Static Analysis

1.1 Calling Conventions

Static Analysis relies on understanding the low-level behavior of functions, which is largely governed by the Calling Convention used. The calling convention is a standardized agreement between the calling function (caller), and the called function (callee) that dictates the precise mechanism of a function call. A calling convention defines three essential aspects:

(1) *Parameter Passing*: How many arguments are passed (on the stack or in registers).

(2) *Return Value*: How the function's result is returned to the caller.

(3) *Stack Cleanup*: Which function (caller or callee) is responsible for cleaning up the parameters pushed onto the stack.

The function result is typically returned via the `EAX` (32-bit) or `RAX` (64-bit) register.

Common x86 Calling Conventions. The following conventions are frequently encountered during static analysis, particularly in 32-bit executables, and they primarily differ in their stack cleanup and parameter passing rules:

Convention	Cleanup	Parameter Order	Mechanism
`_cdecl`	Caller cleans the stack.	Right-to-Left (First parameter is pushed last, at the lowest address).	The standard convention for C variable-argument functions (e.g., `printf`). The caller cleans up, making it suitable for functions with a variable number of arguments.
`_stdcall`	Callee cleans the stack.	Right-to-Left.	Used by most Windows API functions. Since the callee cleans the stack, the function definition must know the exact number and size of its parameters.
`_fastcall`	Callee cleans the stack.	Primarily uses registers (`ECX` and `EDX` on x86, with `EAX` sometimes used), falling back to the stack for additional arguments.	Optimized for speed by minimizing memory access. Best suited for functions with few arguments (typically two to three).
`_thiscall`	Callee cleans the stack.	Right-to-Left (for stack arguments).	Used specifically by C++ member functions. It is similar to `_stdcall` , but the implicit `this` pointer (a reference to the object) is passed in the `ECX` register, rather than on the stack.

Table 1 / Comparison of common x86 calling conventions - `_cdecl` , `_stdcall` , `_fastcall` , and `_thiscall` - highlighting stack cleanup responsibility, parameter-passing order, and usage context.

It is important to remember that different compilers may establish their own conventions or use compiler-specific names for these standards, so an analyst must always verify the actual low-level assembly implementation.

Assembly x86-64. The x86-64 architecture is the 64-bit extension of the original x86 instruction set. This architecture is crucial for modern malware analysis and reverse engineering as it governs the vast majority of current operating systems and applications. The x64 architecture provides 16 general-purpose registers (from `RAX` to `R15`), compared to the 8 registers in 32-bit. All these registers are 64-bit wide, enabling 64-bit operations and data handling. It also uses 64-bit memory addressing, allowing the system to access a vastly larger theoretical memory space than the 4 GB limit of x86.

The standard calling convention for x64 on Windows significantly changes how arguments are passed compared to the x86 stack-based conventions. The first four integer or pointer arguments are passed via specific general-purpose registers, removing the need for immediate stack operations and speeding up function calls. The order of registers used is: `RCX` , `RDX` , `R8` , `R9` . Any subsequent arguments are pushed onto the stack. The caller function is responsible for cleaning up the stack, if any arguments were pushed there. The registers are assigned in order from the first parameter to the fourth parameter, not from the last parameter.

The x64 architecture natively supports more advanced data types and operations, particularly for floating-point and SIMD (Single Instruction, Multiple Data) processing.

- *XMM Registers*: These are special 128-bit registers that were introduced with the Streaming SIMD Extension (SSE) instruction sets. x64 extends the number of these registers to 16.
- *Data Storage*: Each 128-bit XMM register can store:
  - Two 64-bit floating-point values (for double-precision).
  - Four 32-bit floating-point values (for single-precision).
  - Various integer data types for parallel processing.

1.2 Decompilation

Decompilation is an advanced reverse engineering technique that aims to reconstruct a high-level source code representation from the original, low-level assembly language of a compiled program. This process involves more than just translating assembly instructions; a decompiler must perform complex analysis to infer:

- *Data Types and Structures*: The decompiler attempts to identify the original data types, complex structures, and arrays that were abstracted away during compilation.
- *Control Flow*: It analyzes assembly jump and condition instructions to reconstruct high-level control structures like `if/else` statements, `for` loops, and `while` loops.
- *Variable Usage*: It tracks register and stack usage to synthesize human-readable variables (often using custom generic names).

The main advantage of decompilation is the significant increase in code readability. Analyzing thousands of lines of C code is dramatically faster and more intuitive than analyzing the equivalent assembly code. Decompilers allow the analyst to rename custom variables, functions, and structures. This is a crucial step in static analysis, as assigning meaningful names makes the code self-documenting. While an advantage is often cited as the ability to re-compile the decompiled code, this is generally not straightforward and is only possible with significant manual cleanup and correction. Decompilers often use custom types and library calls (like IDA's `defs.h` header) that are necessary to make the decompiled code syntactically correct, but may not perfectly reproduce the original source.

Recognizing Code Constructs. While decompilation offers a high-level view that greatly aids comprehension, it is never a standalone analysis technique. Its output must be continually compared against the original disassembly, which is often more reliable and precise, especially when the decompiler struggles with complex constructs. Decompilers face several inherent challenges when translating machine code back to source code:

- *Distinguishing Code from Data*: Like disassemblers, decompilers can sometimes confuse data sections (e.g., initialized variables, constant tables) with executable code, leading to corrupted or nonsensical output.
- *Inferring Data Types*: The compiler discards high-level type information (like custom `typedef` or complex user-defined structures) during compilation. The decompiler must infer the correct return and variable types based on how they are used, which is prone to error.
- *Complex Control Structures*: Analyzing and reconstructing intricate control flow elements can be difficult, such as translating a jump table back into a high-level `switch` statement, or correctly identifying the boundaries and conditions of multiple nested `for` or `while` loops.

Effective analysis requires understanding how high-level logic is expressed in assembly language, particularly concerning control flow. Conditional branches (like `if` statements) are implemented using two main steps:

(1) *Setting Flags*: Instructions like `CMP` (Compare) or `TEST` perform a subtraction or logical operation between two operands and set specific status FLAGS (e.g., Zero Flag, Sign Flag, Carry Flag) in the `EFLAGS` or `RFLAGS` register, but they do not store the result.

(2) *Conditional Jump*: A subsequent conditional jump instruction then checks the state of one or more flags to decide whether to jump.

Instruction	**Condition**	**Meaning**
`JZ` or `JE`	Zero Flag is set (`ZF = 1`)	Jump if the result of the previous operation was zero (or if operands were equal).
`JNZ` or `JNE`	Zero Flag is not set (`ZF = 0`)	Jump if the result of the previous operation was non-zero (or if operands were not equal).

Table 2 / Common conditional jump instructions in x86 assembly: `JZ` / `JE` and `JNZ` / `JNE` and their dependence on the Zero Flag (`ZF`) to control flow based on equality or zero-result conditions.

Arrays are typically accessed using sequential memory addressing. A general-purpose register (like `RCX` or `RSI`) is often employed as an index or offset, which is scaled by the element size and added to the array's base address. A `for` loop is usually recognized by:

(1) An initialization of the counter variable before the loop starts.

(2) A `CMP` instruction followed by a conditional jump at the beginning or end of the loop, serving as the exit condition.

(3) An increment or decrement instruction applied to the counter variable within the loop body.

(4) A back-arrow jump in the control flow graph, which returns execution to the condition check.

A `while` loop is structurally similar to a `for` loop but often lacks the dedicated increment/decrement instruction within the loop's structure; the condition check is typically placed at the top.

A `switch` statement with many cases is typically compiled into a jump table, offering faster execution than a chain of nested `if/else` conditions. The value of the `switch` variable (the case selector) is used to calculate an index into an array of code addresses (the jump table). The program then uses an indirect jump instruction to transfer control directly to the code block corresponding to that specific case. The array of addresses is stored in a predetermined, often read-only, data section of the file (frequently referenced via a data segment register like `ds:`) and consists of sequential addresses. While the table stores consecutive addresses, the size of the assembly code for each case block can vary, and the case values themselves may not be consecutive. A `switch` statement with only a few cases is more often translated into a series of nested `if` conditions using standard `CMP` and conditional `JMP` instructions.