Packers

(5th April 2010)

Ange Albertini
http://corkami.blogspot.com

Table of contents

3 Models: simple, malware, advanced

4 Categories and Features: compresser, protecter, crypter, bundler, virtualiser, mutater

5 Landscape: Free, Commercial, Malware / Bundlers, Compressors, Virtualizers

6 Detailed features: compression, anti-analysis, anti-debugging, anti-dumping, anti-emulation, bundlers

7 EntryPoints: FSG, UPX (with LZMA), AsPack, PECompact, MEW, Upack

8 Algorithms: aPLib, LZMA, CRC32

Changelog

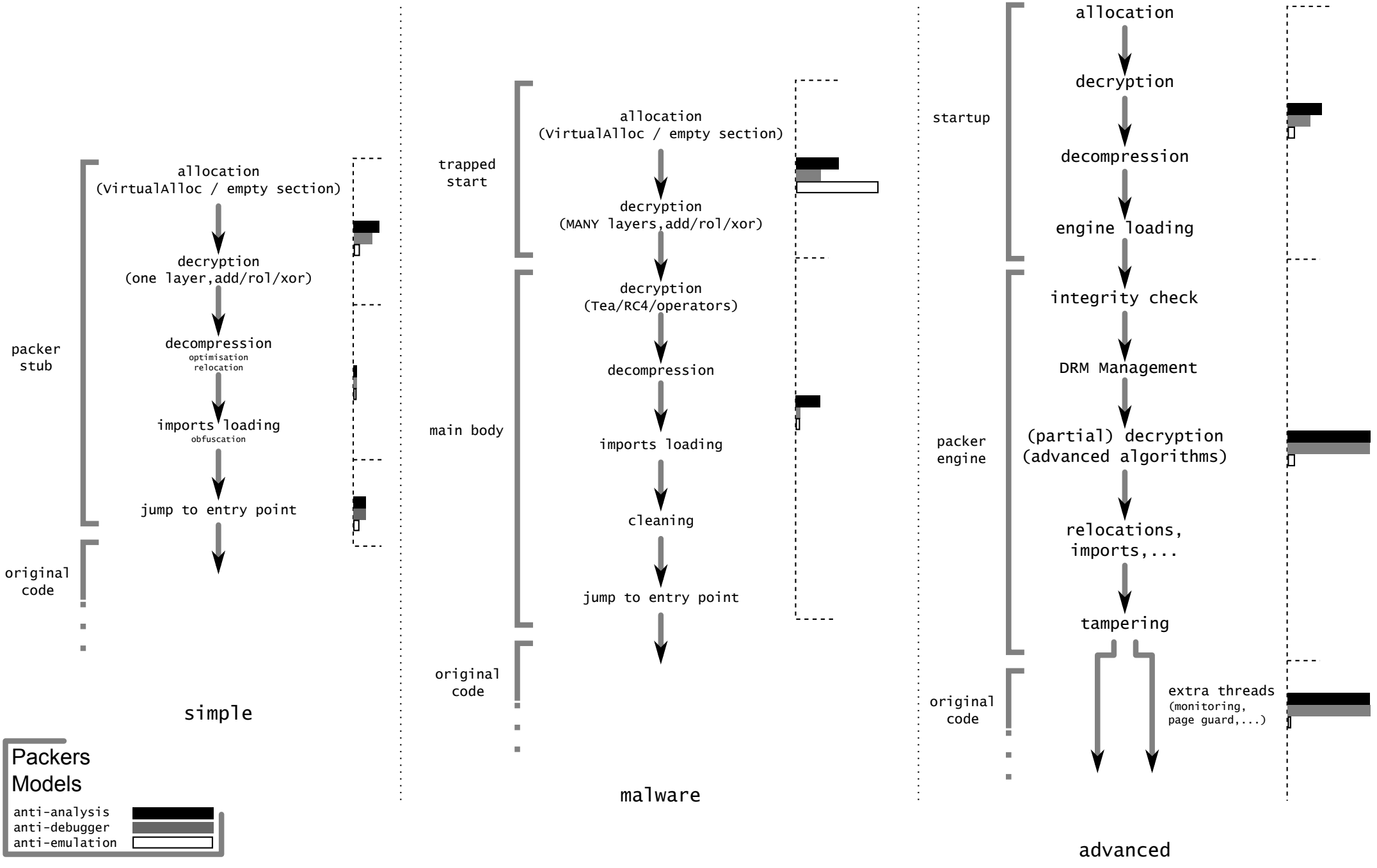
2010/04/05 +algorithms

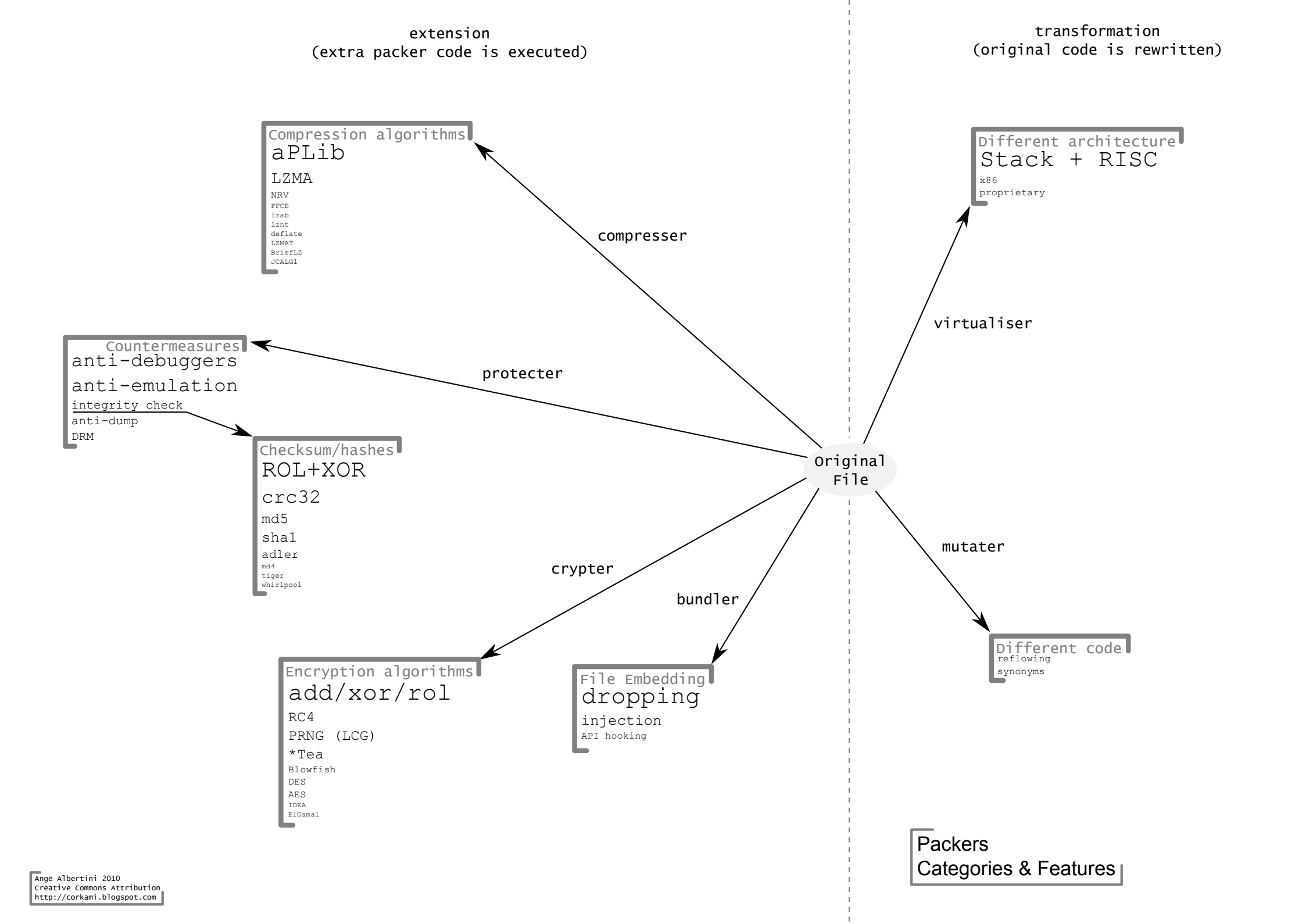
2010/04/04 +models

2010/03/29 +entrypoints

2010/03/24 +categories and features, detailed features

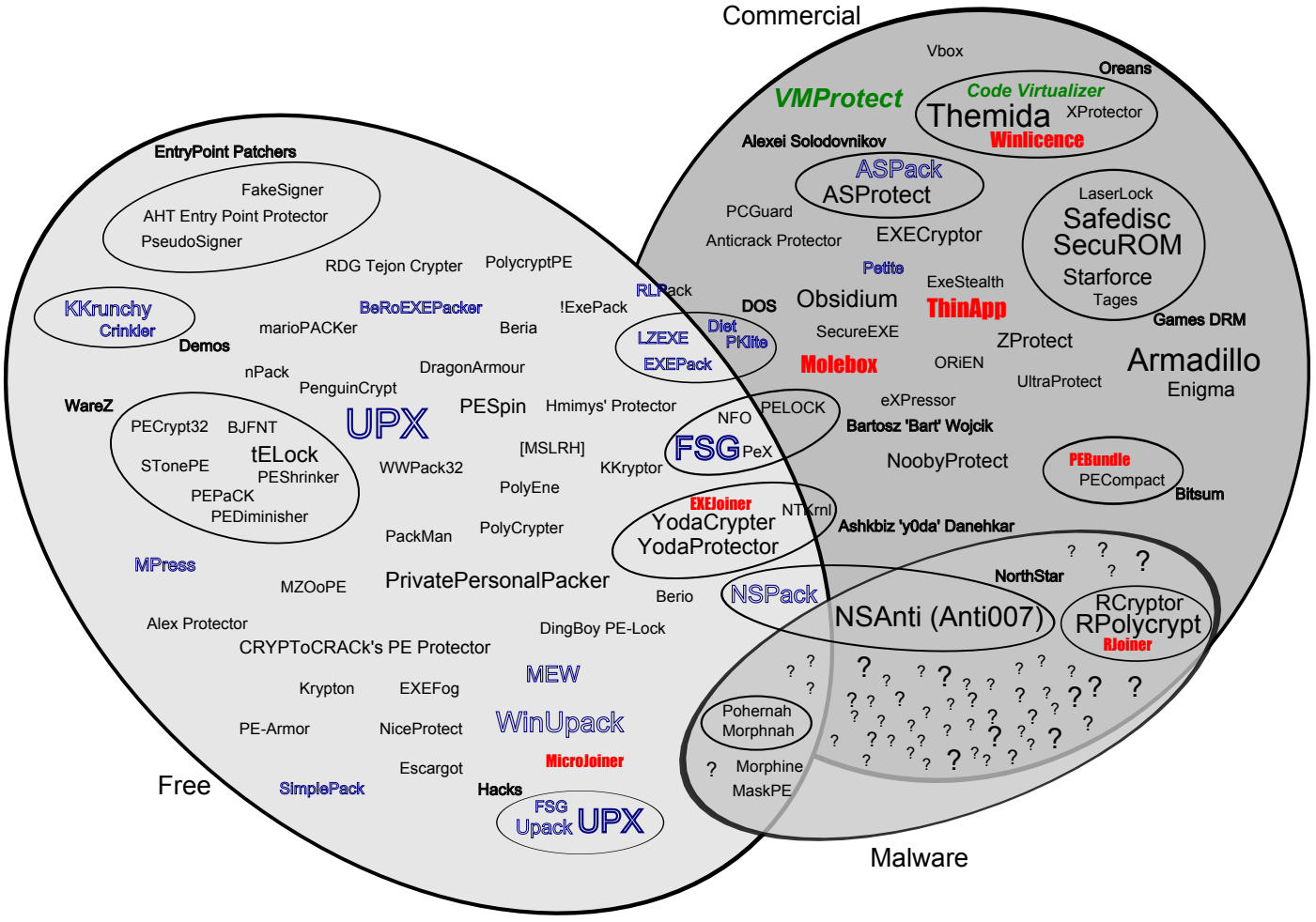
2010/02/23 +landscape (first graphic)




extension
(extra packer code is executed)
transformation
(original code is rewritten)
Compression algorithms
aPLib
LZMA
NRV
FFCE
lzab
lznt
deflate
LZMAT
BriefLZ
JCALG1
compresser
Countermeasures
anti-debuggers
anti-emulation
integrity check
anti-dump
DRM
protecter
Checksum/hashes
ROL+XOR
crc32
md5
sha1
adler
md4
tiger
whirlpool
Original File
virtualiser
Different architecture
Stack + RISC
x86
proprietary
mutater
Different code
reflowing
synonyms
crypter
bundler
Encryption algorithms
add/xor/rol
RC4
PRNG (LCG)
*Tea
Blowfish
DES
AES
IDEA
ElGamal
File Embedding
dropping
injection
API hooking
Packers
Categories & Features
Ange Albertini 2010
Creative Commons Attribution
http://corkami.blogspot.com

Packers Landscape

Bundlers
Virtualisers
Compressors



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

PECOMPACT

```
EntryPoint:
    mov  eax, _1
    push eax
    push dword ptr fs:[0]
    mov  fs:[0], esp
    xor  eax, eax
    mov  [eax], ecx
[...]
_1:
    mov  eax, <random1>
    lea  ecx, [eax + <random2>]
    mov  [ecx + 1], eax
    mov  edx, [esp + 4]
    mov  edx, [edx + c]
    mov  byte ptr [edx], 0e9
    add  edx, 5
    sub  ecx, edx
    mov  [edx - 4], ecx
    xor  eax, eax
    retn

    mov  eax, 12345678
    pop  dword ptr fs:[0]
    add  esp, 4
    push ebp
    push ebx
```

MEW

```
 _1:
     mov  esi, <address>
     mov  ebx, esi
     lodsd
     lodsd
     push eax
     lodsd
     xchg eax, edi
     mov  dl, 80
 _2:
     movsb
     mov  dh, 80
     call [ebx]
     jnb  _2
 [...]
 EntryPoint:
     jmp _1
```

FSG

```
EntryPoint:
    xchg  [_1], esp
    popad
    xchg  eax, esp
    push  ebp
_1:
    movsb
    mov   dh, 80
    call  [ebx]
    jnb   _1
    xor   ecx, ecx
    call  [ebx]
```

UPX (LZMA)

```
EntryPoint:
    pushad
    mov   esi, <address>
    lea   edi, [esi + <negative>]
    push  edi
    mov   ebp, esp
    lea   ebx, [esp - 3E80]
    xor   eax, eax
_1:
    push  eax
    cmp   esp, ebx
    jnz   _1
    inc   esi
    inc   esi
    push  ebx
    push  0C478
    push  edi
    add   ebx, 4
    push  ebx
    push  534E
    push  esi
    add   ebx, 4
    push  ebx
    push  eax
    mov   dword ptr [ebx], 20003
    nop
    nop
    nop
    nop
    nop
    push  ebp
    push  edi
    push  esi
    push  ebx
    sub   esp, 7C
    mov   edx, [esp + 90]
```

UPX

```
EntryPoint:
    pushad
    mov    esi, <address>
    lea    edi, [esi + <negative>]
    push   edi
    or     ebp, ffffffff ; * Not in UPX >3
    jmp    $ + 12
    nop
    nop                  ; *
    nop                  ; *
    nop                  ; *
    nop                  ; *
    nop                  ; *
    mov    al, [esi]
    inc    esi
    mov    [edi], al
```

AsPack

```
EntryPoint:
    pusha
    call  _1
db 0E9h       ; E9 EB045D45 CALL ...
    jmp   _2
_1:
    pop   ebp
    inc   ebp
    push  ebp
    retn
_2:
    call  _3
db 0EBh       ; EB54 JMP <garbage>
_3:
    pop   ebp
```

Packers EntryPoints

Upack

```
EntryPoint:
    mov  esi, <address>
    lodsd
    push eax
    push dword ptr [esi+34]
    jmp  short _1
[...]
_1:
    push dword ptr [esi+38]
    lodsd
    push eax
    mov  edi, [esi]
    mov  esi, <address2>
```

APLIB

```
start:
    pushad
    mov esi, [esp + 24]
    mov edi, [esp + 28]
    cld
    mov dl, 80
    xor ebx, ebx
copy_literal:
    movsb
    mov bl, 2
next:
    call getbit
    jnb short copy_literal
    xor ecx, ecx
    call getbit
[...]
    sub esi, eax
    rep movsb
    pop esi
    jmp next
getbit:
    add dl, dl
    jnz skip
    mov dl, [esi]
    inc esi
    adc dl, dl
skip:
    retn
[...]
end:
    sub edi, [esp + 28]
    mov [esp + 1c], edi
    popad
    retn 0c
```

LZMA

```
start:
    push ebp
    mov ebp, esp
    add esp, -54
    push ebx
    push esi
    push edi
    mov [ebp - c], ecx
[...]
$+84:
    add ecx, [ebp - 34]
    mov eax, 300
    shl eax, cl
    add eax, 736
    dec eax
    test eax, eax
    jb no_init
    inc eax
    mov [ebp - 2c], 0
init_buffer:
    mov edx, [ebp - 10]
    mov ecx, [ebp - 2c]
    mov [edx + ecx * 4], 400
    inc [ebp - 2c]
    dec eax
    jnz init_buffer
no_init:
[...]
    mov al, 1
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    retn 10
```

CRC32

```
crcloop:
    test eax, 1
    jz   no_xor
    shr  eax, 1
    xor eax, 0EDB88320h
    jmp  loop

no_xor:
    shr eax, 1
loop:
    loop crcloop
```

Packers
Algorithms