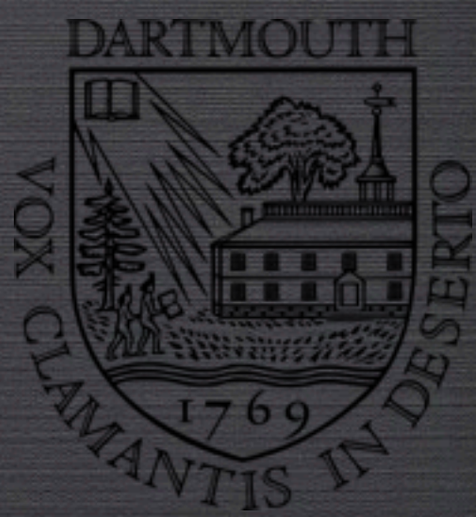
Weird Machines and revisiting "Trusting Trust" for binary toolchains

Julian Bangert, Rebecca Shapiro,
Sergey Bratus

Dartmouth College



Outline

- Trust chains **without bugs**: what can go wrong?
- A chain of **trust** is a chain of **parsers/loaders**: how bug-less Babel breaks (badly!)
- Case studies:
  - Any ~~input~~ table is a program (recall 29c3)
  - ELF signing, Mach-O signing
  - ELF kernel loader vs. RTLD/ld.so

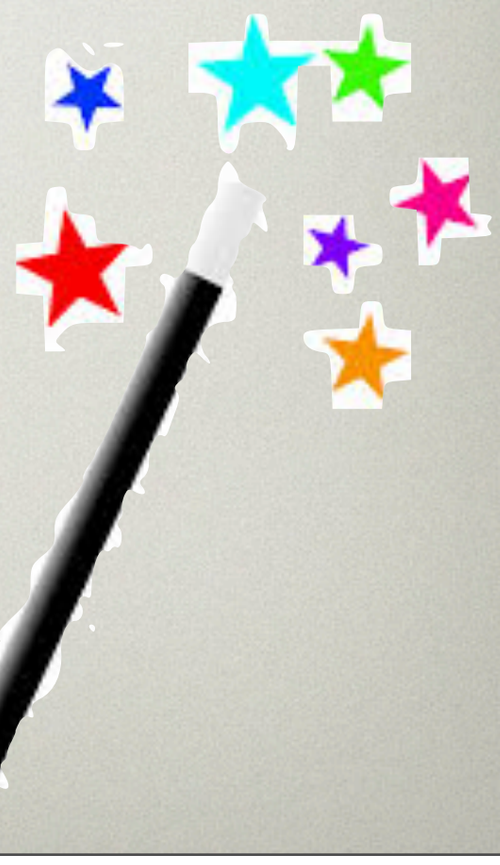
LangSec
vs
Chains of Trust

Further Reflections on Trusting Trust

- Ken Thompson, "**Reflections on Trusting Trust**", 1984
  - (almost) 30 years ago
  - *"You can't trust code that you did not* ***totally*** *create yourself"*
  - invisible links in the chain-of-trust (..."*well-installed microcode bugs*"...)

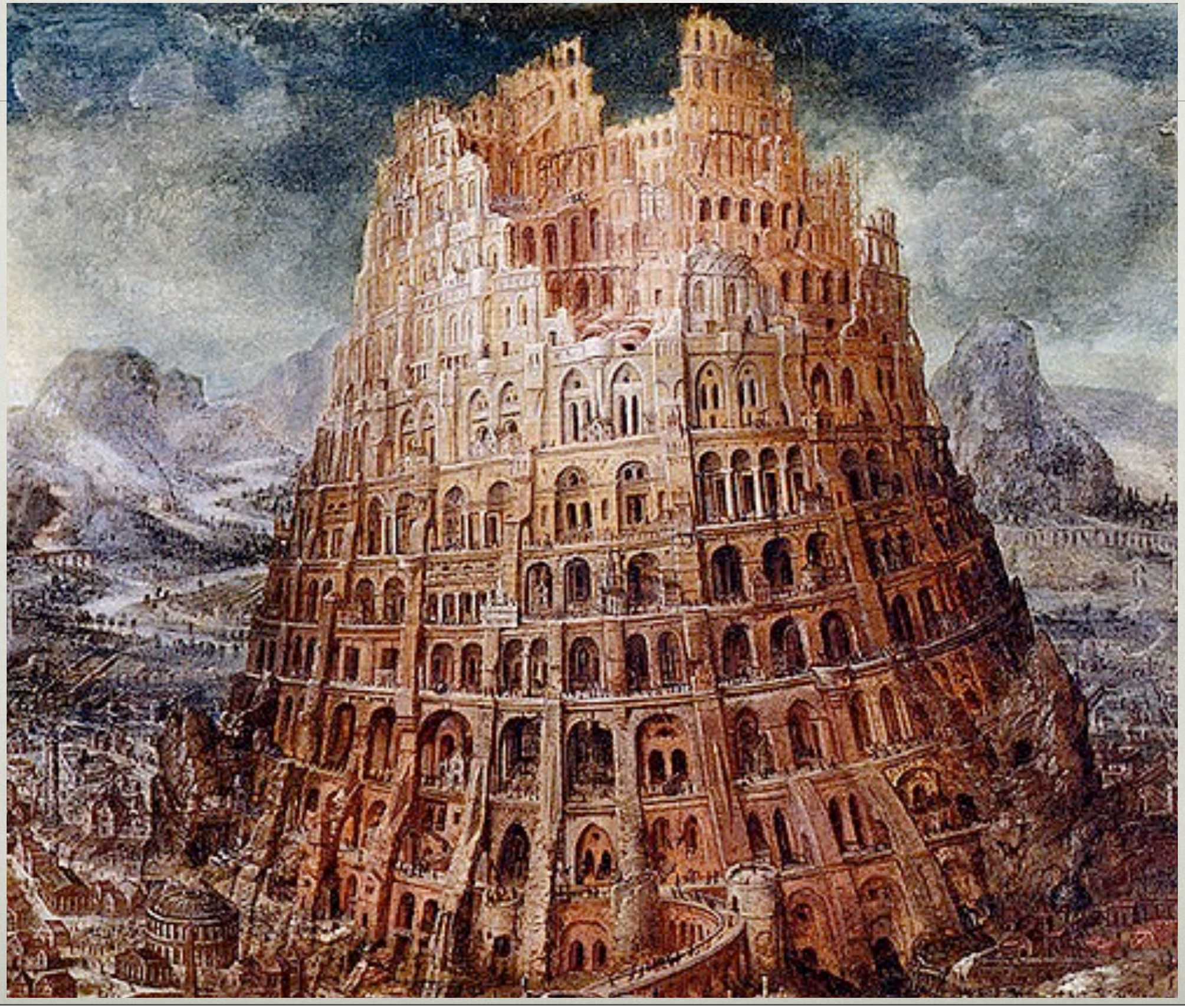
Beyond the bugs in Trusting Trust

- What if there were **no bugs** in any given piece of sw/hw link of the trust chain?
  - What if the code did exactly what the author **intended**, and
    - you can **trust** the author?
- Would we solve "*trusting trust*"?



Hell No!

Because we are in Babel



Dialects of Input?

I can has ur trust chainz?

Chain of Trust

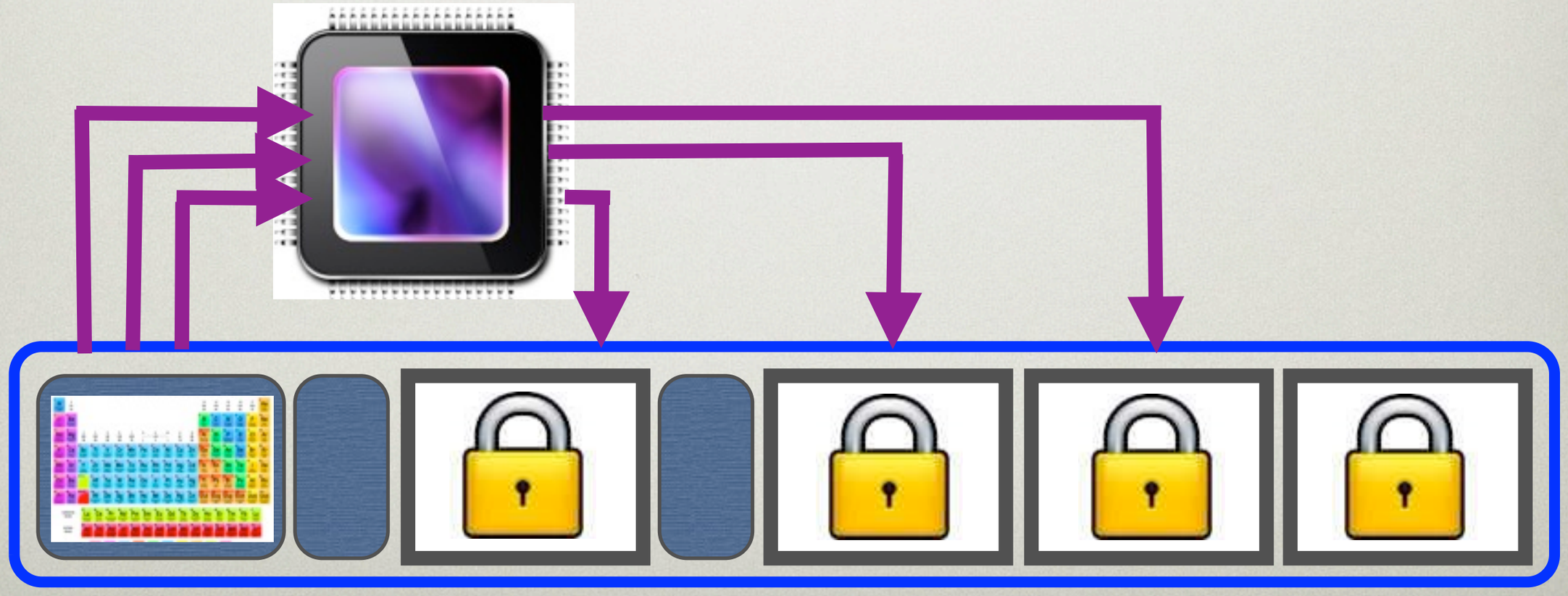
- **Chain:** execution environments of increasing complexity and power (from boot to full OS ABI)
- **Goal**: no **unexpected computation** throughout
- Same code/data bytes interpreted (i.e., **executed**) by several consecutive environments
- Two kinds of trust in data or code:
  - input (code/data) can be **checked** for effects
  - input (code/data) was **signed** & been immutable since someone checked it for effects

Trusted bits: hard vs soft

- It's **hard** to statically find out what code does
  - So we "**freeze**" (sign, etc.) code 
- But we can't freeze **full** binary images without impairing **composition**
  - Libraries, dynamic modules, ASLR,...
- So we add "**tables**" to drive composition/mutability mechanisms

Any Table Is a Program

- "Tables" drive computation that locates signed ABI sections & their signatures
- Tables are **bytecode** for automata in signature verifiers/loaders/parsers



“Any input is a program”

- (Meta)data is just a **program** for code that interprets it. [Hopefully, analyzable for effects]
- Any sufficiently complex input data is indistinguishable from **byte code** driving a VM
- Parser code for any sufficiently complex input format is indistinguishable from a **VM** for its inputs (= “byte code”)
- Input **validation** is “runtime **verification**” of inputs as programs

What can go wrong?

- Input **not** well-defined / recognized
  => code's assumptions about "checked" input **will** be violated (bug / vuln)

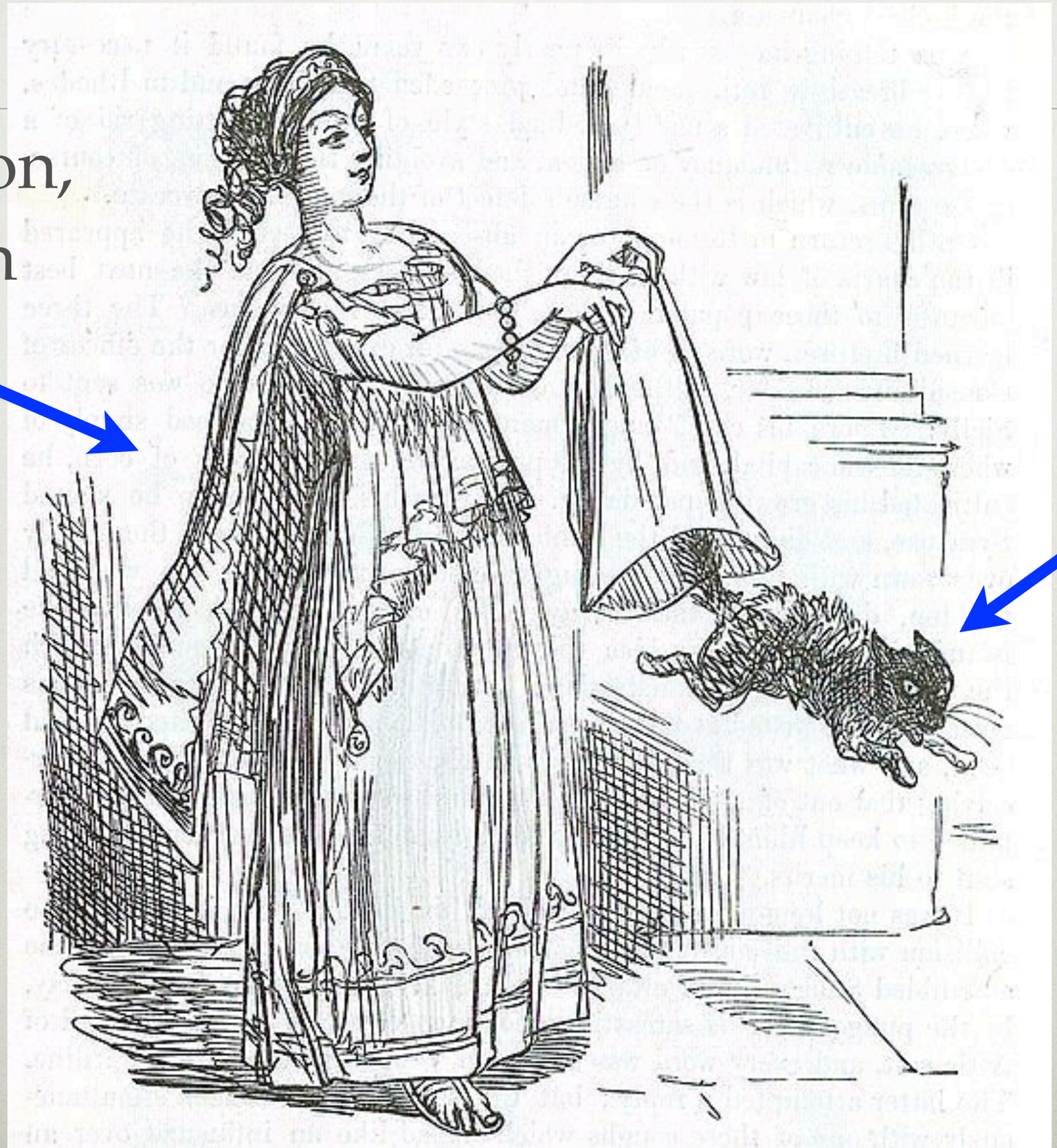
- Input **well-formed** but so complex there's no telling what it does

- Input is seen **differently** by different pieces of program / toolchain

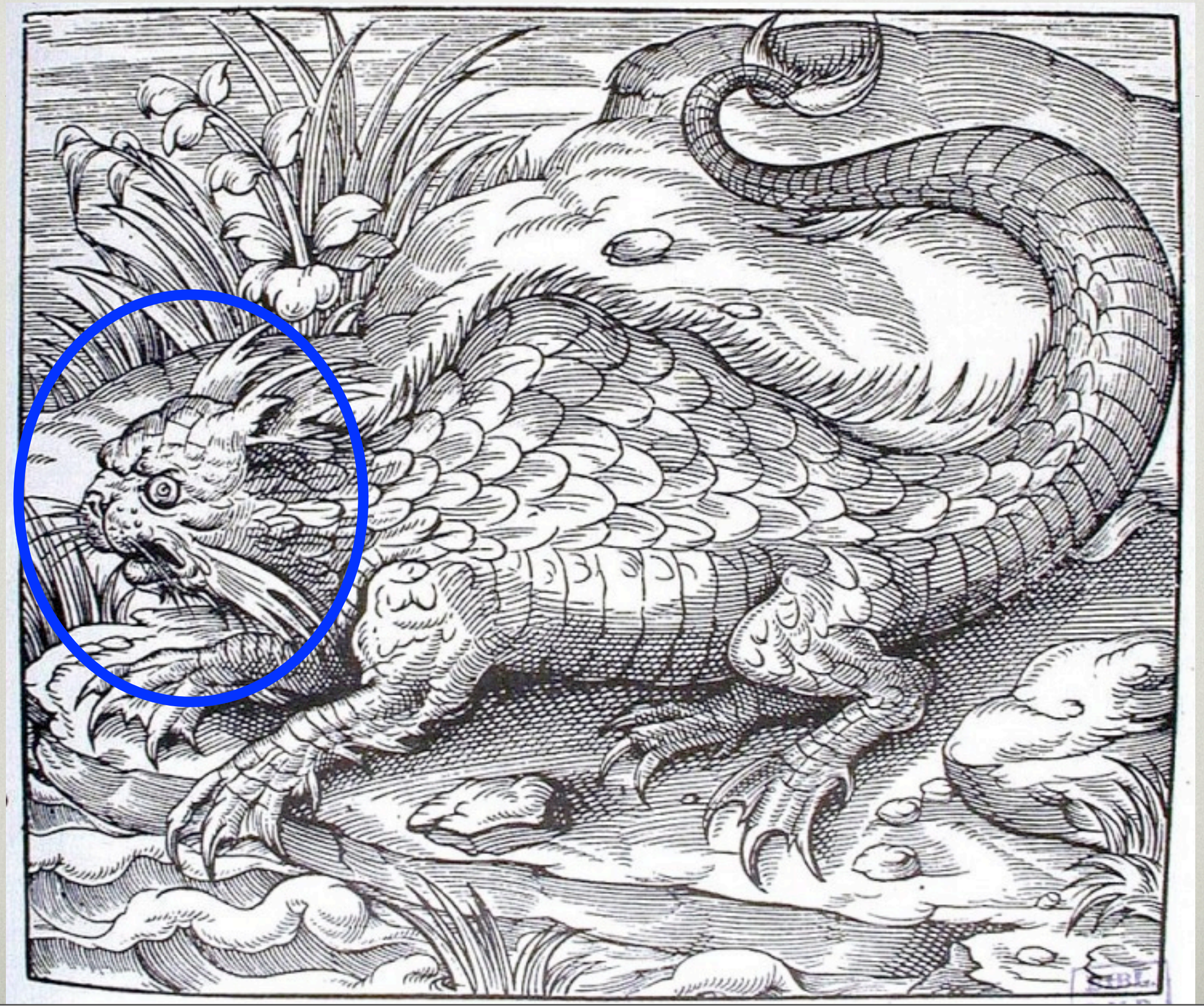
Liberating the soft bits

composition,
mutation

tables,
"staging"
data



“Liberated soft bits”



The ELF/ABI case study

Ken Thompson's planted bug

Compiler

ld.so (RTLD)

Linker

Relocator ("dl-machine")

DWARF exceptions

Loader

binfmt_elf

#PF

#DF

The ELF/ABI case study

Compiler

Linker

ld.so (RTLD)

Relocator
("dl-machine")

DWARF
exceptions

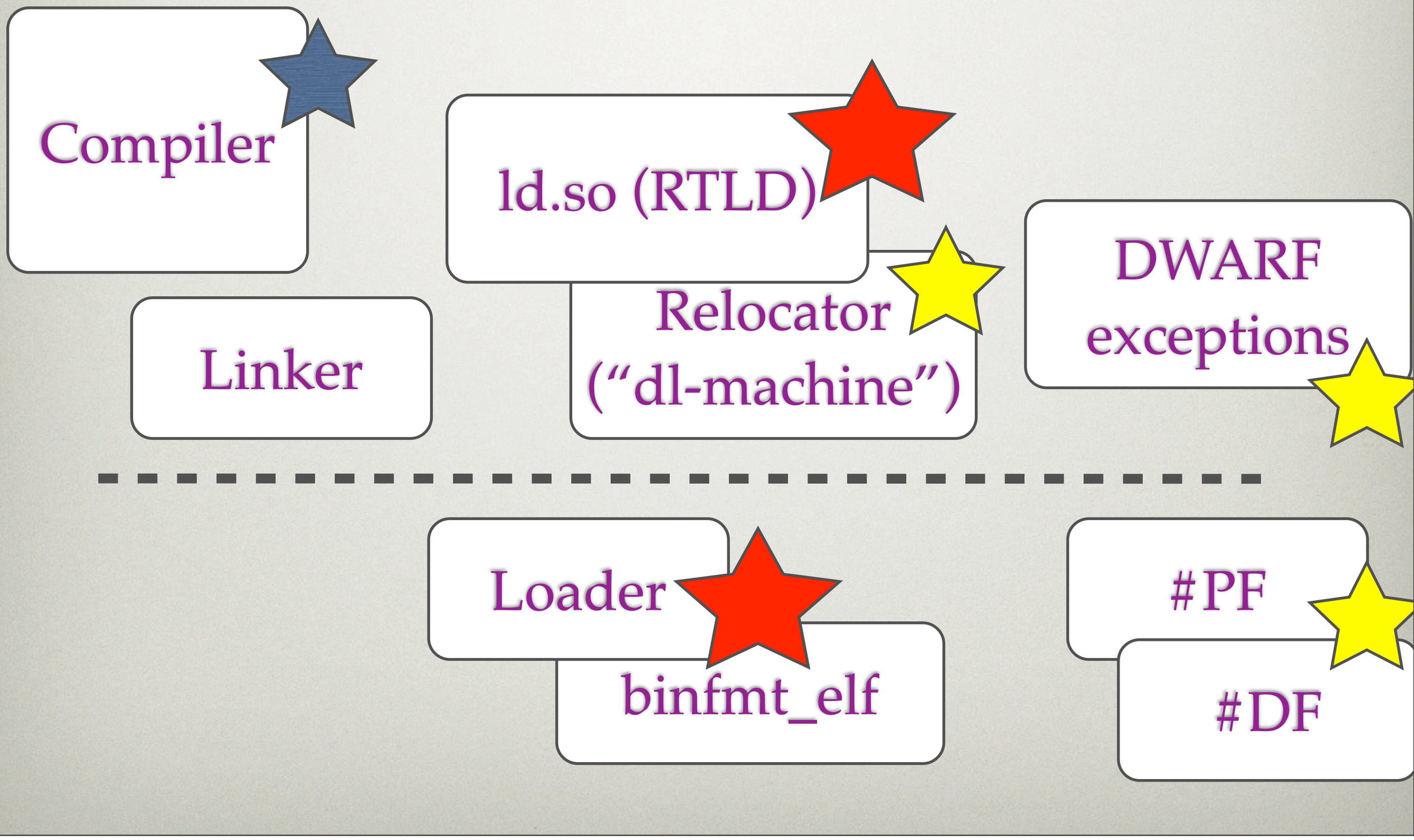
Loader

binfmt_elf

#PF

#DF

The ELF/ABI case study

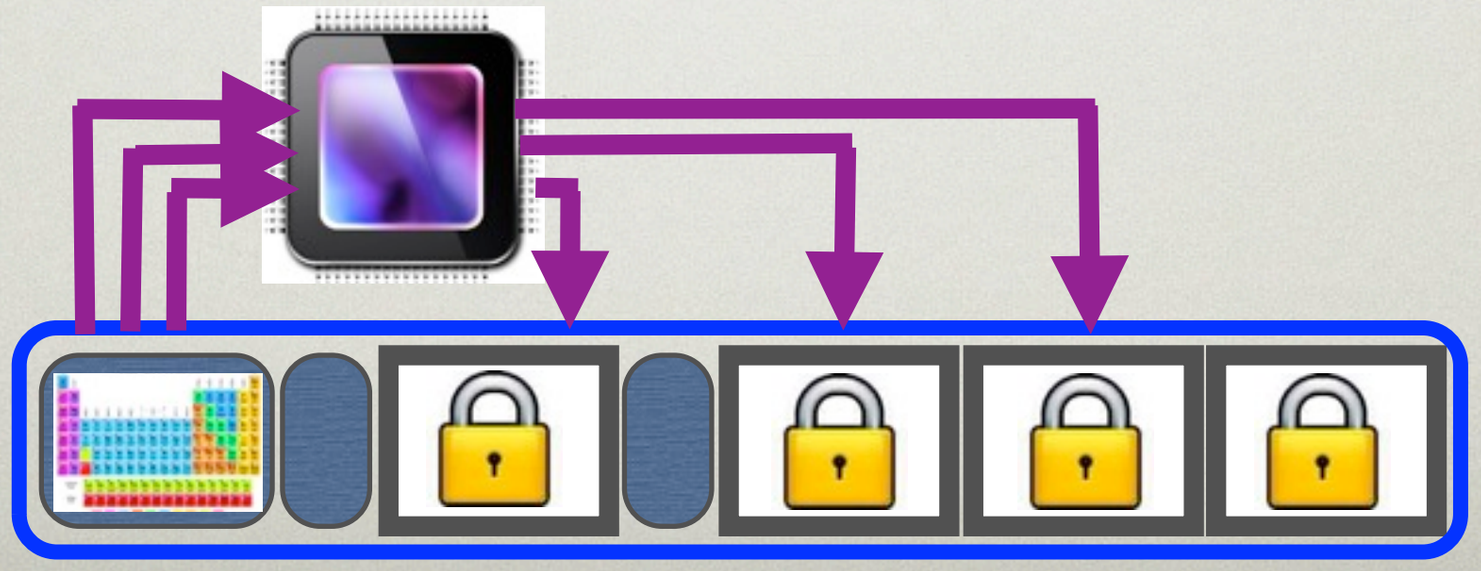


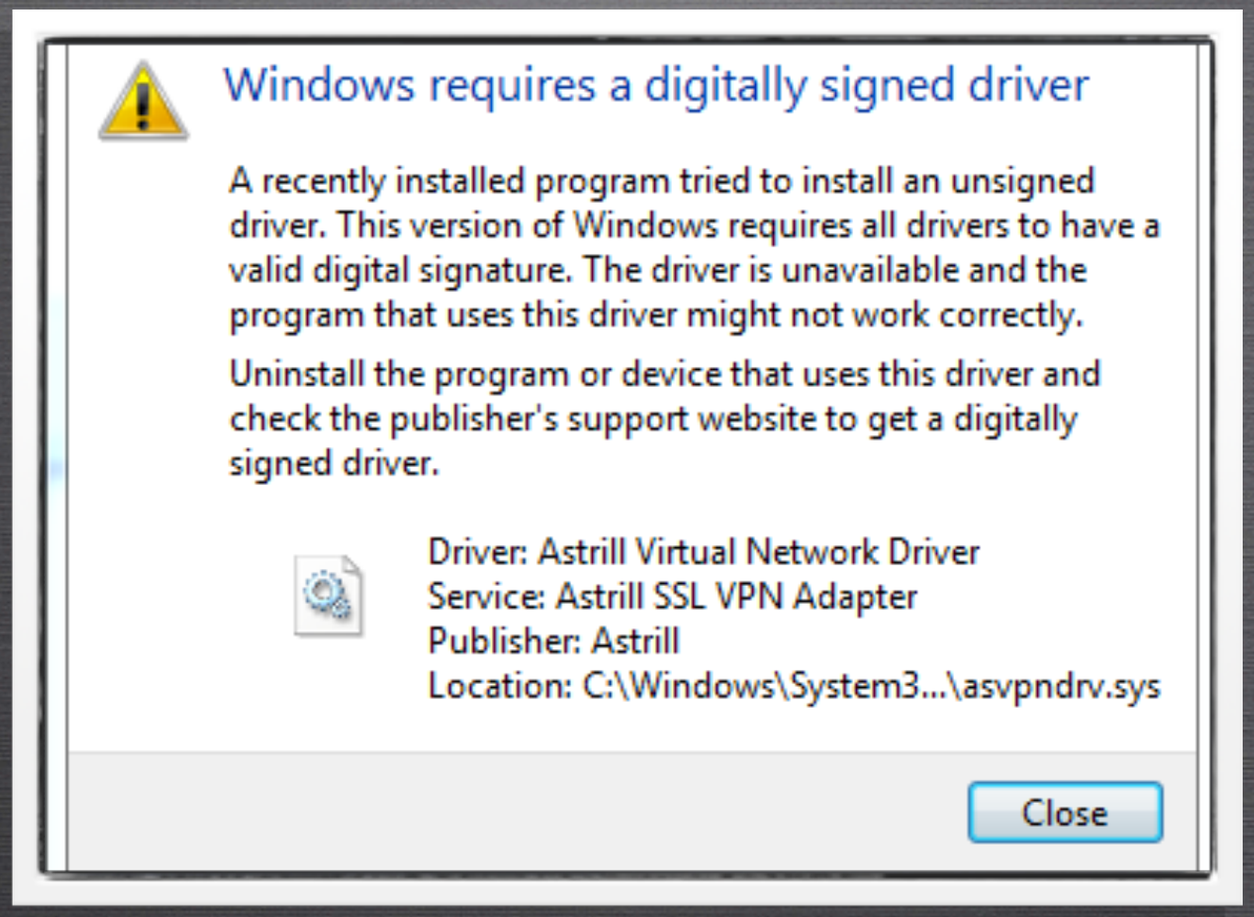
"Weird Machines"

- DWARF **exception** handling data + .eh_frame + Glibc = Turing machine (WOOT 2011)
- **Relocation** entries + dynamic **symbols** = Turing machine on process' address space
- GDT + IDT + TSS + page tables + **#PF** + **#DF** = Turing machine in ia32 (WOOT 2013)
- More coming :)

Validation IS verification

- Tables are trusted when **"valid"** <=>
  drive computation as expected
- **Validation** of tables is **static analysis** of computations they induce on parsers & loaders
- Code that **interprets** ("executes") tables must be simple enough to allow trust via static analysis



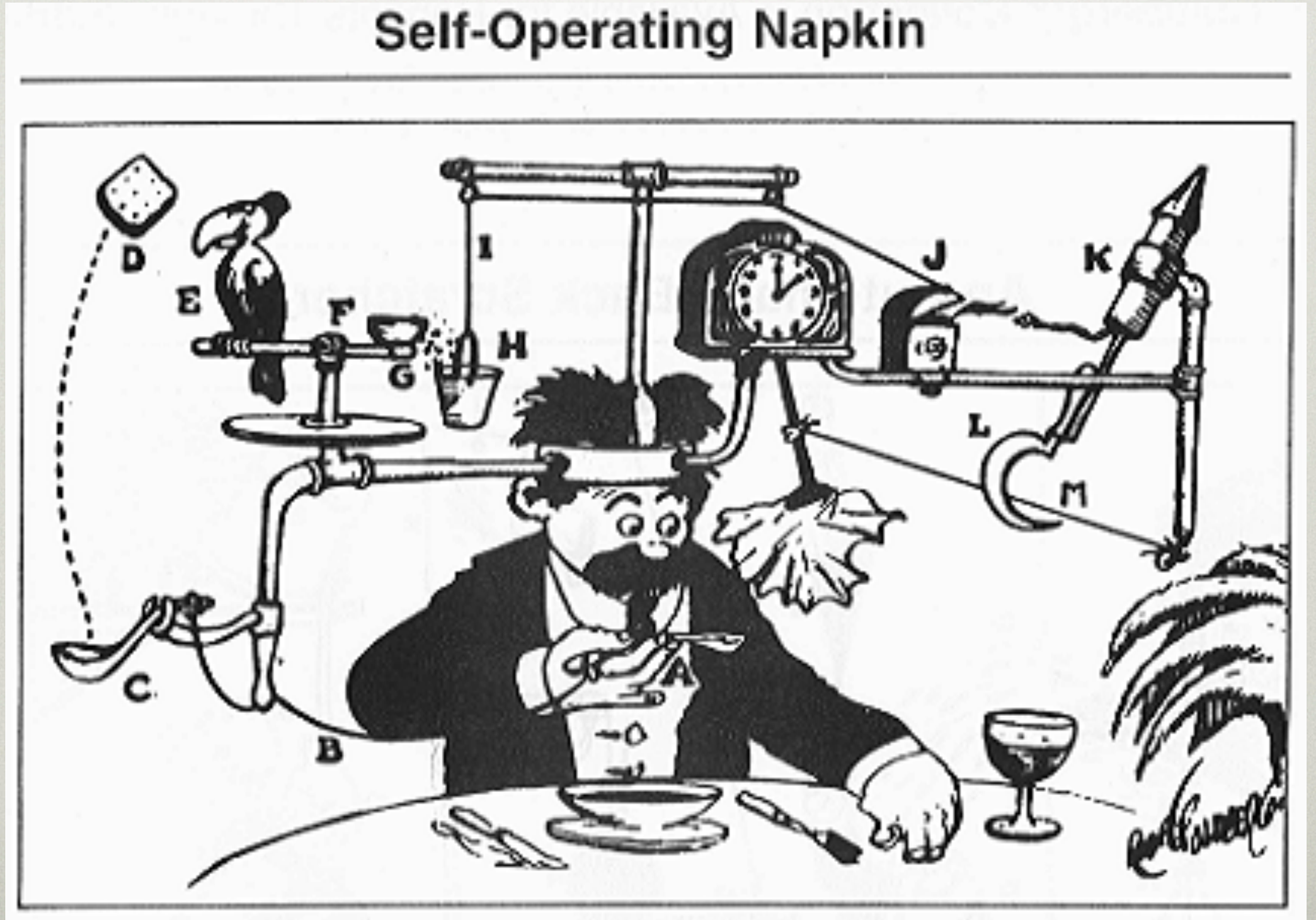


The machines of code signing

Code signing

- Code signing -> primary **trust evidence** for binaries:
  - "**trustworthiness** from **static measurements**"
- Developer/distributer digitally signs bytes in binary
  - Integrity and attribution
- Easy to implement poorly
- **It's not just an algorithm, it's a lifestyle**
  - Key management
  - Program in memory =/= program on disk
    - It is merely *influenced* by what is on disk
  - Many "**machines**" involved in verification
    - Parsers, interpreters, validators

Code signing machine composition



On trusting signature validation

- Are our machines **correctly implemented**?
- Do we understand what our machines are **capable of**?
- Do different machines **agree** on how to parse/**understand** input?
- Do the tables carry **correct** and **complete** data?
- Can we trust **transformations** made after this static analysis?
- **Enforcement**?

ELF case study

- Parsers
  - Signature and signature metadata
- Interpreters / translators
  - Binary -> hashes
- Validators
  - Validate certificates, signatures, hashes

ELF code signing

- Executable signing implementations
  - bsign (Marc Singer)
  - elfgpg (Bart Trojanowski)
  - elfsign (skape)
  - SignELF (Joe Fox)
  - signelf (Vivek Goyal, proposed to kernel developers)
  - elfsign (Solaris)
  - ^ **incompatible** with each other
- Kernel module signing (3.7+, evolved over time)
  - DigSig (until 2009)

Are our machines correctly implemented?

- XML parsers (in the case of Mach-0)
- ASN.1 BER parsing (easy as pie, right?)
- Most written in C/C++ (...)

How powerful are our machines?

- 29c3 “The Care and Feeding of Weird Machines in ELF Metadata”
- Metadata-driven root shell backdoor in ELF and Mach-O
- LOCREATE (skape)
  - unpacker written in PE metadata

Dropping a shell via an executable's metadata

ping backdoor in ELF

```
Symbol table '.sym.p' contains 90 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 000000000060dff0     8 FUNC    LOCAL  DEFAULT  UND

Relocation section '.rela.p' at offset 0xf3a8 contains 14 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend

00000060dfe0  002d00000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
00000060e9e0  004e00000005 R_X86_64_COPY     000000000060e9e0 __progname + 0
00000060e9f0  004b00000005 R_X86_64_COPY     000000000060e9f0 stdout + 0
00000060e9f8  005100000005 R_X86_64_COPY     000000000060e9f8 __progname_full + 0
00000060ea00  005600000005 R_X86_64_COPY     000000000060ea00 stderr + 0
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000001 R_X86_64_64                       0000000000000018
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000001 R_X86_64_64                       0000000000000018
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000005 R_X86_64_COPY                     0000000000000000
00000060eb40  000000000001 R_X86_64_64                       00000000000be6e0
00000060e028  000000000001 R_X86_64_64                       0000000000000000
00000060e218  000000000008 R_X86_64_RELATIVE                 0000000000401dc2
```

ping backdoor in Mach-O

```
> 0016720: 7400 5f67 6574 7569 6400 5f73 6967 6e61  t._getuid._signa
---
> 0016740: 005f 7373 6361 6e66 005f 6578 6563 6c70  ._sscanf._execlp
> 0016750: 0012 1212 125f 7374 7263 6872 005f 7374  ....._strchr._st
```

Can these be trusted?

```
bool is_elf (char* pb, size_t cb)
{
  if (cb < sizeof (HDR_ELF32))
    return false;

  check_byte_sex (pb);

  HDR_ELF32& header = *(HDR_ELF32*) pb;
  if (memcmp (header.rgbID, "\177ELF", 4) != 0
      || header.bitclass < 1
     ...
      || (   _v (header.cbEntryProgram)
     && _v (header.cbEntryProgram) != sizeof (PROGRAM_ELF32))
      || _v (header.cbEntrySection) != sizeof (SECTION_ELF32)
      || _v (header.iSectionNames) >= _v (header.cEntrySection))
    return false;
 // *** FIXME: I don't recall why we need more than a header test.

(goes on to check section/program headers)
```

(from bsign)

Parser differentials

- PKCS 7 crytographic message
- ELF, and its multiple interpretations
  - Sections v. segments
- Multiple ways to locate a section
  - **Is this the signature you are looking for?**

```
_debug("looking for sig section '%s'\n", modsign_note_section);
   for (loop = 1; loop < mvdata->nsects; loop++) {
      switch (sechdrs[loop].sh_type) {
      case SHT_NOTE:
         if (strcmp(mvdata->secstrings + sechdrs[loop].sh_name,
               modsign_note_section) == 0)
            mvdata->sig_index = loop;
         break;
      }
   }
```

(Kernel patch published by David Howells on 02 Dec, 2011)

Data completeness & correctness

- How much of the file is signed?

```
/* only look at interesting sections */
if( !sname || s->shdr->sh_type == SHT_NULL ) {
      //|| s->shdr->sh_type == SHT_NOBITS ) {
   ES_PRINT("skipping null section\n");
   continue;
}
/* skip over the .pgptab and .pgpsig sections */
if( !strcmp( sname, ".pgptab" )
      || !strcmp( sname, ".pgpsig" ) ) {
   ES_PRINT(skipping internal section\n")
   continue;
}
```

(from **elfgpg**)

Data completeness & correctness

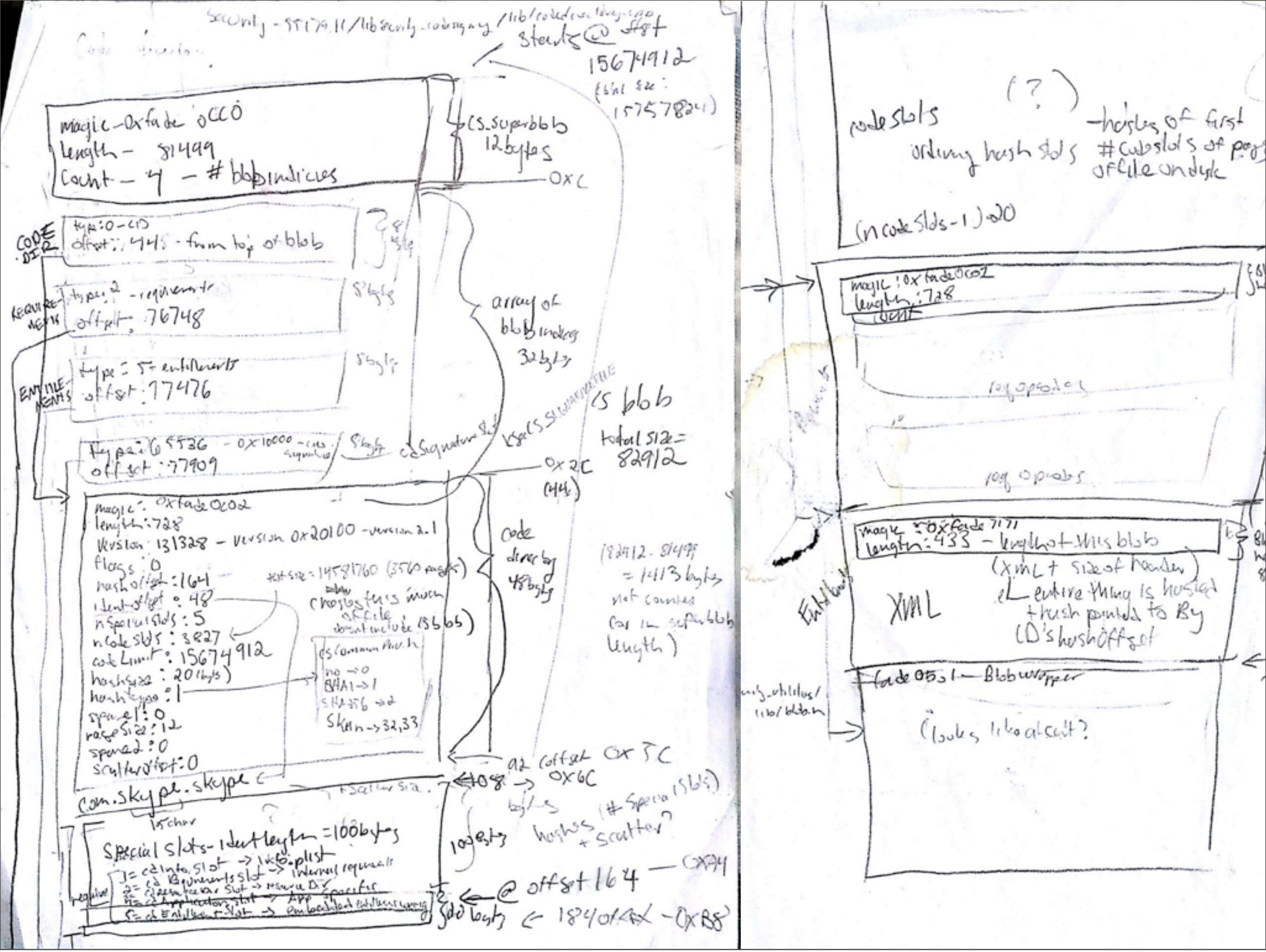
- We cannot sign the signatures, but they are loaded

```
// Include the ELF header, but with the number of sections set minus one,
// under the assumption that any binary having its checksum
// calculated will already have a signature header added to it.
// Yes, I can hear you screaming now.  This makes my life easier. :P
//
// Note that elfsign, the tool, always creates the signature section before
// calculating the checksum.
elfHeader = melf_elfGetRaw(melf);

numSections         = melf_elfGetSectionHeaderCount(melf);
sectionTableOffset = melf_elfGetSectionHeaderOffset(melf);

melf_elfSetSectionHeaderCount(melf, numSections - 1);
melf_elfSetSectionHeaderOffset(melf, 0);
```

(from **elfsign**)

What about Mach-O code signing?

```
:06:58 PM kernel: CODE SIGNING: cs_invalid_page(0x1000): p=1994[GoogleSoftwareUp] clearing CS_VA
:05:41 PM kernel: CODE SIGNING: cs_invalid_page(0x1000): p=2034[GoogleSoftwareUp] clearing CS_VA
:04:24 PM kernel: CODE SIGNING: cs_invalid_page(0x1000): p=2088[GoogleSoftwareUp] clearing CS_VA
```



magic - 0xfade 0cc0
length - 81499
count - 4 - # blob indices
cs superblob 12 bytes
0xC
starts @ offset 15674912
total size: 15757824
CODE DIR
type: 0 - CD
offset: 44 - from top of blob
REQUIREMENTS
type: 2 - requirements
offset: 76748
ENTITLEMENTS
type: 5 = entitlements
offset: 77476
type: 65536 - 0x10000
offset: 77909
array of blob indices 32 bytes
cs blob
total size = 82912
0x2C (44)
magic: 0xfade0c02
length: 728
version: 131328 - version 0x20100 - version 2.1
flags: 0
hash offset: 164
ident offset: 48
nSpecialSlots: 5
nCodeSlots: 3827
codeLimit: 15674912
hashSize: 20 (bytes)
hashType: 1
spare1: 0
pageSize: 12
spare2: 0
scatterOffset: 0
code directory 48 bytes
no -> 0
SHA1 -> 1
SHA256 -> 2
com.skype.skype
Special slots - ident length = 100 bytes
100 bytes
@ offset 164
codeSlots
(?)
(ncodeSlots - 1) x 20
magic: 0xfade0c02
length: 728
req opcodes
magic: 0xfade7171
length: 433 - length of this blob
XML
0xfade0b01 - Blobwrapper

Mach-O code signing data

Signature metadata

Internal Requirements (special bytecode)

Entitlements (XML)

Hashes

Signature (ASN.1 BER encoded derived from PKCS 7)

```
//require Apple's Perl interpreter
host => anchor apple and identifier com.apple.perl
designated => anchor /my/root and identifier com.bar.foo
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
...
```

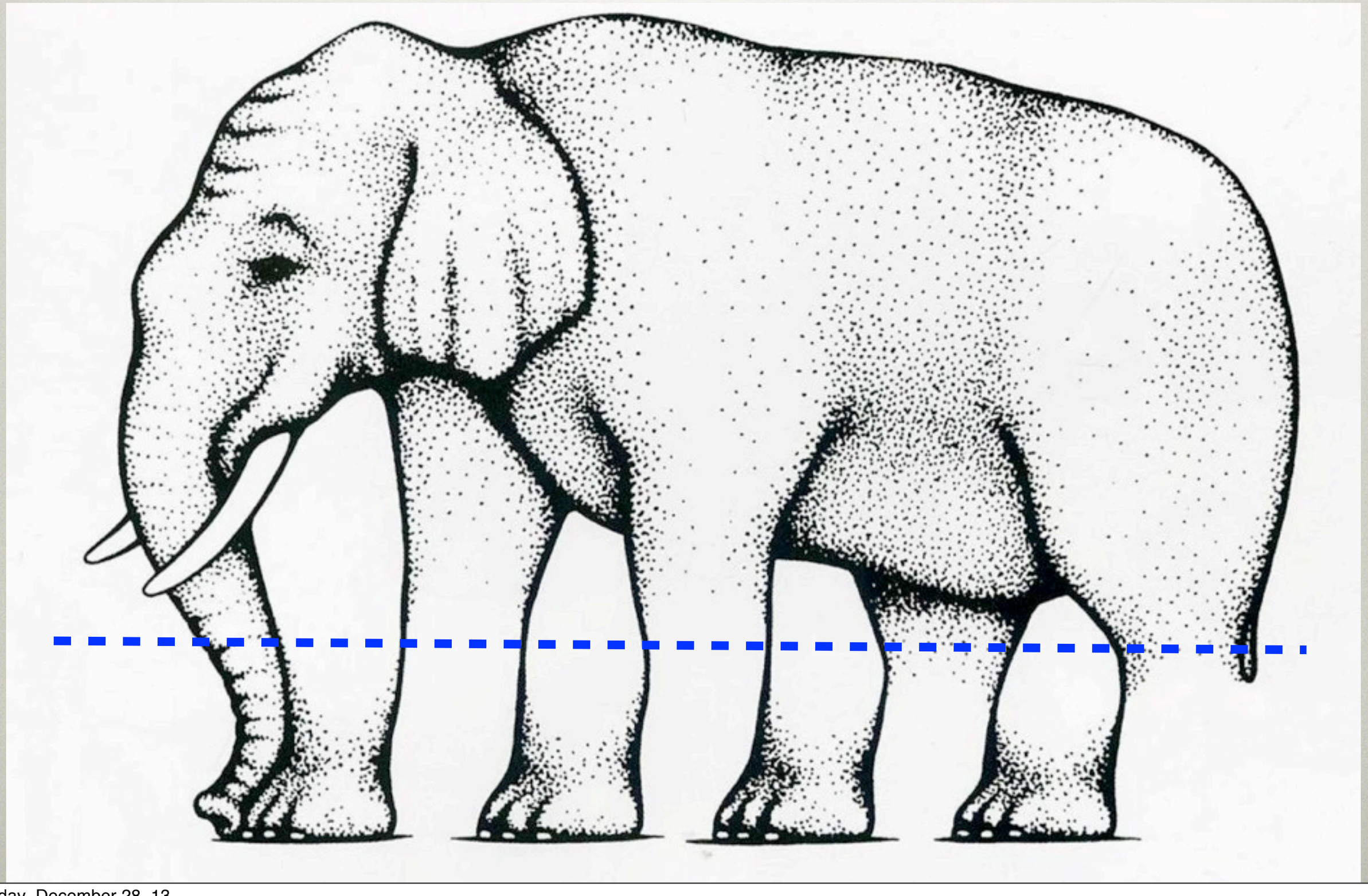
```
c8a47e06c8372dca11ab0bf03f121e1f69638a54
```

```
SignedData ::= SEQUENCE {
  version CMSVersion,
  digestAlgorithms DigestAlgorithmIdentifiers,
  encapContentInfo EncapsulatedContentInfo,
  certificates [0] IMPLICIT CertificateSet OPTIONAL,
  crls [1] IMPLICIT RevocationInfoChoices OPTIONAL,
  signerInfos SignerInfos }
```

*"You can't trust code that you did not **totally** create yourself"*

COROLLARY: You can't trust code that you did not totally **LOAD** yourself

Parser differentials



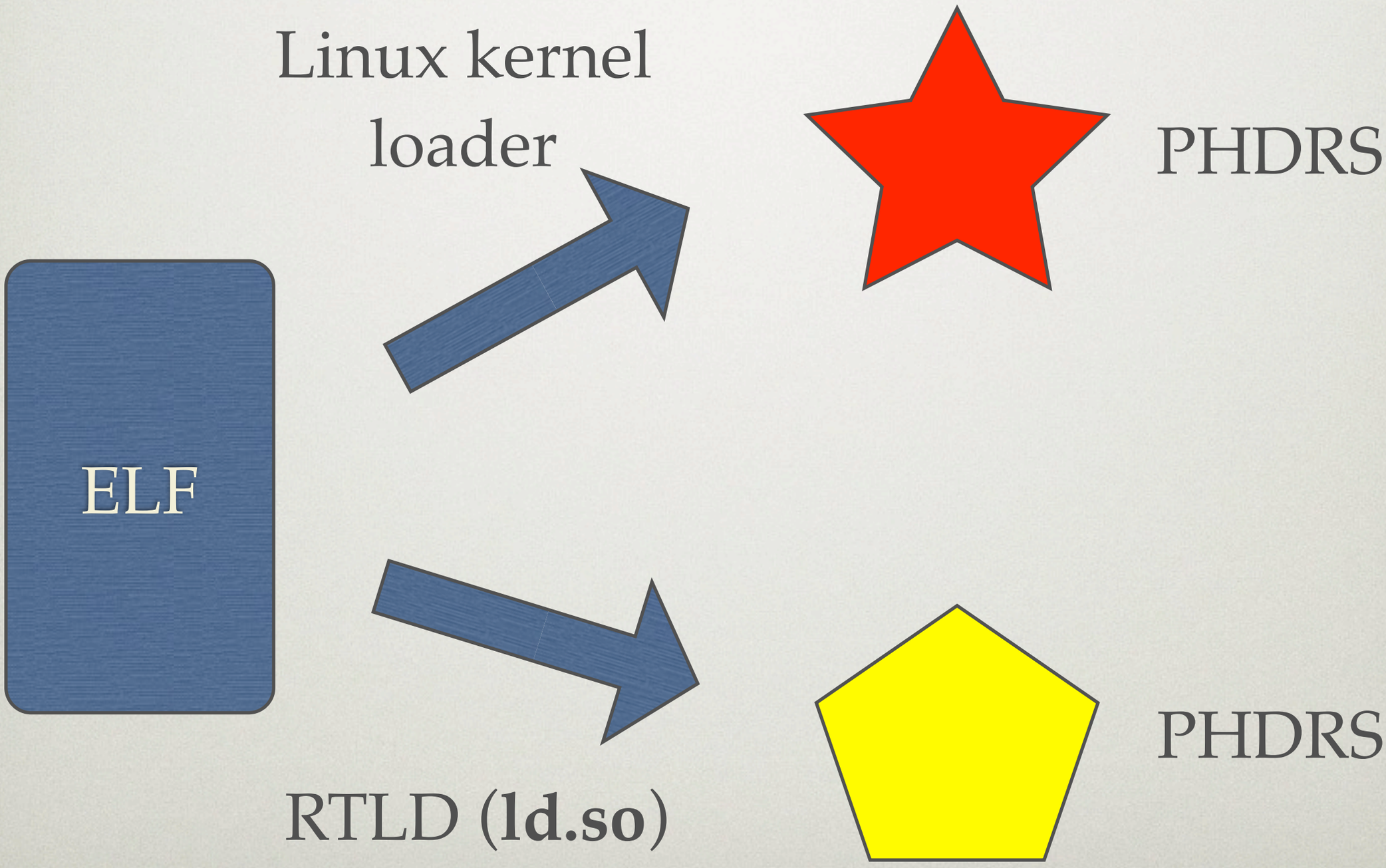
Chain of trust = chain of parsers

- **Parser differentials break chains of trust**
- Two views of the same data => confusion
  - **Android Master Key**, *http://**saurik**.com/id/{17,18,19}*
    - Package structure seen differently by signature verifier & installer (Java vs C++)
  - Mach-O signed loading
  - Linux kernel module signing
  - Even Linux **ld.so** vs. **kernel**!

Case study: ELF ABI chain

- How many **ELF parsers** in ABI toolchain?
  - Do they all see the same view of sections/segments?
- Kernel's **binfmt_elf loader** vs userland **ld.so**

Double the parsing, double the fun



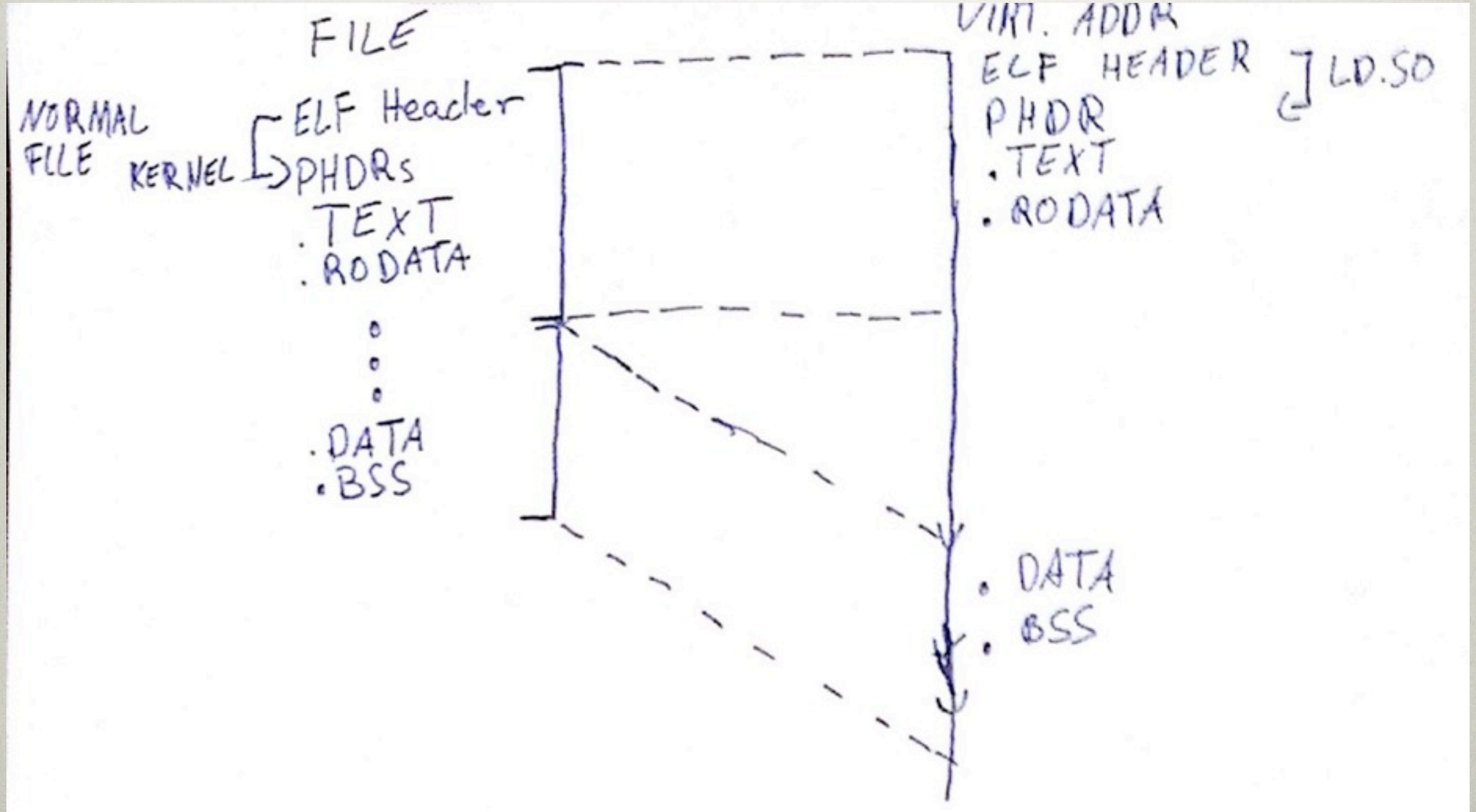
Program headers

- PT_LOAD maps bytes to address range
- Implementation: **mmap()**
- **Evad3rs** noted that (fixed) mmap **replaces** existing mappings
- **Order** of PT_LOAD is important, but that's not in the spec.
- Automaton, not just data

Does this look like a parser to you?

```
/*From binfmt_elf.c, Linux 3.4, GPLv2*/
elf_phdata = kmalloc(size, GFP_KERNEL);
kernel_read(bprm->file, loc->elf_ex.e_phoff, (char
*)elf_phdata, size)
for(phdrs...){
/*..., scans for PT_INTERP and PT_LOAD*/
if(phdr->p_type == PT_LOAD)
if (!load_addr_set) {
    load_addr = (elf_ppnt->p_vaddr - elf_ppnt-
>p_offset);
    load_addr_set = 1;
  }
}
NEW_AUX_ENT(AT_PHDR, load_addr + exec->e_phoff);
                    /*^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^FAIL*/
```

LD.SO PHDR


FILE
NORMAL FILE
KERNEL
ELF Header
PHDRs
.TEXT
.RODATA
.DATA
.BSS
VIRT. ADDR
ELF HEADER
PHDR
.TEXT
.RODATA
LD.SO
.DATA
.BSS

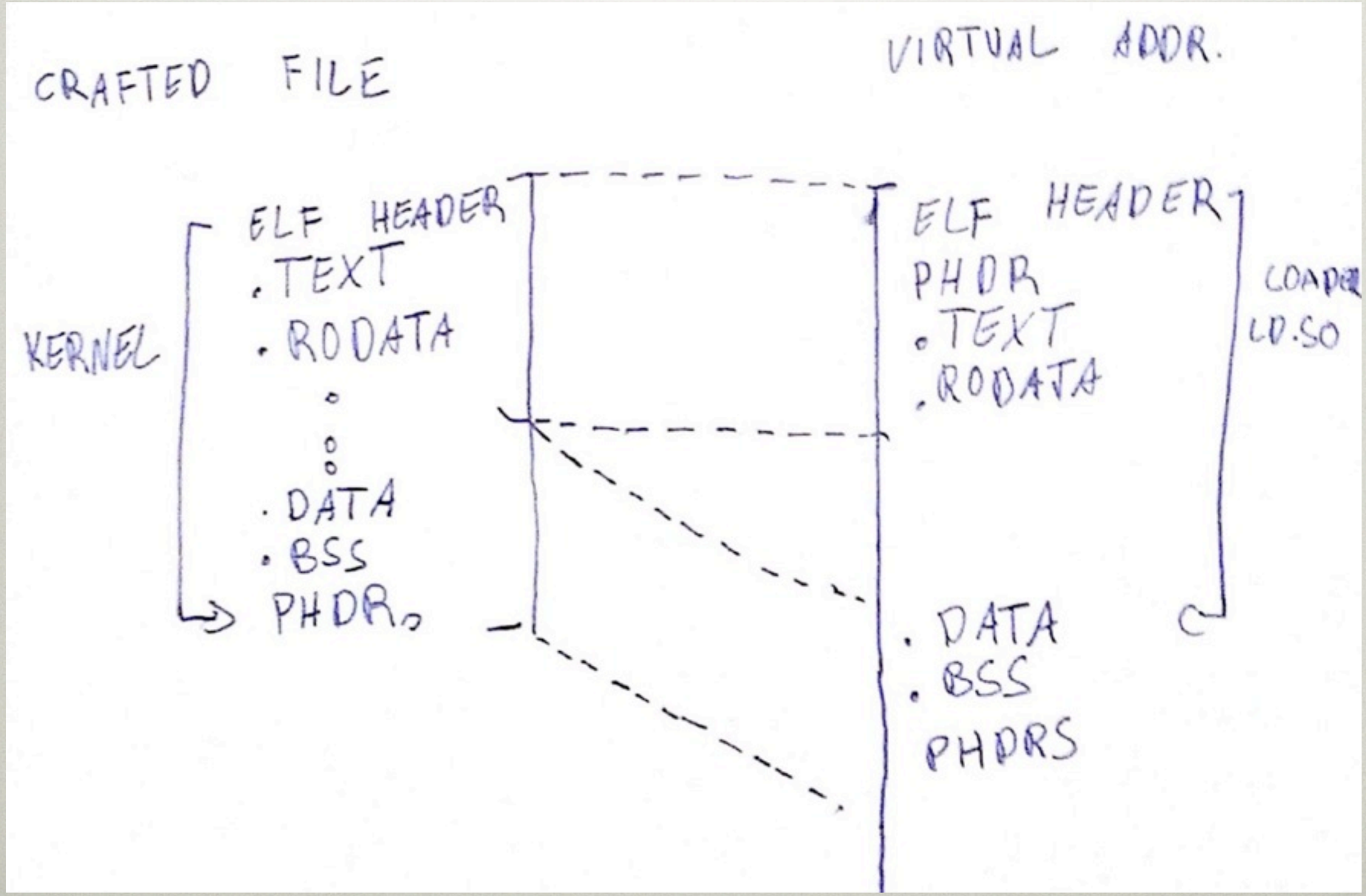
The birth of an Elf

1. Kernel reads PHDR table into buffer
2. mmap PT_LOADS
3. Loads PT_INTERP (ld.so)
4. writes (addr of first PT_LOAD)+ hdr.phoff to loader (“**AUX vector**”)
5. ld.so looks at aux vector and processes PHDR table

PHDR bug

- Works if program headers in **first segment**.
- Otherwise, points to **some other memory**
- We can “finger-paint” memory to our liking, and **ld.so** will use **different** PHDRS
- Hide your PHDRs, the reverse engineers are coming!

Crafted file



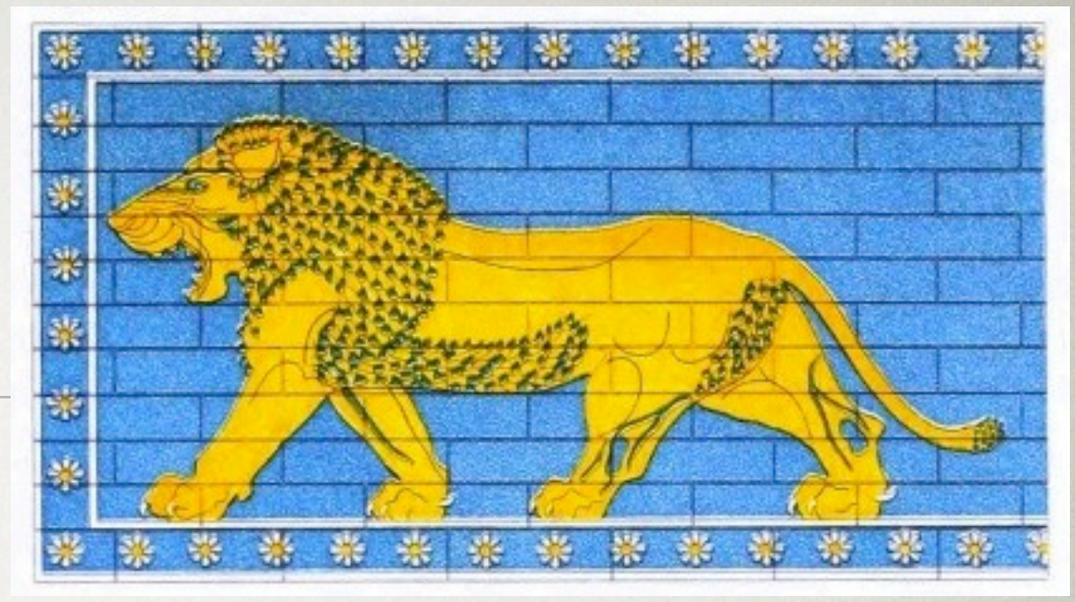
Demo: .so backdoor

- Fun Fact: you can execute .so files (try /lib/libc.so.6)
- Kernel parser for exec(), different parser in ld.so for .so
- POC loads a different library (libevil.so) when loaded by kernel

Rewriting programs in memory with ld.so?

- We control **ld.so**'s idea of all relevant sections: **GOT**, dyn symbols, ...
- **ld.so** resolves (what it thinks are) symbols, writes (what it thinks is) **GOT**
- Now we can **rewrite** a loading program via **only** crafted .dynamic + library **symbols**

"Trusting Trust" in Babel



- Trusting computers is not only about bugs! Bugs are part of a problem, **but not by far all of it**
- **Complex data formats >> bugs**
- **There is no "chain of trust" in Babel!**

Solutions for Babel

- **Squeeze complexity out of data** until it stops being "**code equivalent**"
  - UEFI? Software package formats?
- **Hobble unexpected computation** by blocking implicit flows
  - (see our ELFbac TR **http://elfbac.org/**)
- Use new hardware security primitives to isolate parsers

Thank you

IEEE SPW 2014 LangSec workshop,
May 18, 2014

Collocated with IEEE Secuity & Privacy Symposium 2014

http://spw14.langsec.org/