compression	(used on top of compression algorithms)
section merging	merge all sections (just one entry in the section table)
imports	imports are stored and loaded with a more compact import table format
imports by hash	exports are parsed until it matches a specific hash, instead of a *GetProcAddress* call
call optimisation	turn relative operands of jumps and calls into absolute → better compression
resources	compresses resources, avoiding critical ones (main icon, manifest,...)
protection	
token check	presence check to allow the program to run: `dongle, CD/DVD, key, file, network...`
fingerprinting	token is specific to a hardware element: `disk/OS/CPU/MAC/...`
demo mode	inclusion of a demo binary/mode that is executed when token is absent or not enough privileged
integrity	check the contents are unmodified with checksum or hash
anti-analysis	
overlap	jumping after the first byte of an instruction
illusion	makes the analyst the something incorrect happened
junk	insertion of dummy code between relevant opcodes
jumps	insertion of jumps to makes analysis visually harder
polymorphism	different but equivalent code → 2 packed files of the same source are different
self generation	packer stub generates polymorphic code on the fly → same file executes differently
virtualization	virtualizes (part of) packer stub code → harder analysis
stack	strings are built and decrypted before use, then discarded → to avoid obvious references
faking	add fake code similar to known packers to fool identifcation
thread	use several parallel threads to make analysis harder
timing	comparing time between two points to detect unusual execution
anti-debugging	(and anti-tools, by extension)
detect	detect the presence of an attached debugger: `IsDebuggerPresent`
prevent	prevent a debugger to attach to the target itself or stay attached
nuisance	make debugger session difficult: `BlockInput, slow down...`
thread	spawn a monitoring thread to detect tampering, breakpoints, ...
artifacts	detects a debugger by its artifact: `window title, device driver, exports, ...`
limitation	prevent the use of a tool via a specific limitation
exploit	prevent the use of a tool via a specific vulnerability
backdoor	detect or crash a debugger via a specific backdoor
self-debugging	debug itself to prevent another debugger to be attached
int1	block interruption 1 → debuggers stop working
fake	add code of known packer to fool identification
anti-dumping	(prevent making a working executable from a memory image)
tampering	erase or corrupt specific file parts to prevent rebuilding (header, packer stub,...)
imports	add obfuscation between imports calls and APIs (obfuscation, virtualization, stealing, ...)
on the fly	API address is resolved before each use to prevent complete dumping
API hooking	alter API behavior: `redirect benign API to a critical one` → dump not working
inlining	copy locally the whole content of API code → no more 'import calls'
relocate	relocate API code in separate buffer → calls don't lead to imported DLLs
byte stealing	move the first bytes of the original code elsewhere → harder rebuilding and bypasses breakpoints
page guard	blocks of code are encrypted individually, and decrypted temporarily only upon execution
flow	flow opcodes are removed and emulated (or decrypted) by the packer during execution → incorrect dump
virtualization	virtualizes (part of) original code, API start... → dump not working without VM code
anti-emulation	
opcodes	using different opcodes sets (FPU, MMX, SSE) to block emulators
undoc	use of rare or undocumented opcodes to block non-exhaustive emulators
API	unusual APIs are called to block non-exhaustive emulators (anti-virus)
loop	extra loops are added to make time-constraint emulators give up
bundlers	
drop	original file is written to disk then executed
injection	original file is injected in existing process → no new file on disk + higher privileges
hooking	file handling APIs are modified to make embedded files usable like external ones