Overview of Capstone

Capstone is a lightweight disassembly framework, designed to work with multiple CPU architectures, with that target of being the go-to disassembly engine for binary analysis and reversing in the security community. Tools enabling binary analysis, debugging, and exploitation development sit a layer above Disassembly Engines, which is exactly what capstone is. Known as disassembly frameworks, engines, or libraries, they are responsible for decoding binary code properly to return some assembly code. They can also help to break assembly down in detail to understand its internal workings. Capstone satisfies the complex requirements of working on multiple architectures (x86, Arm, Arm64, MIPS, PPC, 4 more), multiple platforms (Windows, MacOSX, Linux, iOS, BSD, Android, Solaris), and multiple bindings (9 different languages). It does this while implementing a simple, intuitive, and architecture-neutral API, while also giving break-down details on instructions and having a friendly, BSD license. Capstone also has the added benefit of dealing with more tricky x86 instructions than others. Until its release in 2013, no other such disassembler even came close to satisfying these requirements.

Capstone uses a modified fork of the LLVM project, a set of frameworks to build a compiler, which contains a disassembler, Machine code (MC) module. Using this disassembler comes with the benefits of being maintained by top experts of each architecture, including x86 (intel), Arm + arm64 (arm & apple), Mips (imgtec), and others. It has the added advantage of receiving new instructions and bug-fixes incredibly frequently. The disadvantage of using the MC is that its code is in C, not designed for thread-safety, and isn’t designed for windows. Using the MC would also mean keeping up with upstream code to maintain it. Capstone works on the basis of making a compromise in that it must cut off some of the CM code such that it wouldn’t change the code structure, so that upstream changes couldn’t be ported. The other consideration is cutting off too little, resulting in keeping too much redundant code. Capstone works on the basis of taking the disassembler core and making minimal changes, re-implementing required layers of the disassembler. It effectively works as a re-write of LLVM’s MC, but replacing dependent C++ classes and methods with pure C function pointers and structures. Additionally, certain global variables were replaced to make Capstone thread-safe.

Capstone is far superior to LLVM’s disassembler, working as an independent framework more compact in size, providing much more information than just assembly code. Its other additional features include support for Big-Endian machines, Arm and Arm64, along with optimization towards disassembly, and further instruction support. LLVM’s disassembler is considered second class and doesn’t cover tricky cases of binary code. Capstone’s priority is to cover tricky and corner cases of code and as such has taken measures skipped by LLVM’s disassembler. It handles all notable x86 malware tricks available at the time of development.



Figure 1 – tricky x86 instructions handled by Capstone

Capstone is capable of handling more instructions than other frameworks, and even supports most modern CPU extensions. The designers guarantee to keep Capstone updated in the future too. Its clean and intuitive API makes it simple for new users to start writing tools, with an API independent of hardware. Its detailed instruction breakdown makes it straightforward to access instruction operands and other internal instruction data. Additionally, it can provide important semantics of the disassembler instruction such as a list of implicit registers read and written, or if the instruction belongs to a group of instructions. This gives it the additional feature of being a decomposer as well as a disassembler.