Weird Machines aren’t that weird: We are

Work-in-progress presentation

With inspirational thanks to:

- Erik Poll
- Dan O’Keeffe
- Julien Vanegue
- Lorenzo Cavallaro

Jordy Gennissen

S3Lab, Royal Holloway, University of London &
S2Lab, King’s College London (visiting)

Jordy.Gennissen@rhul.ac.uk

@jordygennissen

Outoftheheap.com

Weird machines

- **Unintended** machines/instructions (loosely translated from literature)[1][2]
  - E.g. as a result of invalid input being handled
- Used for reasoning about exploitation
- Existing definitions are lacking
  - Miss out on corner-cases
  - Or difficult to reason about

[1]Weird machines, exploitability, and provable unexploitability, Tomas Dullien, 2017, TETS

[2]The weird machines in proof-carrying code, Julien Vanegue, 2014, LANGSEC

Levels of abstraction

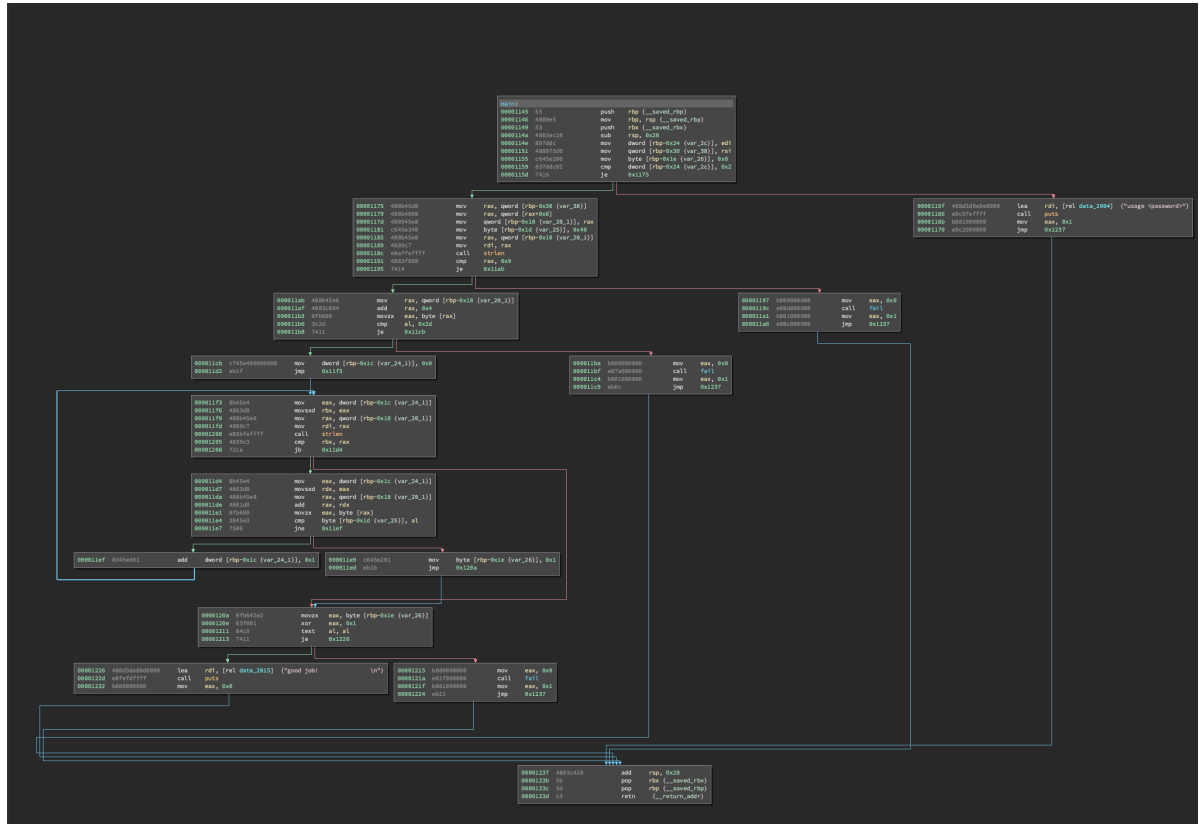
- Weird machines occur in translation
- Hask(ell) -> Python -> C -> asm -> objs -> executable -> process
- They're all our own construction!

Comparing Abstractions

- Source code program

```
if (Fn->isDeclaration() && !Fn->isVarArg()) {
  FunctionDependency *dep = FnDeps->getOrCreateDependency(Fn);
  dep->resetCurrentSens();
  dep->setSensitive(dep->getArgSize());
  for (size_t index = 0; index < dep->getArgSize(); index++) {
    if (dep->isSensitive(index)) {
      Argument *Arg = getArgumentFromIndex(Fn, index);
      Value *val = getValueFromArgument(Arg, Caller);
      addValue(val);
    }
  } // end for loop
  return;
} // end isDeclaration and not isVarArg
ValuePropagator::handleCallToFunctionBackwards(Caller, Fn);
```

- Executable (target) format



Incorrectness and Undefinedness

- A weird machine vulnerability is always a result of:
  - **Incorrect** behaviour (a.k.a. unsound or *inconsistent* behavior)
    - E.g. a compiler bug
  - **Undefined** behaviour, where anything can happen in practice
    - E.g. buffer overflow

Defining weird machines

A weird machine is the machine that appears when either:

1. the execution of the **target program** is **inconsistent** with the defined behaviour*;* or,

2. the **source** code ***program*** has **undefined behaviour.**

Example: integer overflow

$$\frac{e \mapsto v}{\langle x = e, s\rangle \to s[x \mapsto v]}$$

(a) Naive semantics

$$\frac{e \mapsto v}{\langle x = e, s\rangle \to s[x \mapsto v \ (mod\ 2^{64})]}$$

(b) 64-bit semantics

$$\frac{e \mapsto v,\ \ v < 2^{64}}{\langle x = e, s\rangle \to s[x \mapsto v]}$$

(c) 64-bit partial semantics

Figure 1: Different assignment semantics

@JordyGennissen, Royal Holloway

Example: Incorrectness

Source → Target

$$\frac{e \mapsto v}{\langle x = e, s\rangle \to s[x \mapsto v]}$$

(a) Naive semantics

$$\frac{e \mapsto v}{\langle x = e, s\rangle \to s[x \mapsto v \ (mod\ 2^{64})]}$$

(b) 64-bit semantics

$$\frac{e \mapsto v,\ \ v < 2^{64}}{\langle x = e, s\rangle \to s[x \mapsto v]}$$

(c) 64-bit partial semantics

Figure 1: Different assignment semantics

18446744073709552000 ← 2^{64} → 0

Example: Undefinedness

Target Source

$$\frac{e \mapsto v}{\langle x = e, s\rangle \to s[x \mapsto v]}$$

(a) Naive semantics

$$\frac{e \mapsto v}{\langle x = e, s\rangle \to s[x \mapsto v \ (mod\ 2^{64})]}$$

(b) 64-bit semantics

$$\frac{e \mapsto v,\ \ v < 2^{64}}{\langle x = e, s\rangle \to s[x \mapsto v]}$$

(c) 64-bit partial semantics

Figure 1: Different assignment semantics

$0 \leftarrow 2^{64} \rightarrow$ undef

Flaw Identification

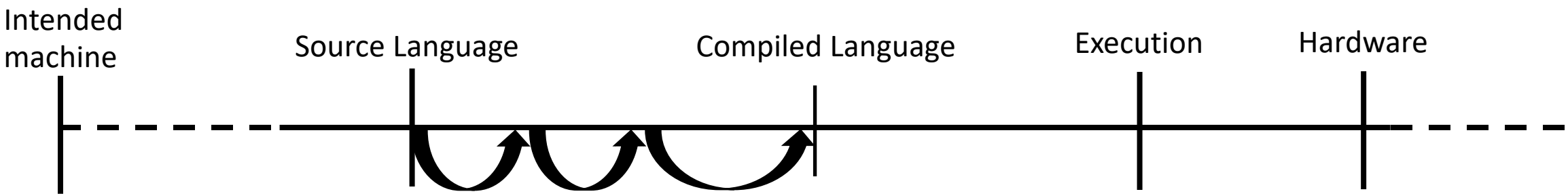
- If we found a weird machine:
  - What caused the weird machine?
  - Does this persist, or does it terminate or *self-heal*?
  - Alternatively, (how) can we program this weird machine?

The bigger picture

More than compilation

- Abstraction line can contain more than compilation

- Other steps can add
  - hardware flaws, side-channel attacks (right-hand side)
  - Logic flaws (left-hand side) / semantic issues
  - Compiler optimisations, etc.



Summarising

- Weird machines (and its vulnerability) are an abstraction discrepancy
  - Either through **incorrectness** or **undefinedness**

- We can observe this through small program transformations

- We hope to expand this notion along the full breadth of abstraction