Safety to the Weak!

Security Through Feebleness: An Unorthodox Manifesto

Rick McGeer, US Ignite
Outline

● What does malware exploit?
● Why can’t we find bugs?
● The Turing Hierarchy and the Verification Hierarchy
● Language Matters: A Cautionary Tale
● A Practical Example: Verifying Software-Defined Networks
● An Extension to Software-Defined Infrastructure
● A Wild Idea: Is Verifying a Turing Machine Really Undecidable?
● Closing Thoughts: Multi-Model Descriptions
How Does Malware Work?

- In general, exploits a bug
- Broad categories cover most:
  - Memory safety (e.g. buffer overflow and dangling pointer bugs)
  - Race condition
  - Insecure input and output handling
  - Faulty use of an API
  - Improper use case handling
  - Improper exception handling
  - Resource leaks, often but not always due to improper exception handling
  - Preprocessing input strings after they are checked for being acceptable.
Prevent Bugs, Stay Safe
Easier Said Than Done! Why Can’t We Find Bugs?

- All this guy’s fault!
- Turing’s Theorem: Determining if a Turing Machine Halts is Undecidable
  ⇒ Can’t tell if a line of code is executed
  ⇒ Can’t find a bug deterministically
- But: Do We Really Need Turing Machines for everything?
Turing-Complete Languages

✓ Easy to build
✓ Powerful
  ○ “I can do anything in language XYZ”
✗ Often more powerful than required
✗ Impossible to verify
The Turing Hierarchy And the Verification Hierarchy

<table>
<thead>
<tr>
<th>Model of Computation</th>
<th>Complexity of Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic-Free (Isomorphism Check)</td>
<td>Polynomial</td>
</tr>
<tr>
<td>State-Free</td>
<td>NP-Complete</td>
</tr>
<tr>
<td>Finite-State</td>
<td>Various from NP-Complete to P-Space Complete*</td>
</tr>
<tr>
<td>Turing Complete</td>
<td>Undecidable</td>
</tr>
</tbody>
</table>

* Powerful enough for many applications but largely unused in programming!
* Depends on exact variant of temporal logic being used
A Cautionary Tale: Verilog and VHDL

- Hardware (Chips) are finite-state
  - “Datapath” is combinatory (state-free)
  - “Control” is a collection of finite-state machines

- Many Verification Technologies!
  - BDD-based, powerful SAT tools...
  - Products from every major Electronic Design Automation vendor
  - Various startups over the years…

- But...hard to get designs described in appropriate languages

- Design-tool ecosystem has grown up around Verilog and VHDL...
Verilog and VHDL

- Both date from the 1980’s
  - Before Formal Verification in the early ‘90s
  - Era of 1980’s:
    - Mostly hand design (of logic, at least)
    - Only really prevalent EDA tool (for logic) was simulation
  - Grew up as simulator programming languages

- Verilog: hacked-together commercial product without clean semantics
  - Metaphors appealing to designers

- VHDL: Born from losing Ada effort
Verilog and VHDL

- **Mixed Simulator Control With Hardware Description**
- **Result**
  - Hard to tell what the hardware was
  - Couldn’t formally verify a design
  - Couldn’t even “synthesize” (aka, compile) it into hardware
- **Finally**
  - “Synthesis” semantics (aka, figure out what was really hardware)
  - “Synthesizable” subsets of languages
- **Imagine…**
  - “Computational” semantics of C/Java/Python/etc…
  - “Compilable” subsets of programming languages….
A Positive Tale: How Networking Became Verifiable

- Control Signaling
- Forwarding Tables (State-Free)
- Data Plane
- Control Plane
- Control Signaling
- Turing Machine
- Packets
- Traditional Switch

- Network Did Control + Data
- Ran autonomously (no external control)
- Verification Undecidable
Software-Defined Networking: Off With Its Head!

- External Controller
  - Network Forwards Packets, sends state information to controller

- SDN Switch
  - Requires External Controller
  - Verification In NP

- Data Plane
  - Forwarding Tables (State-Free)

- Packets
  - State Information
Key Points

- Done to make networks programmable (controller could be programmed) but also provided verification
- Network of Forwarding Tables isomorphic to state free logic network
  - Could verify network of forwarding tables with SAT engine
- Verification Methodology
  - Don’t verify controller -- verify its output before updating network
- Better: Safe Update
  - Update schedule that preserved invariants (aka, bug-free network)
- Still better: Network specifications often state-free
Anatomy of an SDN Ecosystem

Finite-state or state-free, verifiable!

State-free, verifiable!

Controller

Desired Network Specifications

Desired Network Specifications

Input verifiable, output verifiable ⇒ System verifiable

State-Free Forwarding Table

State-Free Forwarding Table
Extension to SDI Deployment

- SDI: Collection of VMs, containers, networks tuned to particular application
- Key problem: Configuring underlying infrastructure to accomplish task
  - Allocate VMs and Containers
  - Use SDN to configure networks
  - Use Orchestration Engines (Ansible, Heat, e.g.)
    - Finite-state or restricted-state
- Can we verify/what-ifs about SDI deployment and action?
- Key task: extract formal model from Ansible/Heat OR extend SDN specification languages to generate SDI Specs
Wild Speculation... Is Verifying Turing Machines Really Undecidable?

- Sure, but...
- Notice that a software-defined networks *still* has an unverifiable TM at its heart
- But network as a whole is verifiable
  - Verify the *inputs* to the TM and its outputs
- Can we do the same with programs?
  - Surround the program with a verifiable model and verify that
A “Verifiable” Program

Inputs → Program → Outputs

Unverifiable

Finite-State Input Generator → Finite-State Program → Finite-State Output Spec

Verifiable
A “Verifiable” Program

- Already informally done through `assert` statements
- But mixed with execution code
- Complicates execution and makes model hard to extract
- Not coupled to FV

- Significant Research Opportunity
A Final Word...

● We need to design computational environments/languages with an eye to verification
● Need a mix of models -- weak for verification, strong for execution
● Key is clean separability for each task
● Use runtime information to validate correspondence