On algorithms operating in adversarial conditions

Allison Bishop
IEX and Columbia University
A Bottom-Up Process of Knowledge Generation:

Stage 1: Simple building blocks
A Bottom-Up Process of Knowledge Generation:

Stage 2: Complexity Introduced
A Bottom-Up Process of Knowledge Generation:

Stage 3: Complexity Maturing
A Bottom-Up Process of Knowledge Generation

Stage 4: Complexity Celebrated
A Bottom-Up Process of Knowledge Generation:

Stage 5: Complexity Subsumed
A Bottom-Up Process of Knowledge Generation:

Stage 6: Building Blocks Rejected
We often think of knowledge as evolving like this:

When it actually goes more like this:
What are our building blocks for thinking about algorithms?

What implicit assumptions are inherent in our abstractions?

Are these assumptions reasonable? Are they avoidable?
“Algorithms are Recipes”

We conceptualize an algorithm as a sequence of steps.

If you follow the steps, you perform an intended function from inputs to outputs.

Algorithms -> functions is a many-to-one mapping.
The environment as ideal
Basic metrics of algorithm evaluation

- Correctness
- Running Time
- Memory Usage
- Parallel vs. Sequential
- Distributed vs. centralized
- Robustness to error (mostly in the distributed setting)
Testing An Algorithm

- We tend to think of testing implemented algorithms as testing function correctness and resource use:
  
  • Does the code give the right answer on average cases?
  • Does the code give the right answer on edge cases?
  • What is its run time on average cases?
  • What is it run time on worst cases?
  • What is it memory usage? Etc.

An "ideal" test suite can catch any incorrect function evaluations, no matter how rare.

*What we are neglecting here is the algorithm’s environment!*
The environment as non-ideal
The environment as malicious
What deeply ingrained assumptions are likely to be violated?

- External Inputs will conform to expectations
- Good Code will produce only good outcomes
- Code will run in isolation
- Code will run sequentially
“Algorithms are People Too”

- They get interrupted
- They get quoted out of context
- They are under surveillance

“You’d be lucky to get him to work for you”
We are *not* talking about AI...
On Interruption ...
Common Problem: User Input

<table>
<thead>
<tr>
<th>Account number</th>
<th>username</th>
<th>What happens when user input doesn’t conform to our expectations?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bah52</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>abb31</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mnd17</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>asifek4#$asdfs$!349$t45sdfg0%60$349...</td>
<td></td>
</tr>
</tbody>
</table>

> enter username with 3 letters and 2 numbers

> asifek4#$asdfs$!349$t45sdfg0%60$349...
Buffer Overflow/ SQL Injection

In our imaginations:

In reality:
### The Instruction Pointer

- **Instruction Pointer** keeps track of where the next instruction is stored in memory. Controls the flow of the program.

```c
int i = 0
int j = 0
i++
j = j + 5
int k = 0
int k = i + j
k++
return
```

...
Proposed Defense: Good Fences Make Good Neighbors

Idea: W XOR X

- dedicated, fixed memory portions for writing data versus executable code.

Limitation 1: this approach doesn’t make much sense in some contexts, e.g. websites where people have come to expect the flexibility of some kinds of executable code from untrusted sources.

Limitation 2: the implicit assumption underlying this defense is often false
On Quoting Out of Context...
Implicit Assumption => Explicit Attack Strategy

Undeniable Truth: Bad code can lead to bad behavior. This is why code injection attacks are scary.

(False) Converse: Bad behavior is the result of bad code.

Human analogy: Regular people can be manipulated into doing bad things.
Return-Oriented Programming
[S07,BRSS08]

• Idea is to exploit snippets of legitimate code to achieve an unintended outcome
• Can be done successfully when control flow is subverted, no code injection necessary
• In retrospect, unsurprising that executing “good” code in an incorrect order can have “bad” consequences.
Return-Oriented Programming

Gadgets of code that end with “return”:
- Can be strung together or malicious effects
On Surveillance ...
Side-Channel Attacks on Cryptography
[K96, KJJ97, BS97, BB05, … and many more]

In our imaginations:

In reality:
Proposed Defense: Leakage-Resilient Cryptography [CDHKS00, ISW03, MR04, ... and many many more]

Allow attacker to learn some limited information about the secret key

Try to prove security still holds
Leakage-Resilient Cryptography

Example guarantee:

- design algorithms for public key encryption so we can prove that:
  *even if attacker learns 100 bits of information about a 1000-bit secret key, the desired security properties still hold!*

- drawbacks: very difficult to decide if enough to capture real side-channel attacks, the changes to the algorithms that allow us to prove this might even exacerbate side channel attacks!
Meltdown and Spectre [LSGPHMKGYH17, KGGHHLMPSY17]

Speculative execution:

- Later instructions may start executing ahead to save time.
- Effects will "revert" for branches not taken.

Some effects may linger - like what is stored in caches!

This may render access checks ineffective!
A Nuclear Option: Program Obfuscation
[BGIRSVY01, ... and many many more]

Inputs → Obfuscated Program → Outputs

Extra information can be unnecessarily leaked by code. Only input-output behavior can be learned.
Possible Application: Software Patching

Distributing security-critical update to many users:

Hmm... so that's where the vulnerability was.

Patch itself may reveal an exploit that can be carried out on yet-to-be patched machines!

Obfuscation has the potential to fix this, and so much more!
A Paradigm Shift?

How we test algorithms today

How we will test algorithms tomorrow
Principles of Threat Modeling for Algorithms?

- Articulate clear, specific, narrow security goals
- Modular design: achieve high level security properties as a consequence of low level security properties
- Identify assumptions
- Test the viability of assumptions
- Model what happens when assumptions are violated:

Do modest violations of assumptions lead to modest or extreme violations of the security properties?
A new regime for testing code?

Expand testing of correctness with differing inputs to testing of security properties in differing environments:

- Shared hardware
- Adversarial inputs
- Speculative execution
- ...