **sockaddr** structure is definded in the glibc source code as:

```
struct sockaddr {
    __uint8_t   sa_family;
    char        sa_data[14];}
```

- **socklent_t** is definded in the glibc source code as:

```
typedef __socklen_t socklen_t;
typedef __uint32_t  __socklen_t;
```

## 5.2 Type conversion

This section discuss on the conversion of types in C language to JNA after successfully obtained type information in Section 5.1. As described in 4.2, passing the data from registers in the emulated environment (in Java) to the native stack requires correct identification of JNA types. To correctly invoke JNA function, Java types must be matching native types of the same size. However, only primitive types are directly passed by value through the registers, while pointers are passed by reference to a cell or a block of cells in memory. Therefore to convert C types to JNA equivalents, there are two cases: *primitive types* and *pointer types.* Noted that only types of parameters require correctly mappings, while parameter names are optional.

### Primitive types

The value of the primitive type variable is stored in the register or stack. As previously show in Table 4.1, the primitive types are directly mapped from native C to Java.

Table 5.1: Mapping primitive parameter types [41]

| C Type | Size | Java Type |
|---|---|---|
| char | 8-bit | byte |
| wchar_t | 16-bit | char |
| short | 16-bit | short |
| int | 32-bit | int |
| enum | 32-bit | int |
| long long, __int64 | 64-bit | long |
| float | 32-bit | float |
| double | 64-bit | double |
| long | 32-bit | NativeLong |
| other | integer types | int |

### Pointer types

In case of pointer types, the value of a pointer variable is a cell or a block of cells in the memory, and it is passed as a pointer that pointing to the memory location. In C language, array and pointer notations are closely similar and even can be used interchangeably in some cases.

Table 5.2: Supported pointer types in JNA [41]

| **C Type** | **Java Type** |
|---|---|
| pointer (e.g., int*,char*) // to element | Pointer (extends ByReference) |
| pointer (e.g., int*,char*) // to array/buffer | Buffer (extends ByReference) |
| array (e.g., void[], int[]) | P[] or Buffer |
| const char* | String |
| char** | String[] |
| void** | Pointer[] |
| struct* | Structure |
| struct[] | Structure[] |
| void (*FP)() | Callback |

As shown in Table 5.2, similar pointer declarations in C can have different meanings and mappings to Java. Therefore, it is necessary to correctly interpret C pointer variables. Pointer variables can be generally divided into several kinds in JNA: **Structure pointer, cell pointer, buffer, array, string and function pointer**. It is also important to note that the term "*pointer*" is strongly associated with the C/C++ and Java doesn't support pointer explicitly. In java, *reference* is used to point object/values.

- **Structure pointer (struct*)**

  Structure is an data type that can combines data items of different kinds. The structure in Java needs to be correctly defined corresponding to the native structure in C. The *Java Structure* represents a native *struct*. This type by default is *struct** - a pointer to structure on the native side. The reason is that if we passed the entire structure by value, then every byte of the structure would have to be copied. This is insufficient and may not be possible when the data of the structure is large. Therefore with passing by a pointer, we can access and modify the object through the pointer. In JNA, the data in a *jna.Java Structure* is automatically written to and read from the native memory. To pass a structure pointer as an argument, we need to define a subclass of *Java Structure* beforehand. Therefore, the structure descriptions in C have to be collected and mapped to the corresponding *Java Structure*. The types of members in structure are recursively mapped from C to the correct types in Java.

```
// Original C code
    typedef struct timeval {
        time_t tv_sec;
        suseconds_t tv_usec;
    } timeval;
    int gettimeofday(struct timeval *tp, void *tzp);
```

```
 // Equivalent JNA mapping
   @Structure.FieldOrder({ "tv_sec", "tv_usec" })
   public static class timeval extends Structure {
       public int tv_sec;
       public NativeLong tv_usec;
   }
   int gettimeofday(timeval tp, Pointer tzp);
```

  The contents of a *jna.Java Structure* will be updated after the function returns.

- **Cell pointer (e.g., int*, void*, char*)**

  A parameter can be a pointer that points to a memory cell. One of the main reasons for passing data by a pointer is to allow the function to modify the data. In this case, an address, which defines where the value is stored, is passed. As stated previously, *reference* is used to point to object/values in Java. Both cell pointer and buffer pointer are passed as type *ByReference* in JNA, but they still need to be distinguished to correctly load referenced value of the parameter. However, it is not clear from the API declaration that the pointer is pointing to one element or a sequence of elements.

```
// C code
int getsockname(int sockfd, struct sockaddr* addr, socklen_t* addrlen);
```

```
    // Equivalent JNA code
    int getsockname(int sockfd, sockaddr addr, IntByReference addrlen);
```

- **Array and Buffer (e.g., int[], gid_t[], int*, void*, char*)**

  A parameter can be a pointer that points to an array or a buffer (a sequence of variables). To use array/buffer as an argument, a pointer pointing to the first element is passed. In JNA, arrays of primitive type and sequences of elements can be defined as Java arrays or a *JNA Buffer* (extends *ByReference*).

  When passing an array to a function, an addition information also needs to be specified is the number of elements in the array. Based on the API signature, we have to decide which parameter describes the array/buffer length. In Linux, the immediate parameter after the buffer pointer isn't guaranteed to be the buffer length.

- **String (char* or char[])**

  String is an array of characters that ends by the null character "/0". For example, "debian" string is saved in the memory 0xbefff880 as:

  ```
  0xbefff880: 0x69 62 65 64   0x6e  61    0x0
          ->    i  b  e  d      n   a     /0
          ->    debian/0
  ```

  Different from an array, a string is not required to be passed along with the string length. In C function signatures, both array of characters and string are defined as *char** or *char[]*, so we have to distinguish such cases.

## 5.3 Deciding on pointer types for parameters

Although passing parameters from the Java stack to the native stack is handled by JNA, Java parameters have to be in the same size and same type as native ones in C language. As stated in 5.2, argument types in C can be divided into two main kinds: primitive and pointer types. In general, for correctly invoking JNA call, the types needed to be mapped following the ***Type Conversion Rule*** shown in Figure 5.2. Primitive types are directly mapped from C to Java following Table 5.1. As for pointer types, there are cases when the same C type identification corresponds to different types in Java (e.g., `char*` in C can be mapped to `Buffer` or `String`). We decide the type mapping statically based on some observations on the naming convention of parameters in the Standard C library.

There are three problems regarding deciding pointer types for JNA parameters.

- ⟨*Problem* 1⟩ Differentiate character array and String

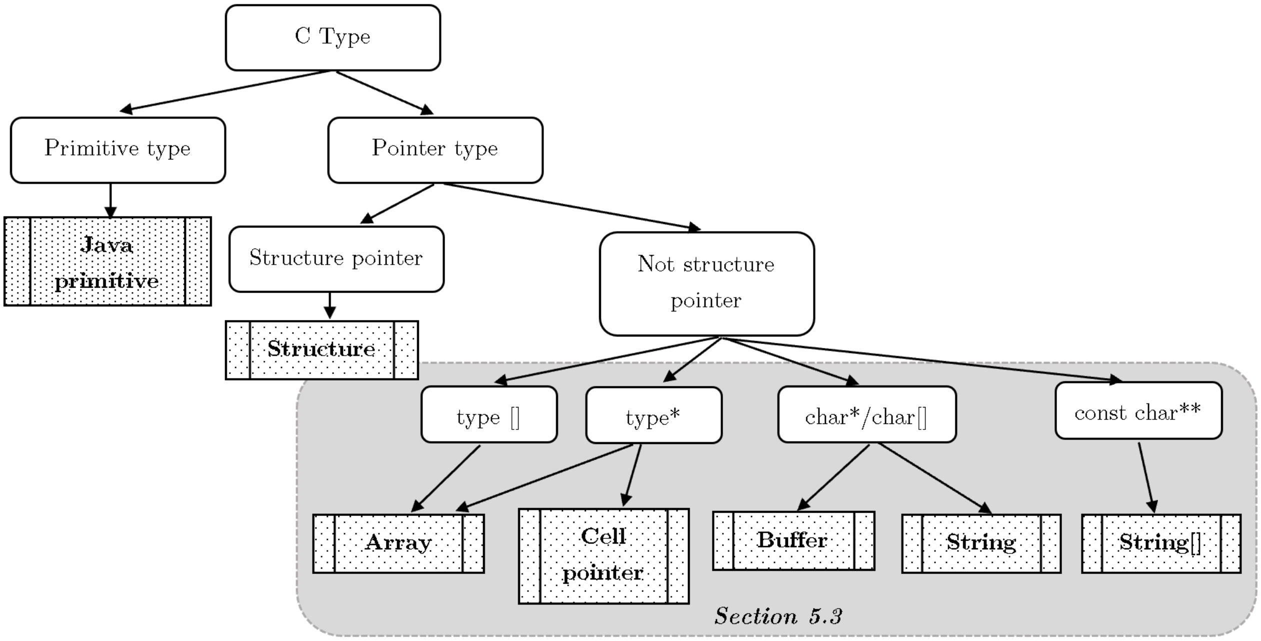


Figure 5.2: Type conversion rule from C to JNA

From the function declaration, both character array and String are defined as `char*` or `char[]`. Fortunately, we can decide these cases based on parameter naming convention of the standard C library.

- String parameters in C library function are defined as type `const char*`.
- Buffer array parameters are usually named as `buf`, `buffer`, `__buf`, . . .
- When passing a buffer as parameter, it is required for the function to have another parameter that describe the size of the buffer, since the C compiler needs to know how many bytes from the memory the buffer will take. Therefore, if the passing argument is a buffer, there exists a parameter in the signature that has type `size_t`, `len_t` or named as `size`.

- ⟨*Problem* 2⟩ Find the parameter that specify number of elements in array

  Similar to the previous case, the argument that specify the number of elements in an array needs to be passed along with the array pointer. That argument is usually set as type `size_t`, `len_t`, `int` or named as `size`.

- ⟨*Problem* 3⟩ Differentiate cell pointer and array/buffer pointer

  A pointer variable can point to either a cell or a sequence of elements. Based on the observation, it is required that there is another parameter with type `size_t`, `len_t` or named as `size` to specify the number of elements if the function needs to pass a pointer as an array/buffer pointer.

# Chapter 6

# Automated API Stub generation

To automated generate API Stubs, there are three main phases: Extraction, Conversion, and Generation. From the collected documents (e.g., Linux manual page, GlibC index page, GlibC source code), the required API elements are extracted, then C types are converted into JNA as described in Chapter 5. This chapter illustrates the process of generating three types of Java classes needed for API Stub, which are structure class, library proxy interface class, and API stubs.

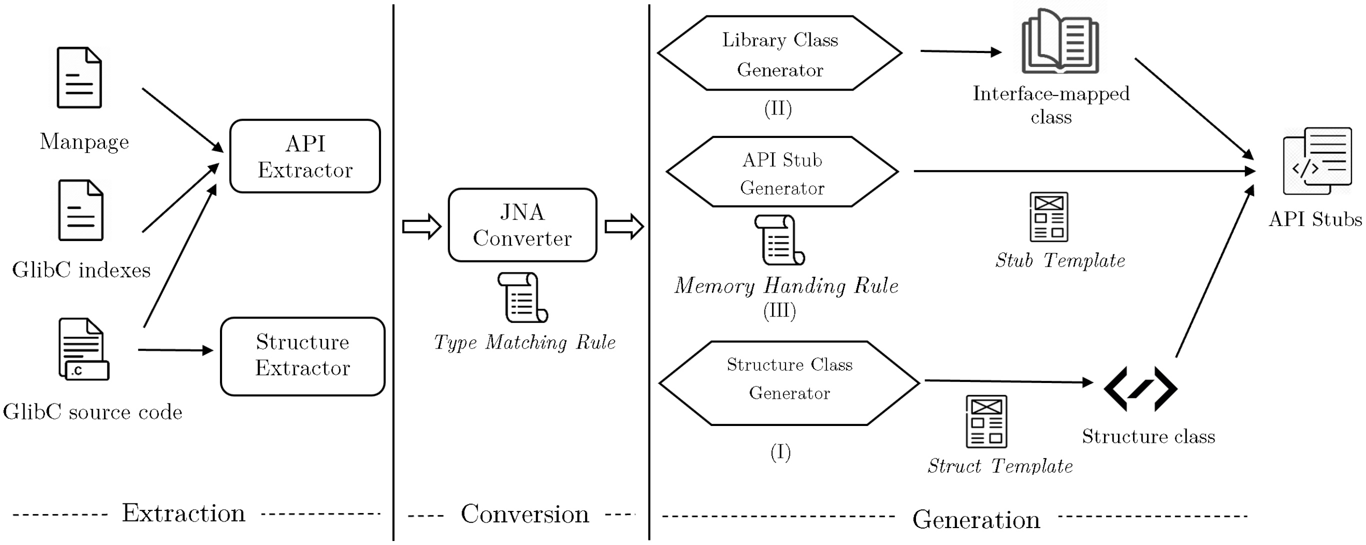


Figure 6.1: Generation of structure, library interface and API stub classes

## 6.1 Generated Java Classes

### 6.1.1 Structure definition class

Structure description is given by source code in the GlibC library. To use the native structure in Java, an equivalent Java class derived from `Structure` needed to be defined. For example, the `termios` struct in C taken from the GlibC source code is given as follows.

```
struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t     c_cc[32];
    speed_t  c_ispeed;
    speed_t  c_ospeed;
};
```

The `termios` class is defined Java as below.

```
@Structure.FieldOrder({"c_iflag", "c_oflag", "c_cflag","c_lflag",
"c_cc","c_ispeed","c_ospeed"})
public static class termios extends Structure {
        public int c_iflag;
        public int c_oflag;
        public int c_cflag;
        public int c_lflag;
        public byte[] c_cc = new byte[32];
        public int c_ispeed;
        public int c_ospeed;
```

Each JNA `Structure` class needs to satisfy the following conditions.

- The struct class is derived from the class `JNA.Structure`
- The number and order of struct fields have to be kept.
- All fields of the class in Java need to match all fields of the structure in C by the type conversion rule as described in 5.2. If a structure is a nested structure, other classes of sub-structure also have to be generated.
- The list of struct fields are defined at the beginning with `@Structure.FieldOrder`. This annotation is mandatory and needs to be defined for each `Structure` class.

### 6.1.2 C library interface class

To access methods in the native C library, a class interface corresponding to that library, which contains native proxies, is needed. The generated interface has to meet some required conditions.

- The generated interface is derived from `jna.Library`.
- Function names are mapped directly to the native library function.

- Types of parameters and return values are converted based on type matching rule in 5.2.

- The name of fields is kept as the original name although it is optional.

For example, the `CLibrary` class that defining some functions such as `printf, sprintf, ...` can be defined as follows.

```
public interface CLibrary extends Library {
    CLibrary INSTANCE = Native.loadLibrary("C", CLibrary.class);
    // printf (const char __restrict__fmt, ...)
    void printf(String __ restrict_fmt, Object... args);

    //int sprintf (char* str, const char* format, ...)
    int sprintf(char[] str, String, format, Object... args);
}
```

### 6.1.3 API Stub class

The API Stub class is the main component that actually performs the needed actions to invoke and manage the external call. Each API Stub performs the following sequence of actions.

- Firstly, the parameter addresses are getting from the registers based on ARM Calling Convention (AAPCS).

- Secondly, the parameter variables are initialized and loaded from the memory. To retrieve values from memory, the loading function for each type has been prepared based on memory allocating rules.

- Thirdly, the parameters are passed to the API and invoked from the instance of the mapped interface. This library function proxy needs to be predefined in the mapped interface class (as mentioned in 6.1.2).

- In the end, the return value from the invoked function is saved into the register R0. The referenced memory contents are also updated based on the memory allocating rules if needed.

```
public static void gettimeofday (Environment env) {
        // I. Retrieve parameters' addresses
        BitVec t0 = env.register.get('0');
        BitVec t1 = env.register.get('1');

        // II. Load parameters from memory
        timeval param0 = new timeval();
        timezone param1 = new timezone();
        param0.tv_sec = env.memory.getIntFromRef(t0);
        param0.tv_usec = env.memory.getNativeLongFromRef(t0.add(4));
        param1.tz_minuteswest = env.memory.getIntFromRef(t1);
        param1.tz_dsttime = env.memory.getIntFromRef(t1.add(4));

        // III. Invoke JNA interface
        int ret = CLibrary.INSTANCE.gettimeofday(param0, param1);

        // IV. Update the memory
        env.register.set('0', new BitVec(ret));
        env.memory.setInt(t0, param0.tv_sec);
        env.memory.setNativeLong(t0.add(4), param0.tv_usec);
        env.memory.setInt(t1, param1.tz_minuteswest);
        env.memory.setInt(t1.add(4), param1.tz_dsttime);
    }
```

## 6.2 Automated code generation

Based on the descriptions of each class type in 6.1, we automated the generation processes of those classes using the previously discussed conventions and observations. To prepare for the API Stub classes, Structure classes and C library interface classes need to be generated beforehand.

### 6.2.1 Structure class generation

As mentioned before, to deal with native `struct`-type variable, an equivalent Java `Structure` needs to be defined in JNA. For each `struct` type in C, the following generation flow is applied to produce the `Structure` class in JNA.

- First, native structure definition is extracted from C Library (GLibC) source code using pattern matching.

- After that, each member and its corresponding type are identified and changed to the peer member in `Structure` by applying *Type Conversion Rule*. JNA also requires the `@FieldOrder` annotation to serialize data into memory buffer before using the structure as an argument.

- Finally, the converted members are filled based on the following prepared template to generate a structure class.

```
// Template for Structure class
@Structure.FieldOrder(){"member_0", "member_2", ...}
public static class $struct_name extends Structure{
    public type_0 member_0;
    public type_1 member_1;...}
```

**Example 6.2.1**
To illustrate for the generation process, we consider the library function `connect()`. In the declaration of the function `connect()`, the struct `sockaddr` is one of the parameter fields. Therefore, the structure class needs to be defined in order for the function `connect()` to be correctly invoked.

```
@Structure.FieldOrder({"sa_", "sa_data"})
   public static class sockaddr extends Structure {
       public int sa_;
       public int[] sa_data = new int[14];
   }
```

### 6.2.2 Library interface-mapped class generation

As stated in 6.1.2, an interface class corresponding to target C library, which contains native proxies, is needed. We collected 1659 library function names from the function index page of GlibC [1]. For each function, its native declaration is extracted through *Linux man-page*. In general, the function interface in JNA is similar to the native function definition. Function names are mapped directly from the symbol exported by the native library to Java interface name, while types are converted to JNA following *Type Conversion Rule*.

**Example 6.2.2**
The C declaration of function `connect()` is extracted from the C source code then converted to JNA as follows. `sockaddr` class is generated as in Example 6.2.1.

```
   // C function declaration
       int connect(int __fd, sockaddr *__addr, socklen_t __len);

   // Library class with the JNA function proxy
   public interface CLibrary extends Library {
       CLibrary INSTANCE = Native.loadLibrary("C", CLibrary.class);
       int connect(int __fd, sockaddr __addr, int __len);
   }
```

### 6.2.3 API stubs generation

The API Stub class is the main component for dealing with the external call. Same as the previous section, the function declarations needs to be collected from *Linux man-page*. After obtaining the function declaration in C, the API Stubs generation process is performed by three main stages as illustrate by Figure 6.2.

(I) From the function declaration, API elements (e.g., function name, parameter fields, return type) are extracted.

[1] *https : //www.gnu.org/software/libc/manual/html_node/Function − Index.html*

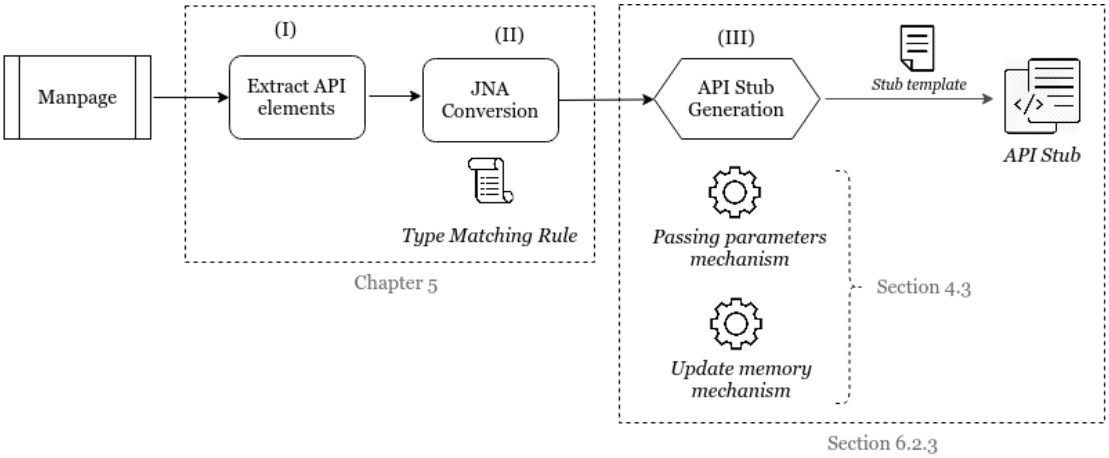


Figure 6.2: API Stub generation

(II) The return type and parameter type are converted from C to JNA based on the *Type Matching Rule* in Section 5.2.

(III) After deciding the types of parameters, API stub is generated adhere to the passing arguments mechanism in ARM and the memory update mechanism (as discussed in Section 4.3). Based on C memory allocating rules, we prepared required functions for retrieving parameter values from the memory and saving updates to the memory. For example, the function `getTextFromAddress()` returns a `String` by continuously read the memory at the input address until find the "/0" character.

```
    public String getTextFromAddress(BitVec atAddress) {
        String text = "";
        String word = atAddress.getSym(); // The address in hexadecimal
        while (!Memory.getValue(word).contains("00")) {
            text += HexToASCII(Memory.getValue(word));
            word = nextWord(word);
        }
        return text;
```

(IV) Finally, API Stubs are generated follows the defined template.

```
public static void $functionName (Environment env) {
//1: Get original parameters from registers
       BitVec t0 = env.register.get('0');
       BitVec t1 = env.register.get('1');
        ...
//2: initialize input parameters
       $type0 param0 = new $type0();
       $type1 param1 = new $type1();
//3: read parameter values from memory
       param0 = env.memory.$getMemoryValue(t0);
       param1 = env.memory.$getMemoryValue(t1);

//4: call API function
$return_type ret = CLibrary.INSTANCE.$functionName(param0, param1, ...);

//5: update registers and memory
       env.register.set('0', new BitVec(ret));
       env.memory.$setMemoryValue(t0, param0);
       env.memory.$setMemoryValue(t1, param1);
}
...
```

### Example 6.2.3

Continuing from Example 6.2.1 and Example 6.2.2, we consider the API Stub generation of function `connect()`. Below is the declaration in C language.

```
int connect(int __fd, struct sockaddr *__addr, socklen_t __len);
```

- Firstly, the API elements are extracted from the declaration.
  - Function name: `connect`
  - Parameter types: `int, struct sockaddr*, socklen_t`
  - Return type: `int`
- Based on Section 5.2, we translate C types to equivalent JNA types.
  - `int` → `int`
  - `struct sockaddr*` → `sockaddr` (derived from `JNAStructure`)
  - `socklen_t` → `int`

  The JNA class of `sockaddr` is generated as in Example 6.2.1. Noted that, JNA structure by default is loaded by reference.
- The passing parameters and update memory mechanisms are applied to generate the API Stub.

```
    public static void connect(Environment env) {
        BitVec t0 = env.register.get('0');
        BitVec t1 = env.register.get('1');
        BitVec t2 = env.register.get('2');

        int param0;
        sockaddr param1 = new sockaddr();
        int param2;
        param0 = env.memory.getIntFromReference(t0);
        param1.sa_ = env.memory.getIntFromReference(t1);
        param1.sa_data = env.memory.getIntArray(t1.add(4), 14);
        param2 = env.memory.getIntFromReference(t2);

        int ret = CLibrary.INSTANCE.connect(param0, param1, param2);

        env.register.set('0', new BitVec(ret));

        env.memory.setIntReference(t1, param1.sa_);
        env.memory.setIntArray(t1.add(4), 14, param1.sa_data);
    }
```

# Chapter 7

# Experiments

This chapter presents the result of the API Stub generation. After that, to demonstrate how the supported API Stubs help extend CORANA ability to analyze ARM binaries, we show the execution trace performance of the extended CORANA with the API supports (CORANA/API). Finally, we use CORANA/API to do a thorough analysis on a binary sample of Mirai malware.

## 7.1 CORANA/API performance

### 7.1.1 Generated API Stubs

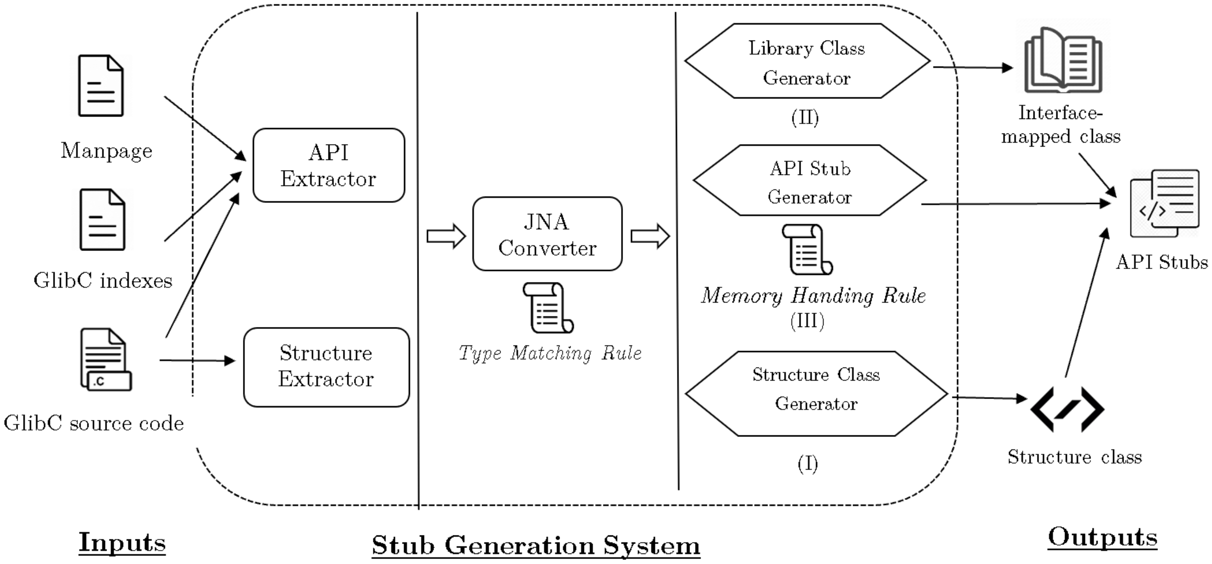


Figure 7.1: API Stub generation system

The implementation module for API Stub generation is written in Python. The generated API Stub themselves are in Java since the previous extracted formal semantic of

ARM instructions [21] are also written in Java. For automated API Stub generation, we collected the following documents:

- Linux Manual Page
- GNU Standard C Library (GlibC) source code
- GlibC function indexes[1]

Since the source code of GlibC contains a huge number of functions but not every function is an interface that allows systems to use, we restrict the extracted API descriptions according to the GlibC function index page. Based on 1659 API descriptions, we generates:

- 267 structure definition classes
- 1129 API Stubs

The API stub generation fails on about 500 APIs due to the presence of function pointers and unknown parameters.

### 7.1.2 Trace generation performance

We perform experiments of execution trace construction for real-world IoT malware binaries taken from *VirusShare*, and non-malware samples from *the Linux official repository*. The trace generation is performed on Ubuntu with Intel i9-10900K, 3.70GHz, and 128GB.

Previously, we have introduced the DSE for ARM Cortex-M tool CORANA in [21]. Although CORANA (FM19) has high coverage of implemented ARM instructions, with an average of 66% coverage across 6 ARM Cortex-M variants, it lacks support for memory data loading and external calls handling. Therefore, we use both versions CORANA and CORANA/API[2] (extension with API support) to generate execution trace of malware samples to show how API handling support affecting to the result of DSE tools.

Table 7.1 shows the result of execution trace generation on some malware and non-malware samples comparing between CORANA with and without API support, in which `Time` is reported in milliseconds. As shown in the table, two most common error leads to the termination of the previous version of CORANA (FM19) is unloaded memory and non-supported external library calls. The higher number of nodes in the trace results partially shows that CORANA/API can continue the execution longer and explore more states of the program.

However, with some cases such as `Generic.2e5841` and `Generic.34b8ad`, the nodes of CORANA (FM19) is higher. This is due to the fact that CORANA (FM19) does not distinguish between external call and normal jump instruction (Figure 7.2). Therefore

[1] https://www.gnu.org/software/libc/manual/html_node/Function-Index.html

[2] https://github.com/vananhnt/corana

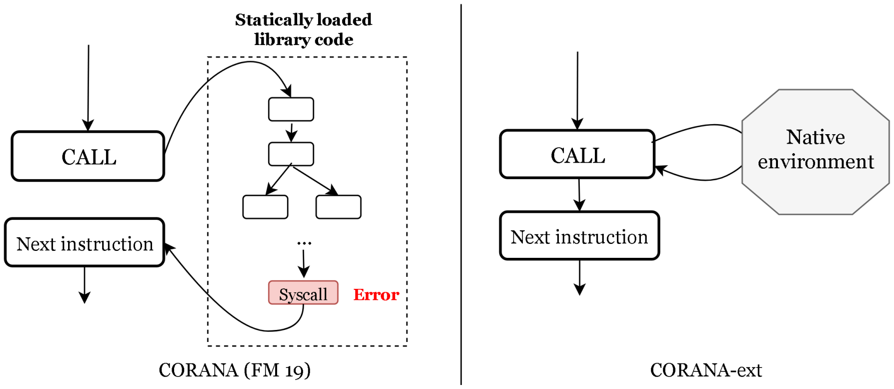
with the statically loaded library, CORANA (FM19) when encounters library call will jump to the library code instructions, and catch an error when reaching to system call since the library code is just a wrapper of the system call.

Table 7.1: Result of execution trace generation

| **Example** | **Size(kB)** | **Ext. calls** | **CORANA (FM19)** | | **CORANA/API** | | |
|---|---|---|---|---|---|---|---|
| | | | **Nodes** | **Edges** | **Nodes** | **Edges** | **Time (ms)** |
| malware | | | | | | | |
| ① *CORANA(FM19) ends due to unloaded memory* | | | | | | | |
| Mirai.Gen10.31814e | 147 | 85 | 9 | 8 | 2956 | 3314 | 625740 |
| Mirai.Gen10.32caff | 157 | 85 | 9 | 8 | 3595 | 4033 | 867740 |
| Mirai.348b61 | 117 | 93 | 9 | 8 | 2306 | 2544 | 835449 |
| Generic.34a264 | 147 | 132 | 20 | 21 | 494 | 514 | 27870 |
| Gafgyt.Gen5.341feb | 93 | 112 | 36 | 36 | 500 | 551 | 448481 |
| Gafgyt.Gen7.34dbce | 159 | 140 | 36 | 36 | 588 | 617 | 31620 |
| Gafgyt.Gen44.34a085 | 150 | 128 | 36 | 36 | 835 | 904 | 195338 |
| Tsunami.Gen4.35a82c † | 741 | 223 | 22 | 23 | 47 | 46 | 12863 |
| | | | | | | | |
| ② *CORANA(FM19) ends due to non-supported library call* | | | | | | | |
| Tsunami.Gen5.34c430 † | 112 | 123 | 74 | 74 | 193 | 197 | 5186 |
| Gafgyt.Gen15.34d921 | 117 | 119 | 74 | 74 | 432 | 465 | 7911 |
| Gafgyt.3493e6 | 133 | 115 | 73 | 73 | 410 | 434 | 17772 |
| Gafgyt.5.0ac271 | 208 | 93 | 128 | 132 | 577 | 611 | 3832 |
| Generic.2e5841 | 141 | 129 | 112 | 117 | 408 | 436 | 7508 |
| Generic.64110.34b8ad | 103 | 115 | 844 | 843 | 422 | 453 | 19989 |
| | | | | | | | |
| non-malware | | | | | | | |
| chcon | 706.8 | 177 | 255 | 263 | 278 | 368 | 26046 |
| chgrp | 759.9 | 202 | 275 | 274 | 233 | 299 | 26371 |
| chmod | 682.3 | 175 | 157 | 156 | 294 | 382 | 25174 |
| getlimits | 807.2 | 182 | 9 | 8 | 63 | 78 | 24056 |
| hostid | 607.1 | 190 | 6 | 6 | 56 | 61 | 17494 |

On termination, among the examples, CORANA/API successfully performs symbolic execution until reaching the end of the programs on several variants except for two Tsunami malware samples. Their execution is interrupted due to encountered an unsupported API or an unsupported instruction.

In short, the comparison shows that by extending the support of API call handling, CORANA/API has been able to produce more reasonable traces of binaries, although there are cases that CORANA/API encounters errors due to unsupported instructions or library calls. To display the potential of CORANA/API, we conduct a more detailed observation on the produced traces of CORANA/API against some typical obfuscation techniques and compare them to `angr`.



CORANA (FM 19)

CORANA-ext

Figure 7.2: Comparison on external call handling

Noted that, `angr` is a powerful binary analysis framework, and its capability depends on how we define the running scripts. As discussed before, `angr` allows users to hook to modify program behavior, especially hooks can be used to fix the result of the external call as specified by users. In this experiment, we only compare CORANA/API with the default symbolic execution strategy in `angr` to produce the control flow graph, which is `CFGEmulated`. In case `CFGEmulated` is fail, the static analysis `CFGFast` is used to generate control flow graph of `angr`.

**Indirect Jumps.** CORANA/API is able to dynamically handle the indirect jump. In sample `Mirai.32caff`, at `0x175c0`, there is an indirect jump to the address stored at *lr*. After resolving the value of *lr*, we found the destination of the indirect jump is `0x17514` and determined the next instruction accordingly. `angr` fails to resolve the indirect jumps at this location.

```
CORANA
0x175b4      add ip,ip,#1
0x175b8      cmp ip,r2
0x175bc      bne #0x175ac
0x175c0     bx lr
-> Found the destination: 0x17514
0x17514      mov r3,#0x2e4
```

```
angr
0x175b4      add ip,ip,#1
0x175b8      cmp ip,r2
0x175bc      bne #0x175ac
0x175c0     bx lr
0x17c0       mov ldr r3,[r0,r3]
```

**Self-modification.** A common strategy in many binary analysis tools is lifting binary code to an Intermediate Representation (IR) such as VEX, LLVM, and BAP. However, VEX assumes the code being lifted is not self-modifying, which leads to analysis tools being unable of discovering actual actions of code. *angr* is a binary analysis tool that lifting binary code as basic blocks of *VEX IR Statements*. We consider an example collected from the *ARM Community page* that describes self-modifying code[3]. The execu-

[3] https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/caches-and-self-modifying-code

tion trace below shows that the code tries to append instructions at the memory location 0x109c0. At instruction 0x10bb8, **angr** fails to analyse the memory that loaded into r3 at [fp, #-0xc], while CORANA/API successfully resolved the value stored at r3 and executed the instructions at 0x109c0.

```
CORANA
0x10aec      movw r3,#0x9c0
0x10af0      movt r3,#1
0x10af4      ubfx r3,r3,#0,#0xc
0x10af8      orr r3,r1,r3
0x10afc      orr r3,r2,r3
...
0x10bb4      ldr r3,[fp,#-0xc]
0x10bb8      blx r3
        -> Indirect Jump to 0x109c0
0x109c0      push fp,lr
...
```

```
angr
0x10aec      movw r3,#0x9c0
0x10af0      movt r3,#1
0x10af4      ubfx r3,r3,#0,#0xc
0x10af8      orr r3,r1,r3
0x10afc      orr r3,r2,r3
...
0x10bb4      ldr r3, [fp, #-0xc]
0x10bb8      blx r3

0x10bbc      ldr r3, [fp, #-0x10]
...
```

**Anti-debugging (External call handling).** The library function call ptrace can be utilized to detect debugging process. The reason is that ptrace[PTRACE_TRACEME] cannot be called in succession more than once for a process and debuggers use this call to setup debugging. In the code belows[4], ptrace(PTRACE_TRACEME,...) is called at 0x109d8. In this case, CORANA/API executes the API Stub of ptrace and throws the function to the actual OS environment. Since the process of CORANA/API does not use another ptrace process likes debuggers (e.g., GDB, EDB), the return value will be $\neq -1$ and saved to the register. At 0x109fc, the jump instruction is satisfied and the analysis is continued since ptrace returns $\neq -1$. angr also correctly traces the next destination of the conditional jump instruction at 0x109e4. This technique is normally used to deter dynamic analysis of binary executables.

```
CORANA
0x109d4      mov r3,#0
0x109d8      blx #0x218a4
        -> Call to: ptrace
0x109dc      mov r3,r0
0x109e0      cmn r3,#1
0x109e4      bne #0x109fc
0x109fc      movw r0,#0x4e20
0x10a00      movt r0,#5
```

```
angr
0x109d4      mov r3,#0
0x109d8      blx #0x218a4  // ptrace

0x109dc      mov r3,r0
0x109e0      cmn r3,#1
0x109e4      bne #0x109fc
0x109fc      movw r0,#0x4e20
0x10a00      movt r0,#5
```

[4] https://gist.github.com/vananhnt/34eb34f92026fd0384782cb78ec39776

## 7.2 Analysis of Mirai malware sample

In this section, we show how our Dynamic Symbolic Execution Tool CORANA/API explores the execution traces of a sample of Mirai - a popular IoT Botnet. Mirai has been used to compromise approximately 500,000 IoT devices to perpetrate some of the largest DDoS attacks [49], one of them is the take-down of Dyn DNS service (2016) with the biggest traffic of a DDoS attack ever recorded [50]. The malware can be used to perform several types of DDoS attacks, from basic SYN Flood to exploiting many protocols (e.g., GRE, TCP, UDP, DNS, and HTTP).

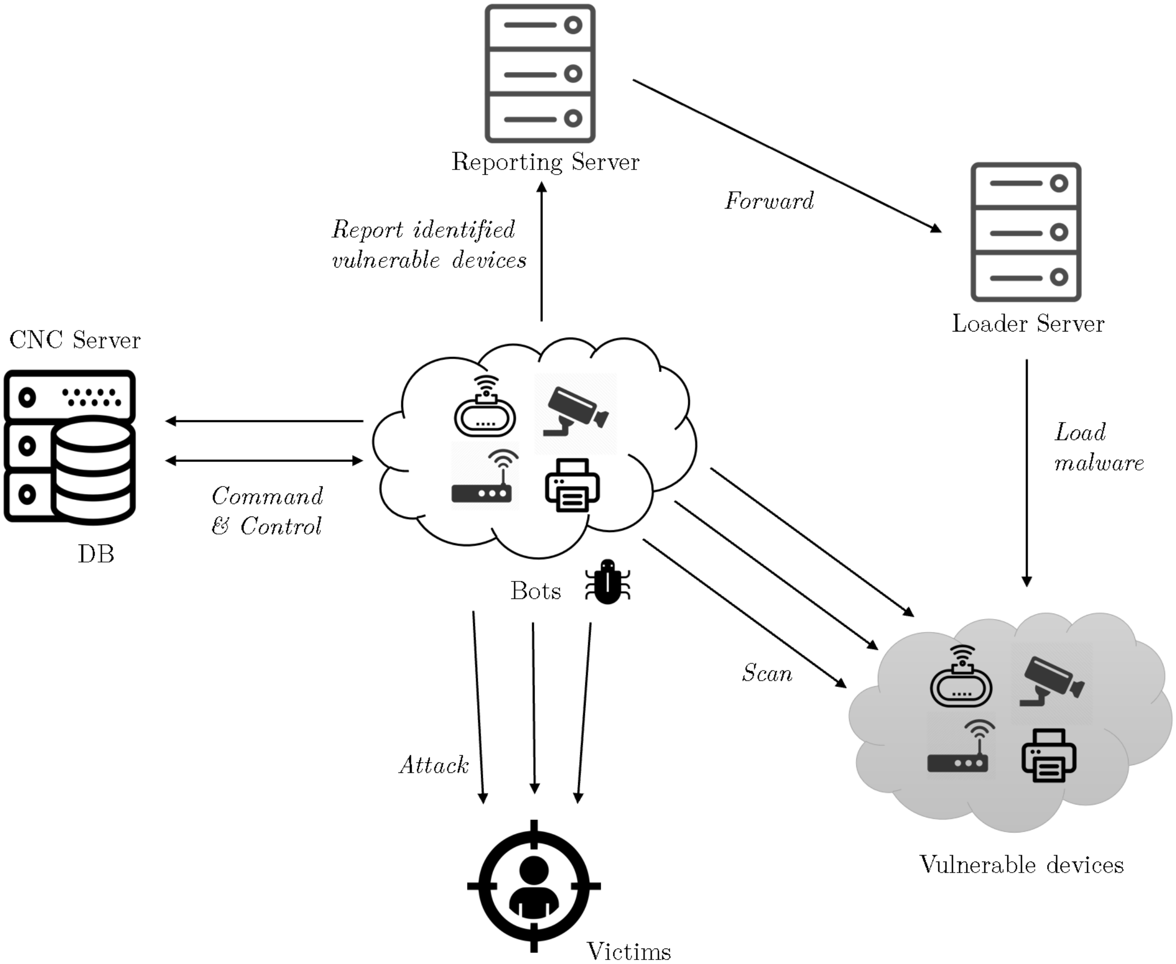


Figure 7.3: Mirai logical infrastructure [51]

A bot is the actual Mirai malware that runs on the infected device (Figure 7.3). It runs several tasks in both the main thread and background. Based on statically analyzing the published source code of the malware, we observed that the Mirai bot is composed of a main part and three submodules:

- **Scanner:** Scans for new vulnerable IoT devices, tries to access the devices, and sends them back to the Reporting Server.

- **Killer:** Kills other running malware on the infected device to protect all the computational resources and prevent itself from being removed.

- **Attack:** This module performs the DDoS attack when received requests from the CNC server.

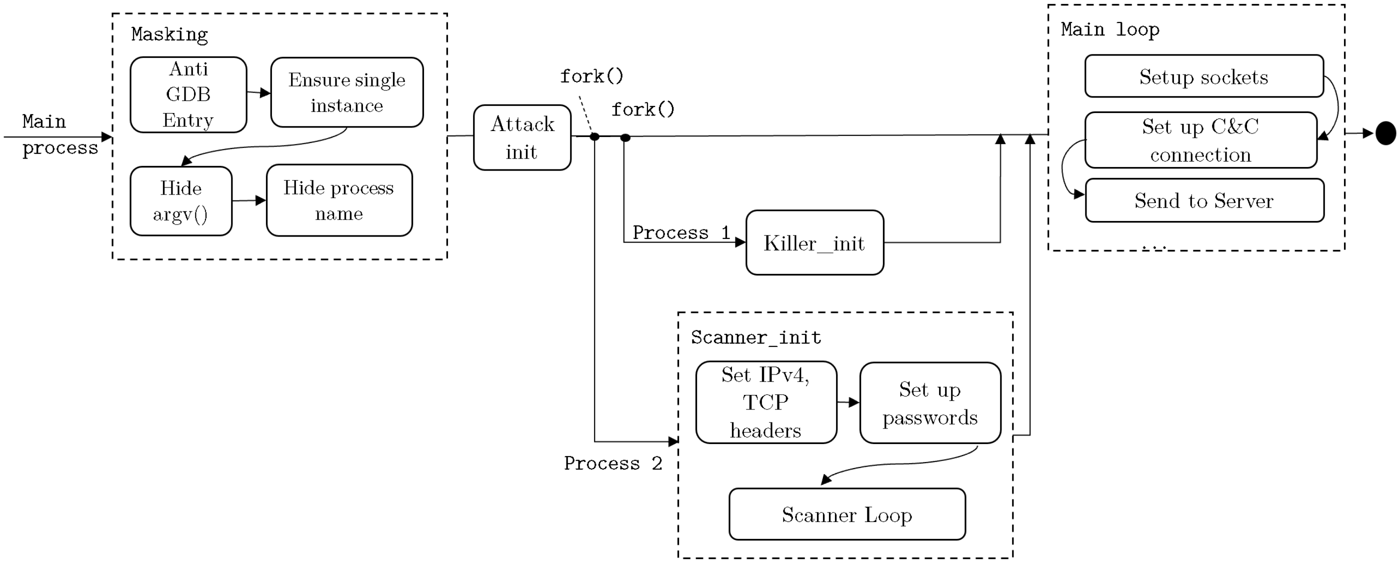


Figure 7.4: Mirai execution flow

The execution of the Mirai bot is analyzed starting from the beginning of the main() function using CORANA/API. We observe the execution flow of the analyzed Mirai variant, which is generally described in Figure 7.4. The execution trace[5] produced by CORANA/API are analyzed in detail in Appendix A.

```
+ Analyzing samples/32caf/32caff26a4dfa373cd0ed869544a30b7$ ...
-> Capstone disassembler elapsed: 1036ms
-> Exporting to .capstone-asm file ...
 + Parsing samples/32caf/32caff26a4dfa373cd0ed869544a30b7$ ...
-> Executing 11d80 : push {r4,r5,r6,r7,r8,sb,sl,fp,lr}
-> Executing 11d84 : sub sp,sp,#0x540
-> Executing 11d88 : sub sp,sp,#0xc
-> Executing 11d8c : add r4,sp,#0x530
-> Executing 11d90 : mov r5,#0
-> Executing 11d94 : mov r6,r0
-> Executing 11d98 : mov r0,r4
-> Executing 11da4 : bl #0x18db8
     === Call to library function: sigemptyset ...
```

[5] https://gist.github.com/vananhnt/7440272e27106f835261f92db3d3452e

The detail analysis of Mirai is explained in Appendix A. Table 7.2 shows the summarized results of the execution traces produced by CORANA/API and `angr` when analyzing the same malware sample. `angr`'s CFGEmulated cannot be generated due to `angr` unable to analyze the large binary file. Therefore, only static analysis is possible when using `angr` to analyze the malware sample. CFGFast computes a much larger control flow graph than CORANA/API in a short time since the goal of CFGFast is to generate a graph with high code coverage without caring about reachability. The high number of edges generated by `angr` shows that `angr` detects multiple paths but lacks most of the control flow. This graph is similar to other static analysis tools such as IDA Pro and Ghidra, which by nature are fast but easily cheated by control flow obfuscation techniques (e.g., opaque predicate, self-modifying code).

Table 7.2: Results for execution trace generation

| | **Nodes** | **Edges** | **Time (ms)** |
|---|---|---|---:|
| CORANA/API | 3595 | 4033 | 867740 |
| Angr (CFGFast) | 6025 | 12957 | 491 |
| Angr (CFGEmulated) | | | *Error* |

# Chapter 8

# Conclusion

## 8.1 Conclusion and Current limitation

The formal semantics of ARM instructions have been extracted to preliminarily built a dynamic symbolic execution tool CORANA[21]. To extend the ability of CORANA, throughout our study, we investigate the feasibility of using API Stub to handle the interaction between the dynamic symbolic execution process and the operating system. Based on the observed conventions of the C function interfaces and Linux system on ARM, this thesis proposed an approach to systematically generate Linux API Stub from the C library function interface description. We have constructed a system for API stub generation.

- The system can automatically extract API and structure descriptions from Linux manual page and Glib source code.
- With 1659 collected API descriptions, we have successfully auto-generated 267 structure definition classes and 1129 API Stubs for library functions calls of Linux system run on ARM.
- Our generation method can be generalized to multiple platforms when handling external interaction with Linux OS.

After successfully generated the API Stubs, we extend CORANA/API to be able to deal with external system function calls.

- The generated APIs Stub allows CORANA/API to be able to continue the execution when encountering external function calls. When analyzing some malware samples, we have confirmed that the execution trace correctly produces under the presence of obfuscated code.
- The path condition is kept the same before and after performing the API Stub.
- We handle the concurrent processes by serialization with the assumption that processes are executed without overlapping and do not interact with each other. Thus,

we are able to trace the action performs by child processes, which is really important since malware often creates multiple processes in its execution.

### Current limitation on API Stub generation

Since our approach determined the parameter's types statically, there are cases when the type cannot be decided and need more detailed investigation.

**Void pointer.** In C, a void pointer is a pointer that has no associated data type, but the type of parameters has to be defined beforehand so the compiler knows how to load the function. Generally, from the API description, it is not clear what kind of objects a void pointer points to. However, in some cases, even if a parameter is a void pointer, we successfully derived what kind of objects are pointed from our statically matching type rule. Still, a void pointer can be decided more correctly if we investigate the natural language description of the function calls.

**Function pointer.** A function pointer points to code, not data. We do not know how many cells needed to be copied into the emulated memory. Moreover, as the emulated memory in Java and actual memory of the system is not the same, the return address of the function pointer in JNA will point to a location in the actual system memory. Therefore, a method needed to be built to correctly load function pointers into the emulated memory.

## 8.2 Future works

### Automatic generation of test cases for API Stubs

To perform conformance testing for verifying the generated API Stubs, we need to automatically generate test cases of C library function calls. A method is to randomize input values and execute the function call in the actual environment to obtain the test case. However, this is shown to be difficult in our preliminary investigation. The main reason is that function call requires valid inputs to be able to execute in the system. Another reason is that we need to generate C files to execute the library functions in the actual system. Due to the complexity of the C syntax, the automatically generated C files might encounter multiple errors in the compilation. We can also consider another approach, which is investigating a third-party tool to automatically generate test cases.

### Control flow graph construction

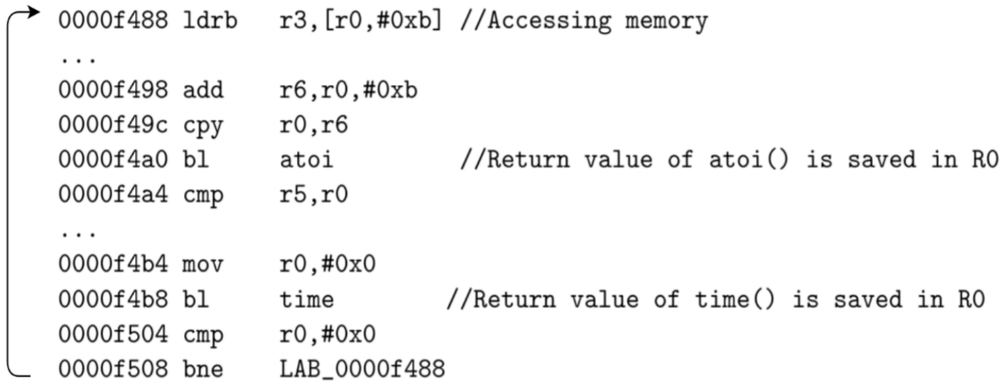
Currently, CORANA is able to generate execution traces of the analyzed binary samples. However, to correctly construct a CFG from the trace, we have to investigate how to define a model for the CFG, especially in the presence of typical obfuscations (e.g., self-modification).

### Stub for external system communication

A user process program is not only interacting with the underlying kernel system but also with external devices (e.g, servers, peer devices) over the internet network or other connections. This is especially true in the case of malware, C&C server is the headquarter that perform attacks and control the malware running on the infected devices. Therefore, the behavior of malware largely depends on the packets and signals they receive from other systems. A detailed investigation on how to construct *stubs* for connection with external systems is necessary to exploit all behavior of malware samples.

### Loop invariant generation

In the current implementation, we have to set an upper bound on the number of loop unrollings. *Loop invariant* can be used to handle loop. Dealing with loops is one of the main difficulties in symbolic execution, especially in the binary case. Due to the lack of syntactic structure, *"what is a loop"* is not clear. Therefore, it is important to specify the definition of loop and Hoare-logic rules in binary. The goal is to propose a loop invariant generation method targeting binary executables of typical loop structures in IoT malware (e.g., ARM Cortex-M-based malware). Automatically constructing inductive loop invariants is a classical problem in program analyses, however, they mainly focus on high-level languages whose syntax of a loop statement is clearly defined.



```
0000f488 ldrb   r3,[r0,#0xb] //Accessing memory
...
0000f498 add    r6,r0,#0xb
0000f49c cpy    r0,r6
0000f4a0 bl     atoi         //Return value of atoi() is saved in R0
0000f4a4 cmp    r5,r0
...
0000f4b4 mov    r0,#0x0
0000f4b8 bl     time        //Return value of time() is saved in R0
0000f504 cmp    r0,#0x0
0000f508 bne    LAB_0000f488
```

Figure 8.1: An example of loop in Mirai

Many methods (e.g., Farkas' Lemma [52], Craig interpolation [53], and the learning-based approach [54]) had been proposed. Farkas' Lemma is an effective technique to produce linear inductive invariants by extracting nonlinear constraints on the coefficients of the target invariant. The targets are mainly high-level languages whose syntax of a loop statement is clearly defined. However, binary code has no syntactical structure, so applying invariant generation techniques for the binary code is a challenging task. Moreover, since there are no existing loop invariant generation methods on the bit-vector theory, on which the semantics of ARM instructions are represented, some techniques such as constraint solving based on Farkas' Lemma need detailed modification from linear arithmetic to the bit-vector theory. Another challenging problem is invariant generation methods such as Farkas' Lemma rely on Hoare Logic and are limited to simple invariants (e.g., linear invariant), while binaries have to access the memory and tend to contain internal function calls and system calls (Figure 8.1).

# Appendix A

# Execution trace of Mirai

We will analyzed the execution trace[1] of Mirai which is produced by CORANA/API in detail. The leaked source code of Mirai [29] is used accordingly to explain the execution trace result.

### Masking

When the botnet starts to run on the device, it performs some setup actions (e.g., avoiding debugger, preventing the device from rebooting, killing other malware instances and hiding the malware process). At the beginning of the execution, Mirai hides the CNC address via the signal() function. The signal() function is used to registers the *anti_gdb_entry()* as a handler for SIGTRAP. The *anti_gdb_entry()* when be invoked will return a real CNC address, if not the CNC address is set to a fake address. Before connecting to the server, a SIGTRAP signal be raised by the bot. If Mirai is analyzed in the debugging environment, the signal will be handled by the debugger and *anti_gdb_entry()* handle will not be invoked. On the other hand, if Mirai is not running on a debugger, *anti_gdb_debug()* is invoked and a real CNC address is obtained.

```
// Signal based control flow
sigemptyset(&sigs);
sigaddset(&sigs, SIGINT);
sigprocmask(SIG_BLOCK, &sigs, NULL);
signal(SIGCHLD, SIG_IGN);
// return real CNC address if SIGTRAP is raised
signal(SIGTRAP, &anti_gdb_entry);
...
// if not, assign server address as FAKE_CNC_ADDR
srv_addr.sin_addr.s_addr = FAKE_CNC_ADDR;
srv_addr.sin_port = htons(FAKE_CNC_PORT);
```

The execution trace from instruction `#0x11e14` enter the *ensure_single_instance()* function to maintain that each time only one instance is running.

[1] https://gist.github.com/vananhnt/7440272e27106f835261f92db3d3452e

```
static void ensure_single_instance(void) {
    static BOOL local_bind = TRUE;
    struct sockaddr_in addr;
    int opt = 1;

    if ((fd_ctrl = socket(AF_INET, SOCK_STREAM, 0)) == -1) return;
    setsockopt(fd_ctrl, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof (int));
    ...
    if (bind(fd_ctrl, &addr, sizeof (struct sockaddr_in)) == -1) {
        printf("Another instance is already running: Sending kill request...");

        sleep(5);
        close(fd_ctrl);
        killer_kill_by_port(htons(SINGLE_INSTANCE_PORT));
        ensure_single_instance();
    } else {
        listen(fd_ctrl, 1);
    }
    ...
```

At #0x11ac8 the function *socket()* is called to create an unbound socket in AF_INET domain with TCP connection. If the socket is created successfully (i.e. the return value is not -1) then it sets up the socket option and tries to bind to the control port (*SINGLE_INSTANCE_PORT*). The binding fails indicate that there is another Mirai instance running on the device. If the binding is successful, then listen() is called to mark the socket as accepting connections. In the experiment, since there are no instances of Mirai running on the device, the conditions at #0x11ad8 and #0x11b74 are not taken.

```
-> Executing 11e14 : bl #0x11aac
        -> Direct Jump to 11aac
         ==+ Call to: ensure_single_instance
...
-> Executing 11ac8 : bl #0x18d24
         === Call to library function: socket
-> Executing 11acc : ldr r6,[pc,#0x12c]
-> Executing 11ad0 : cmn r0,#1
-> Executing 11ad4 : str r0,[r6]
-> Executing 11ad8 : b #0x11b88
        -> Direct Jump to 11b88 if EQ
         //  if (socket(AF_INET, SOCK_STREAM, 0) == -1)
        -> Checking path constrains by Z3 (= #x0000003c #x00000000) ... UNSAT
        -> Checking path constrains by Z3 (not (= #x0000003c #x00000000)) ... SAT
        -> Start Jumping from 11ad8 --> 11adc
...
-> Executing 11b6c : bl #0x18808
```

```
         === Call to library function: bind
-> Executing 11b70 : cmn r0,#1
-> Executing 11b74 : b #0x11b94
        -> Direct Jump to 11b94 if EQ //  if (bind(fd_ctrl, &addr, ...) == -1)
        -> Checking path constrains by Z3 (= #x00000001 #x00000000) ... UNSAT
        -> Checking path constrains by Z3 (not (= #x00000001 #x00000000)) ... SAT
...
-> Executing 11b84 : bl #0x18990
         === Call to library function: listen
...
```

The malware also deleting itself and alternating its name to random value, then it prevents the watchdog from rebooting the devices, which would delete the malware from the device memory. After finishing set up the above actions in the foreground, it calls fork() function and immediately return to the main thread. The attack is carried out in the child process. The main function invokes *attack_init()* to initialize the data to perform attacks, *killer_init()* to start killer process and *scanner_init()* to start background scanner process.

```c
int main() {
    // Hide argv0 and assign a random value
    name_buf_len = ((rand_next() % 4) + 3) * 4;
    rand_alphastr(name_buf, name_buf_len);
    name_buf[name_buf_len] = 0;
    util_strcpy(args[0], name_buf);
    // Hide process name and assign a random value
    name_buf_len = ((rand_next() % 6) + 3) * 4;
    rand_alphastr(name_buf, name_buf_len);
    name_buf[name_buf_len] = 0;
    prctl(PR_SET_NAME, name_buf);
    ...
     //terminating the foreground process
    if (fork() > 0) return 0;
        pgid = setsid();
        close(STDIN); close(STDOUT); close(STDERR);

    // initialize data structure for attacks
        attack_init();
    // initialize process for killing other malware instances
        killer_init();
    #ifdef MIRAI_TELNET
    // initialize process for scanning new vulnerable devices
        scanner_init();
    #endif
    ...
}
```

### Attack_init()

The execution trace from CORANA/API also successfully captures the action of the child processes after the main process is terminated. After the *fork()* call, the *attack_init()* function is invoked on child process. At first the function sets up the pairs of ATTACK_VECTOR and ATTACK_FUNC, where ATTACK_VECTOR is an identifier of the DDoS attack type and ATTACK_FUNC is the pointer to the implemented attack function. Every time the bot received the attack identifier, its corresponding attack function is launched. The types of DDoS attacks that Mirai bot implemented by default are listed in */bot/attack.h* in the Mirai source code (Appendix 1).

```
BOOL attack_init(void) {
    int i;
    add_attack(ATK_VEC_SYN, (ATTACK_FUNC)attack_tcp_syn);
    add_attack(ATK_VEC_ACK, (ATTACK_FUNC)attack_tcp_ack);
    add_attack(ATK_VEC_UDP, (ATTACK_FUNC)attack_udp_generic);
    add_attack(ATK_VEC_VSE, (ATTACK_FUNC)attack_udp_vse);
    ...
}
static void add_attack(ATTACK_VECTOR vector, ATTACK_FUNC func){
    struct attack_method *method = calloc(1, sizeof(struct attack_method));
    method->vector = vector;
    method->func = func;
    methods = realloc(methods,(methods_len+1)*sizeof(struct attack_method*));
    methods[methods_len++] = method;
}
```

```
// Start the attack initialization in the child process
-> Executing 11e70 : bl #0xa590
        -> Direct Jump to a590
        ==+ Call to: attack_init
-> Executing a590 : push {r4,r5,r6,r7,lr}
-> Executing a594 : mov r1,#8
-> Executing a598 : sub sp,sp,#4
// Load the TCP_SYN attack
-> Executing a59c : mov r0,#1
-> Executing a5a0 : bl #0x198ac
        -> Direct Jump to 198a
        === Call to library function: calloc
-> Executing a5a4 : ldr r4,[pc,#0x390]
-> Executing a5a8 : ldrb r1,[r4,#0x0]=>methods_len
-> Executing a5ac : ldr r6,[pc,#0x38c]
-> Executing a5b0 : ldr r2=>attack_method_tcpsyn,[->attack_method_tcpsyn]
-> Executing a5b4 : mov r5,r0
-> Executing a5b8 : mov r3,#0
-> Executing a5bc : add r1,r1,#1
-> Executing a5c0 : ldr r0,[r6,#0x0]=>methods
-> Executing a5c4 : str r2=>attack_method_tcpsyn,[r5,#0x0]
...
// Load the TCP_ACK attack
-> Executing a5f8 : ldrb r1,[r4,#0x0]=>methods_len
-> Executing a5fc : ldr r3=>attack_method_tcpack,[->attack_method_tcpack]
...
```

### Killer_init()

The *killer_init()* process kill competing processes to ensure only the malware is running on the system. First, the process *killer_init()* is invoked by the main process to start the background killer process. After the *fork()* call at ee30, the parent process would be continued on the main thread, while the killer process runs in the background.

```c
void killer_init(void)
{
    int killer_highest_pid = KILLER_MIN_PID,
    int last_pid_scan = time(NULL), tmp_bind_fd;
    uint32_t scan_counter = 0;
    struct sockaddr_in tmp_bind_addr;
    // Let parent continue on main thread
    killer_pid = fork();
    if (killer_pid > 0 || killer_pid == -1)
        return;
    ...
```

```
-> Executing 11ef8 : bl #0xee18
        -> Direct Jump to 60952 if null
         ==+ Call to: killer_init
        -> Start Jumping from 11ef8 --> ee18 ...
-> Executing ee30 : bl #0x1adfc
         === Call to library function: fork
         === Fork a new process. Run parent process:
-> Executing ee34 : cmn r0,#1 ...
-> Executing ee3c : beq 0xee3c
    // if (killer_pid > 0)
        -> Checking path constrains by Z3 (= #x00000011 #x00000000) ... UNSAT
        -> Checking path constrains by Z3 (not (= #x00000011 #x00000000)) ... SAT
        // Return SAT since in the parent processes, killer_pid > 0
        -> Start Jumping from ee3c --> ee40
-> Executing ee40 : cmp r0,#0
...
```

### Scanner_init

The *scanner_init()* process also create a new child process to perform the scanning. First, the scanner process set up socket connection by calling `socket` and `fcntl`.

```
void scanner_init(void)
{
    int i;
    uint16_t source_port;
    struct iphdr *iph;
    struct tcphdr *tcph;

    // Let parent continue on main thread
    scanner_pid = fork();
    if (scanner_pid > 0 || scanner_pid == -1)
        return;

    LOCAL_ADDR = util_local_addr();

    rand_init();
    fake_time = time(NULL);
    conn_table = calloc(SCANNER_MAX_CONNS, sizeof (struct scanner_connection));
      // Set up raw socket scanning and payload
    if ((rsck = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) == -1){
        exit(0);
    }
    fcntl(rsck, F_SETFL, O_NONBLOCK | fcntl(rsck, F_GETFL, 0));
    ...
```

CORANA/API also successfully tracked the execution of the scanner process.

```
-> Executing 11f00 : bl #0x12d18
        -> Direct Jump to 77080 if null
         ==+ Call to: scanner_init
        -> Start Jumping from 11f00 --> 12d18
-> Executing 12d18 : push {r4,r5,r6,r7,r8,sb,sl,fp,lr}
...
=== Run child process from 12d28
-> Executing 12d28 : cmn r0,#1
-> Executing 12ea8 : bl #0x18d24
         === Call to library function: socket
-> Executing 12eac : ldr fp,[pc,#-0x14c]
-> Executing 12ec0 : bl #0x17e24
         === Call to library function: fcntl
-> Executing 12ec4 : mov r1,#4
-> Executing 12ed0 : bl #0x17e24
         === Call to library function: fcntl
```

After setting up sockets, the scanner process set up possible default configuration and passwords to try to gain administration control of target devices.

```
// Set up IPv4 header
    ...
// Set up TCP header
    ...
// Set up passwords
add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x41\x11\x17\x13\x13", 10); // root xc3511
add_auth_entry("\x50\x4D\x4D\x56", "\x54\x4B\x58\x5A\x54", 9);      // root vizxv
add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x4B\x4C", 8);      // root admin
add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7);  // admin admin
add_auth_entry("\x50\x4D\x4D\x56", "\x1A\x1A\x1A\x1A\x1A\x1A", 6);  // root 888888
add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x4F\x4A\x46\x4B\x52", 5);  // root xmhdipc
add_auth_entry("\x50\x4D\x4D\x56", "\x46\x47\x44\x43\x57", 5);      // root default
add_auth_entry("\x50\x4D\x4D\x56", "\x48\x57\x43\x4C\x56\x4A", 5);  // root juantech
add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16\x17\x14", 5);  // root  123456
add_auth_entry("\x50\x4D\x4D\x56", "\x17\x16\x11\x10\x13", 5);      // root  54321
    ...
```

```
-> Executing 12d2c : mov r3,#0
-> Executing 12fb8 : mov r2,#0xb
-> Executing 12fbc : bl #0x12bb8
        -> Direct Jump to add_auth_entry
-> Executing 12fd0 : ldr r0,[pc,#-0x20c]        // \x50\x4D\x4D\x56
-> Executing 12fd4 : ldr r1,[pc,#-0x264]        // \x5A\x41\x11\x17\x13\x13
-> Executing 12fd8 : mov r2,#9
-> Executing 12fdc : bl #0x12bb8
        -> Direct Jump to add_auth_entry
-> Executing 12fe0 : ldr r0,[pc,#-0x21c]        // \x50\x4D\x4D\x56
-> Executing 12fe4 : ldr r1,[pc,#-0x270]        // \x54\x4B\x58\x5A\x54
-> Executing 12fe8 : mov r2,#0xa
-> Executing 12fec : bl #0x12bb8
        -> Direct Jump to add_auth_entry
...
```

When the scanner process is initialized, the scanning loop is started. Two loops continuously generate random IP and checksum values to find vulnerable IoT devices.

```
add_auth_entry("\x56\x47\x41\x4A", "\x56\x47\x41\x4A", 1);// tech     tech
printf("[scanner] Scanner process initialized. Scanning started.\n");

  // Main logic loop
  while (TRUE)
  {
          for (i = 0; i < SCANNER_RAW_PPS; i++) {
              iph->id = rand_next();
              iph->saddr = LOCAL_ADDR;
```

```
               iph->daddr = get_random_ip();
               iph->check = checksum_generic((uint16_t *)iph,
                                            sizeof (struct iphdr));
           }

    while (TRUE){
           ...
           setup_connection(conn);
           //Attempting to brute found IP %d.%d.%d.%d\n",
           // conn->fd, iph->saddr & 0xff, (iph->saddr >> 8) & 0xff,
           //(iph->saddr >> 16) & 0xff, (iph->saddr >> 24) & 0xff);
       }}
```

As shown by the trace, after executing setting up functions, the direct jump to `0x12af0` which is the function `setup_connection()` is called in a loop to continuously scan the network for vulnerable devices.

```
-> Executing 1321c : str r3,[sp,#0x7c8]
-> Executing 13220 : bl #0x12420
        -> Direct Jump to rand_next()
 // executing get_random_ip()
-> Executing 13224 : ldr r4,[pc,#-0x420]
-> Executing 13238 : bl #0x12420
        -> Direct Jump to rand_next()
        ...
-> Executing 14b64 : bl #0xe7bc
        -> Direct Jump to checksum_generic()
-> Executing 14f08 : bl #0x12af0
         ==+ Call to: setup_connection
         ...
-> Executing 14f08 : bl #0x12af0
         ==+ Call to: setup_connection
        -> Start Jumping from 14f08 --> 12af0
```

# Bibliography

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[3] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *ESEC/FSE-13*, New York, NY, USA: Association for Computing Machinery, 2005, pp. 263–272.

[4] S. Anand, C. Pasareanu, and W. Visser, "Jpf-se: A symbolic execution extension to java pathfinder," vol. 4424, Mar. 2007, pp. 134–138.

[5] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps, "Directed proof generation for machine code," Jul. 2010, pp. 288–305.

[6] F. Desclaux, in *Actes du SSTIC*, 2012.

[7] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," pp. 380–394, May 2012.

[8] A. Romano, "Methods for binary symbolic execution," PhD Dissertation, Stanford University, 2014.

[9] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm," Oct. 2015, pp. 745–756.

[10] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Computer Architecture News*, vol. 39, Jun. 2012.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," *S&P*, pp. 138–157, 2016.

[12] N. M. Hai, M. Ogawa, and Q. T. Tho, "Obfuscation code localization based on cfg generation of malware," in *FPS*, 2015.

[13] G. Xu and A. Rountev, "Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis," in *International Symposium on Software Testing and Analysis, ISSTA 2008*, ACM, 2008, pp. 225–236.

[14] X. Li and M. Ogawa, "Stacking-based context-sensitive points-to analysis for java," in *Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference, HVC 2009*, ser. LNCS, vol. 6405, Springer, 2009, pp. 133–149.

[15] *Ida.* [Online]. Available: `https://hex-rays.com/products/ida` (visited on Jul. 29, 2021).

[16] *Capstone engine.* [Online]. Available: `http://capstone-engine.org` (visited on Jul. 25, 2021).

[17] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *DIMVA*, vol. 10885, Springer, 2018, pp. 372–392.

[18] M. Nguyen, M. Ogawa, and T. Quan, "Packer identification based on metadata signature," in *The 7th Software Security, Protection, and Reverse Engineering Workshop (SSPREW-7), ACM*, 2017.

[19] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *IEEE Symposium on Security and Privacy (S&P 2008)*, 2008, pp. 143–157.

[20] N. L. H. Yen, "Automatic extraction of x86 formal semantics from its natural language description," Master Thesis, JAIST, March, 2018.

[21] A. Vu and M. Ogawa, "Formal semantics extraction from natural language specifications for arm," in *FM*, ser. LNCS, Sep. 2019, pp. 465–483.

[22] Q. T. Trac and M. Ogawa, "Formal semantics extraction from MIPS instruction manual," in *FTSCS*, Springer, 2019, pp. 133–140.

[23] L. Vinh, "Automatic stub generation from natural language description," Master Thesis, JAIST, August, 2016.

[24] T. Izumida, K. Futatsugi, and A. Mori, "A generic binary analysis method for malware," in *Advances in Information and Computer Security*, Berlin, Heidelberg, 2010, pp. 199–216.

[25] M. Mues and F. Howar, "Jdart: Dynamic symbolic execution for java bytecode," in. Apr. 2020, pp. 398–402.

[26] B. Anckaert, M. Madou, and K. De Bosschere, "A model for self-modifying code," in *Information Hiding*, Berlin, Heidelberg, 2007, pp. 232–248.

[27] S. Bardin, R. David, and J.-Y. Marion, "Backward-bounded DSE: targeting infeasibility questions on obfuscated codes," in *SP 2017*, 2017, pp. 633–651.

[28] J. Yuan and S. Ding, "A method for detecting buffer overflow vulnerabilities," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, 2011, pp. 188–192.

[29] *Leaked linux.mirai source code.* [Online]. Available: `https://github.com/jgamblin/Mirai-Source-Code` (visited on Jul. 29, 2021).

[30] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," vol. 4963, Apr. 2008, pp. 337–340.

[31] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, Berlin, Heidelberg, 2011, pp. 463–469.

[32] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *SP*, 2010, pp. 317–331.

[33] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *SANER - Volume 1*, 2016, pp. 653–656.

[34] A. Djoudi and S. Bardin, "Binsec: Binary code analysis with low-level regions," in *TACAS*, 2015.

[35] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[36] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *PLDI*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 1133–1148.

[37] *Arm architecture reference manual.* [Online]. Available: `https://www.documentation-service.arm.com` (visited on Jul. 29, 2021).

[38] V. V. Anh, "Formal semantics extraction from natural language specifications for arm," Master's Thesis, School of Information Science, JAIST, December 2018.

[39] *Arm developers.* [Online]. Available: `https://developer.arm.com` (visited on Jun. 7, 2021).

[40] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, no. null, pp. 993–1022, Mar. 2003, ISSN: 1532-4435.

[41] *Jna api documentation.* [Online]. Available: `https://java-native-access.github.io/jna/4.2.1/overview-summary.html` (visited on Jul. 29, 2021).

[42] D. Scott, "Domains for denotational semantics," in *9th Colloquium on Automata, Languages and Programming (ICALP)*, ser. LNCS, vol. 140, Springer, 1982, pp. 577–613.

[43] A. Baltag, L. Moss, and S. Solecki, "The logic of public announcements and common knowledge and private suspicions," in *7th Conference on Theoretical Aspects of Rationality and Knowledge (TARK-98)*, Morgan Kaufmann, 1998, pp. 43–56.

[44] J. van Benthem and F. Velázquez-Quesada, "The dynamics of awareness," *Synthese*, vol. 177, no. Supplement-1, pp. 5–27, 2010.

[45] G. Belardinelli and R. Rendsvig, "Awareness logic: A kripke-based rendition of the heifetz-meier-schipper model," in *Dynamic Logic. New Trends and Applications - Third International Workshop, (DaLı 2020)*, ser. LNCS, vol. 12569, Springer, 2020, pp. 33–50.

[46] H. Attiya, G. Ramalingam, and N. Rinetzky, "Sequential verification of serializability," *SIGPLAN Not.*, vol. 45, no. 1, pp. 31–42, Jan. 2010.

[47] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *S&P*, 2018, pp. 161–175.

[48] *Linux manual page*. [Online]. Available: `https://man7.org/linux/man-pages/` (visited on Jul. 29, 2021).

[49] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of network-based defense mechanisms countering the dos and ddos problems," *ACM Comput. Surv.*, vol. 39, no. 1, 3–es, Apr. 2007.

[50] K. Angrishi, "Turning internet of things(iot) into internet of vulnerabilities (iov) : Iot botnets," *ArXiv*, vol. abs/1702.03681, 2017.

[51] M. De Donno, N. Dragoni, A. Giaretta, and A. Spognardi, "Ddos-capable iot malwares: Comparative analysis and mirai investigation," *Security and Communication Networks*, vol. 2018, pp. 1–30, Feb. 2018.

[52] M. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *CAV*, 2003.

[53] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 413–427.

[54] Y. Jung, W. Lee, B.-Y. Wang, and K. Yi, "Predicate generation for learning-based quantifier-free loop invariant inference," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2011, pp. 205–219.