# Reliable Software and Security Engineering with Unreliable Tools

*Lecture 1 - Introduction to Security*

David G Khachaturov[1, 2]

[1]St Catharine's College, University of Cambridge
[2]Computer Laboratory, University of Cambridge

**19th February 2026**

## Course Philosophy

This course is a **bridge** between:

- Classical security engineering
- The modern era of AI-assisted software development

**Core question:** How do we build reliable systems when our tools are unreliable?

*This course draws heavily from the work of the late Professor Ross Anderson*

## Unreliable Tools

Traditional view:

- Development tools are trusted “oracles”
- Compilers, IDEs, and libraries are assumed correct

Modern reality:

- Generative AI tools are **unreliable components**
- They require **guardrails**
- We cannot blindly trust their output, at least without **verification**

## AI Writes Most of the Code

Recent Y Combinator statistic:

**“AI writes 95% of the code for 25% of YC startups”**

Questions to consider:

- Who reviews this code for security?
- What assumptions are baked in?
- How do you debug what you didn’t write?

## Economic and Societal Drivers

*"Move fast and break things."*
*— Silicon Valley motto, popularised by Facebook*

Why do companies often prioritise features over security?

- **First-mover advantage** - get to market before competitors
- Security is "invisible" when it works
- Breaches are probabilistic; features are immediate

The result: **A legacy of insecure infrastructure**

- Technical debt accumulates
- Retrofitting security is expensive
- Users bear the cost of breaches

## Learning Objectives

By the end of this course, I aim to shift your focus from:

| **Syntax** | | **Semantics** |
|---|---|---|
| Writing code | $\longrightarrow$ | Verifying system intent |
| “Does it compile?” | $\longrightarrow$ | “Does it do what we mean?” |
| Trusting tools | $\longrightarrow$ | Testing assumptions |

## What is Security Engineering?

*"Security engineering is about building systems to remain **dependable** in the face of **malice**, **error**, or **mischance.**"*
*— Ross Anderson, Security Engineering*

Three threats to dependability:

1. **Malice**: intentional attacks
2. **Error**: bugs and mistakes
3. **Mischance**: accidents and plain ol' bad luck

## What is Security Engineering?

A system being *dependable* implies that it possess the following properties:

### The CIA Triad

- **Confidentiality**: preventing unauthorized disclosure of information
- **Integrity**: ensuring information is accurate, complete, and trustworthy, and hasn't been altered
- **Availability**: guaranteeing that systems and data are accessible and usable by authorized users when needed

## Evolution of Security Engineering

**Physical Security $\rightarrow$ Software-Defined Security**

- Locks, guards, and vaults $\rightarrow$ Encryption, access control, firewalls
- The principles remain the same
- The attack surface has expanded dramatically

## Reflections on Trusting Trust

Ken Thompson’s 1984 Turing Award Lecture posed a fundamental question:

> *“You can’t trust code that you did not totally create yourself.”*

**Even if** all the source code is open and audited:

- The *compiler* could be compromised
- The *environment* could be hostile
- The *hardware* could have been tampered with

What about when AI writes our code?

- Without auditing the *training data* (environment) or the *model weights* (compiler-ish), you rely on blind trust
- Susceptible now to both **deterministic** (malice) and **probabilistic** (hallucination) errors
- Very recent research on tampering with *hardware* to attack ML models

## The Syntax Trap

AI is excellent at generating **syntactically correct** code:

- Compiles without errors
- Follows language conventions
- "Looks right", because this is what the models are optimised to produce

But AI, unless paired with specific tool-calling and verification, lacks understanding of **security semantics**:

- What does this code *mean*?
- What are the security implications?
- Are there hidden vulnerabilities?

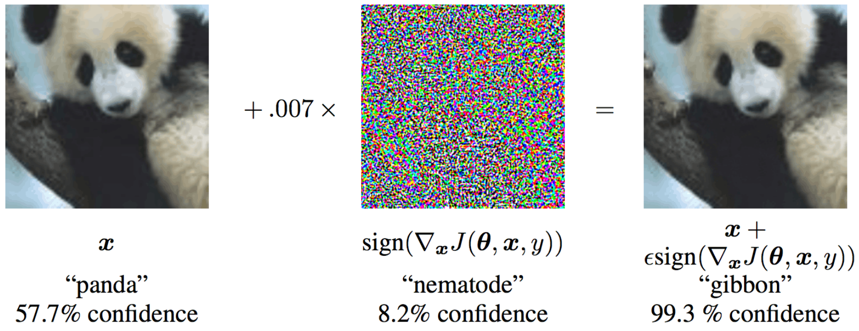
## Increasing Dependence on Brittle Systems

As we integrate AI into engineering:

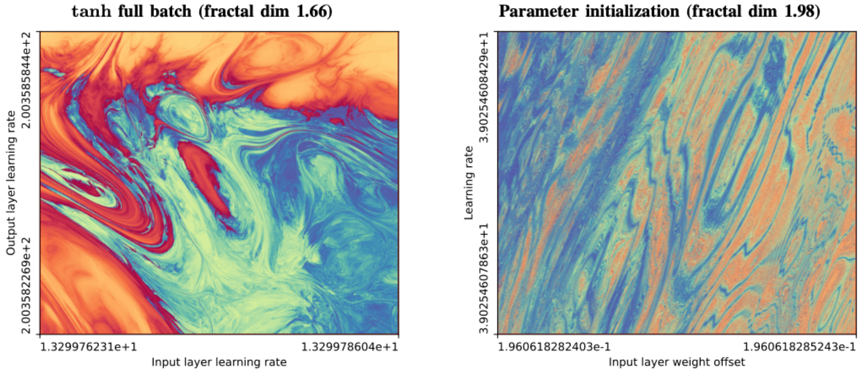
- Systems become more **capable** but also more **brittle**
- Failures are often sudden and catastrophic
- We don't always understand *why* something works

### Definitions

- **Brittle failure:** A system that works perfectly until it doesn't work at all.
- **Graceful degradation**: A system that operates at a reduced (and ideally, *predictable*) level of performance after some component fails.


$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

FGSM Panda, courtesy of Goodfellow et al.
"*Explaining and Harnessing Adversarial Examples*". 2014. ICLR.



Courtesy of Sohl-Dickstein J.

"*The boundary of neural network trainability is fractal*". 2024.

The implication here is that formal verification is intractable when the failure boundary is fractal. *We will cover this in more detail in Lecture 3.*

## Defining the Attack Surface

The **attack surface** is the boundary where an adversary can interact with, and potentially exploit, a system.

Components of an attack surface:

- Network interfaces and APIs
- User inputs and data entry points (e.g. *training data*)
- Authentication mechanisms
- Third-party dependencies (i.e., *supply chain* attacks)
- Model weights
- *AI-generated code* itself

## Introducing Alice and Bob

### Standard notation for security protocols

- **Alice (A)** - initiator
- **Bob (B)** - responder
- **Eve (E)** - eavesdropper (passive)
- **Mallory (M)** - malicious attacker (active)

Example notation:

$$A \rightarrow B : \{P\}_{K_{AB}}$$

*“Alice sends message P to Bob, encrypted with their shared key”*

## Theoretical Attacks: Man-in-the-Middle

**Intended communication:**

$$A \to B : P$$

**With Mallory intercepting:**

$$A \to M : P$$
$$M \to B : P'$$

Mallory intercepts, reads, and potentially modifies messages (in this case, $P$) between Alice and Bob.

**Example:** Using a bank website over public Wi-Fi may leave you vulnerable.

## Theoretical Attacks: Man-in-the-Middle

**Fixed protocol:**

$$B \to A : K_B, \mathsf{Cert}_B$$
$$A \to B : \{P\}_{K_B}$$

This is fixed via **Authentication** (Confidentiality + Integrity).

- Confidentiality is ensured via *encryption*, i.e., $\{\dots\}_{K_B}$
- Integrity provided by a trusted third party (e.g., a Certificate Authority) to verify identities

This is how internet traffic gets authenticated, via, for example, TLS.

## Theoretical Attacks: Replay Attack

**Original valid message:**

$$A \to B : \{P\}_{K_{AB}}$$

**Mallory captures and replays:**

$$M \to B : \{P\}_{K_{AB}}$$

Mallory captures the encrypted string. She cannot read $P$, but she can send the string again. Bob decrypts it and assumes it’s a valid new request because there is no timestamp/nonce to distinguish it.

## Theoretical Attacks: Replay Attack

**Fixed protocol:**

$$A \rightarrow B : \{P, T_1\}_{K_{AB}}$$

To prevent this, Alice must include a timestamp $T_1$ that can be checked by Bob.

**Example:** Unlocking your car/garage via a fob.

## Theoretical Attacks: The Confused Deputy

**Definition:** A privileged program (the “*Deputy*”) is innocently tricked by a lower-privileged user (*Mallory*) into misusing its authority

**“Valet” Analogy**

- Deputy: A Valet who holds the keys to every car.
- Attacker: Mallory (who owns a Ford) tricks the Valet into retrieving Alice’s Ferrari.
- Outcome: The Valet acts with valid authority (they have the keys) but invalid intent (they were fooled).

**Examples:**

- Classical: A compiler writing to a restricted system file.
- LLM Prompt Injection: The AI has “excessive permissions” (read email, delete files) and is confused by a malicious prompt into using them against the user’s interest.

## Classical Attack Surfaces

Traditional vulnerabilities:

- **Buffer overflows** — writing beyond allocated memory
- **SQL injection** — malicious database queries
- **Network vulnerabilities** — protocol weaknesses, unencrypted traffic

These are well-studied and have known mitigations.

## Modern/AI-Assisted Attack Surfaces

New vulnerabilities introduced by AI:

- **Training data poisoning** — corrupting the model's learning
- **Prompt injection** — manipulating AI behaviour through inputs
- **AI-generated vulnerabilities** — insecure code from models

These attack surfaces are still being understood.

## Threat Modelling

A systematic approach to security:

1. **Identify assets** — What are you protecting?
2. **Identify adversaries** — Who might attack?
3. **Identify attack vectors** — How might they attack?
4. **Prioritise threats** — What is most likely/damaging?
5. **Design defences** — How do you mitigate?

*We will explore this more in later lectures.*

## The Asymmetric Cost of Errors

| Stage | Cost | Example |
|---|---|---|
| Design | 1x | *Architecture:* Deciding to use Rust vs C |
| Code | 10x | *Linter:* Flagging `verify=False` or hardcoded keys |
| Test | 100x | *Red Team:* Fuzz-testing reveals a crash/injection |
| Prod | **1,000x+** | *Crisis:* e.g., CrowdStrike (2024). Lawsuits, emergency patching follow |

This is an example of the **Shift-left Principle**: fixing a vulnerability becomes exponentially more expensive the later it is discovered.

Hidden non-monetary costs of errors:

- **Human:** Developer burnout from “firefighting” brittle systems.
- **Liability:** Negligence lawsuits (e.g., Air Canada’s chatbot).
- **Reputation:** Trust is hard to gain, but instant to lose.

AI tools accelerate the *Code (10x)* but often skip *Design (1x)*, pushing invisible risks into *Production (1000x+)*.

## Conclusion

As tools become less reliable, foundational CS skills become **more critical**:

- **Systems design** — understanding how components interact
- **Formal verification** — proving correctness
- **Threat modelling** — anticipating attacks
- **Critical thinking** — questioning assumptions

**Verify intent.**

## Recommended Reading

1. Chapters 1 and 3 of **Anderson, R.** *Security Engineering* (3rd Ed.)
2. **Thompson, K.** *Reflections on Trusting Trust* (1984)

## Questions?

*Course page with feedback form and recommended reading*

**David G Khachaturov**

`dgk27@cam.ac.uk`

*Next week: Death of Syntax*