Applied Binary Analysis

Angewandte Binäranalyse

ABA Script

by

Xhoan Qazimi

submitted to
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

Bonn, March 31, 2023

Contents

1 Basic Static Analysis

Static analysis is the process of analysing software artefacts without execution. The purpose of static analysis is to find potential security violations, runtime errors and logical inconsistencies in an artefact. The given pieces of Software are examined with external tools to determine their structure and function. Static analysis discovers bugs rather than failures. To better comprehend how static analysis works, it is necessary to understand how the given artefacts or software are structured.

1.1 Files

A file is a collection of data stored in one unit, identified by a filename. With this broad definition of a rather abstract concept, it is necessary to distinguish files from one another and categorise them into file types. From a security perspective, it is also crucial to check their integrity. Crucial pieces of information, such as the file type, are usually embedded in the binary representation of any file.

1.1.1 Magic Numbers

A magic number, a.k.a. a file signature, is a number embedded at the beginning of a file to indicate its format. While files also have name extensions, different extensions can describe the same file type. The name extension only hints at the file format, it does not necessarily define it. The name extension can be changed without affecting the file itself. Table 1 shows some common file types examples with their name extension and magic numbers.

1.1.2 Hashes

Hashes are one-way functions that take inputs of variable (arbitrary) length and convert them into fixed-length strings of Bytes according to preset rules. A file is a string of Bytes from its

Table 1: *Common file types, their extensions and magic numbers.*

Filetype	Typical Extension	Magic Bytes	Ascii
Bitmap	.bmp	42 4d	BM
GIF	.gif	47 49 46 38	GIF8
JPEG	.jpg	ff d8 ff 30	....
ELF	**[none]/.so**	7f 45 4c 46	.ELF
PE	**.exe/.dll**	4d 5a	MZ

magic number to its EOF. In general, when talking about hashes, we refer to the input as input data, whether it is a file or arbitrary bytes.

This script differentiates between three categories of hash functions according to the features of their output.

1. **Data-Digest**: This group of hash functions map their input to their output without considering constraints other than the fixed length of the output.

   Usecases: Given that these functions do not have many constraints, they are fast. They are suitable to be used in Databases, especially in search queries, when confidentiality is not a concern.

2. **Cryptographic Hashing**: This group of hash functions has to ensure that the chances of collision in output are negligible. In addition to that, they must be extremely difficult to break. Last but not least, small differences in the input must lead to entirely different outputs.

   Usecases: Being hard to break makes this group of hash functions suitable for storing and exchanging sensible data such as passwords.

3. **Similarity Hashing**: As opposed to cryptographic hashing, this group of hash functions allows similar inputs to lead to similar outputs.

   Usecases: The given attributes allow for quick data comparison. These functions can be used for pattern recognition. For example, some mutated malware mutates parts of its code more heavily than others. Following these patterns can lead to efficient malware detection.

1.1.3 Tools

Analysing files requires tools. The goal of the tools presented in this section is to give the user a first impression of the file, i.e. the program, being analysed is all about.

- Binwalk is a handy tool used to identify, extract and analyse filesystem images, compressed data, executables and file formats. Binwalk offers thorough signature matching. Another feature of Binwalk is the entropy analysis, which enables the analyzer to find interesting segments in the file.
- Strings is a simple old-fashioned tool that extracts strings from a given file. This can be useful to read debugging or feedback messages. Hardcoded phrases, i.e. Passwords, can also be extracted.
- Hexdump or alternatives, like hexyl, display files or data in Byte form.
- Objdump is a tool that helps collect information about a given binary. It can extract and read the headers, symbols and strings in an ELF file. It can also show the assembly code of the program.

1.2 ELF Files

ELF abbreviates Executable and Linkable Format. It was first published in the specification for the application binary interface (ABI - System V Release 4) of the Unix operating system. ELF is flexible, extensible and offers cross-platform support. By design, ELF supports different architectures and instruction sets.

1.2.1 Composition

An ELF file always starts with the ELF-Header, which stores crucial information about the binary such as the supported architecture, the byte order, target ABI, etc. The ELF-Header instructs the System on how to read the binary.

Regarding how they are loaded, access permissions and other criteria, different parts of the program must be dealt with differently. In the ELF the program is divided into segments. The Program Header Table describes the properties of each segment.

Table 2: *The ELF Header : `ELF64_ident` struct*

Offset		Size		Name	Purpose
32-bit	64-bit	32-bit	64-bit		
0x00	0x00	4	4	signature	Identify the file format
0x04	0x04	1	1	file_class	Signifies 32- or 64-bit
0x05	0x05	1	1	encoding	Identifies the byte-order
0x06	0x06	1	1	version	The version of the ELF header
0x07	0x07	1	1	os	Identify the OS ABI
0x08	0x08	1	1	abi_version	The version of the ABi
0x09	0x09	7	7	pad	Padding at the end

ELF also provides a division of its content by categorising it into sections. The section table tags areas in the ELF file with semantic information facilitating the search for symbols, debug information and other meta-data.

1.2.2 The ELF Header

(will be changed to itemize for better readability) The ELF header starts with an `ELF64_ident` struct that describes the target machine that can execute the binary. The struct contains a four-byte long magic number `7F 45 4C 46 - a.k.a. 0x7fELF` followed. The next two bytes define whether the binary should run on 32 or 64bit and the byte order. The next byte defines the version of the ELF Header, from which there is only one, the original. The following two bytes identify the OS ABI and its version. In case the identified OS ABI was `0x03`, which means Linux ABI, the version field identifies the version of the dynamic linker. Seven bytes of padding conclude the `ELF64_ident` struct.

Following the ident struct, an enum variable of two bytes identifies the type of object for the OS to map and use correctly. Next to type, another enum variable determines the instruction set that the object uses. The header continues with the ELF version as an unsigned integer. After the version, the ELF header contains a pointer to the starting point of the program. The following 16 bytes contain the offset of the program and section header table, each as long int of 8 bytes. The flags integer, that follows the section headers offset, is used by different architectures in different ways. The rest of the header contains six short integers that identify the size of the ELF header, the size and number of program and section headers and the index of the section header table entry that contains the section names.

Table 3: *The ELF Header*

Offset		Size		Name	Purpose
32-bit	64-bit	32-bit	64-bit		
0x10	0x10	2	2	type	Identify Object Type
0x12	0x12	2	2	machine	Identify the instruction set
0x14	0x14	4	4	version	Version of the ELF
0x18	0x18	4	8	entry	Pointer to start of program
0x1C	0x20	4	8	program_header_offset	The offset where the p-headers start
0x20	0x28	4	8	section_header_offset	The offset where the s-headers start
0x24	0x30	4	4	flags	Flags for different architectures
0x28	0x34	2	2	header_size	Size of this header
0x2A	0x36	2	2	p_header_size	Size of the program headers
0x2C	0x38	2	2	p_header_count	Count of the program headers
0x2E	0x3A	2	2	s_header_size	Size of the section headers
0x30	0x3C	2	2	s_header_count	Count of the section headers
0x32	0x3E	2	2	string_table	Offset of section names

1.2.3 The Program Header

When executing an ELF object the OS must correctly map the content of the ELF file into a process image. The program headers tell the system how to map and use different parts of the object. The program header contains the following fields:

- **Type** - It defines the type of the segment. A segment can be loadable, contains information on dynamic linking or interpretation, be reserved, etc. This information tells the OS how to manage the defined segment.
- **Flags** - This field identifies the privileges of the segment. A segment can be readable, writable, executable, none of those or a combination of the three.
- **Offset** - This field tells the OS the location of the described segment in the ELF object.
- **Virtual Address** - As the name gives it away, this variable identifies the offset in the virtual memory where the segment is mapped.
- **Physical Address** - It exists for Systems where the physical address is relevant and it is reserved for the physical address of the segment.
- **File Size** - Defines the size in bytes of the segment in the file. It can be zero.
- **Memory Size** - Defines the size in bytes of the segment in memory. It may be zero and it may be different from the size in file.
- **Align** - An integer power of two that specifies the alignment under the constraint that the virtual address of the segment equates to its offset modulo this variable.

1.2.4 The Section Header

ELF organizes the content of its object in sections according to its purpose. The section header has the following fields defined:

- **name** - The offset of the name of this section in the .shstrtab section
- **type** - Identifies the type of the described section. Depending on its content a section can be classified as program data, symbol table, dynamic linking information, etc.
- **flags** - It is used to describe attributes of the section.
- **address** - Defines where in the virtual memory the section is loaded if loaded.
- **Offset** - Defines where in the ELF file the setion is located.
- **Size** - This is the size in bytes of the section in the ELF file.

Table 4: *The Program Header*

Offset		Size		Name	Purpose
32-bit	64-bit	32-bit	64-bit		
0x00	0x00	4	4	type	The type of the segment
-	0x04	-	4	flags64	Only in 64-Bit | Segment flags
0x04	0x08	4	8	offset	Segment offset in the file
0x08	0x10	4	8	virtual_address	Virtual address of the segment
0x0C	0x18	4	8	physical_address	Physical address of the segment
0x10	0x20	4	8	file_size	Size of Segment in file
0x14	0x28	4	8	memory_size	Size of segment in memory
0x18	-	4	-	flags32	Only in 32-Bit | Segment flags
0x1C	0x30	4	8	align	Alignment of the file

- **Link** - This field is used for different purposes depending on the section type. It can be used to identify associated sections by index.
- **Info** - This field is used for different purposes depending on the section type. It is usually used for extra information.
- **Align** - This field defines the required alignment of the section.
- **Entry Size** - If the entries in the described section are fixed-sized, then this field contains the size, else it contains zero.

1.2.5 Sections to Segments Mapping

In a nutshell, a segment defines the privileges of the content, and a section determines the type of content. ELF maps sections into segments. A section containing the code of the binary must be mapped into an executable segment. On the other hand, user data should not be executable. Accessing parts of the object without having the necessary privileges, for example, trying to execute in a read-only segment, leads to a failure, in this case, segmentation fault.

1.3 The Making of an Executable

During the building process, the linker uses the information provided by the section header to correctly link a working executable. While every object can be linked differently, most of the

ELF executables follow similar patterns. Some sections are mandatory for an executable to function while others are optional. The linker can also add custom sections. This section of the script concentrates on the common sections.

1.3.1 Symbol Tables

Two sections .symtab and .dynsym, with the second being a subset of the first, contain symbol information. The section .dynsym contains the global symbols that are necessary during runtime for dynamic linking and cannot be stripped. The rest of .symtab contains symbol information that is useful for debugging but not needed for execution. Hence, it is often stripped in release builds. These two sections contain references to .dynstr for .dynsym and .strtab for .symtab.

1.3.2 The Beginning and The End

A program needs constructors and destructors to run properly. The construction and destruction of a process can be seen as arrays of functions being called one after the other. In an ELF file these arrays are contained in two sections .init_array for the constructor and .fini_array for the destructor.

1.3.3 The Data

Several sections hold data. One of them is .bss. This section is responsible for uninitialized data. Data in .bss works like a booking. Statically allocated variables that are declared but not assigned a value are stored in .bss.

Data that is already initialized is stored in .data or if the data is read-only, then it is stored in .rodata. These two sections are very similar to each other, but they are located in different segments. Static and global initialized variables are stored in these sections.

1.3.4 Dynamic Linking

The code of the object is located inside the .text section. This section is executable and not writable. The OS maps the code into the process image and delivers the external addresses, i.e. addresses of functions from libc. However, loading all reachable code and addresses inside the process image would be inefficient. The ELF format uses the Procedure Linkage Table (.plt) and the Global Offset Table(.got) to avoid loading everything at the beginning of the process. This procedure is called lazy-loading.

The .got section contains the addresses of external functions, i.e. functions from shared libraries. To speed up the start of the process .got is not directly filled with every possibly reachable function address instead, some of the entries point to .plt. The .plt is responsible for resolving the missing addresses during runtime.

When an external function is called, its address is looked up in GOT. If the address in GOT is valid, then the execution jumps to that address. Otherwise, PLT has to resolve that address. Once the PLT resolves the address once, it stores it in GOT.

1.3.5 Static Linking

A static linker copies all the required libraries into the binary file. The linker performs this task as the final step of the compilation process. The external references, such as those from libc are resolved during compilation and copied into the binary. The final product contains the calling program and every program called. This leads to big binary files. Changes in called programs require the binary to recompile and re-link.

1.4 The PE Format

Like the ELF, the PE format encapsulates all the information needed for an executable to run correctly. PE stands for Portable Executable and is the standard format used by Microsoft Windows since Windows 3.1 NT (around 1993). The data encapsulated in a PE file falls under two categories: Headers and Sections.

1.4.1 The Headers

The headers give the OS the necessary information on how to map and execute the file. They define the format, the supported architectures, instruction sets, etc.

DOS Header

A PE file starts with a DOS-Header, followed by the DOS stub. The purpose of this header is to make the file a valid DOS executable. The original purpose of the stub was to let potential

DOS users know, that the file couldn't run in DOS. This header contains the offset of the PE header.

PE Header

The PE header contains the PE signature (`"PE\0\0"`), the file header and the optional header. The file header is needed for the execution while the optional header, as the name gives it away, is optional. The file header contains information that is found in the ELF Header as well, like machine type, number of sections, a pointer to the symbol table, etc, and it also contains the size of the optional header, which can be set to zero.

The Optional Header

The optional header has a lot of entries. The most important are: AddressOfEntryPoint, SectionAlignment, and SizeSectionHeader. Noteworthy is also the fact that the PE optional header contains a checksum and information about the size of code, stack and heap.

1.4.2 Sections

The PE format contains nine predefined sections: .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata and .debug. Note that the PE format does not define a construct such as segments in ELF. The information that ELF saves in the Program Headers is given by the section headers.

Section Headers

The following list contains the variables in a section header and their purpose. Definitions from `Win32/API/winnt.h`

- **`Name[IMAGE_SIZEOF_SHORT_NAME]`** : An 8-byte, null-padded UTF-8 string. There is no terminating null character if the string is exactly eight characters long.
- **Misc.PhysicalAddress** : In the Misc Union, gives the file address.
- **Misc.VirtualSize** : In the Misc Union, defines the total size of the section when loaded into memory, in bytes.
- **VirtualAddress** : The address of the first byte of the section when loaded into memory.
- **SizeOfRawData** : The size of the initialized data on disk, in bytes.
- **PointerToRawData** : A file pointer to the first page within the object.

- **PointerToRelocations** : A file pointer to the beginning of the relocation entries for the section.
- **PointerToLineNUmbers** : A file pointer to the beginning of the line-number entries for the section.
- **NUmberOfRelocations** : The number of relocation entries for the section.
- **NumberOfLineNumbers** : The number of line-number entries for the section.
- **Characteristics** : The characteristics of the image. The following values are defined.

Commonly used Sections

The PE format contains nine predefined sections. The text, data, rodata and bss sections are the same as in the ELF. The other five sections are listed below.

- **.rsrc** : Resource Section. Similarly to XML technologies, this section contains the hierarchy of the resources used by the Object and also some configurations.
- **.edata** : Export Section. This section contains the Export Directory for an application or DLL.
- **.idata** : Import Section. This section contains various information about imported functions including the Import Directory and Import Address Table.
- **.pdata** : Pointer Data Section. The .pdata section is an array of RUNTIME_FUNCTION.
- **.debug** : Debug Section. This sections holds Debug information.

2 Basic Dynamic Analysis

Dynamic analysis is the process of testing, analysing and evaluating a piece of software during runtime. This type of analysis does not assume access to the object file or its source code. The dynamic analysis treats the binary as a BlackBox and observes its behaviour during runtime. Monitoring activities, interactions and effects on a given environment, we can make assumptions on the binary and later verify them.

Basic dynamic analysis concentrates on the environment and the effects of the binary on it, rather than the intern state of the process. The intern state of the process is covered in the Advanced dynamic analysis chapter(5).

This chapter starts with an overview of how processes work to build a solid understanding of the matter. After that, the focus shifts to specific aspects of the analysis such as Library and System Calls, Memory Activity, activity on storage devices, user input and network activity. This chapter gives an introduction to virtual machines as well.

2.1 Limitations

Dynamic Analysis can be very productive and efficient. However, similar to static analysis, there are some limitations.

When analysis techniques develop, so do techniques against analysis. It is common for malware to detect analysis environments and run in a different mode, thus hiding its true nature.

Coverage is another major weakness of dynamic analysis. It is practically impossible to simulate all execution paths. It is difficult to anticipate every possible condition. Furthermore, dynamic analysis only analyses effects without giving reasons for their occurrence. Last but not least executing an unknown binary can be dangerous if not appropriately secured.

2.2 Processes

To perform dynamic analysis efficiently, an analyst should know how a process, a running piece of software, works. Observing a process during runtime requires a working knowledge of how activities, interactions and effects present themselves.

2.2.1 Process Management

"In the beginning, there was the (Word) program." Then the program became a process after being mapped and linked as discussed in the previous chapter. There are five different states a process can be in during runtime: New, Ready, Running, Waiting and Terminated.

- **New**: The State in which a process is created.
- **Ready**: Process is fully loaded and ready to execute.
- **Running**: The process is being executed by a CPU.
- **Waiting**: The process was held by an interrupt or scheduling and is waiting to continue.
- **Terminated**: The execution finished and the process is ready to be destroyed.

The operating system keeps track of existing processes through the process table. This table consists of two columns: Process ID and a PCB(Process Control Block) pointer. The PCB contains information about the process. It saves the PID(Process ID) and the priority of the process. It also keeps track of the Program counter, register values, opened files and run time.

OS as Coordinator

The OS keeps track of the hierarchy of the processes. The kernel starts the core functionality. One process can start another. In this context, we say that a parent's process spawns a child. The OS keeps track of threads too. Threads are different execution paths of the same process. They share the same code and memory but each has its own register set, program counter, and stack space. Considering all processes and threads, the OS schedules the execution according to the set scheduling strategy.

Virtual Memory

Each process gets its Virtual Memory. Virtual Memory is an idealised abstraction of the storage resources that are available on a given machine. Its goal is to simulate the physical main memory for each process. The OS uses a paging system to achieve this goal. Both the virtual and physical memory are divided into pages of typically 4kB(or 2mB for BigPages). The content inside a single page is identical in the physical and virtual memory but not all of the virtual pages are mapped into the physical memory. The OS(partially the firmware too) decides which pages reside in the physical space and which are moved to the disk. To keep track of this mapping the OS uses Page Tables. The translation between physical and virtual memory is done by the MMU (Memory Management Unit - If not present in Hardware then emulated). The paging system also helps manage permissions. The OS can flag pages as writable, readable, executable or a combination of the three.

A process accessing a page with no proper entry in the page table or without permission leads to a page fault error. The OS handles the error by either loading the page into the memory or throwing a Segmentation Fault error. This usually happens when an invalid page is accessed.

2.3 Performing Analysis

Due to security concerns, a normal process has restricted privileges. A process cannot access hardware or the main memory directly. These operations are done by using the System API and shared libraries. Most common operating systems enforce the Protection Ring Model. Userland processes run on ring 3(the least privileged level), and the kernel runs on ring 0(the most privileged level in the OS context).

2.3.1 Library Calls and System Calls

The OS provides the Application System Calls and Library Calls as part of its functionality. Library Calls are Userland functions that use code from the shared libraries. If needed, these functions are mapped in the virtual space of the process. System Calls are a way of communication between Ring 3 and Ring 0. The application uses System Calls to ask the OS to perform a privileged task on behalf of the application. Both System and Library Calls are traceable and helpful for dynamic analysis.

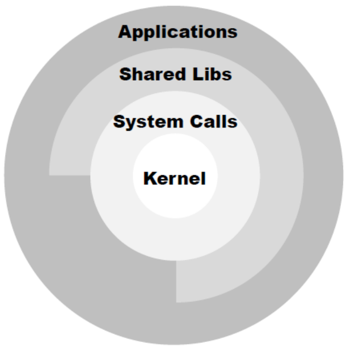


Figure 1: *System API & Shared Libraries*

System Calls offer functionalities such as reading, writing, connecting to a socket, etc.Library Calls hold similar information. Knowing which System and Library Calls the application uses can help understand its functionality. The program `ltrace` helps trace Library Calls on a Linux-based OS and `strace` does the same for System Calls. Process Explorer is a Windows tool that offers monitoring functionality. It is capable of examining the registry, the file system, network activity, processes, and threat activity and can thus be used in Windows as an equivalent of strace and ltrace.

2.3.2 Main Memory Activity

When a process is being created it is assigned a Process ID (PID) and the OS fills out the PCB. The OS keeps vast information about the process, which is very helpful for analysis.

Windows

Microsoft's Process Explorer offers a GUI Tool to help analyze what files, connections, registry keys or DLLs a process accesses or manipulates. It makes it easy to find information about running processes.

Linux

Unix systems use pseudo-files to manage the processes. These files can be found in the `proc` folder. Each process has its subfolder at `/proc/[PID]`. The OS creates many pseudo-files for each process. Some interesting files are:

1. `/proc/[PID]/status` -> Gives an overview of the function.
2. `/proc/[PID]/cmdline` -> shows the command line for the process. If you check content of this file, you will see the command, but arguments are not separated by space.

   {`cat /proc/PID/cmdline  | xargs -0 echo`} for space-separated arguments
3. `/proc/[PID]/environ` -> shows all the environment variables that were passed when calling the process.

   {`cat /proc/3940/environ | tr '\0' '\n'`}
4. `/proc/[PID]/fd` -> This is a subdirectory containing one entry for each file which the process has open, named by its file descriptor.
5. `/proc/[PID]/maps` -> A file containing the currently mapped memory regions and their access permissions.

2.3.3 Activity on Storage Devices

A process with access to storage devices can manipulate(create, edit, delete) accessible files. Processes often create files in their subdirectories or the temporary folder of the system. But they can also change system configurations.

System Configurations on Windows

Windows stores configurations for one or more users, for applications and hardware in the Windows Registry. This structure is primarily changed during the installation, configuration and removal of apps. It can also be edited through a registry editor (RegEdit). Malware often uses the registry for persistence. The registry is located at `%SytemRoot%\system32\config`.

The Windows registry is organized in Hives. A hive is the first level of registry key in Windows. It contains other keys and subkeys. The following list gives a brief description of the standard hives in Windows 10 Enterprise and their purpose.

- `HKEY_CURRENT_USER HKU\<USERID>` : Configurations for the specified user.

  Path: `%systemdrive%\Users\%username%\NTUSER.DAT`
- `HKLM\BCD00000000` : Boot Configurations Database.

  Path: `\Device\HarddiskVolume1\Boot\BCD`

- `HKLM\COMPONENTS` : State of Windows features and updates.

  Path: `%systemroot%\System32\config\COMPONENTS`
- `HKLM\HARDWARE` : Information on connected hardware. Only stored in memory (at each boot) and not stored within a file.

  Path: Only in Memory
- `HKLM\SAM` : User information such as user names and passwords.

  Path: `%systemroot%\System32\config\SAM`
- `HKLM\SECURITY` : Security policies and user permissions

  Path: `%systemroot%\System32\config\SECURITY`
- `HKLM\SOFTWARE` : System-wide Windows configuration that is not used for the boot process, e.g. application configuration.

  Path: `%systemroot%\System32\config\SOFTWARE`
- `HKLM\SYSTEM` : Information on Windows configuration required during boot process.

  Path: `%systemroot%\System32\config\SYSTEM`
- `HKU\.DEFAULT HKU\S-1-5-18` : Hive of the account Local System.

  Path: `%systemroot%\System32\config\.DEFAULT`
- `HKU\S-1-5-19` : Hive of the account Local Service.

  Path: `%systemroot%\ServiceProfiles\LocalService\Ntuser.dat`
- `HKU\S-1-5-20` : Hive of the account Network Service

  Path: `%systemroot%\ServiceProfiles\NetworkService\Ntuser.dat`

System Configurations on Linux

Linux stores its configurations a bit differently. Under the root folder of a Linux System, there are the subfolders `/etc`, `/var`, `/home`.

- `/etc` : Short for "et cetera", the etc folder is a system configuration directory that contains files used to configure the system. These files include the network configuration, the user and group configuration, and the system services configuration.

- `/var` : Short for "Variable", the var folder contains variable data files. This can include spool directories(contains data which is awaiting some kind of later processing, often through peripheral devices) and files, administrative and logging data, and transient and temporary files.
- `/home/[Username]` : Contains user-specific data and configurations, such as terminal properties.

2.3.4 User Input

Expected command line input and output are among the obvious pieces of information to observe. Binaries, with the exception of malware, usually give hints of their purposes. For example, if strace was to be analysed looking at the input and output of the command line would be a fast giveaway of the purpose of the binary. The assumption, based on the output would be that strace is used to list system calls performed by certain programs.

2.3.5 Network Activity

Network activity is crucial for dynamic analysis. It can provide information on the specifics of how the binary is communicating and what information is being transmitted. To be able to perform network analysis it is necessary to understand how applications transmit data through the network. Most applications use TCP or UDP as their transport protocol. The application data is encapsulated by the transport protocol and given to the Network layer protocol, which would be IP. IP encapsulates the data and adds its own header to it then passes it to the Data link layer where a MAC header and footer are added.

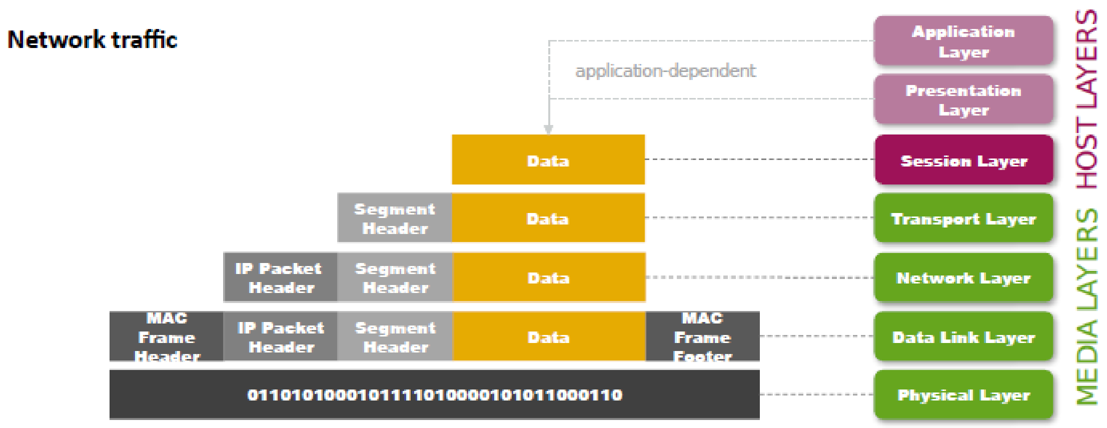


Figure 2: *Network Packages*

Analysing network data requires capturing or sniffing network packets. Depending on which Layer the capturing(or sniffing) takes place, the captured packets contain varying amounts of information. The following list gives different positions for network sniffing and the data that can be extracted from there.

Note: The lower levels can extract data from the higher ones. I.e. 3. can read information from 1.

1. Transport Layer : Contains the segment header and the data. The two most prominent protocols at this layer are TCP and UDP.
2. Network Layer: At this point, the data already has the transport layer header attached and gets the IP header too. This allows us to read the source and destination address, as well as the time to live.
3. Data Link Layer: Information like source and destination hardware, frame check Sequence and frame type is added to the packet.

2.4 Virtual Machines

A virtual machine is a computer file, typically called an image, that behaves like an actual computer. It can run in a window as a separate computing environment, often to run a different operating system—or even to function as the user's entire computer experience—as is common on many people's work computers. Analysing an unknown binary by executing it can be dangerous. Executing a virus on a computer just for the sake of analysing it can be expensive and have consequences(Infection of the whole company network). We use a controlled environment to help minimise these risks. Usually, this environment is implemented as a virtual machine. Virtual machines come with many perks. They can save their current state through snapshots and return to it later. The same setup is reproducible.

2.4.1 Availability

Analysing applications for different systems requires each system to be available. Having each system on a piece of hardware can be very expensive. Virtual machines, on the other hand, require much less and can be very efficient.

Virtual machines also enable snapshots, making it easier to test a given application on different versions of the system without wasting time or other resources.

2.4.2 Analysis

A virtual machine can be configured to be a fully isolated environment. If a binary happened to be malicious and changes or damaged the system, it is easy to recover through the snapshot mechanism. The host can monitor changes on a virtual machine, making the detection of advanced malware, such as rootkits, easier. Virtual networks(several VMs connected) can simulate real-world scenarios. To stop a virus from spreading and infecting other machines, the network on a VM can be fully disabled or redirected to a dummy.

2.4.3 Sandbox

A sandbox is a special kind of virtual machine. The idea behind it is to perform quick and dirty analysis on a given binary. These binaries are run in a safe environment automatically. The sandbox then creates a report with information about the binary.

2.4.4 Limitations

Emulating full-scale systems costs a lot of resources. Most of the time the emulated system is limited in processing power, memory, etc. Malware can test the limits of a system and decides if it is being run in an emulated environment or not. Malware can avoid running in emulated environments to prevent dynamic analysis.

2.5 Tools

- **PS** -> Linux. Allows the user to list all visible processes and give information such as user, ID, parent ID, time running, command line and path.
- **pstree** -> Linux. Shows the process hierarchy as a tree diagram.
- **top/htop** -> Linux. Live version of ps combined with pstree, useful but very volatile.
- **inotifywait** -> Linux. A watcher that can be set on a folder or file to notify the calling process if changes occur.
- **Process Explorer** -> Windows. Read information about the process. List open files, connections, threads, environment and performance.
- **Process Monitor** -> Windows. Process Monitor is an advanced monitoring tool for Windows that shows real-time file system, Registry and process/thread activity.
- **RegShot** -> Windows. Allows for taking screenshots of the registry and comparing between them.
- **RegEdit** -> Windows. Used to edit the registry.
- **QEMU** -> Cross-Platform. Virtualisation/Emulation Platform. Can simulate firmware and operating systems.
- **VirtualBox** -> Cross-Platform. Virtualisation Software. Beginners friendly.
- **Wireshark** -> Capture network traffic and analyse packets sent and received through a monitoring pointing.
- **Any.Run** -> Innovative cloud-based sandbox with full interactive access
- **Cuckoo** -> Cuckoo Sandbox is an advanced, modular and open source automated malware analysis system with lots of features.

3 Assembly

Assembly language, commonly known as simply assembly and abbreviated as asm, is the direct interpretation of binary code given the target machine, instruction set and alignment. While being a direct interpretation of binary code, assembly also works as a low-level programming language. Different instruction sets and architectures require each their specific assembly language. This chapter gives an overview of Assembly Languages and their details.

3.1 Syntax

There are two prominent flavours of assembly syntax: Intel and AT&T. The syntax of the language defines how the instructions and operands are ordered, but does not change the behaviour of the machine executing the code. They help with readability. The rules are simple. The instructions are generally divided into three parts: Op-Code, destination, and source.

3.1.1 Intel

Intel writes the op-code (Operation code, a.k.a. instruction) first, the destination second and the source third. If there is no source(or the source is at the top of the stack), then there is just the op-code and the destination. If there is no destination (or the destination is at the top of the stack) then there is the op-code and the source. The registers are used as variables, by their name only, and addresses use square brackets. If the operation requires a size variable, the size comes before the operand.

3.1.2 AT&T

AT&T writes the op-code (Operation code, a.k.a. instruction) first, the source second and the destination third. If there is no source(or the source is at the top of the stack), then there is just the op-code and the destination. If there is no destination (or the destination is at the top of the stack) then there is the op-code and the source. The registers are used as variables, by their name with a $ in front, and addresses use $ and square brackets. If the operation requires a size variable, then the op-code is completed with a suffix to define it.

3.2 Architectures

Different types of machines and architectures support different instructions. This leads to different assembly languages. The different instruction sets are divided into two categories: RISC(reduced instruction set computing and CISC(complex instruction set computing). CISC architectures tend to optimise the work done by one instruction, while RISC architectures tend to optimise the time needed for each instruction. Overall the performance of machine architectures is evaluated as work done over time. Both RISC and CISC aim to optimise performance. This section goes over some of the specifics of common architectures.

- **i386** The name stands for Intel 386, previously known as 80386. Intel 386 was the first processor to use the i386 instruction set. It ran on a 32-bit system and had backwards compatibility for 16-bit mode. It uses 32-bit registers and it can address up to 4GB. This instruction set is also known as x86.
- **ARMv8** ARMv8, the successor of ARMv7, is another instruction set supported by the AArch64 machine architecture. ARMv8 adds 64-bit compatibility to its predecessor while defining a relationship to the prior 32-bit (referred to as AArch32). This makes the ARMv8 compatible with 32- and 64-bit. When in 64-bit mode, this architecture supports up to 48-bit of virtual addresses.
- **MIPS** (discontinued since 2021) MIPS has a clear RISC structure. The latest version(R3000) runs on 32-bit hardware and supports 32-bit operations exclusively. It has 32 general-purpose registers and a few special registers. Only two instructions, load and store, use memory addressing, while all the others require immediate or register operands.
- **AMD64** AMD64 is a 64-bit processor architecture that was developed by Advanced Micro Devices (AMD) to add 64-bit computing capabilities to the x86 architecture. It is sometimes referred to as x86-64, x64, and Intel 64. It uses 64-bit registers and a total of 16 general-purpose registers, 16 vector registers(these are 128-bit long) and other registers like floating point registers.

3.3 Understanding Assembly

Assembly is different from higher programming languages. To understand assembly, it is necessary to know what is "under the hood" of the machine. This section takes a closer look at assembly tools and equipment at disposal.

3.3.1 Operand Types

Operand in higher programming languages can be of different sizes and types. In assembly, operands are a string of bits with a given length and it is up to the instruction to interpret them. The operands passed to an instruction can be immediate, registers or in memory.

Immediate operands are numbers that can only be used as sources. The only restriction is the size. Some architectures require fixed-length instructions(length includes the operands), others have an upper limit to how long an immediate operand can be.

Operands can be registers. Registers are built-in data storing units. They can be source and destination. Registers have a fixed length, however, most architectures offer a way to address parts of a register. (i.e. eax is the lower half of rax)

Data stored at a memory address can be an operand as well. When using memory data as operands we distinguish between two categories: Direct addressing and indirect addressing. The first category uses the address directly and loads a given number of bytes as an operand, the second takes a base address and adds offset multiplied by size to it, similar to the way arrays are addressed.

3.3.2 Registers

Registers are built-in, fast and fixed-sized and dedicated storage devices. They are memory cells directly accessible to the CPU. Despite being fixed-size, most architectures support accessing parts of them. X64 supports addressing: lower 8 Bits(0:7), higher 8 Bits(8:15), 16 Bits(0:15), 32 Bits (0:31) and the full 64 Bits. The X64-specific registers r8-r15 cannot access the higher first Byte(7:15) directly.

Some registers have a designated function they are assigned. Some of them are(X64 architecture):

- **rax**: Arithmetics and Return Values

- **rcx**: Loop Counter
- **rdx**: Extension of rax for arithmetics
- **rsi**: Source Index
- **rdi**: Destination Index
- **rsi**: Stack Pointer
- **rdi**: Stack Frame Base Pointer
- **rip**: Instruction Pointer. **cannot be read directly**
- **rflags**: Flags holding control and optimisation values

3.4 Commonly used Instructions

Instructions are grouped into four categories depending on their purpose: Data Transfer, Binary Arithmetic, Logical Operations and Shifts and Miscellaneous.

3.4.1 Data Transfer

The operations that transfer or convert data fall under this category. The following list contains some important operations and their meaning.

- **mov**: Move operations from source to destination.

  Intel Syntax Usage: mov dest, source
- **cmov**: Conditional move. After an assessment such as cmp, execute mov if the assessment was correct.

  Intel Syntax Usage: cmov dest, source
- **xchg**: Exchange operands values

  Intel Syntax Usage: xchg operand1, operand2
- **movsx**: Mov and sign extend. If destination register is bigger than the source, extend preserving the sign and execute mov

  Intel Syntax Usage: movsx rax, eax
- **cwd**: Convert word to doubleword. Length conversion. instruction extends the sign bit of AX into the DX register

Intel Syntax Usage: cwd (the registers used are predefined to be AX and DX).

- **cdq**: Convert doubleword to quadword. Similar to cwd. It uses EAX and EDX. instruction extends the sign bit of EAX into the EDX register.

  Intel Syntax Usage: cdq (the registers used are predefined to be EAX and EDX)
- **cbw**: Convert byte to word. Instruction extends the sign bit of AL into the AH register.

  Intel Syntax Usage: cbw (the registers used are predefined to be AL and AH)

3.4.2 Binary Arithmetic

The operations that fall under this category perform arithmetic operations on the operands. The following list gives the most commonly used arithmetic operations:

- **add/sub:** Addition or Subtraction between two integer values.

  Intel Syntax Usage: add sum/summand, summand || sub minuend/difference, subtrahend
- **adc/sbb:** Integer Addition/Subtraction with carry/borrow

  Intel Syntax Usage: same as add/sub
- **adcx:** Unsigned integer Addition/Subtraction with carry(IA64?)

  Intel Syntax Usage: same as add
- **adox** Unsigned integer Addition with overflow.

  Intel Syntax Usage: same as add
- **imul/idiv:** Integer multiplication/division of two factors.

  Intel Syntax Usage: imul factor1, factor2 result is written in rdx:rax
- **mul/div:** Unsigned integer multiplication/division of two factors.

  Intel Syntax Usage: mul factor1, factor2 result is written in rdx:rax
- **inc:** Add one to the given operand

  Intel Syntax Usage: inc operand
- **dec:** Subtract one from the given operand

  Intel Syntax Usage: dec operand
- **neg:** Negate the given operand.

Intel Syntax Usage: neg operand

- **cmp:** Compares two values and sets flags depending on the result. Equal or not equal.

  Intel Syntax Usage: cmp val1, val2

3.4.3 Logic and Shifts

These are logic operations that are usually fast and widely used.

- **and:** Bitwise AND operator. If both operands have a one in a given (bit)position, that position is a one in the result and zero otherwise.

  Intel Syntax Usage: and operand1, operand2
- **or:** Bitwise OR operator. If either operand has a one in a given (bit)position, that position is a one in the result and zero otherwise.

  Intel Syntax Usage: or operand1, operand2
- **xor:** Bitwise XOR operator. If the operands have different values for a given (bit)position, that position is a one in the result and zero otherwise. ***xor operand1, operand1*** is a quicker option for ***mov operand1, 0***

  Intel Syntax Usage: xor operand1, operand2
- **sar/sal:** Arithmetic shift to the right(sar) or left(sal). Also known as signed shift. Preserves the sign bit and moves the rest in the given direction. Vacant positions(sal only) are filled with zeros. Handy for multiplications with powers of two, faster than mul/imul.

  Intel Syntax Usage: sar/sal operand, steps
- **shr/shl:** Logical shift to the right(shr) or to the left(shl). Moves every bit one position to the given direction, filling vacant bit positions with zeros.

  Intel Syntax Usage: same as sar/sal

3.4.4 Miscellaneous

These operations have diverse purposes.

- **lea**: lea(load effective address) is an efficient operation that computes the effective address in memory of a source operand and places it in a general-purpose register. As a rule of thumb, lea is the version of mov for addresses.

  Intel Syntax Usage: lea dest, [base+offset*size]

- **nop**: NOP instruction is the short form for 'No Operation' and is often useful to fine tune delays or create a handle for breakpoints. The NOP instruction is sometimes required during sensitive sequences in the hardware

  Intel Syntax Usage: nop

- **cpuid:** CPUID (CPU Identification) is a set of instructions that is used to fetch information about the Processor such as Processor Topology, Cache Size and much more. Takes up to two parameters.

  Intel Syntax Usage: mov eax, param1; mov ecx, param2; cpuid

- **rdtsc:** Time stamp counter. Returns the time stamp in 64-bit.rdtsc returns results in edx:eax.

  Intel Syntax Usage: rdtsc

- **pushad/popad:** Push/Pop all general-purpose registers on/from the stack. Only supported in x86, can be simulated in x64 with a macro.

  Intel Syntax Usage: pushad/popad

3.4.5 Ambiguity

There is more than one way to achieve the same result in assembly. This results in ambiguity when looking at compiled code. The same C source code can seem different in assembly yet achieve the same. A very good example of ambiguity in assembly is the fact that mov is Turing-complete. Another example, that is often useful when forging shellcode, is that the xor operator with the same operand as source and destination is the same as setting the operand to zero.

3.5 Stack

Do you remember Stack Overflow? Well, they stole their name from this guy's drama and that's why it is okay to steal code from there.

A stack is a kind of list, that allows adding and removing elements only from the top. These lists follow the Last-in-First-out model. The stack is usually mapped on top of the address space assigned to a process. It grows towards the lower addresses.

3.5.1 Managing the Stack

The OS implements the stack as a chunk of memory on top of the virtual address space. The OS has to manage the stack properly in order for it to be useful. The registers rsp and rbp help with the task.

The rsp register keeps track of the top of the stack. Every time data is pushed on the stack the rsp register moves to the end of it. If the program pushes an integer on top of the stack, rsp moves four bytes up in its growth direction(meaning the address of rsp gets 4 Bytes smaller). A pointer(8 Bytes in X64) would move rsp 8 Bytes in the growth direction.

The rbp helps with Stack frames. Every time the program calls a function it creates a stack frame. The beginning of the frame is saved in the rbp register.

The operations that influence stack management are:

- **push**: Push data on stack, move rsp in the growth direction according to the size of the data
- **pop**: Pop data form stack, move rsp against the growth direction according to the size of the data
- **call**: push rip; push rbp;mov rbp, rsp; move rip to the called address
- **ret**: pop rbp; pop rip; back to the calling function

3.5.2 Stackframes

A function in assembly consists of three parts: Prolog, Body and Epilog. The Prologue creates a new stackframe. It pushes rbp on the stack and writes the current value of rsp to rbp. This way the local variables and data stored on the stack by the calling function are saved.

The body is the code of the routine. If the routine needs to return a value it writes that value in rax.

The epilogue takes care of restoring the frame of the calling function. If there were local variables, the rsp is moved before those variables, effectively "forgetting" those variables. Then it pops rbp from the stack and returns to the calling function.

3.5.3 Subroutines

Assembly does not support functions as we know them from higher programming languages, but it does support a similar technique. When talking about functions in assembly we actually refer to subroutines.

Calling conventions define how the program manages subroutines. There are different standards. Basically, they all try to achieve the same result: Transition between caller and callee without data loss or corruption.

cdecl

Cdecl is the default calling convention for C and C++ programs and the de facto standard for X32 Architectures. Because the stack is cleaned up by the caller, it can do vararg functions. The cdecl calling convention creates large executables because it requires each function call to include stack cleanup code. The return value is written in eax. Everything but eax, ecx, edx is preserved.

AMD V 64 Bit Convention

The calling convention of the System V AMD64 ABI is followed on GNU/Linux. The registers RDI, RSI, RDX, RCX, R8, and R9 are used for integer and memory address arguments and XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 are used for floating-point arguments. For system calls, R10 is used instead of RCX. Additional arguments are passed from right to left on the stack and the return value is stored in RAX. The clean-up is managed by the caller. This convention is only used in User-Mode. Everything but rax, rcx, rdx is preserved.

X64 (Microsoft) ABI Convention

It uses registers RCX, RDX, R8, R9 for the first four integer or pointer arguments (in that order), and additional arguments are pushed onto the stack (right to left). Integer return values are returned in RAX if 64 bits or less. In the Microsoft x64 calling convention, it's the caller's responsibility to allocate 32 bytes of "shadow space" on the stack right before calling the function (regardless of the actual number of parameters used), and to pop the stack after the call. The shadow space is used to spill RCX, RDX, R8, and R9, but must be made available to all functions, even those with fewer than four parameters. Caller cleans up and rbx, rbp, rdi, rsi, rsp, r12, r13, r15 are preserved.

AArch64

It uses the registers x0-x7 for the first 8 parameters and pushes the rest on the stack. The same registers are used for return values and x8 is used for indirect return values. The caller cleans up and the registers x16 – x30 are preserved.

3.6 Control Flow

Linear programs are easy to understand but they are inefficient, less usable and repetitive. Conditional statements and loops make the program much more efficient and usable. Changing the control flow of a program can be very useful. This section takes a look into conditional statements and loops.

3.6.1 Flags

Most of the operations in assembly take source and destination operands or modify the stack. Operations that check for conditions usually modify the flags register. The flags register consists of single-bit values assigned meaning and purpose. For example, when the cmp operation takes place, it sets the zero bit if the given values are equal.

Some of the important flags in the `x86_64` Architecture are:

- **CF**: Carry Flag is set if the previous operation generated a carry.

  Position: 0
- **PF**: Parity Flag is set if the number of bits set during the previous operation was even.

  Position: 2
- **AF**: Auxiliary Flag is set if the previous operation generates a carry or borrow out of the four least significant bits. Important for decimals.

  Position: 4

- **ZF**: Zero Flag is set if the result of the previous arithmetic or logical operation was zero.

  Position: 6
- **SF**: Sign Flag is set if the result of the previous operation was negative.

  Position: 7
- **OF**: Overflow Flag is set if the previous operation generated an overflow.

  Position: 11

3.6.2 Conditional Operations & Loops

Conditional operations check if a condition is satisfied, and if that is the case, perform the operation. To check if the condition is met, the CPU reads one or more flags before following the instruction. Usually, the test or cmp operation is used before a conditional operation. This is done to set the flags.

Conditional operations like conditional jumps help implement loops and if-statements as we know them from higher programming languages. For example in a for statement, we compare an integer i with a given number and keep jumping back to that comparison after every iteration until the condition isn't met.

3.6.3 Position Independence

Since most platforms nowadays relocate the code and randomise the addresses, it becomes necessary to have position-independent code. This means that jumping to concrete addresses is not supported. Jumps use an offset from the current position. Compilers of modern higher languages use position-independent code by default.

4 Advanced Static Analysis

Performing static analysis on binary artefacts tends to be a demanding task. The difficulty resides in the amount of text that needs to be processed and in the way this information is represented. There are differences between human-written (or human-readable) code and machine code. When dealing with machine code, it becomes a necessity to transform it into a human-readable form. This chapter takes a look into intermediate representations, their usage and value, and decompilers.

4.1 Intermediate Representations

The process of transforming human-written code into machine code is called compilation. Transforming machine-code into human-readable code is called decompilation. Both processes, compilation and decompilation, include intermediate representations.

An intermediate representation is a form of code, used to represent a state of the source code (or the artefact for decompilers) without any information loss. These representations highlight some aspects of the code, making processing and optimisation easier.

Intermediate Languages are forms of intermediate representation. These are abstract languages designed to help the analysis of programs. The design is usually simple and suitable for low-level optimisation. They abstract away from different architectures, allowing for a single analysis of all of them. These are some of the concepts that are subjected to abstraction:

- **Register Name**: Every Architecture uses different names for registers. Intermediate languages use their structures to abstract away from these differences. They also define how these structures are addressed.
- **Memory Access**: Several architectures address the memory differently, for example, ARM can use both little-endian and big-endian modes. Intermediate representations understand these differences and abstract away from them.

- **Memory segmentation**: Most architectures support memory segmentation but they don't implement it in the same way. An IR should understand the different methods of segmentation and present the same idea in a unified way.
- **Instruction side-effects**: Many instructions have side effects, like modifying the stack or the flags. These modifications are information that must be accounted for. An IR should keep track of these modifications and represent them in a unified way.

4.1.1 VEX

VEX is an intermediate representation used in Valgrind and angr (if you don't know these two don't worry, you will). VEX is an architecture-agnostic, side-effects-free representation of several target machine languages. It abstracts machine code into a representation designed to make program analysis easier. VEX is well-suited for decompilers.

The VEX Representation consists of four main classes of object:

- **Expressions**: An expression is a representation of a calculated, loaded or constant value. This includes reading from registers, memory or results of arithmetic and logical operations.
- **Operations**: Describe a modification of an expression. It include bit-wise, arithmetic, logical operations and so forth.
- **Temporary Variables**: VEX uses temporary variables as internal registers: IR Expressions are stored in temporary variables between use. The content of a temporary variable can be retrieved using an IR Expression.
- **Statement**: They model changes in the state of the target machine, such as modifications of the stack, writing and reading memory or registers.

4.1.2 LLVM

LLVM is without a doubt one of the most prevalent intermediate representations. LLVM is primarily used by compilers for optimisation. It supports compile-time, link-time, run-time and idle-time optimisation. While originally developed for C and C++, LLVM now supports a variety of programming languages. The LLVM IR is a strongly typed reduced instruction set computing (RISC) instruction set which abstracts away most details of the target. For example, the calling convention is abstracted through call and ret instructions with explicit arguments. Also, instead of a fixed set of registers, LLVM uses an infinite set of fixed memory locations of the form %0, %1, etc. LLVM supports three equivalent forms of IR: a human-readable assembly format, an in-memory format suitable for frontends, and a dense bitcode format for serializing.

4.2 Decompilation Step by Step

The goal of decompilation is to turn binary artefacts into some form of source code. Analysing source code, or a form of code similar to it, is much easier than analysing assembly. However, the ambiguity, variation of architectures and limitations of assembly make the process rather complex. This section takes a look into the process of decompilation as done by Binary Ninja.

4.2.1 Lifted Intermediate Language

Lifted Intermediate Language(LIL) is the first step of decompilation. It is a straightforward translation of instruction from assembly (any supported assembly) to the instruction set defined for the Low Level Intermediate Language. This is the step where the architecture is abstracted away.

4.2.2 Low Level Intermediate Language

The Low Level Intermediate Language(LLIL) takes the output of LIL and optimises it. It removes the NOP instructions and turns reading flags into conditional statements. Other than that, it looks a lot like LIL.

4.2.3 Mapped Medium Level IL

In this stage of the process, the stack is replaced by variables and abstracted away. The instruction set is changed again to that of Medium Level Intermediate Language.

4.2.4 Medium Level Intermediate Language

Medium Level IL (MLIL) translates registers and memory accesses into variables, types are associated with variables, platform information is used to generate call sites with types (both inferred and explicit) and parameters, data flow is calculated and constants are propagated.

4.2.5 High Level Intermediate Language

High Level IL (HLIL) builds on MLIL by adding higher level control flow instructions, a number of dead-code and variable passes as well as other simplification/folding. HLIL is a structure that can be easily read and analysed.

4.3 Abstract Syntax Tree

An abstract syntax tree is a form of representation of formal language texts. An abstract syntax tree can represent a program or any other text written in some formal language. The AST represents the procedural logic, data definition and workflow composition in a granular level. This section concentrates on how these abstract tree help with static analysis.

4.3.1 Usecases

Since the ASTs represent the structure of a program and its workflow in a formal unified way, it becomes easier to read and analyse them automatically. ASTs are useful in static analysis for different reasons. Some of these reasons are listed below:

- **Error Detection & Warning Generation** - Following the structure of a program using ASTs, we can automatically check for error-prone code. Reading from a file with a *libc* function can return an error code that needs to be checked. If the return value of such functions isn't checked a warning is generated.
- **Code Transformation** - Transforming code to different representations or different languages requires information on the procedural logic, data definition and workflow. Given that ASTs contain this information it can be helpful to integrate them into the process.
- **Overview of the Code** - Reading the tree or parts of it is usually quicker than reading the text itself. This allows for a quick overview of the program through the abstract tree.

4.4 Tools

This section lists a set of helpful tools. Guess what: you get to keep your soul if you use them, contrary to what the presentation says.

- **Ghidra** : Ghidra is a tool developed by the NSA in Java designed for reverse engineering. Ghidra helps analyse software through disassembly, assembly, decompilation, graphing, and scripting. It is available on Mac, Windows and Linux.
- **Rizin** : Rizin is a fork of the radare2 reverse engineering framework with a focus on usability, working features and code cleanliness. Rizin is portable and it can be used to analyze binaries, disassemble code, debug programs, as a forensics tool, and as a scriptable command-line hexadecimal editor able to open disk files, etc. Rizing is well documented.
- **Cutter** : Cutter is an open-source reverse engineering suite designed for creating advanced and costume environments with usability as a high priority. It has a ton of built-in features and it is still beginner friendly.

5 Advanced Dynamic Analysis

Basic Dynamic Analysis considers activities, interactions and effects on a given environment and makes assumptions on a binary. While this method gives us a fair amount of information on the process, we can achieve more if we consider the internal state of a process, variables, register values, and instructions. In Advanced Dynamic Analysis we do that by using a debugger.

Using a debugger allows us to inspect dynamically calculated variables, such as passwords or addresses. Furthermore, we can choose our execution path to better target interesting code. In this way, we can obtain a good understanding of the binary with little effort.

5.1 Prerequisites

To perform dynamic analysis efficiently, it is necessary to understand the internals of a process. The following list is a short reminder of some basic concepts from the previous lectures.

- **Virtual Memory** is an idealised abstraction of the storage resources that are available on a given machine. Its goal is to simulate the physical main memory for each process. (2.2.1)
- **Calling Conventions** define how the program manages subroutines. There are different standards. Basically, they all try to achieve the same result: Transition between caller and callee without data loss or corruption. (3.5.3)
- The **stack** is a kind of list, that allows adding and removing elements only from the top. These lists follow the Last-in-First-out model. The stack is usually mapped on top of the address space assigned to a process. It grows towards the lower addresses. (3.5)
- A **File Format** is a standard way that information is encoded for storage in a computer file. (1.2 , 1.4)
- **Registers** are built-in, fast and fixed-sized and dedicated storage devices. They are memory cells directly accessible to the CPU. (3.3.2)

5.2 Capabilities of Debuggers

The Debugger gives insight into the execution of a program. It can read the currently executing assembly instruction. Debuggers allow for interaction with the process, like reading or writing memory, registers or even changing the execution path.

In debug releases, it can read the currently executing line of the source code. Debug releases contain debugging symbols. The debugging symbols allow the debugger to interact with the source code, like reading or manipulating variables.

5.3 Compiling and Debugging

Debuggers use the information stored in the binary to give a better representation of the current state. They use the symbol table to define the symbols of the process. This allows the debugger to address the function by its name when using "break", for example. In production, this information is stripped to reduce the size of the binary.

Debug releases contain debugging symbols. This allows the debugger to access the variables, arguments and functions by name. It also allows the debugger to map the currently executing instruction with the corresponding source code line.

Use "`~strip mybinary.out`" to strip the symbol table and debugging symbols.

Use "`~gcc -g`" or "`~clang -g`" when compiling to add debugging symbols.

There are three different ways to start debugging a process:

- **Out of Shell**: Start gdb with the path of your program as an argument. The debugger then starts the process.

  Usage: `~gdb /path/to/program`
- **Out of Debugger**: Start the debugger and then load the program with the file command. The debugger then starts the process.

  Usage: after starting gdb `~file /path/to/program`
- **Attach to process**: The debugger connects to a running process and halts it. The process is put under debugging.

  Usage: `~gdb -p <PID>` or in gdb `attach <PID>`

5.4 Setting Breakpoints & Watchpoints

A debugger uses breakpoints and watchpoints to control the flow of the process and halt it when needed. This section takes a closer look into these two concepts.

5.4.1 Breakpoints

Breakpoints are instructions in the text section of the process. These instructions halt the flow of the program and transfer control to the debugger.

The debugger sets these trap instructions by replacing the actual instruction in the text section and saving it to execute after the debugger continues the process. To do so, it uses PTRACE.

When a breakpoint is reached, the debugger halts the process and saves the state of the process. After finishing, the debugger resets the previous state of the process (resets the register values and executes the saved instruction)

Breakpoints can be set as following:

- **On Symbol**: Sets the breakpoint on a symbol. This requires the Symbol table. It allows us to break execution when a specific function is called.

  Usage: In gdb " break symbol" or " b symbol"
- **On Addresses**: Sets the breakpoint in an address. When the address is reached, the trap instruction halts the process.

  Usage: In gdb " break *0xdeadbeef" or " b *0xdeadbeef"
- **By Code line**: Sets the breakpoint on the instruction corresponding to the line of the source code. Requires the source code included in the binary.

  Usage: In gdb " list" lists the lines of code and " break 3" breaks by line 3

5.4.2 Watchpoints

Watchpoints are similar to breakpoints. They check if an address in memory is accessed. This can be a read or write access. If debugging symbols are included, it is possible to watch variables using gdb.

There are three types of watchpoints:

- **On Write**: If the process writes in the watched address the process is halted.

  Usage: In gdb " watch *0xdeadbeef"
- **On Read**: If the process reads in the watched address the process is halted.

  Usage: In gdb " rwatch *0xdeadbeef"
- **On Read/Write**: Halts the process if any of the conditions above is satisfied.

  Usage: In gdb " awatch *0xdeadbeef"

5.4.3 Managing Breakpoints and Watchpoints

While debugging it becomes necessary to manage the breakpoints(BP) and watchpoints(WP). In gdb you can read the information on your WP/BP with the `"~info breakpoints"` command. This command lists all your WP/BP and their assigned integer(each BP/WP is assigned an integer value). You can enable and disable WP/BPs with the `"~enable <int>"` and `"~disable <int>"` commands. To permanently remove a BP/WP use `"~del <int>"`.

5.5 Control Execution

Debuggers use a variety of features. Some of them are to get information on the state of the process while others help with navigation, reading and writing. This section examines some important gdb commands.

5.5.1 Navigation

After starting gdb and attaching it to a process, you need to navigate through the execution of the process. Setting breakpoints in every step of the execution or stepping through every instruction becomes hard and confusing. When working with projects of considerable size, it becomes a necessity to navigate properly. The list below introduces some important commands.

- **nexti || ni**: Executes the current instruction and moves to the next. If the current instruction is a function call, then it executes the function and moves to the instruction after the return of the function called.

- **stepi || si**: It is similar to nexti, but if a function is called, then it steps into that function, moving to the next executed instruction(beginning of the called function).
- **run || r**: It (re)starts the process under debugging. It lets the process run until it reaches a breakpoint
- **continue || c**: After a breakpoint the continue command allows the process to run until it hits the next breakpoint.
- **jump <location> || j <location>**: Jump to a location in memory. Continues execution from there.

5.5.2 Reading the State and Memory

Reading the state of the process and reading from memory is not always a straightforward task. It requires interpreting the data, especially when it comes to complicated data structures like structs or unions. This section examines some useful commands related to the task.

- **disassemble**: Gives the assembly representation. If the symbol table is present, then you can disassemble single functions by using their name.

  Usage: `"~disassemble main"` to disassemble main.
- **info || i**: Gives information on a specified part of the process. You need to specify what information you require. GDB supports the following: address, registers, files, functions, line, source, symbol, types, variables, vector, vtbl. Some of these require the symbol table.

  Usage: Take a look at the following site.

  `https://visualgdb.com/gdbreference/commands/information_displaying_commands`
- **print**: Prints the value of a given expression.

  Usage: Take a look at the following site.

  `https://visualgdb.com/gdbreference/commands/print`
- **x**: Displays the memory contents at a given address using the specified format.

  Usage: Take a look at the following site.

  `https://visualgdb.com/gdbreference/commands/x`

5.6 Advanced Topics and Usability

GDB allows the user to program scripts to automate debugging. It is possible to pass python scripts to gdb allowing the debugging process to be much faster.

To further enhance the experience, you can also use conditional breakpoints. You can write the condition when setting the breakpoint or add it later with the condition (or short cond) command.

GDB also allows moving back in the timeline of the execution. It manages to do so by keeping a full record of each step executed. The "record-full" option allows for "Time Travel".

In the debugging environment, everything is can be set to be writable. That includes the instructions. Overwriting instructions is called patching. To patch an instruction in gdb a user should activate the "write" option (" set write on"). Then the user can assign a value to some address with a typecast. For example, set int0x4023a3 = 0x90909090 sets four bytes of memory, since int is four byte. The given example writes four NOP instructions.

GDB is also highly customisable. The user can set custom fonts, colours, layouts, etc. To spare time, the user can hook the stop function of gdb to execute some commands after every stop. PwnGDB uses this technique to print information, like the register values, the stack and the instructions.

Debuggers are fun. Here are some fun activities for debugging kiddies:

- Control the execution path by changing register values. For example, a method checks a password and returns zero if the password was correct.
- Change the value of variables in a program. For example, a certain function only gets called if x is 42.
- Change the control flow by patching instructions. Skip a sanity check.
- Change the execution path by changing the address of a jump instruction.

5.7 Tools

- **GDB**: GDB is a Linux command-line debugger with a wide range of functionality. There are a lot of resources on gdb, making it easy to find information or tutorials on it.
- **EDB**: A Linux Debugger with a GUI. It contains a lot of features and it promises usability.

- **Windbg** : WinDbg is a kernel-mode and user-mode debugger that is included in Debugging Tools for Windows.
- **x64dbg**: Windows debugger with a GUI interface. can debug 64 and 32 Bit applications. Easy to learn.

6 Data-Flow Analysis

Data-Flow Analysis is a technique that analyses how the data changes during execution. The purpose of Data-Flow Analysis is to derive information about runtime-behaviour through static information. In this context, we use abstraction to combine information for all possible states of a process at some point in the program.

This chapter introduces Data-flow Analysis along with some typical problems and algorithms. This chapter also presents the lattice theory and its usage in Data-Flow Analysis.

6.1 Introduction

Data-Flow analysis can be used to collect information on variables defined and used by the program at specific program points. This information is collected by approximating possible runtime-behaviour. It serves the purpose of computing universal properties and constraints on variable values. The properties and constraints derived through Data-Flow analysis are valid for one specific point of the program, e.g. one line of the source code or one CFG-Block.

6.1.1 Usages

Data-Flow analysis is fruitful in many fields of program analysis. The following list gives some examples:

- **NULL-Pointer Dereference**: By defining the constraints and properties of the point in the program where a pointer gets dereferenced, we can examine if the value of that pointer can be NULL.
- **Follow User Input**: Following the flow of data, we can follow the user input to analyse what the program does with it. We can analyse which parts of the programs affect the data and which get affected by it.

- **Optimisation**: Data-Flow analysis helps optimise programs by detecting unnecessarily repeated instructions. For example, if a variable gets redefined during a loop without it changing through the iterations, then that variable can be defined before the loop.
- **Finding Bugs**: Data-Flow analysis can discover many bugs. One of them is infinite loops. If a variable defines the conditions of a loop but never gets changed during the loop, then we have an infinite loop.
- **Others**: Data-Flow analysis is also used in (de)compiler, refactoring, finding vulnerabilities, etc.

6.1.2 Control-Flow vs Data-Flow

The control flow of a program defines the order of execution for instructions and function calls. It gives a representation of the program's execution paths. The control flow depicts relationships between operations in a program, such that one operation will be executed after the other.

The data flow gives a representation of the life cycle of data in a program. It defines how values are computed and modified during the process. The data flow depicts relationships between operations and data(e.g. Variables) in a program, such that data produced by one operation is consumed by the other.

6.2 Abstraction

As defined, the data-flow analysis examines all possible states of a program at a certain point of it. However, when it comes to loops(or even jump instructions), the number of program paths leading to the same program point grows quickly. It is not possible to keep track of all the possible states. Hence it is necessary to use abstraction and only keep useful information.

Data-flow analysis discriminates between obligatory and potential information. An obligatory statement gives information that **must** be true, while potential statements give information that **may** be true. To reduce the amount of data analysed and make the process more efficient it discards one of the categories and keeps the other.

6.2.1 Definitions

For the purposes of this chapter we define the following:

- **Domain**: A set of possible data-flow values, e.g. variables, definitions, expressions, etc.
- **`IN[s] or IN[B]`**: Data-flow value before statement s or Block B.
- **`OUT[s] or OUT[B]`**: Data-flow value after statement s or Block B.
- **Transfer Function** f_s: Relation between `OUT[s]` and `IN[s]` inside a Block. This relation has two cases:
  - *Forward*: `OUT[s]` $= f_s($`IN[s]`$)$
  - *Backward*: `IN[s]` $= f_s($`OUT[s]`$)$

Note that the order in which the blocks are executed is not necessarily immutable. Data-Flow analysis has to consider that. In order to be able to define a similar transfer function for Blocks we define the following:

- **Block Transfer Function**: The transfer functions of each statement in the block executed in order. $f_B = f_{s_n} \circ f_{s_{n-1}} \circ ...f_{s_2} \circ f_{s_1}$
- **Meet Operator**: In math, the meet of a set S is the infimum (greatest lower bound) of that set. In this context, we will use the meet operator to define how the information on branching points is dealt with. Remember that we usually keep only one information category (obligatory or potential) this will influence our meet operator.

These definitions allow us to define the following statements:

- In Forward-Analysis `OUT[B]` $= f_B($`IN[B]`$)$

  `IN[B]` $= \bigwedge_{P \in N^-(B)}$`OUT[P]`
- In Backward-Analysis `IN[B]` $= f_B($`OUT[B]`$)$

  `OUT[B]` $= \bigwedge_{S \in N^+(B)}$`IN[S]`

6.2.2 Abstraction for concrete Algorithms

In this chapter, we will define concrete algorithms using the following table:

Abstraction	Possible Values
Domain	A Set
Flow Direction	Forward or Backward
Transfer Funtion	Data modification in each step
Meet Operator	How branching is handled
OUT[B]	The Output of block B
IN[B]	The input of block B

6.3 Data-Flow Algorithms

Through abstraction, we can define concrete problems and their solutions. This section presents some typical problems concerning data flow and the algorithms used to solve them.

6.3.1 Reaching Definitions

Accessing undefined variables leads to unexpected behaviours. It is necessary to detect if undefined values are used in a program. Another problem of similar nature is the Loop-Invariant Code Motion. This problem occurs when a statement or an expression inside of a loop can be moved outside the body of the loop without affecting the program. It is convenient to solve this problem automatically through Data-Flow analysis.

The Reaching Definitions algorithm is well-suited to solve problems of nature. In Reaching Definitions, we say that a definition **d** of a variable **v** **reaches** a point **p** in the program if there is a path from **d** to **p** that does not contain a redefinition of **v**. We say that a definition **d** of **v** is **killed** if there is any redefinition of **v** anywhere along the path. If the statement **d** at a program point **p** is a definition, we say that point p generates definition **d** or short "$gen_p = \{d\}$". Consider the following code:

```
int x = input;
int y = constant;
if (x < y) {
    x = x + 42;
}
print(x);
```

Lines 1, 2 and 4 generate definitions. There are two possible paths for the execution of this program. The condition at line 3 is either satisfied or not. If the condition is not satisfied, then the definition of x(**v**) at line 1(**d**) reaches line 6(**p**) of the program, else we say that the definition of x at line 1 is killed at line 4. Note that the Reaching Definitions algorithm does not evaluate the conditions, and it assumes both results are possible. This leads to the following inaccuracy:

```
 1  int foo(int x, int y) {
 2      int z = 0;
 3      if ( x != y) {
 4          z = 5;
 5      } else{
 6          z = 6;
 7      }
 8      if ( x == y) {
 9          x = y + z;
10      }
11      y = bar(z);
12      return x + y;
13  }
```

Following the Reaching Definitions algorithm, both definitions of z at lines 4("z = 5") and 6("z = 6") could reach line 9 because the condition at line 8 is not evaluated and compared to the condition at line 3.

According to Rice's theorem, the set of possible paths in a CFG is undecidable. Instead of deciding which paths are possible, we consider them all as equally possible for the algorithm.

The Reaching Definitions algorithm follows this abstraction scheme:

Abstraction	Values
Domain	set of definitions
Flow Direction	Forward
Transfer Funtion	$f_d(x) = gen_d \cup (x - kill_d)$ $f_B(x) = gen_B \cup (x - kill_B)$
Meet Operator	Union $\cup$
OUT[B]	$f_B(IN[B])$
IN[B]	$\bigcup_{P \in N^-(B)} OUT[P]$

The domain that the algorithm considers is the set of definitions. The flow direction is forward. The observed differences after each point in the program (or after each block) can be generations of definitions or kills of definitions. Hence the transfer function considers definitions generated and killed. It adds generated definitions to the set and removes the killed definitions. The chosen meet operator is the union operator since we need to consider all incoming information.

The following is a pseudo-code representation of the algorithm. The algorithm computes the output of each block regarding the information on the block itself and the output of the blocks preceding it. As long as the output of one Block changes, it means that the others can change too since the information on other blocks has changed.

```
OUT, IN = dict(), dict()
for B in CFG nodes:
    OUT[B] = set()
while OUT changes:
    for B in CFG nodes:
        IN[B] = union(OUT[P] for P predecessor of B)
        OUT[B] = gen[B]  | (IN[B] - kill[B])
```

Listing 6.1: *Reaching Definitions*

To prove that the algorithm terminates we have to consider the following:

1. The number of definitions in the IN[B] & OUT[B] function has an upper bound.
2. The size of OUT[B] never decreases.

The first is trivial since the number of total definitions in the program gives an upper bound. For the second statement, we need to take a look at the transfer function. The **kill** function executes only if there is a **gen** function. The transfer function does one of the following: nothing, replace one definition with another or add a new definition. The size of OUT[B] never decreases.The two statements combined imply the algorithm can only run for a finite number of steps.

6.3.2 Liveness Analysis

There are problems in Data-Flow analysis that we cannot solve with the reaching definition algorithm. One of these problems is detecting the redefinition of variables before use. If a variable is redefined before being used, the first definition is not needed.

Another problem is the management of restricted finite resources, e.g. Registers in a machine. For performance, we want to change register values as few times as possible. Which current register values should the program overwrite to optimise performance?

Liveness Analysis is a technique that determines if a variable **v** is used after a point **p** in the program without being redefined. In Liveness Analysis, we say that a variable **v** is (a)**live** at a point **p** if there is a path from **p** to a usage of **v** that does not contain a redefinition of **v**. A variable **v** is **dead** at a point **p** if it is not (a)**live**. Consider the following program:

```
1  int foo (int a){
2      x = 0;
3      y = 1;
4      if (a < 20){
5          x = y;
6          y = 2;
7      } else {
8          x = 5;
9      }
10     return a + x - y;
11 }
```

The variables x and y are defined in lines 2 and 3. The program has two possible execution paths. The first satisfies the condition on line 4, and the second does not. Now, consider line 10 as a point p in this program. The first path redefines both x and y, while the second redefines only x. In this case, we say that the definition of y is still live at line 10 since there is a path from line 3 to line 10 where y is not redefined. The definition of x at line 2 is not live at line 10 because every possible path redefines x. For the abstract representation of this problem we define the following functions:

- $def_p = \{v|\text{v defined at p}\}$: This function returns the variable v defined at point p.
- $use_p = \{v|\text{v used at p}\}$: returns all variables used at point p

The following table defines the abstract representation of the Liveness Analysis algorithm:

Abstraction	Values
Domain	set of variables
Flow Direction	Backward
Transfer Funtion	$f_d(x) = use_d \cup (x - def_d)$ $f_B(x) = use_B \cup (x - def_B)$
Meet Operator	$Union \cup$
IN[B]	$f_B(OUT[B])$
OUT[B]	$\bigcup_{S \in N^+(B)} IN[S]$

The domain, in this case, is the set of variables. The order in which the definitions and blocks are analysed is the execution order in reverse. The algorithm examines the usage and (re)definition of variables. Since a variable is (a)live, if it is used in any path, the meet operator is Union.

The algorithm for Liveness Analysis looks similar to reaching definitions with small differences. A pseudo-code representation of the algorithm is given below. Note the usage of IN[B] and OUT[B].

```
OUT, IN = dict(), dict()
for B in CFG nodes:
    IN[B] = set()
while IN changes:
    for B in CFG nodes:
        OUT[B] = union(IN[S] for S successor of B)
        IN[B] = use[B]  | (IN[B] - def[B])
```

Listing 6.2: *Liveness Analysis*

This algorithm terminates. The proof is the same as for Reaching Definitions with the difference being that for Liveness Analysis we need to show that IN[B] never decreases instead of OUT[B].

6.3.3 Available Expression

The previous two algorithms examine if there is at least one path that satisfies one condition. Sometimes it is necessary to know if all paths satisfy one condition.

Consider the following function foo(1). Note that the expression "x * 2" is used several times. Whenever the execution reaches line 9, it is certain that the expression "x * 2" has already been evaluated once. To optimise the program the compiler may save the value and avoid reevaluating. Note that this is not the case for line 11.

```
1
2       int foo(int x, int y){
3           if (x > y){
4               if (x < 20){
5                   y = x * 2
6               } else {
7                   x = x * 2
8               }
9               x++;
10              z = x * 2
11          }
12          y = x * 2
13          return z
14      }
```

Listing 6.3: *Already Defined Expression*

Unnecessarily reevaluating expressions is problematic considering performance. In Data-Flow analysis, we use the Available Expression algorithm to find the already-defined expressions.

An expression **e** is **available** at a point **p** if every path to **p** evaluates **e** without redefining any used variable after the last evaluation. A point **p kills** an expression **e** if it defines one of the used variables.

The fitting meet operator for this algorithm is the intersection operator because the statement has to be valid for every possible path. The transfer function is defined as follows:

- $f_d(x) = e\_gen_d \cup (x - e\_kill_d)$

  with:

  - $e\_kill_p = \{e | v \in e\}$ expressions using variable defined at p
  - e_gen_p = {e | e evaluated at p ∧ v ∉ e} all expressions evaluated at point p not using the defined variable v at p
- $f_B(x) = e\_gen_B \cup (x - e\_kill_B)$

The following table defines the abstract representation of the Available Expression algorithm:

Abstraction	Values
Domain	set of expressions
Flow Direction	Forward
Transfer Funtion	$f_d(x) = e\_gen_d \cup (x - e\_kill_d)$ $f_B(x) = e\_gen_B \cup (x - e\_kill_B)$
Meet Operator	Intersect ∩
OUT[B]	$f_B(IN[B])$
IN[B]	$\bigcap_{P \in N^-(B)}$ OUT[P]

The following is a pseudo-code representation of the algorithm:

```
OUT, IN = dict(), dict();

for B in CFG nodes:
    OUT[B] = U;
while OUT changes:
    for B in CFG nodes:
        IN[B] = intersection(OUT[P] for P predecessor of B)
        OUT[B] = e_gen[B] | (IN[B] - e_kill[B])
```

Once again, OUT[B] never increases. This means that the algorithm terminates.

6.3.4 Very Busy Expression

You probably know the drill by now: This chapter goes through one last algorithm that is important for data-flow analysis.

Available Expression finds expressions that are already evaluated at point p because they are needed in every possible path. However, Available Expression does not consider parallel paths. Consider the following code:

```
 1  int foo ( int a, int b){
 2      if ( a > b ){
 3          x = a - b;
 4          y = b - a;
 5      }
 6      else {
 7          x = b - a;
 8          y = a - b;
 9      }
10      ...
11  }
```

Consider lines 3 and 4 and compare them to lines 8 and 9. The expressions used for x and y are the same. They are just assigned to different variables. Because these assignments are on parallel paths, Available Expression wouldn't find them. For problems of this nature, we use the Very Busy Expressions algorithm.

An expression **e** is **very busy** at point **p** if every path from **p** uses **e** before any variable used in **e** is redefined. A point **p kills** an expression **e** if it defines one of the used variables.

The Very Busy Expressions algorithm defines the following:

- Meet Operator : Intersection. The algorithm delivers a statement on all the paths leading to a point p.
- $f_d(x) = e\_use_d \cup (x - e\_kill_d)$

  with:

  - $e\_kill_p = \{e|v \in e\}$ expressions using variable defined at p
  - $e_use_p = \{$e | e used at p$\}$ all expressions used at p

- $f_B(x) = e\_use_B \cup (x - e\_kill_B)$

These definitions lead to the following abstraction:

Abstraction	Values
Domain	set of expressions
Flow Direction	Backward
Transfer Funtion	$f_d(x) = e\_use_d \cup (x - e\_kill_d)$ $f_B(x) = e\_use_B \cup (x - e\_kill_B)$
Meet Operator	Intersect $\cap$
IN[B]	$f_B(IN[B])$
OUT[B]	$\bigcap_{S \in N^+(B)}$ IN[S]

The following is a pseudo-code representation of the algorithm:

```
OUT, IN = dict(), dict();

for B in CFG nodes:
    IN[B] = U;
while IN changes:
    for B in CFG nodes:
        OUT[B] = intersection(IN[S] for S successor of B)
        IN[B] = e_use[B] | (OUT[B] - e_kill[B])
```

Once again, IN[B] never increases. This means that the algorithm terminates.

6.4 Lattice Theory

The Lattice theory lays fundamental work for Data-Flow analysis. It can be used in various ways to define problems and algorithms in a unified way. Using the Lattice theory, we can define a Data-Flow Analysis framework.

6.4.1 Foundations of Data-Flow Analysis

A data-flow analysis framework can be defined as a four-tuple (D, V, lang, F) as follows:

1. **D**: Direction of the flow.

2. **V**: A semilattice with domain V and a meet operator ∧
3. A family F of transfer functions

In the previous section, we defined the four algorithms using a similar framework. A quick reminder of that:

Algorithm	D (flow direction)	V (domain)	∧ (meet operator)
Reaching Definitions	Forward	2^U with U={all definitions}	$\bigcup$
Liveness Analysis	Backward	2^U with U={all variables}	$\bigcup$
Available Expression	Forward	2^U with U={all expressions}	$\bigcap$
Very Busy Expression	Backward	2^U with U={all expressions}	$\bigcap$

Algorithm	F (transfer functions family)
Reaching Definitions	$f_d(x) = gen_d \cup (x - kill_d)$ $f_B(x) = gen_B \cup (x - kill_B)$
Liveness Analysis	$f_d(x) = use_d \cup (x - def_d)$ $f_B(x) = use_B \cup (x - def_B)$
Available Expression	$f_d(x) = e\_gen_d \cup (x - e\_kill_d)$ $f_B(x) = e\_gen_B \cup (x - e\_kill_B)$
Very Busy Expression	$f_d(x) = e\_use_d \cup (x - e\_kill_d)$ $f_B(x) = e\_use_d \cup (x - e\_kill_d)$

6.4.2 Semilattice and Partial Order

The definition of the framework includes the concept of a semilattice. In general, a semilattice is a set which is equipped with a binary operation that is associative, commutative and idempotent. For the purposes of this chapter we define a semilattice and partial order as follows:

A semilattice is a set **V** and a binary meet operator ∧ with $\wedge : VxV \rightarrow V$ such that for all x,y,z ∈ **V** holds:

1. x ∧ x = x (**idempotent**)
2. x ∧ y = y ∧ x (**commutative**)
3. x ∧ (y ∧ z) = (x ∧ y) ∧ z (**associative**)

- A semilattice has a top element $\top$ such that $\top \wedge x = x$
- Optionally, it also has a bottom element $\perp$ such that $\perp \wedge x = \perp$

A partial order is a set **V** and a binary relation $\sqsubseteq$ such that for all x,y,z $\in$ **V** holds:

1. $x \sqsubseteq x$ (**reflexive**)
2. $x \sqsubseteq y$ and $y \sqsubseteq x \rightarrow x = y$ (**antisymmetric**)
3. $x \sqsubseteq y$ and $y \sqsubseteq z \rightarrow x \sqsubseteq z$ (**transitive**)

- $(V, \sqsubseteq)$ is called partially ordered set.
- $(x \sqsubseteq y \wedge x \neq y \iff x \sqsubset y)$

A semilattice is a partially ordered set. Consider the three requirements for a set and a relation to be partially ordered:

- **reflexive**: $(V, \wedge)$ from the idempotence follows: $(x \wedge x = x) \rightarrow x \sqsubseteq x$
- **antisymmetric**: $(V, \wedge)$ commutative $(x \wedge y = y \wedge x) \rightarrow (x \sqsubseteq y \wedge y \sqsubseteq x \rightarrow x = y)$
- **transitive**: $(V, \wedge)$ associative $(x \wedge (y \wedge z) = (x \wedge y) \wedge z) \rightarrow x \sqsubseteq y \wedge y \sqsubseteq z \rightarrow x \sqsubseteq z$

6.4.3 GLB and LUB

The Greatest Lower Bound (GLB) and the Least Upper Bound (LUB) are important concepts of theoretical informatics and Data-flow analysis. Consider their definitions:

Let $(V, \wedge)$ be a partially partially ordered set and x,y $\in V$. $g \in V$ is a greatest lower bound for x and y if the following holds:

1. $g \sqsubseteq x$
2. $g \sqsubseteq y$
3. $z \in V \wedge z \sqsubseteq x \wedge z \sqsubseteq y \rightarrow z \sqsubseteq g$

Let $(V, \wedge)$ be a partially ordered set and x,y $\in V$. $u \in V$ is a least upper bound for x,y if the following holds:

1. $x \sqsubseteq u$
2. $y \sqsubseteq u$
3. $z \in V \wedge x \sqsubseteq z \wedge y \sqsubseteq x \rightarrow u \sqsubseteq z$

With these concepts defined we can define a lattice. A lattice is a partial order $(V, \sqsubseteq)$ with a meet operator $\wedge$ and a join operator $\vee$, so that $\forall x, y \in V$ holds:

- $x \wedge y \in V$ glb of x and y
- $x \vee y \in V$ lub of x and y

6.4.4 Lattice Diagrams

A lattice diagram is a form of visual (graph) representation of a lattice. They are a quick way of communicating the information on a lattice through a graph. Consider the following definition:

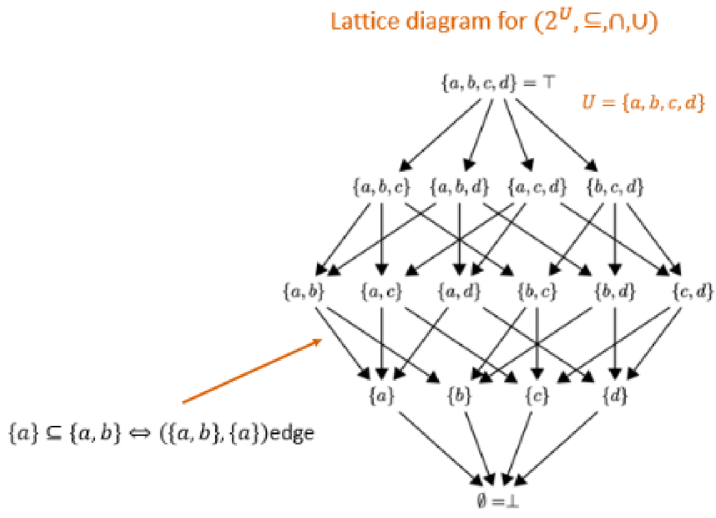
Let $(V, \sqsubseteq)$ be a (semi-)lattice with a meet operator $\wedge$ and a join operator $\vee$. The (semi-)lattice diagram is a directed Graph G with:

- $V(G) = V$ domain of lattice
- $E(G) = (x, y) \in VxV | y \sqsubseteq x$

A lattice diagram has the following properties:

- The **height** of a partially order $(V, \sqsubseteq)$ is the largest number of $\sqsubseteq$ relations in any ascending chain $x_1 \sqsubseteq x_2 \sqsubseteq ... \sqsubseteq x_n$
- GLB $g = x \wedge y$ is the node that is reachable from x and y and closest with this property.

This is how a lattice diagram looks like:



Lattice diagram for $(2^U, \subseteq, \cap, \cup)$
$\{a,b,c,d\} = \top$
$U = \{a,b,c,d\}$
$\{a,b,c\}$ $\{a,b,d\}$ $\{a,c,d\}$ $\{b,c,d\}$
$\{a,b\}$ $\{a,c\}$ $\{a,d\}$ $\{b,c\}$ $\{b,d\}$ $\{c,d\}$
$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$
$\emptyset = \bot$
$\{a\} \subseteq \{a,b\} \Leftrightarrow (\{a,b\},\{a\})$edge

Figure 3: *A Lattice Diagram*

6.4.5 Family of Transfer Functions

In Data-Flow analysis we define a family of transfer functions as follows:

A family F of transfer functions is a collection of functions from V to V fulfilling the following properties:

1. F contains the identify function id s.t. $\forall_{x \in V} : id(x) = x$
2. F is closed under composition, i.e., $f, g \in F \rightarrow f \circ g \in F$

With the aforementioned definitions, we can now define the properties of Data-Flow Frameworks. A data-flow analysis framework $(D, \vee, \wedge, F)$ with associated partial order $(V, \sqsubseteq)$ is called:

- **monotone**, if: $\forall_{x,y \in V} \forall_{f \in F} : x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$
- **distributive**, if: $\forall_{x,y \in V} \forall_{f \in F} : f(x \wedge y) = f(x) \wedge f(y)$

A data-flow analysis framework $(D, \vee, \wedge, F)$ is **monotone** if and only if $\forall_{x,y \in V} \forall_{f \in F} : f(x \wedge y) \sqsubseteq f(x) \wedge f(y)$

6.4.6 Algorithm for data-flow Framework

Previously we designed algorithms for specific problems of data-flow analysis. As you may have noticed they all had a similar form. This section shows a general approach to an unspecified data-flow problem. We define such an algorithm as follows:

Input:A data-flow framework with the following components

1. A data-flow graph, with an Entry(Forward) or Exit(Backward) node
2. A direction of the data-flow D
3. A set of values V
4. A meet operator $\wedge$
5. A set of function F, where f_B in F is the transfer function for Block B
6. A constant value v_ENTRY or v_EXIT representing the boundary condition for forward and backward framework

Output: Values in V for IN[B] and OUT[B] for all nodes

```
OUT, IN = dict(), dict();
    IN[EXIT] = v_EXIT
    for B in CFG nodes:
        IN[B] = ⊤
    while IN changes:
        for B in CFG nodes:
            OUT[B] = ∧(IN[S] for S successor of B)
            IN[B] = $f_B$(OUT[B])
```

Listing 6.4: *Forward Flow Direction*

```
OUT, IN = dict(), dict();
IN[ENTRY] = v_ENTRY
for B in CFG nodes:
    OUT[B] = ⊤
while OUT changes:
    for B in CFG nodes:
        IN[B] = ∧(OUT[S] for S successor of B)
        OUT[B] = $f_B$(IN[B])
```

Listing 6.5: *Backward Flow Direction*

- The algorithm converges if the framework is monotone and the semilattice $(V, \wedge)$ has a finite height.
- The algorithm delivers a solution to the data-flow equations (IN and OUT) if it terminates.
- If the framework is monotone, then the solution is the maximal fixed point (MFP) of the data-flow equation.

6.5 Angr

Angr is a cross-platform multi-architecture binary analysis suite. It is developed as a python package and can be imported through pip. Angr offers a lot of services including static, dynamic and symbolic (concolic) analysis.

Angr supports:

- Disassembly and intermediate-representation lifting
- Program instrumentation
- Symbolic execution
- Control-flow analysis
- Data-dependency analysis
- Value-set analysis (VSA)
- Decompilation

Angr: `https://angr.io/`

CTF with Angr : `https://docs.angr.io/examples`

7 Data-Flow Analysis 2

The previous chapter presented a formal definition of a data-flow framework and its usage in defining. This allowed us to define algorithms regarding data-flow problems fairly easily. This chapter builds on the previous one and presents manners of optimising and extending the data-flow analysis.

To build a representation of code, that is better suited for data-flow analysis, this chapter defines the SSA-Form. Afterwards, we define the concepts of Taint Analysis, Satisfiability, Symbolic Execution and Value-Set Analysis.

7.1 SSA-Form

The runtime of the data-flow algorithms as we defined them depends on the form of the code. In the most optimal case, each variable is only defined once. This allows the algorithm to terminate in linear time. The SSA-Form is a representation of Code, in which each variable is only defined once.

7.1.1 Use-Def and Def-Use Chains

While defining the Reaching Definitions algorithm we defined sets of definitions available at a certain point in execution. We can think of these sets as the union of the available definitions for each variable at said point. In doing so, we can define the Def-Use and Use-Def chains.

- **Use-Def:** A **Use-Def** (UD) chain for a variable **v** connects a usage **u** of **v** to all the definitions that may reach usage **u**.
- **Def-Use:** A **Def-Use** (DU) chain for a variable **v** connects a definition **d** of **v** to all possible usages of definition **d**

To optimise our algorithms we need a representation of the program, where each Use-Def or Def-Use chain is a set consisting of one element. This representation is given by the SSA-Form.

7.1.2 Definition of the SSA-Form

SSA is a special intermediate representation that simplifies the data-flow analysis. In this representation, each variable is defined only once.

A program is SSA conform if:

1. Each variable is defined exactly once.
2. Each variable is defined before it is used.
3. Each variable can be used arbitrarily often.
4. The flow can enter a definition multiple times.

Through the character of the SSA-Form, it is easy to find the definitions of variables, which makes the reaching definitions algorithm on an SSA-conform program obsolete. It is easy to compute the Use-Def or Def-Use chains in linear time and identify dead code(unreachable code). The SSA-Form of a program needs to be computed once only, which makes it usable.

7.1.3 Computing the SSA-Form

Programs are rarely written in an SSA-conform way. They usually contain redefinitions of variables. Consider the following snippet of code:

```
int foo(int x){
    a = 2;
    b = 21;
    c = a + b;
    if ( x > c ){
        c = x - c;
    }
    else{
        c = x + c;
    }
    return c;
}
```

Listing 7.1: *Redefinitions in Code*

Function foo defines variable c three times, which means that foo is not SSA-conform. To transform function foo into an SSA-conform function we rename each redefinition of c as follows:

```
1  int foo(int x){
2      a = 2;
3      b = 21;
4      c_0 = a + b;
5      if ( x > c_0 ){
6          c_1 = x - c_0;
7      }
8      else{
9          c_2 = x + c_0;
10     }
11     return c_?;
12 }
```

Listing 7.2: *Renaming in Code*

Note that we do not know which of the two definitions of c returns. During the analysis, it becomes necessary to define a mechanism that decides which definition should be considered. This mechanism is usually called the ϕ-Function.

$$\phi : x_n -> \phi(c_0, c_1 ... c_{n-1})$$

The ϕ-function determines which of the definitions coming from predecessor blocks to consider. It takes as many parameters as there are predecessor blocks, and then chooses between the given definitions depending on what path is being analysed.

Consider the previous code snippet again. Line 11 gives a definition of c using the ϕ-function. This function takes two parameters since there are two definitions of c reaching line 11.

```
1  int foo(int x){
2      a = 2;
3      b = 21;
4      c_0 = a + b;
5      if ( x > c_0 ){
```

```
            c_1 = x - c_0;
        }
        else{
            c_2 = x + c_0;
        }
        c_3 = ϕ(c_1,c_2)
        return c_3;
    }
```

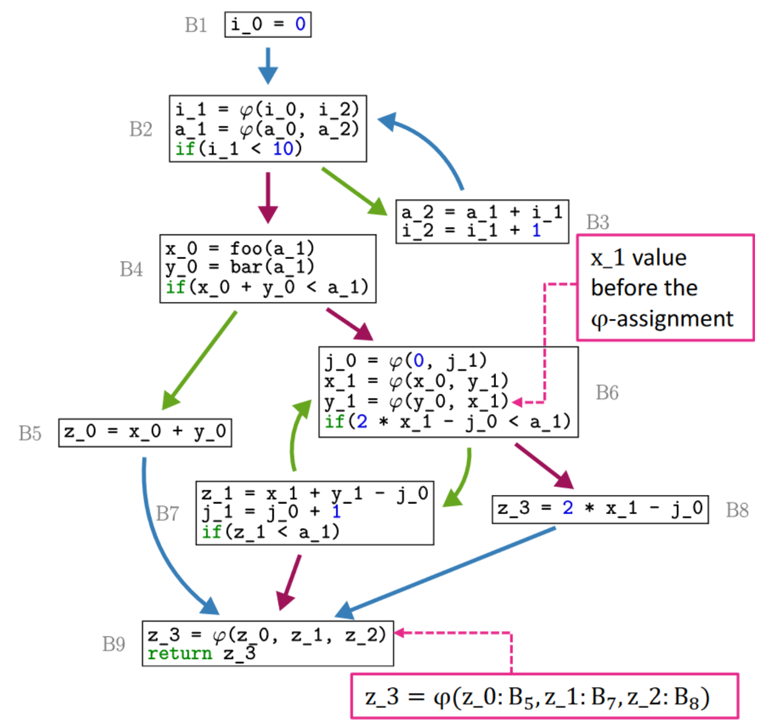
Listing 7.3: *Snippet*

Properties of The ϕ-Function

- ϕ-functions are always at the beginning of a basic block.
- ϕ-functions are all evaluated simultaneously.
- ϕ-functions select between values depending on control-flow

Some notations also indicate the predecessor block in the ϕ-function: c_3 = ϕ(c_1:B_2,c_2:B_3)

To better understand the ϕ function, it is advised to try using it on a CFG as follows:


B1
B2
B3
B4
B5
B6
B7
B8
B9
x_1 value before the φ-assignment
z_3 = φ(z_0: B5, z_1: B7, z_2: B8)

```
B1: i_0 = 0

B2: i_1 = φ(i_0, i_2)
    a_1 = φ(a_0, a_2)
    if(i_1 < 10)

B3: a_2 = a_1 + i_1
    i_2 = i_1 + 1

B4: x_0 = foo(a_1)
    y_0 = bar(a_1)
    if(x_0 + y_0 < a_1)

B5: z_0 = x_0 + y_0

B6: j_0 = φ(0, j_1)
    x_1 = φ(x_0, y_1)
    y_1 = φ(y_0, x_1)
    if(2 * x_1 - j_0 < a_1)

B7: z_1 = x_1 + y_1 - j_0
    j_1 = j_0 + 1
    if(z_1 < a_1)

B8: z_3 = 2 * x_1 - j_0

B9: z_3 = φ(z_0, z_1, z_2)
    return z_3
```

While working with the ϕ-function, there are two main issues to be considered.

- Where to put the *phi*-function.
- How to choose the indexes of the renamed variables.

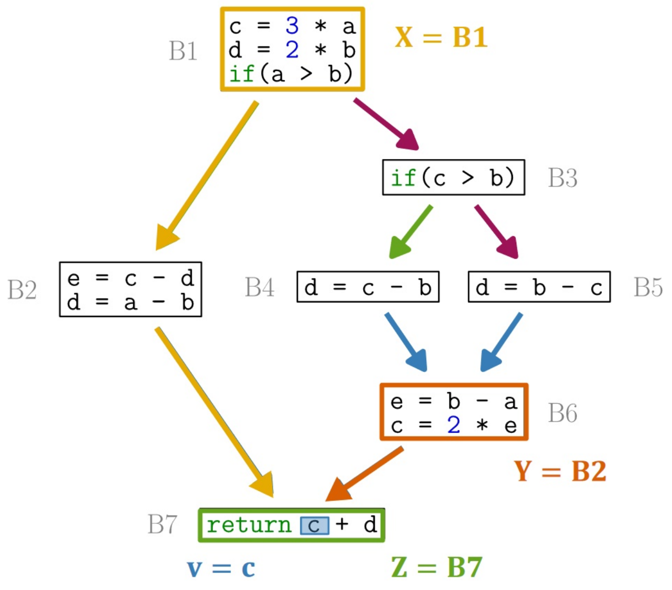
These questions will be answered in the following sections.

Placement of the ϕ - Function

Before defining where the ϕ function is placed, we define which variables need a ϕ function. Consider three Blocks X, Y, and Z. Let X and Y be predecessors of Z, defining a variable c and let Z use the variable c, then the variable c need a ϕ function if none of the paths is a subset of the other. In other words there are three conditions:

1. There are two blocks X & Y defining variable v.
2. There are nonempty paths P_{XZ} and P_{YZ} from X resp. Y to Z
3. $P_{XZ} \not\subset P_{YZ}$ and $P_{YZ} \not\subset P_{XZ}$.

If $P_{XZ} \cap P_{YZ} = \{Z\}$, then the ϕ- Function is placed in Block Z, otherwise in a predecessor of Z. Note that P_{AB} is a path from Block A to Block B, defined as a set of Blocks. This set is not uniquely defined for every A and B Block. Consider the following example. There are different paths from X to Z. One of the paths(marked in yellow) fulfils the conditions. That means that variable c needs a ϕ- function in Block Z.



Dominance and Dominance Frontier

If $P_{XZ} \cap P_{YZ} \neq \{Z\}$ then placing the ϕ-Function in Block Z is not necessarily optimal. A concept known as the dominance frontier helps us find an optimal placement for the phi function. To define the dominance frontier we have to define dominance first.

Let $G = (V, E)$ be a directed Graph (i.e. CFG) and $d, n \in V$

- d dominates n (d dom n) if every path from the entry to n goes through node d
- d strictly dominates n (d sdom n) if d dom n and d ≠ n
- d immediately dominates n (d idom n) if d sdom n and ∄c: d sdom c ∧ c sdom n

For a directed graph $G = (V, E)$ is $T = (V, F)$ the dominator tree if $F = \{(d, n) \in VxV$ | d idom n}.

The dominator tree has the following properties:

1. $\forall n \in V$: n dom n
2. $\forall n \in V \backslash \{entry\} \exists! d \in V$: d idom n
3. $\exists d, n$- Path in T $\Longleftrightarrow$ d dom n

To build the dominator tree it is required to know the dominating sets. A Dominating set of a node D in a directed graph is the set of all Nodes C dominating D. We can use the data-flow framework from the last chapter to compute these sets.

Abstraction	Value
Domain	V set of nodes
Flow Direction	Forward
Transfer Function	$f_B(x) = x \cup \{B\}$
Meet Operator	$\cap$
OUT[B]	$f_B(IN[B])$
IN[B]	$\bigcap_{P \in N^-(B)} OUT[P]$
T	V

With the dominating sets, we can now compute the dominance frontier.

Let $G = (V, E)$ be a directed graph and let $n \in V$. The dominance frontier of **DF** of n is notated as $DF[n]$ and it is a set off all $d \in V$ such that:

1. **n** dominates a predecessor of **d**
2. **n** does not strictly dominate **d**

Let $G = (V, E)$ be a directed graph and $T = (V, F)$ the dominator tree of the graph G. To compute the dominance frontier, consider the following definitions:

1. $DF_{local}[n] = \{s \in Succ_G(n) | \neg(\text{n idom s})\}$
2. $DF_{up}[n] = \{v \in DF[n] | \neg(\text{d sdom v})\}$

$$\begin{aligned} DF[n] &= DF_{local}[n] \cup \bigcup_{v:\text{n idom v}} DF_{up}[v] \\ &= DF_{local}[n] \cup \bigcup_{v \in N_T^+(n)} DF_{up}[v] \end{aligned}$$

The following pseudo-code algorithm computes the dominance frontier for each node n in a dominator tree T of a given CFG.

```
for n in post_order(T):
    DF[n] = set()
    # compute $DF_{local}$
    for s in Successor_CFG[n]:
        if (n,s) not in E(T):
            DF[n].add(s)
    for c in Successor_T[n]:
        # compute $DF_{up}$
        for w in DF[c]:
            if n != w or n,w -path in T:
                DF[n].add(w)
```

Listing 7.4: *Compute DF-sets*

Inserting the ϕ- Function

We use the dominance frontier to decide where to put the phi function. If a variable **v** is defined in block **B**, then we put a ϕ-function for the variable **v** in each block of the dominance frontier of **B**. This method ensures that the computed SSA representation has a minimal number of phi function calls.

The following pseudo-code algorithm gets a CFG as input and returns a set of sets. Each of these sets is mapped to a variable so that the set contains all the nodes in the CFG, that require a ϕ-function.

```
for n in CFG nodes:
    for v in {v variable defined in n}:
        defsites[v].add(n)
        def_of[n].add(v)

for each variable v:
    W = defsites[v]
    while W:
        n = W.pop()
        for y in DF[n]:
            if y not in PHI[v]:
                PHI[v].add(y)
                if v not in def_of[y]:
                    W.add(y)
```

Listing 7.5: *Compute DF-sets*

Renaming the Variables

With the algorithm defined above, we have successfully placed the phi function. To have a valid SSA-Representation, we still need to rename the variables with multiple definitions.

The following algorithm takes a CFG and its dominator tree as input and returns a CFG with renamed variables. If the input has the PHI function already inserted, then the output is the SSA Representation.

```
def rename_variables():
    # initialize
    for each variable v:
        count[v] = 0
        stack[v] = [0]
    rename(entry)
```

```
def rename(n)
    # rename all variables not used in ϕ-functions
    for each statement stmt in n:
        if stmt is not a ϕ-function:
            for each variable x in used[stmt]:
                i = stack[x][-1]
                replace usages of x by x_i in stmt
        for each variable x in defined[stmt]:
            count[x] += 1
            stack[x].append(count[x])
            i = stack[x][-1]
            replace definitions of x by x_i in stmt
    # END: rename all variables not used in ϕ-functions
    # rename variables used in ϕ-functions according to their predecessor
    for S successor of n in CFG:
        j = PredecessorNumber(s, n)
        for each ϕ-function p in S:
            i = stack[x][-1]
            replace the jth operand x of p by x_i
    # END: rename variables used in ϕ-functions according to their predecessor
    for each S successor of n in T:
    # Recursion over dominator tree
        rename(S)
    for variable v in defined[S]:
    # remove added definitions
        stack[v].pop()
```

Listing 7.6: *Compute DF-sets*

7.1.4 Translation Out-Of-SSA-Form

The SSA-Representation contains phi functions. While this representation is helpful for analysis and optimisation, after finishing those steps we need to go back to the standard representation. The transition from the SSA-Form to the standard representation is non-trivial.

To translate out of the SSA-Form, we need to separate the definitions again and move them to other positions. This can be as easy as moving instructions to the predecessor block or as complicated as iterating through the CFG to find the right position. Consider the easy example(4) below.

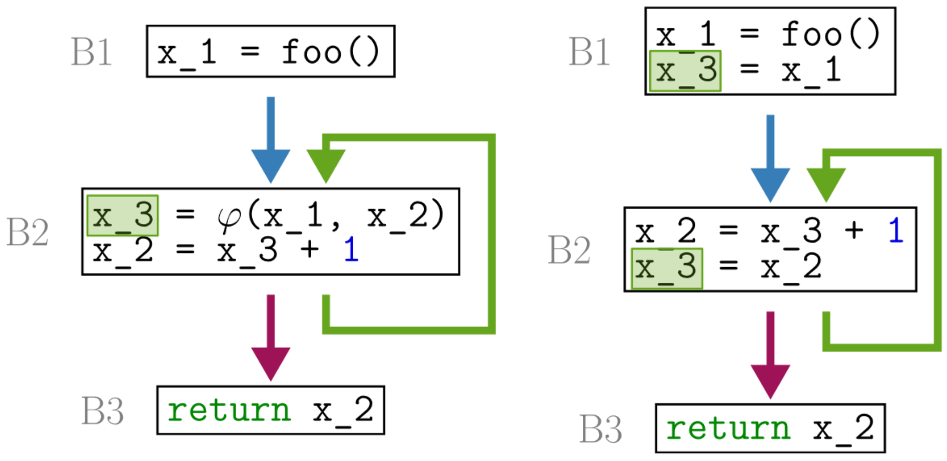


Figure 4: *An Easy Case*

In the given example, the translation is close to trivial. The ϕ function contains two definitions. The phi function chooses the second definition when the control-flow loops from B2 to B2. Otherwise, the ϕ function uses the first definition. In the example, the translation is done by moving the first definition to the predecessor block B1 and the second to the end of B2.

The Lost Copy Problem

Translation out of the SSA-Form for complex programs introduces several problems. One of these problems is known as the "Lost Copy Problem". Consider the following example(5).

In the incorrect approach, the value of x_2 is corrupted after the translation because it is no longer assigned to x_3 + 1 at the end of the block. Therefore, using x_2 after B3 delivers corrupted data. To solve this problem, we introduce copies that help us keep the data consistent. In the example, we remove the phi function on the copy using the naive approach. Since we reassign the data, there is no inconsistency.

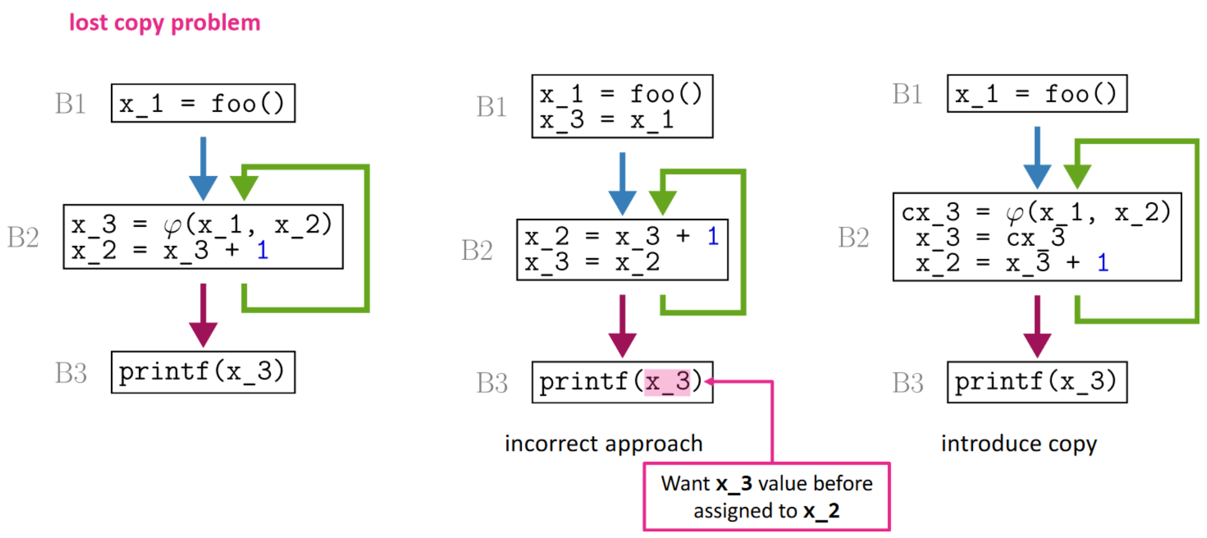


Figure 5: *Lost Copy Problem*

The Swap Problem

The Swap Problem is another issue we have when translating out of SSA. When a block contains more than one phi function, they are all executed at the same time. The swap problem occurs when one phi function uses the value assigned by another. Consider the example(6) below. Once again, we solve this problem by introducing copies to prevent data-corruption.

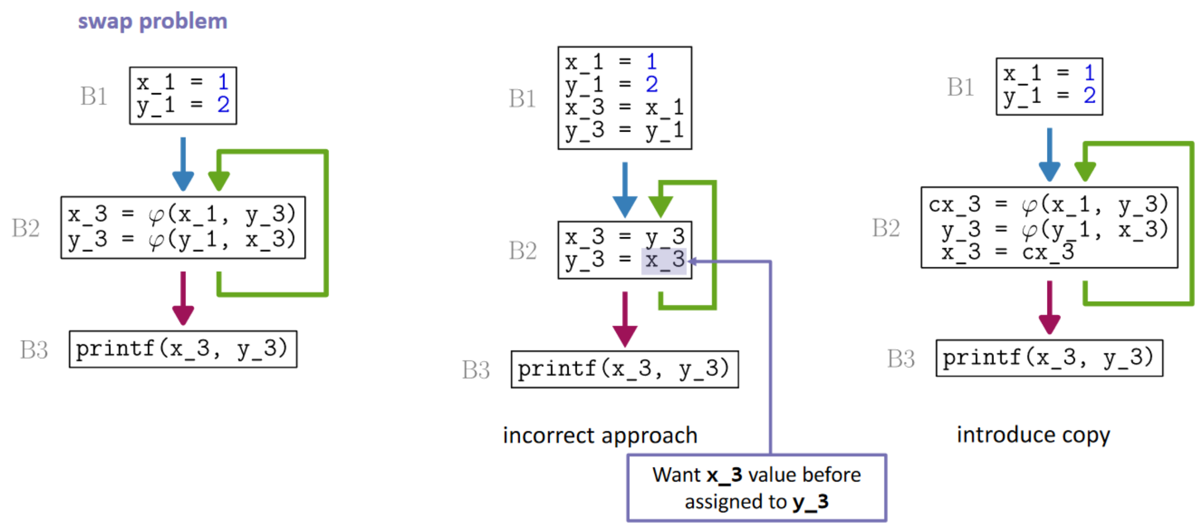


Figure 6: *Lost Swap Problem*

The following paper gives an overview of some problems of translating out of SSA and their solution.https://www.tjhsst.edu/~rlatimer/papers/sreedharTranslatingOutOfStaticSingleAssignmentForm.pdf

7.2 Taint Analysis

User input, in any form, can cause unexpected behaviours. These unexpected behaviours pose a security threat in form of information leakage, unchecked user input vulnerabilities, etc. We can detect such unexpected behaviour by tracking data through the program. In doing so we want to answer two questions:

1. Which values(i.e which variables) do they influence?
2. Which places(i.e. stack address) do they reach?

7.2.1 Definition

To answer these questions we use Taint Analysis. We call a variable or a memory space, such as a buffer containing user input, taint. Usually, we define untrusted input or sensitive data as taint. We define a policy of when and how to propagate(i.e. when does another variable get tainted?). We check for tainted variables or memory space at special instructions such as return instructions, access control calls, etc.

For taint analysis we define:

- **Definition:** A variable or object is called tainted, if its source is untrustworthy, i.e., user input, memory.
- **Propagation:**
    1. If X is tainted and used to derive Y, then Y is tainted.
    2. **Taint operator:** t:X -> t(Y)
    3. t is transitive: X -> t(Y) -> t(Z) → X -> t(Z)

7.2.2 Defining the Algorithm

We want to define an algorithm for taint analysis using the data-flow framework. To do so, we need to define our transfer function.

Consider a definition **d** and a variable **x**. We say that a definition **d** generates a tainted variable **x** if it uses tainted data or if **d** is a taint source. Likewise, we say that a definition **d** kills a tainted variable **x**, if d defines **x** using untainted sources or if **d** validates **x**.

- $gen_d = \{x|$ d defines x and uses tainted data $\vee$ d is a taint source tainting x$\}$
- $kill_d = \{x|$ d:x defined with untainted sources $\vee$ x is validated in d$\}$

We define the algorithm using the data-flow framework:

Abstraction	Value
Domain	Set of Variables
Flow Direction	Forward
Transfer Function	$f_d(x) = gen_d \cup (x - kill_d)$ $f_B(x) = gen_B \cup (x - kill_B)$
Meet Operator	$\cup$
OUT[B]	$f_B(IN[B])$
IN[B]	$\bigcup_{P \in N^-(B)} OUT[P]$
T	{}

The taint analysis algorithm takes a CFG and the sets gen[B] and kill[B] as input to return IN[B] and OUT[B]. Consider the following pseudo-code representation.

```
OUT, IN = dict(), dict()

for B in CFG nodes:
    OUT[B] = set()
    while OUT changes:
        for B in CFG nodes:
            IN[B] = union(OUT[P] for P predecessor of B)
            OUT[B] = gen[B] | (IN[B] - kill[B])
```

Listing 7.7: *Taint Analysis*

A variable **v** is tainted in point **p** if **v** in IN[p]

7.2.3 Taint Analysis on SSA-Representation

The SSA Representation gives the def-use chains implicitly. Since every variable is defined only once, it is easy to check for tainted variables. In an SSA-Representation, if a variable v is tainted at one point p, then v is tainted in the whole program(after being defined of course). This makes finding all usages of tainted variables very easy. Taint analysis is faster on the SSA-Representation. The algorithm continues until each new variable is marked by the previous step, but each definition is only considered once.

7.3 Slicing

Data-flow algorithms often have a prolonged runtime. One of the reasons for that is the number of definitions, that need to be considered. However, not every definition is relevant to a given problem.

Slicing is an algorithm that aims at reducing the number of statements in a program to the number of relevant statements for a given problem. In other words, slicing reduces a program to the number of statements needed for a partial computation.

7.3.1 Definition

If we define a program as a set of statements, we can define any reduced program as a subset of those statements. Consider a reduced program computing variable v. This reduced program contains all the statements that influence v or all the statements that are influenced by v.

A statement s is affected by statement d if there is one of the following:

- **A data dependency:** statement d defines a variable that is used in statement s
- **Control Dependency:** statement d decides whether statement s is executed at all

A reduced program is called a program slice if the following holds:

1. A program slice is a syntactically correct program
2. all variable values at a statement in the slice equal the ones in the program itself

If we determine the statement influencing or being influenced by a variable, we can use the information to perform change impact analysis and achieve better parallelisation and more efficient debugging.

- **Change Impact Analysis:** determines which parts of a program are affected by a change and what should be retested after the change.
- **Parallelisation:** Determine parts of the program that can be computed independently from each other.
- **Debugging:** Focus on parts of a program relevant to a bug. Use slicing to visualise control and data dependencies.

7.3.2 Forward and Backward Slicing

As mentioned earlier, we use slicing to answer two questions:

- Which statements are influenced by a criterion?
- Which statements does a criterion influence?

To answer the first question, we need to examine the statements following the said criterion. Hence we use forward analysis. Conversely, we use backward analysis to determine the statements influencing the criterion. As always, we define the gen and kill function for our data-flow analysis to define the algorithms.

- Forward Slicing:
  - $gen_d = \{\mathrm{x} \mid \mathrm{d\ defines\ x} \wedge \mathrm{d\ uses\ a\ variable\ in\ IN[p]}\} \cup \{\mathrm{v} \mid \mathrm{if\ d} \in \mathrm{S} \wedge \mathrm{v\ marked\ by\ d}\}$
  - $kill_d = \{\mathrm{v} \mid \mathrm{d\ defines\ v}\}$

  Forward Slice = $\{\mathrm{p\ point} \mid \mathrm{p\ uses\ variable\ v} \in \mathrm{IN[p]}\}$
- Backward Analysis:
  - $gen_d = \{\mathrm{v} \mid \mathrm{d\ uses\ v} \wedge \mathrm{d\ defines\ x} \in \mathrm{OUT[p]}\} \cup \{\mathrm{v} \mid \mathrm{if\ d} \in \mathrm{S} \wedge \mathrm{v\ marked\ by\ d}\}$
  - $kill_d = \{\mathrm{v} \mid \mathrm{d\ defines\ v}\}$

  Backward Slice = $\{\mathrm{p\ point} \mid \mathrm{p\ defines\ variable\ v} \in \mathrm{OUT[p]}\}$

Abstraction	Forward Slicing	Forward Slicing
Domain	Set of Variables	Set of Variables
Flow Direction	Forward	Backward
Transfer Function	$f_d(x) = gen_d \cup (x - kill_d)$ $f_B(x) = gen_B \cup (x - kill_B)$	$f_d(x) = gen_d \cup (x - kill_d)$ $f_B(x) = gen_B \cup (x - kill_B)$
Meet Operator	$\bigcup$	$\bigcup$
OUT[B]	$f_B(IN[B])$	$\bigcup_{S \in N^+(B)} IN[S]$
IN[B]	$\bigcup_{P \in N^-(B)} OUT[P]$	$f_B(OUT[B])$
T	{}	{}

7.4 Satisfiability

Satisfiability is another reoccurring problem in Program Analysis. During analysis, it becomes necessary to know if certain conditions can be met. Consider a key checker. A key checker takes an input and checks if the input satisfies a number of conditions. Is there a way to find an input so that the said number of conditions is satisfied?

7.4.1 Definition

A formula, a set of conditions, is satisfiable if it is true under some assignment of the variables. If an assignment of the variables satisfies a formula, then it is called a satisfying assignment for that formula. Satisfiability modulo theory(SMT) is the problem of determining whether a mathematical formula is satisfiable. Note that this problem is very similar to SAT, which contains boolean formulas only. In fact, SMT can be seen as a generalisation of SAT containing arithmetic operations, arrays, functions, etc. While SAT is NP-complete, SMT is NP-hard.

There are no algorithms to solve NP problems efficiently(in polynomial time), but there are ways to solve them.

For example, we use the DPLL-Algorithm for SAT solving. Theoretically, DPLL has exponential runtime, but in practice, it usually runs much faster. The algorithm is described closely under the following link.

DPLL Algorithm

Other examples of solving NP-hard problems include the substitution approach for systems of linear equations or the Fourier-Motzkin Elimination for systems of linear inequalities.

Substitution

Fourier-Motzkin Elimination

7.4.2 Solving for Satisfiability

When performing analysis, we usually need a quick way to achieve results. For the purposes of this script(and by extension the purposes of the lecture) we introduce Z3.

Z3 is a theorem prover from Microsoft Research. It offers a quick way to define and solve different SMT problems. It allows us to work with strings, bit vectors, arrays, quantifiers and other datatypes to define any SMT problem. Z3 allows us to use different levels of precision.

The Z3Py Guide gives a detailed tutorial on using Z3 with python.

When writing Z3 scripts be careful to choose the right type.

7.5 Symbolic and Concolic Execution

Real programs can have an enormous number of paths. With larger numbers of paths, it becomes difficult to use Z3 with handwritten conditions. It becomes necessary to define tools that determine what input leads to the execution of which parts of code. In program analysis, we use symbolic and concolic execution.

7.5.1 Symbolic Execution Definition

Symbolic execution is a way of "executing" a program abstractly using abstract values as input so that one such value covers many concrete values that share a particular execution path.

The goal of symbolic execution is to analyse all paths through a program and find a set of inputs that lead to that execution path.

Consider the following function:

```
int func(int a, int b){
```

```
2       int z = a + b;
3       if (z > 10){
4           funcB();
5       }
6       else {
7           funcA();
8       }
9   }
```

Listing 7.8: *Symbolic Execution*

For the symbolic execution, we would replace a and b with symbolic values, such as α and β. Since z is defined using a and b, we can define the symbolic value of z to be $\alpha + \beta$. Now consider the condition on line 3. We can now say that all pairs of α, β with $\alpha + \beta > 10$ lead to the execution of funcB and all the other pairs lead to the execution of funcA. We can continue to do this with every branching point and collect the conditions for each path. We can then solve for every path using Z3 to find input that leads to it.

Formally we say that Symbolic Execution is a way of executing a program abstractly by assigning each variable/input a symbolic state. At any point the symbolic state is describe as the conjunction of two formulas:

1. σ_S: **Var -> Sym** mapping variables to their symbolic values, called symbolic store,
2. **Path Constraint (pct)**, a logic formula recording all branch constrains taken so far.

7.5.2 Symbolic Execution Limitations

While symbolic execution proves to be an important and powerful tool, it has its limitations. One of the biggest limitations is the path explosion problem. Common contributors to the path explosion problem are:

1. loops and recursion
2. exceptions
3. pointer aliasing
4. concurrency

To see how these factors could influence symbolic execution let's consider loops. Every iteration of a loop introduces another branching point. Consider a for-loop that loops x times. The number of branches created is 2^x. The requirements for the analysis skyrocket.

7.5.3 Concolic Execution Definition

Concolic Execution, also called dynamic execution, is a combination of concrete execution and symbolic execution.

In concolic execution, we start with a random seed input and execute the program while tracking the symbolic execution for the execution path. Then we negate one/ the last conjunction to obtain the constraints for the next run.

Consider the same code snippet again:

```
int func(int a, int b){
    int z = a + b;
    if (z > 10){
        funcB();
    }
    else {
        funcA();
    }
}
```

Listing 7.9: *Concolic Execution*

Concolic execution would generate two random values for a and b and run the program. Naturally, one of the two functions funcA or funcB would get called. In the second iteration we would negate the last conjunction ($z > 10$) using an SMT solver(Z3) and execute the other function.

In other words, we execute the program with concrete values which we change with every execution according to the branching conditions. To adjust the values efficiently we keep the symbolic values. **Con**rete + Symb**olic** = **Concolic** Execution

The main **limitation of concolic execution** is the coverage. It is difficult to predict interesting conjunctions to negate while remaining efficient. Concolic execution usually covers most of the branches around the end of a specific execution path.

7.6 Value Set Analysis

Compilers nowadays are equipped with a lot of tools and algorithms to detect problems during compilation and avoid buggy binaries. A well-known problem that compilers detect is the out-of-Array-Bound problem. We use data-flow algorithms to detect this problem. The applicable approach is called Value-Set-Analysis.

The idea behind Value-Set-Analysis is to compute a (super)-set of possible values for each variable. These sets can be intervals, logical expressions or bounded sets. We use Value-Set-Analysis to detect many distinct problems: Out-of-Bound array access(already mentioned), arithmetic overflows, incorrect memory access (Null pointer), etc.

7.6.1 Definition

A **value set** is a set of possible values associated with a variable at a certain program point **p**. We denote the value set of a variable **v** at program point **p** by **VAL(v,p)**.

Consider the following example:

```
1  int foo(){
2      int a = 30;
3      int b = 9 - a / 5;
4      int c = b * 4;
5      if (c > 10){
6          c = c - 10;
7      }
8       d = bar(c);
9      return d;
10 }
```

This simple program has two possible paths. The first executes line 6, and the second does not. With Value-Set-Analysis we want to answer questions like the following:

- What values can variable c have on line 8?
- What values van variable d have on line 9?

Let us consider a naive approach. To answer the first question, we consider both paths going through **line** 8. If we do not execute **line** 6, then the only possible value of **c** in **line** 8 is **c**=12. If we do execute **line** 6, then the only possible value of **c** in **line** 8 is **c**=2. Note that in this naive approach of value set analysis, we do not evaluate conditions. Therefore the possible values of **c** on **line** 8 are **c** $\in \{2,12\}$.

The value of **d** in **line** 9 depends on the internals of the function bar. Same as with conditions, we do not evaluate functions. Therefore the value of **d** in **line** 9 can be anything. We say d $\in \{-\infty, \infty\}$.

7.6.2 Developing an Algorithm

Note that the given approach in the previous section is neither optimal nor efficient since we are considering invalid paths. In the given example, c is always 12 in line 5, which means that line 6 always executes.

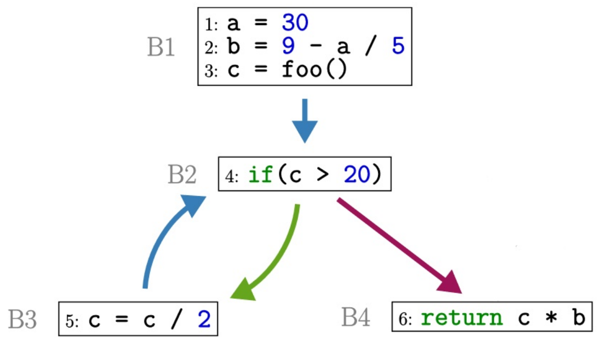
In a better approach, we could use **Symbolic execution** and **Satisfiability** to ensure that all considered paths are valid. However, this approach would inherit the weaknesses of symbolic execution. Using symbolic execution would lead to path explosion or incomplete coverage.

Using Data-Flow Framework

We want to avoid the problems of symbolic execution. To do that, we trade off accuracy for full coverage. We want to accept false positives to get full coverage and avoid path explosion. We use the data-flow framework to define the following algorithm.

Abstraction	Value
Domain	Variables x Value-Sets
Flow Direction	Forward
Transfer Function	$f_d(x) = \bigcup f_{d,v}$ $f_B(x) = \bigcup f_{B,v}$
Meet Operator	$\bigcup$
OUT[B]	$f_B(IN[B])$
IN[B]	$\bigcup_{P \in N^-(B)} OUT[P]$
T	{}
$\perp$	$[-\infty, \infty]$

Consider the following example.



```
a = 30
b = 9 - a / 5
c = foo()
```

```
if(c > 20)
```

```
c = c / 2
```

```
return c * b
```

Figure 7: *Value Set Example Graph*

Block B	OUT[B]^0	IN[B]^1	OUT[B]^1	IN[B]^2	OUT[B]^2
B_1	∅	∅	$\{(a, \{30\}), (b, \{3\}), (c, \perp)\}$	∅	$\{(a, \{30\}), (b, \{3\}), (c, \perp)\}$
B_2	∅	$\{(a, \{30\}), (b, \{3\}), (c, \perp)\}$	$\{(a, \{30\}), (b, \{3\}), (c, \perp)\}$	$\{(a, \{30\}), (b, \{3\}), (c, \perp)\}$	$\{(a, \{30\}), (b, \{3\}), (c, \perp)\}$
B_3	∅	$\{(a, \{30\}), (b, \{3\}), (c, (20, \infty]\}$	$\{(a, \{30\}), (b, \{3\}), (c, (20, \infty]\}$	$\{(a, \{30\}), (b, \{3\}), (c, (20, \infty)\}$	$\{(a, \{30\}), (b, \{3\}), (c, (10, \infty)\}$
B_4	∅	$\{(a, \{30\}), (b, \{3\}), (c, [-\infty, 20])\}$	$\{(a, \{30\}), (b, \{3\}), (c, [-\infty, 20]\}$	$\{(a, \{30\}), (b, \{3\}), (c, [-\infty, 20))\}$	$\{(a, \{30\}), (b, \{3\}), (c, [-\infty, 20)\}$

Figure 8: *Value Set Example Table*

To better understand the transfer function, check the transition for B2 between $IN[B]^2$ and $OUT[B]^2$.

Also, note that despite it is not shown in the data-flow framework, we do apply constraints when computing the $IN[B]$ sets. Check $IN[B]^1$ for B3, the value-set of c.

Unfortunately, even though we now have an effective algorithm, we cannot ensure that this algorithm terminates. Consider the following slightly different example.

In this example, we have no information about the value of c. According to our algorithm, we have to consider $c \in [-\infty, \infty]$. The value of i gets changed in every iteration. This would lead to a semi-lattice of infinite height. Therefore the algorithm does not terminate. (The value-set for b on B2 changes with every iteration, from $\{3\}$ to $[3, \infty]$).

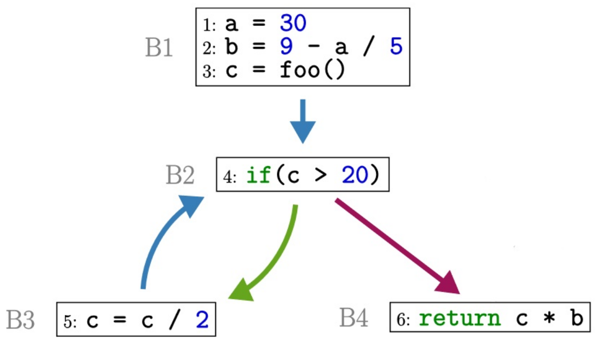


Figure 9: *Value Set Example 2 Graph*

To avoid this problem, we redefine the widening operation. The sole purpose of using a modified widening operation is to ensure that the algorithm terminates.

We define $[a,b] \sqcup [x,y]$ to be:

- $[a,b]$ -> if a $\leq$ x $\land$ y $\leq$ b
- $[a,\infty]$ -> if a $\leq$ x $\land$ y $>$ b
- $[-\infty,b]$ -> if a $>$ x $\land$ y $\leq$ b
- $[-\infty,\infty]$ otherwise

In value set analysis, we attempt to identify a tight over-approximation. However, this leads to many false positives in our results. This is a major drawback of using the data-flow framework with value-set analysis. Another drawback is the information that gets lost by the join-operation.

8 Binary Analysis 2.0

The number of programming languages keeps growing. While many of the new languages find little application, some prove to be appealing and full of features. In binary analysis, we take a special interest in the characteristics of binaries compiled from different programming languages. In this chapter, we take a look at C++, Go, Rust, Objective-C and some Compiler Idioms.

8.1 Application Binary Interface

The Application Binary Interface(ABI) is an interface between two binary programs. One of the programs is often a library or part of an operating system, while the other is a program run by the user. The ABI is hardware dependent. It specifies how data structures or computation routines are accessed at machine code level.

The duty of complying with the ABI falls on compilers, operating systems and library authors. However, understanding how different ABIs work is necessary for many different fields, such as Program Analysis, Software Development(especially when several programming languages are involved), etc.

These are some of the characteristics specified by an ABI:

- Calling Convention
- Processor Instruction Set
- Details on basic Data Types
- System Call access
- File Hierarchy Layout

Some well known ABIs are: Intel Binary Compatibility Standard, System V ABI, Itanium C++ ABI.

8.2 C++ Programming Language

C++ is a general-purpose, object-oriented programming language. It first appeared in 1985 as an extension of the C programming language with support for classes and objects. The development of the language has continued and made major expansions in features since the first version. C++ is designed for embedded and system programming. It is well-suited for programming large systems and resource-constrained software. C++ is very performant, efficient and flexible. C++ is a superset of C, meaning that every C program can be compiled using a c++ compiler, but not every C++ program is understood by C compilers. We want to take a look at C++-specific features and their implementation.

8.2.1 Overloading

C++ allows the programmer to specify more than one definition for a function name or an operator in the same scope. We say that a new definition of a symbol overloads the first if the two have different implementations(i.e. the second definition of a function uses different parameters). The compiler determines the most appropriate definition to be used by comparing the argument types used. This process is called **overload resolution**. Consider the following code snippet.

```
void print(int value){
    printf("%d\n", value);
}
void print(char* value){
    printf("%s\n", value);
}
void print(int value){
    printf("%d\n\n", value);
}
int main(){
    int i = 0;
    char test[] = "test";
    print(i);
    print(test);
}
```

Listing 8.1: *Overloading*

The definition of print, in line 1, is the first definition that the compiler reads. The second definition, in line 4, has the same symbol but different arguments compared to the first. This leads to overloading. The third definition, in line 7, has the same symbol and arguments as the first one. This leads to an error during compilation.

8.2.2 Name Mangling

The compiler needs to distinguish affected functions and operators to support separation by scope and overloading. In C++, this is achieved through the name mangling process.

In C++, we can define namespaces(similar to packages in Java), classes, and methods to define scopes. The compiler has to use this information to distinguish each method and operator.

Every mangled name starts with the prefix "_Z". This prefix is followed by "N*X*", where *X* is the length of the Namespaces identifier, and the namespace identifier.

Example: For namespace "example" the first part of the mangled name would be: "_ZN7example"

The compiler adds the name of the class with its length, as a prefix, to the first part. This is followed by the length of the function name and the name itself. The constructor then adds the prefix "E" and the short version of the types of the parameter.

Example:

- Namespace -> "example"
- Class -> "Test"
- Function -> foo(int i, int j)
- Output -> "_ZN7example4Test3fooEii" is the mangled name

Here is a detailed explanation.

8.2.3 New and Delete

The new() operator in C++ is a request for memory allocation of a given size on the heap. If the request is granted by the OS, then the operator initialises the memory with the given type(primitive or not) and returns a pointer to the memory address. This operator uses malloc() internally, but it manages the size of the allocation automatically.

The delete() operator is the counterpart of new(). It takes a pointer as a parameter and uses free() to deallocate the memory space.

8.2.4 Classes

Classes are an expanded version of data structures. Classes serve as blueprints for objects. Classes can inherit from(or extend) other classes for better code reuse. They contain data and methods, both of which will be called members for the purposes of this script. These members are separated into instance members and class members. In this section, we want to take a look at the memory mapping of objects.

We declare **class variables** using the "static" keyword in the definition of a class. These variables are shared among the instances(objects) of the same class. They are saved in the .data or the .bss section. Variables that are not class variables are **instance variables**. Each instance has its copy of the variable. They are saved in the heap or on the stack in the same order as they are declared in the class definition.

Class and instance methods are both stored in the .text section. The instance knows of instance methods but not of class methods. Class methods are called using the class name (rather than the usual instance name). They have no direct access to instance variables or the keyword **this**. Instances of a class access the instance methods through the **vTables**. Instance methods often use the "this"-parameter. This is implemented by simply passing a pointer to the instance to the instance method called.

Virtual functions are instance members whose behaviour can be overridden in the derived class. The dynamic type of an instance decides which implementation of the function is called during runtime.

8.2.5 Decompilers vs C++

The most notable differences between the decompilation of a C binary and a C++ binary are that C++ uses Objects and Streams. To better understand C++ binaries, we have to look at how objects are instantiated, how instance variables and methods are accessed and how streams work on assembly level.

Working with Objects

C++ binaries instantiate objects using the new() operator followed by the constructor of the class. The new() operator returns a pointer to the allocated memory space. The constructer fills the memory space with a reference to the vTable and the instance variables.

After the initialisation, the objects appear as structs(in C). The variables of an object are often accessed using the pointer to the object with the offset to the variable. The same goes for the functions on the vTable. Note that static variables or methods do **not** share this connection.

Streams

In C++, a stream is a flow of data into or out of the process. For example "cout" and "cin" are streams used to write or read data. There are different categories of streams(detailed explanation). The list below denotes some of the frequently used streams:

- **istream:** General purpose input stream, such as cin.
- **ostream:** General purpose output stream, such as cout, cerr.
- **ifstream:** Input file stream. It is a version of istream that reads from a file.
- **ofstream:** Output file stream. It is a version of ostream that writes to a file.

8.3 Go Programming Language

Go is a relatively new programming language developed by Google. It first appeared in 2009. Go was created with secure and scalable systems in mind. It offers many features like Memory Safety, Garbage Collection, Structural typing and CSP-style concurrency.

8.3.1 Compilation

Go binaries tend to get large, even if the program they execute is trivial. This is a result of the linker in the generic compiler(GC) toolchain. Per default, the linker creates statically linked binaries that include the Go-Runtime along with other feature-related code, such as support for dynamic type checks, reflection, panic-time stack traces, etc.

A simple Hello World program in Go takes roughly 1.9MB with around 2051 functions as it automatically includes all features listed above. Go allows the programmer to call C code, in which case the toolchain creates a dynamically linked binary(only the C part is dynamic).

8.3.2 Goroutines

Goroutines are lightweight execution threads. They execute concurrently with each other and the rest of the program. Compared to traditional threads (like threads in C, for example), goroutines are cheaper to create and faster to start up.

Per default, each goroutine takes 2048 bytes of stack space to start. If needed, then the size of the stack space is increased dynamically. As a result, goroutines always start by checking if more stack is needed. This makes it practical to create hundreds of goroutines in the same address space. A goroutine is created by using "go function()" which translates into "runtime.newproc(function)" after compilation.

8.3.3 Go Calling Convention

The calling convention is part of the ABI. It defines how the caller passes parameters to the callee and how the callee returns results. Programming languages implement different calling conventions. Before version 1.17, Go used a stack-based calling convention. Arguments were passed on the stack and only moved to registers when needed.

Version 1.17 of Go released the register-based calling convention for better performance. Go uses the following registers in order: RAX, RBX, RCX, RDI, RSI, R8, R9, R10, R11.

Functions in Go that include exceptions also preserve a copy of the frame pointer to unwind the stack if an exception was to occur.

Detailed explanation before 1.17

Explanation for after 1.17

8.3.4 Go functions and characteristics

This section is about some of the characteristics of the Go programming language. We are going to take a look at strings, channels, Duff's devices, etc.

Strings

Other than most traditional programming languages, Go does not use Null-terminated strings. Instead, it uses a structure to define strings. The structure StringHeader for Strings in Go looks as follows:

```
struct StringHeader data_address = {
    char* data = address_beginning_of_string;
    int64_t length = length_of_string
}
```

As a result, strings in Go are usually packed without separators next to each other.

Defer

The defer statement in Go defers(delays) the execution of a function until the calling(surrounding) function returns. When defer is called, then the marked function is added to a LIFO List. At the end of the current surrounding function, all the deferred functions are executed in Last In, First out order. Consider the following function:

```
import (
    "fmt"
    "os"
)
func example() {
    f1 := createFile("/tmp/file1.txt")
    f2 := createFile("/tmp/file2.txt")
    defer closeFile(f)
    defer closeFile(f2)
    fmt.Fprintln(f1, "data")
    fmt.Fprintln(f2, "more data")
}
```

Note that we have two defer statements in this example. Both will execute when the example function returns. The second deferred statement will execute first, according to the LIFO order, and then the first will also execute. Using defer is quite similar to using finally in java or python.

Here is one in-depth explanation of how defer works.

Channels

Channels are an important aspect of the Go programming language. They offer a way of communication between Goroutines. They can be thought of as bidirectional pipes connecting two goroutines.

Reading and writing from channels are blocking operations by default. This allows for easier synchronisation since there is no need for locks or similar concepts.

Go uses the function makechan from the runtime package to create a new channel. This function takes two parameters, one of type int and the other of type chanType, which is given below.

```
type rtype struct {
    size uintptr
    ptrdata uintptr
    hash uint32
    tflag tflag
    align uint8
    fieldAlign uint8
    kind uint8
    equal func(Pointer, Pointer) bool
    gcdata *byte
    str nameOff
    ptrToThis typeOff
}
```

```
type chanType struct {
    rtype
    elem *rtype
    dir uintptr
}
```

To send and receive from channels Go uses the following functions from the runtime package:

- func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool
- func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool)
- chansend1 as a wrapper for chansend
- chansend2 as a wrapper for chansend
- chanrecv1 as a wrapper for chanrecv
- chanrecv2 as a wrapper for chanrecv

You can find the implementation here.

Unrolling and Tail Calls

Loop unrolling, also known as loop unwinding, is a space-time tradeoff(most of the time) and optimisation technique that optimises execution speed at the expense of its binary size. Conditionals and branching are expensive for performance. Consider the following example:

```c
int length(int vector[3]) {
    int sum = 0;
    for(int i = 0; i < 3; i++)
        sum += vector[i] * vector[i];
    return sqrt(sum);
}
```

Without optimisation, the compiler would update the variable i and check the condition before using the imul instruction to compute the result in each of the three iterations. Writing three consecutive imul instructions is much more efficient.

A tail call is a function call performed at the end of a given function. Consider the example above where we have a tail call to the sqrt(int i) function. Without optimisation, the compiler would allocate a new stackframe above the current one to execute the called function. However, since the calling function will return after this call, the compiler can avoid allocating the frame. Instead, it uses the existing top frame and simply jumps to the function. When the sqrt function returns, it returns back to the function calling the length function from our example.

Duff's Device

A Duff's device is a way of implementing loop unrolling manually. Tom Duff's problem was copying 16-bit unsigned integers from one array to another. The naive implementation is to copy the data in a loop, copying one 16-bit integer per iteration. As already mentioned in the previous section, this approach is expensive performance-wise.

Since most of the data are divisible by 8 in size, it is more efficient to copy 8 elements per iteration. Copying more data per iteration makes the copying process faster since we run less code to maintain the loop.

The problem with copying units bigger than one(in this case, 8) is that not all datasets can be divided into units of the given size. To avoid reading out-of-bound and unnecessary alignments, Duff implemented a solution to jump into the loop for the first iteration. Consider the following code:

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = count / 8;
    do {
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
    } while (--n > 0);
}
```

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
        } while (--n > 0);
    }
}
```

The function on the left works correctly only if the size of the array is divisible by 8. The one on the right works correctly, regardless of the size. Note that we assume that the size is not zero.

Go implements the functions duffzero and duffcopy from the runtime package using the same principle. Instead of using a do-while loop with a switch statement, the functions have many different entry points. The toolchain decides during compilation what entry point to choose. The function duffzero zeros out an array, while duffcopy copies data.

8.4 Other Programming Languages

In this section, we are going to take a look at other programming languages and their characteristics in a short overview. The languages presented in this section are Rust, Objective-C and Swift.

8.4.1 Rust

Rust is a high-level, general-purpose programming language. It supports multiple programming paradigms. Rust offers type safety, performance enhancement and memory safety. Rust is a relatively new programming language having first appeared in 2015.

Types

Data types look similar in most programming languages. However, every language implements and uses some types differently. Rust offers primitive and advanced(compound) data types. Rust offers the following primitive data types:

- **Signed Integers** - i8, i16, i32, i64, i128 and isize (pointer size) where i stands for signed integer and it is followed by the number of bits that the type occupies.
- **Unsigned Integers** - u8, u16, u32, u64, u128 and usize (pointer size) where u stands for unsigned integer and it is followed by the number of bits that the type occupies.
- **Floating Point** - f32, f64 where f stands for float followed by the number of bits occupied.
- **char** - Unicode characters(not ascii!!!) each takes 4 Bytes.
- **bool** - boolean value for either true or false.
- **Unit type** - denoted by (), the only possible value for this type is an empty tuple.

These are some of the advanced types that Rust offers:

- **&str** - String Slices. A string slice is immutable and has a fixed size. It is implemented as a sequence of UTF-8 bytes. It s not Null-terminated and can only be accessed through a reference (&str) which contains information about the size.
- **String** - Strings allocated on the heap. Also implemented as a UTF-8 sequence. This data-type is mutable.
- **Tuple** - A tuple is a list of elements of various types grouped in a compound type. Per default, tuples are immutable in Rust.
- **Array** - An array is a list of elements of the same type grouped in a compound type. Per default, arrays are immutable in Rust.
- **Struct** - Like tuples, structs hold a list of elements of various types. Unlike in tuples, each element takes a name, by which it can be accessed.

- **Enum** - Enums give the developer the possibility to define types that contain different variants. Each variant has the same size(padded if not) and the necessary definition is chosen from the set during runtime.

Memory Safety

One of the biggest promises that Rust carries as a programming language is memory safety. Rust does not allow the user to write unsafe code unless the programmer explicitly requests Rust to do so. The programmer can write unsafe code using the **unsafe** keyword.

Rust introduces the following principles to keep the development memory-safe.

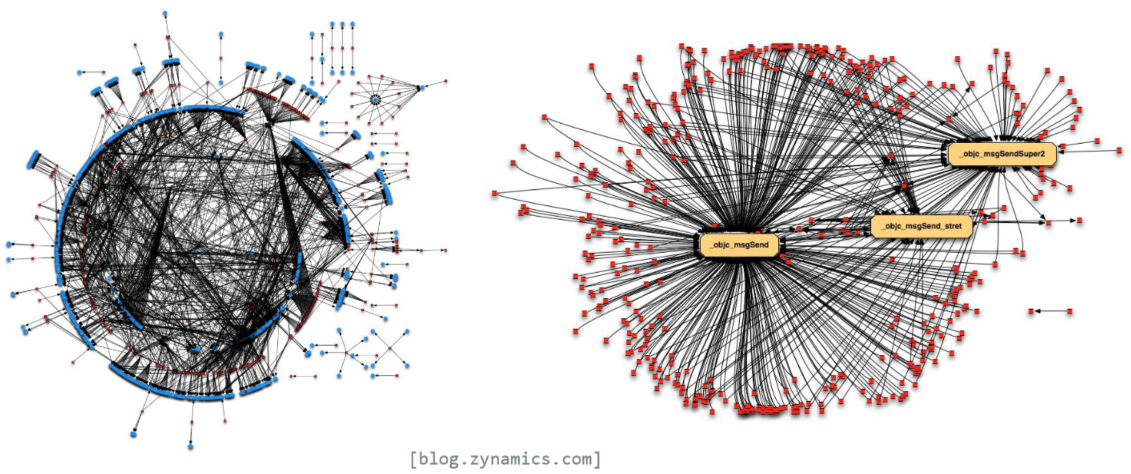
- **Mutability** - By default, all variables are immutable unless explicitly declared as mutable. Furthermore, each variable can only have one reference pointing at it at a time.
- **Ownership** - Rust introduces the concept of ownership for variables to avoid leaks and other problems during runtime. Ownership can be thought of as safety encapsulation.
- **Borrowing** - With the concept of ownership, it becomes necessary to access variables without taking their ownership. The borrow checker ensures that the access is legal.
- **Bounds checking** - Verifying that code accesses are within the boundaries of an array or assigned memory location.

8.4.2 Objective-C and Swift

Objective-C and Swift are both languages used heavily in IOS App development. They both offer interesting characteristics. Objective-C first appeared in 1984 while Swift came 30 years later in 2014.

Objective-C is a superset of C that adds object-oriented capabilities and a dynamic runtime. While the primitive types and the syntax for control flow remain the same as in C, Objective-C adds syntax for defining Classes and methods. Objective-C also adds dynamic typing and binding.

The idea behind Objective-C is to defer as many decisions as possible and remain as dynamic as possible. To achieve that Objective-C leaves the runtime to decide which object or method should be called. This results in the following call-graph.



[blog.zynamics.com]

Figure 10: *C Call Graph vs Objective-C Call Graph*

Notice that most function calls use the runtime(the three functions in orange) as an intermediator to call other functions. From the perspective of binary analysis, this makes locating interesting functions, such as functions that get called very often, more difficult. Since everything is dynamic, it is challenging to analyse the behaviour of the binary through static analysis alone.

In the following piece of code, we want to create an instance of Person and give it the name "Chicnselad" in Objective-C. The piece of code is followed by the internal interpretation of Objective-C.

```
Person *person = [[Person alloc] init];
[person setName:@"Johnny"];
objc_msgSend(
    objc_msgSend(
        objc_msgSend(
            objc_getClass(@"Person"),
            NSSelectorFromString(@"alloc")
        ),
        NSSelectorFromString(@"init")
    ),
NSSelectorFromString(@"setName:"),
@"Johnny");
```

Listing 8.2: *Objective-C COde*

Swift is very similar to Objective-C but it does have some differences. The most notable ones are listed below:

- **Type Inference** - Swift supports type inference, which allows the compiler to infer the type of a variable based on its value.
- **Polymorphism** - Protocols in Swift enable the programmer to write code that applies to any type that conforms to the protocol. Objective-C allows for functionality o be extended but only by specific types.
- **Type Safety** - Swift assigns variables to e new copy of data whenever they are modified. This prevents issues like aliasing.
- **Dynamically Typed** - Objective-C is a dynamically typed language; hence variables can hold values of any type. Swift's variables can only hold values of a specific type.

9 Binary Analysis for Android

Android is one of the most popular platforms for mobile devices. Android controls roughly 71% of the mobile OS market. Given this dominance on the mobile OS market, it is not surprising that the number of applications developed for the platform is growing rapidly. Android is based on Linux, but the structure of applications usually differs from that of a classical Linux program. Furthermore, mobile devices carry a lot of personal data and information. Therefore the analysis of programs that run on these devices carries huge importance.

9.1 Fundamentals

Android applications are usually packed into Android Packages. The Android Package (APK) is the standard file format used for the distribution and installation of mobile applications in android. APKs use Java or Kotlin as a base programming language. An installed APK running on a device is called an activity. Through its lifecycle, an activity has access to runtime components and resources. This section contains an overview of the standard content found in an APK, the components of an application and the lifecycle of an activity.

9.1.1 The Android Package

An APK is a ZIP archive that contains the necessary resources for an application to e installed and ran on a device. The following listing contains some of the most important files and directories found in an APK file.

- **META-INF Directory** - This directory contains different files with meta-data. For example, the meta-data may contain information about the packages used during the development of the APK, certificates, etc.
- **lib Directory** - The lib directory contains the native libraries. More formally, it contains code that is processor specific. To support different architectures, this code is present

for every supported architecture. The most popular architectures for Android native libraries are ARMv8, ARMv7, x86_64, x86, and MIPS.

- **The res Directory** - This directory contains files that are included or referenced by the application. They are easily accessed by the R instance in the android runtime.
- **assets Directory** - This directory contains files in their raw format. Resources packed here are accessible as raw bytes. The programmer needs to use the AssetManager to read them as a stream of bytes.
- **AndroidManifest.xml** - The AndroidManifest.xml file gives us a lot of information on an application. It defines which version of the android SDK it uses, what permissions it has, what is the lowest supported version of Android, where the application starts, etc(more on this later). **classes.dex** - This file contains the java classes compiled in the dex format. This format is understandable for the Dalvik Virtual Machine. **resources.arsc** - This file contains pre-compiled resources for the application.

9.1.2 Android Stack

Android devices(hardware) are very diverse. To support this wide array of devices android introduces the Android Stack. The android stack builds a uniform interface for applications through step-by-step abstraction. These are the different levels on the android stack:

1. **The Linux Kernel** - Provides low-level functionality such as memory management, threading, etc. Optionally a security layer is added at this level, such as Samsungs KNOX.
2. **HAL** - The Hardware Abstraction Layer provides a standard interface according to the capabilities of the hardware for the higher levels such as the Java API.
3. **Android RT and Native Libraries** - Each App runs in its own android runtime instance. The Android Runtime(ART) is designed to run multiple lightweight virtual machines.

   Many core functionalities of the android system are written in C or C++. These libraries are called native libraries. An example is the OpenGL library.
4. **Framework** - The feature set of the android system is available to the applications running on it through APIs written in Java. This includes a view system, resource management, notification management, etc.
5. **Apps** - The applications use the framework to implement diverse functionality. The android system usually comes with applications to support core functionality such

as keyboard, SMS, Dialing, etc. Applications installed by the user can replace those provided by the system (i.e. Gboard over the standard keyboard).

9.1.3 App Components

The android applications consist of essential building blocks. These blocks are called App Components. The App Components are categorised into four categories: Activities, Services, Content Providers and Broadcast Receivers.

Activities

Activities provide a window for the application to draw its UI. Activities are implemented as subclasses of the Activity class. The activities serve as entry points for an applications interaction with the user. An activity facilitates different interactions between the user, application and system. Some examples of these interactions are listed below:

- The system keeps track of running activities (what is shown on the screen for example) to ensure that the application the user is using keeps running.
- Keeping track of activities stopped, or processes recently running in general, to prioritise keeping processes that may be needed again alive.
- Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. For example, when sharing a video, users usually follow a certain trail of events.

The lifecycle of an activity on an android device depends on the system and the user. The lifecycle is described in the following diagram(11).

Services

A service is a general-purpose process that runs in the background. It does not provide any UI. A service is a process that performs long-running operations (such as syncing) or work for remote processes(like fetching data from the network). When talking about android, we distinguish between two categories: Started Services and Bound Services.

Started services are started by the user or by a process and tell the system to keep a task running until the work is done. An example is playing music in the background, downloading a file or syncing data.

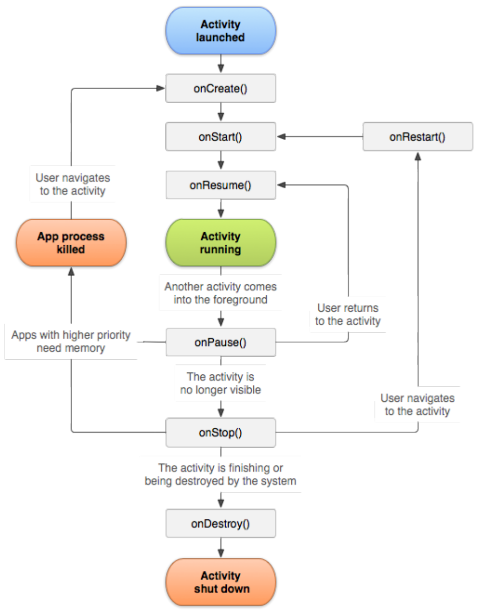

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 11: *The lifecycle of an activity*

A bound service runs because some application or the system itself needs the services provided. These services basically provide APIs for the applications and the system to use. The system keeps track of the dependencies between the processes and services.

Content Providers

A content provider manages a shared set of application data that can be found in the filesystem. The content provider makes the data accessible for other applications. It also decides if the applications that access the data can modify it. For example, the android system provides a content provider for the contacts saved in the device.

Broadcast Receivers

A broadcast receiver is a component that enables the application to receive broadcast messages from the system outside of its regular flow. This allows the app to react to events and notifications. The system can deliver broadcasts to apps even if they are not running. For example, the user can schedule an alarm to go off, but there is no need for the Alarm application to keep running until the alarm rings.

9.1.4 Inter-Process Communication

The android system supports different Inter-Process Communication (IPC) mechanisms. These mechanisms include intents, inter-process method invocation(AIDL: Android Interface), Binder and the Android shared memory.

- **Intents** - An intent is an abstract description of an operation that needs to be performed. For example, the intent startActivity is used to start an activity. The launcher uses startActivity intents to start applications on click.
- **Android Interface Definition Language(AIDL)** - Provides a way of defining programming interfaces that can be passed from one process to another by the system. This usually requires objects to be parsed into primitives.
- **Binder** - The binder class is the core of a lightweight remote procedure call mechanism. It serves as a base class for a remotable object.
- **Android Shared Memory** - The android shared memory allows the creation of memory regions that can be accessed by multiple processes at once.

The AIDL implements the base class Binder and is used to implement Intents. An intent is a messaging object to request an action from another app component. We distinguish between two types. An explicit intent targets an explicit component. An implicit intent specifies the action but not the component to perform that action. Android developers use intents to start activities and services or deliver a broadcast message.

9.1.5 The Android Manifest

Every valid APK must contain a file named AndroidManisfest.xml at the top of its hierarchy. This file gives essential information about the APK to the build tools, android operating system and Google Play.

Among other information, the XML file is required to declare the following information:

- A list of the components of the application, declaring all activities, content providers, broadcast receivers and services.
- A list of the permissions needed by the application to access protected parts of the android system or other Apps.
- A list of the requirements, software and hardware.

The AndroidManifest.xml gives information on the packages and SDK used for the development, the set of features and permissions used and the exported activities(these are activities accessible to any other app).

9.2 Android Bytecode

The android runtime (ART) is responsible for executing the code of an application. The android runtime distinguishes between three categories of code: DEX Code, JIT-Code and AOT-Code. This section takes a look at each of the categories.

9.2.1 DEX Code

The Dalvik EXecutable is the format used by the android OS to store code for the Dalvik Virtual Machine. It contains information about the classes, types and methods used. The full explanation of the format is given under this link.

From an analysis perspective, one of the most important aspects of the format is the **class_def_item** structure. This structure gives information about the Java/Kotlin classes defined and used in the given application. This structure contains a pointer to **class_data_item**. As the name gives away, this structure contains the data of a given class. Among this data, we find a pointer to **encoded_methods**. Each method of the class is described in this structure, and a pointer points to the actual code of the method, given in a **code_item**. We can use this information to better analyse android applications.

9.2.2 AOT and JIT Code

Running code in a Dalvik Virtual Machine has performance drawbacks. To avoid these drawbacks, android introduces the complementary Just-In-Time(JIT) and Ahead-Of-Time(AOT) compilation. The JIT compiler translates the Dalvik code into machine code dynamically. As the execution of the application progresses more and more of the code will be translated.

The AOT translation is a one-time event that happens during the installation of the application. Frequently used pieces of code are translated into machine code during installation.

To support both JIT and AOT compilation android requires profiles for the applications. These profiles mark the important and frequently used parts of code for AOT or JIT compilation.

The AOT compiler creates the following three types of files during compilation:

- ***.art** - These files are not necessarily included. They contain a ART internal representation of some strings and classes listed in the APK.
- ***.odex** - These files contain the translated code for the methods of the APK.
- ***.vdex** - These files contain the DEX representation of the translated code with some additional metadata.

9.2.3 ODEX and VDEX Format

The ODEX and VDEX data are packed inside an ELF file. This file follows the standard ELF format. The dynamic symbols table contains three entries relevant to the AOT compilation:

- **OAT data section** - Read-Only data on the classes
- **OAT exec section** - Executable Machine Code
- **oatlastword** - End of oAT sections

Trivia: OAT, meaning "Of Ahead Time", is a wordplay for AOT "Ahead of Time". The developers liked cereal better than Attack on Titan.

Consider the following overview given below(12. The OAT data section contains **OAT-DexFiles** (ro data) that point to **DexFile**s(ro data) and **OatClass**es(ro data). The **Oat-Class**es point to **OatMethod**s(ro data) which themselves point at the machine code of the method(executable).The DexFile entries are Vdex files that contain the Dalvik Bytecode that corresponds to the translated machine code along with some metadata. This allows us to read the DEX Bytecode even after translation.

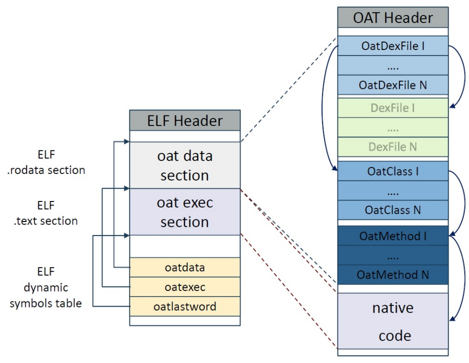


Figure 12: *ODEX and VDEX*

9.2.4 SMALI

Smali is a representation of DEX bytecode that can be thought of as the equivalent of assembly for C programs. To be precise, SMALI is an intermediate representation between Source Code and DEX Bytecode. This section gives an overview of Smali.

Types

Smali understands several types. Each type has its descriptor given by one character. The listing below gives the types and their descriptors.

- **Void** - V -> can only be used for return value types
- **Boolean** - Z
- **Byte** - B
- **Short** - S
- **Char** - C
- **Float** - F
- **Int** - I
- **Long** - J
- **Double** - D

To indicate arrays of each given type Smali adds a "[" before the type descriptor. So the type descriptor for **int**[][] would be [[**i**.

Classes and Methods in Smali

The name of classes is given by the full namespace separated by slashes, prefixed by "L" and suffixed by ";". For example, the class StringBuffer from the package java.lang becomes "Ljava/lang/StringBuffer;".

Nested classes are accessible by using $ as a separator. Consider two classes OuterClass and InnerClass, where the second is nested inside the first. The InnerClass is accessible through "name/space/OuterClass$InnerClass".

If it is an anonymous or private inner class, then Smali gives a unique decimal identifier instead of a name. In our example "name/space/OuterClass$42". The same is true for fields.

Smali indicates methods by the keyword ".method". This keyword is followed by the access modifier and the name of the method. Next to the name, inside round brackets, Smali indicates the type of the parameters by the type descriptor. Smali adds the type descriptor of the return value at the end. For example, ".method private someMethod([BCI[[C)I" is the Smali representation of "int private someMethod(byte[] a, char b, int c, char[][] d)".

Variables and Operations

Smali does not use registers. Instead, it adds a layer of abstraction and distinguishes between parameters and local variables. In general, a Smali statement consists of an operation followed by the operands needed for the operation. The operands can be parameters or local variables.

Parameters are denoted by p followed by a decimal number. Variables are denoted by v followed by a decimal number. The Smali code of a method declares the number of local variables at the beginning of the method by using ".locals <num>" and the number of parameters by "<num> parameter". Note that the first parameter refers to the same object as the "this" keyword.

Smali has numerous opcodes. A list of these opcodes along with their explanation is given here.

9.2.5 Native Libraries

Android allows developers to use native libraries through Android NDK (Native Development Kit). The Android NDK allows for source code to be compiled into an ELF file that is compatible with running in an app. The native code is usually written in Assembly, C/C++ or Rust and is compiled to the target architecture.

The NDK uses the Bionic C library which is lighter than the GNU C library. This library is not POSIX complient and does not support System V IPCs but has better access to android system properties. Android can make use of native code through the Java Native Interface(JNI). This interface allows for methods to be declared in java using the "native" keyword and implemented in other languages.

The compiled native libraries are stored under the lib directory in the APK. They are ELF library files. Each of these libraries contains the JNI_onLoad() method. The developer imports these libraries using the System.loadLibrary() or System.load() method.

The JNI is responsible for registering the native methods and making them accessible during runtime. How these methods are registered depends on whether they are linked statically or dynamically.

- **Dynamic Linking** - The function names and signatures in the library file match the names and signatures declared in the java code.

  They follow the format: *Java_<magled class name>_<mangled method name>*

  The first parameter of these methods needs to be a pointer to a JNIEnv object. This object is a singleton.
- **Static Linking** - Static linking is done manually during JNI_OnLoad. The methods are registered using the RegisterNatives method.

  Following format: *jint RegisterNatives(JNIEnv *env, jclass clazz, const JNINativeMethod *methods, jint nMethods);*

  Requires string of the method name and the string of the method signature

The JNI uses slightly different types. Please consider the table below:

Java Type	Native Type	Descriptor	Description
Boolean	jBoolean	Z	unsigned 8 bits
Byte	jByte	B	signed 8 bits
Char	jChar	C	unsigned 16 bits
Short	jShort	S	signed 16 bits
Int	jInt	I	signed 32 bits
Long	jLong	J	signed 64 bits
Float	jFloat	F	32bit float
Double	jDouble	D	64 bit double

These types are defined in the jni.h Header.

9.3 Android Runtime

The Android Runtime(ART) is the application runtime environment in the Android OS. It is one of the core building blocks of the android ecosystem. Understanding the android runtime is important for dynamic analysis on android. This section takes a look at the Android Runtime and some important concepts.

9.3.1 Initialisation

When an android device turns on it starts the bootloader. The bootloader loads the kernel in a section of the memory that is the kernel space. After the kernel is fully loaded, it starts the first process in the user space. This is the Init process, and it always has PID = 1. The Init process initialises the native deamons, service manager and media server. The Init process also initialises an ART VM process, known as Zygote. Consider the picture below.

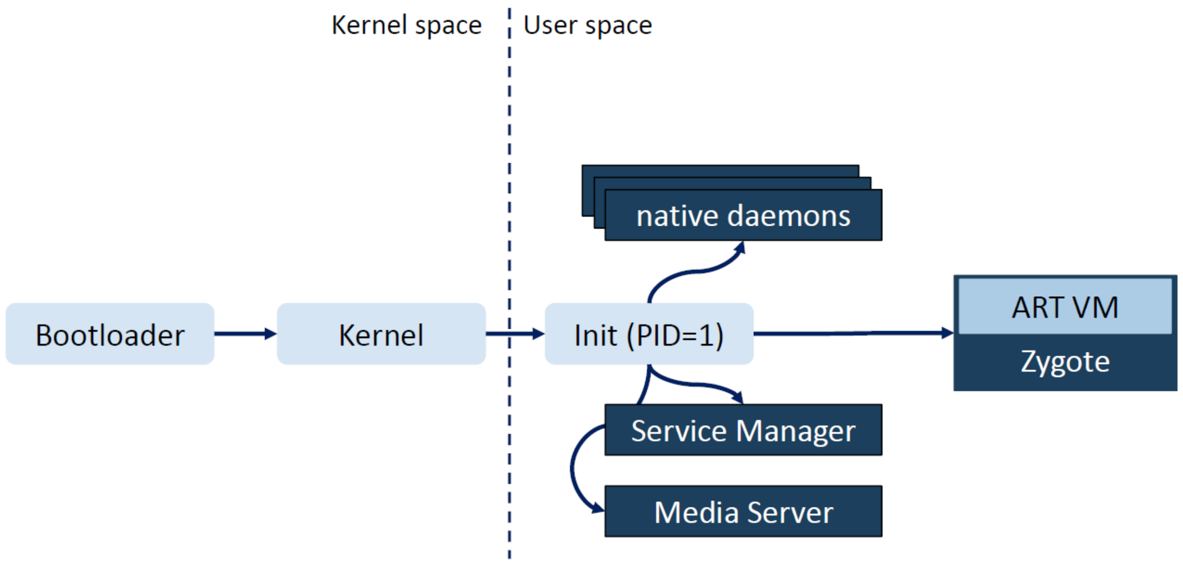


Figure 13: *ART Initialisation*

The Zygote forks itself to create the System Server process, which initialises(or forks into) several Services and Managers. Every user-space application that starts in android is initialised by forking from the Zygote process. This means that every application contains the android runtime library. In the mapping of each application, we find app_process64. This is the Zygote. The different name comes from /system/bin/app_process64 which starts the Zygote.

9.3.2 The Zygote

The Zygote process starts with a minimal C++ script that is responsible for loading the android runtime (libart.so) into the memory of the process. After the runtime is loaded, it takes control of the process. The ART loads the precompiled Java libraries (as boot.oat) and the native libraries. After everything is loaded into the memory of the newly created process, it is the duty of the ART to link the different libraries(oat, native, java). In the end the ART starts the execution with the entry point of the application.

9.4 Analysis Techniques

The analysis techniques of the previous lectures work on android as well. However, they are some differences in the way they are conducted. In this section, we are going to take a look at static and dynamic analysis for android and introduce instrumentation as a technique of dynamic analysis.

9.4.1 Static Analysis

When conducting static analysis on android programs, it is important to consider all parts of the program. An android program consists of Java/Kotlin Code, OAT Code and native libraries. As discussed earlier, the OAT Code contains the DEX representation as ro data. This makes it possible for us to treat Java/Kotlin code and OAT code the same, as we can decompile both. The rather rigid structure of Java usually leads to qualitative decompilation compared to C programs.

The native libraries are written in C, C++, Rust or other low-level programming languages. Analysing these libraries requires tools that support these languages. Furthermore, we need to consider how native libraries are linked. For static linking, we need to consider the JNI_OnLoad method to find the methods we need to analyse. Dynamic linking requires an understanding of the name-mangling process.

It is also important to keep in mind that the processes run inside the ART. It may become necessary to understand how IPC mechanisms work. Another important skill in static analysis for android is the understanding of Smali code. Decompilers are not always reliable. When decompilers fail, the analysis falls back on Smali.

9.4.2 Dynamic Analysis

Dynamic analysis on android is usually done through the Android Debug Bridge (ADB). The ADB allows us to connect with an android device through an internet or USB connection. ADB gives us a shell on which we can execute commands. If the android device is rooted, then we can act as root through ADB.

The environment, the system status and the logs are important for dynamic analysis. We can check the environment and system data using ADB. We can access the logs using the logcat command of ADB.

There are two established ways to debug an android application. The first is using a gdb client that connects to a gdb server running on the android device. The server takes control of the target process and acts on behalf of the commands the client gives. The second way is to use a JDB(Java Debugger) client. This debugger also connects with a server(jdwp server) but this one runs inside the ART VM. The debugger takes control of the process running inside the ART VM.

Basically, there are two types of debuggable entities on Android OS:

- Android Processes - Here we debug native code using ptrace, with the debugger running alongside the target process.
- ART vM - Here we debug inside a process. This is made possible by the JDWP protocol, whuch runs inside the hosted ART vM.

9.4.3 Instrumentation

Instrumentation is an essential analysis technique used to understand program behaviour and quality. Instrumentation refers to the technique of adding code to the target program or process with the purpose of monitoring, logging, or simply collecting information about program execution. We distinguish between static instrumentation and dynamic instrumentation.

Static instrumentation refers to the technique of adding snippets of code during compile time. This can be done before or during compilation. A well-known example of static instrumentation is compiling binaries with Adress Sanitiser(ASAN). Static instrumentation requires full access to the source code or the compiling toolchain.

In dynamic instrumentation, we add extra code to a binary during or just before runtime. The advantage of dynamic instrumentation is the access to runtime behaviour and libraries. Dynamic analysis requires full control over the process running or the binary that is about to run. This usually translates to root privileges.

9.5 Tools

In conclusion of this lecture, the following list introduces some of the established tools for analysing android applications.

- Android Studio - Android development toolkit. It also contains an android emulator with built-in logging and adb.
- JADX - Command line and GUI tools for producing Java source code from Android Dex and Apk files
- APK Lab - Visual Studio extension for APK analysis
- JNI Analyser - This Ghidra extension contains various scripts that assists in analyzing Android NDK applications.
- Ghidra
- ADB - Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device
- FRIDA - Cross-Platform Instrumentation and Analysis Toolset