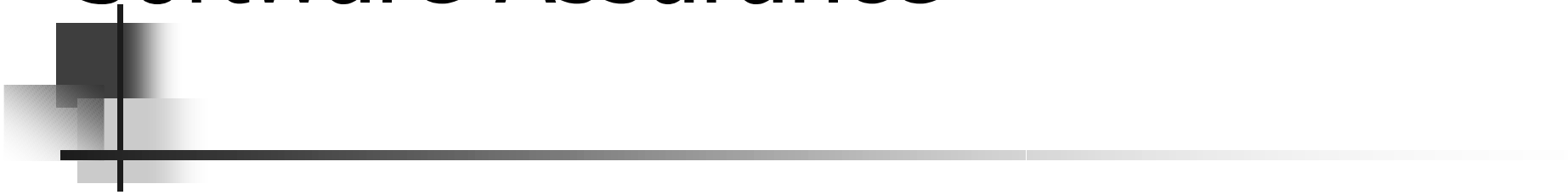
Static Analysis and Software Assurance



David Wagner
U.C. Berkeley

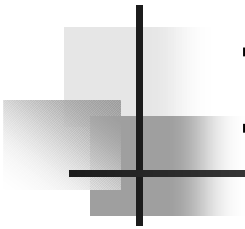
The Problem

- Building secure systems is hard
  - 2/3 of Internet servers have gaping security holes
- The problem is buggy software
  - And a few pitfalls account for many vulnerabilities

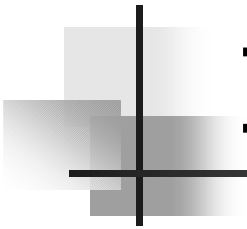
The Problem

- Building secure systems is hard
  - 2/3 of Internet servers have gaping security holes
- The problem is buggy software
  - And a few pitfalls account for many vulnerabilities

- Challenge: Improve programming technology
  - Need way to gain *assurance* in our software
  - Static analysis can help!



In This Talk…



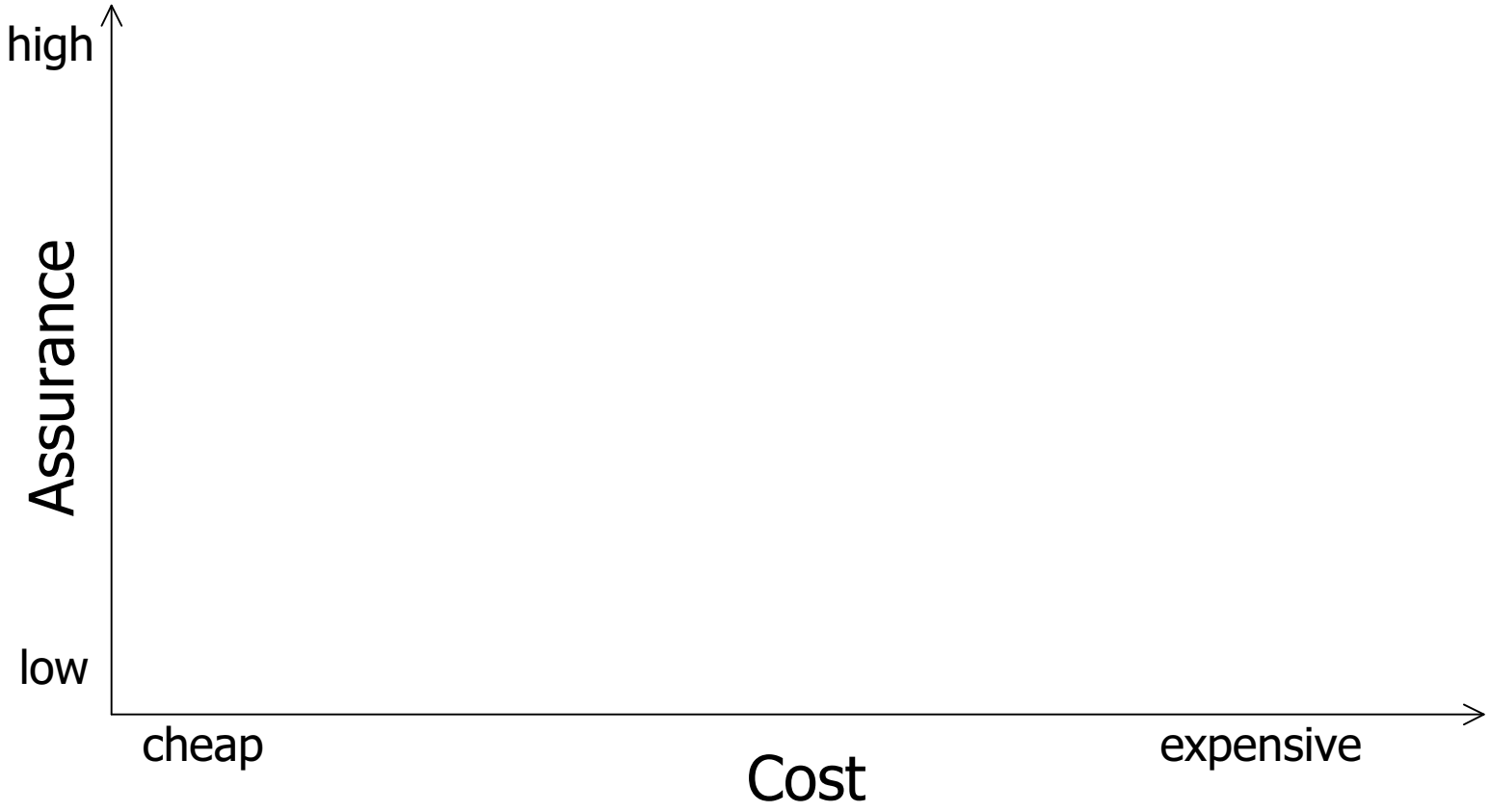
In This Talk…

- I *won't* discuss:
  - Cryptographic protocols
  - Information flow, covert channels
  - Mobile and malicious code

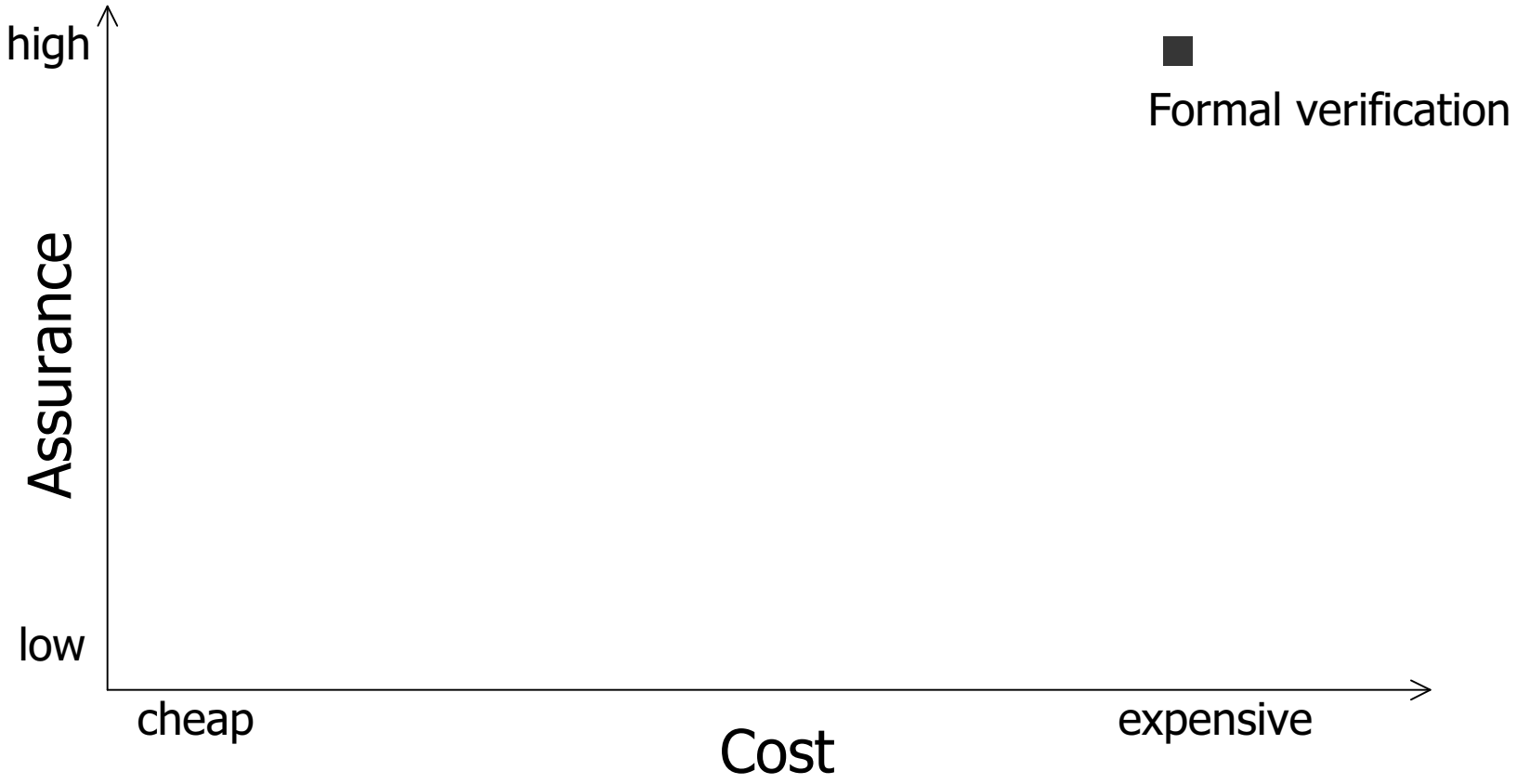
In This Talk…

- I *won't* discuss:
  - Cryptographic protocols
  - Information flow, covert channels
  - Mobile and malicious code
- I *will* discuss:
  - Software security
  - Three examples of interesting research challenges

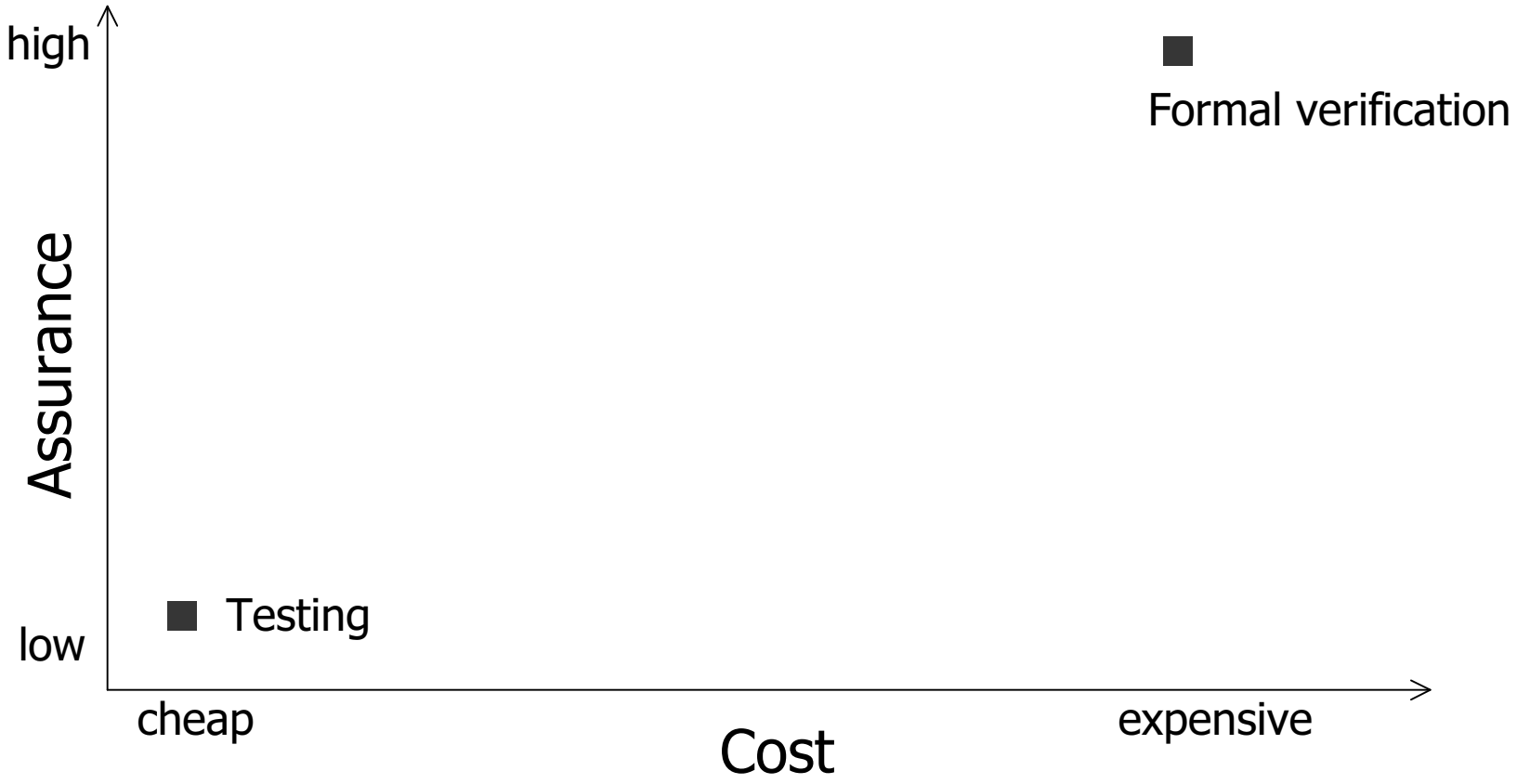
Existing Paradigms



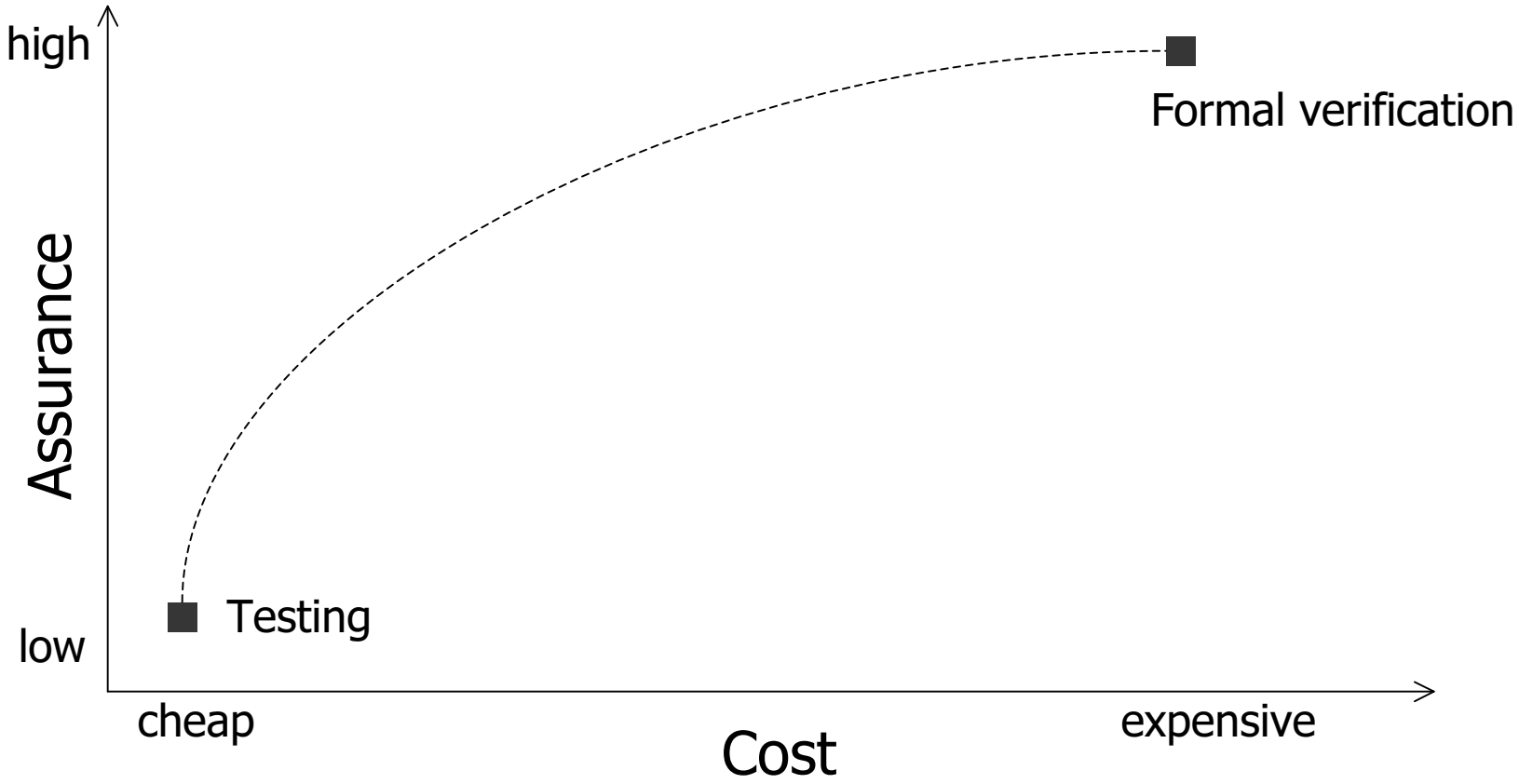
Existing Paradigms



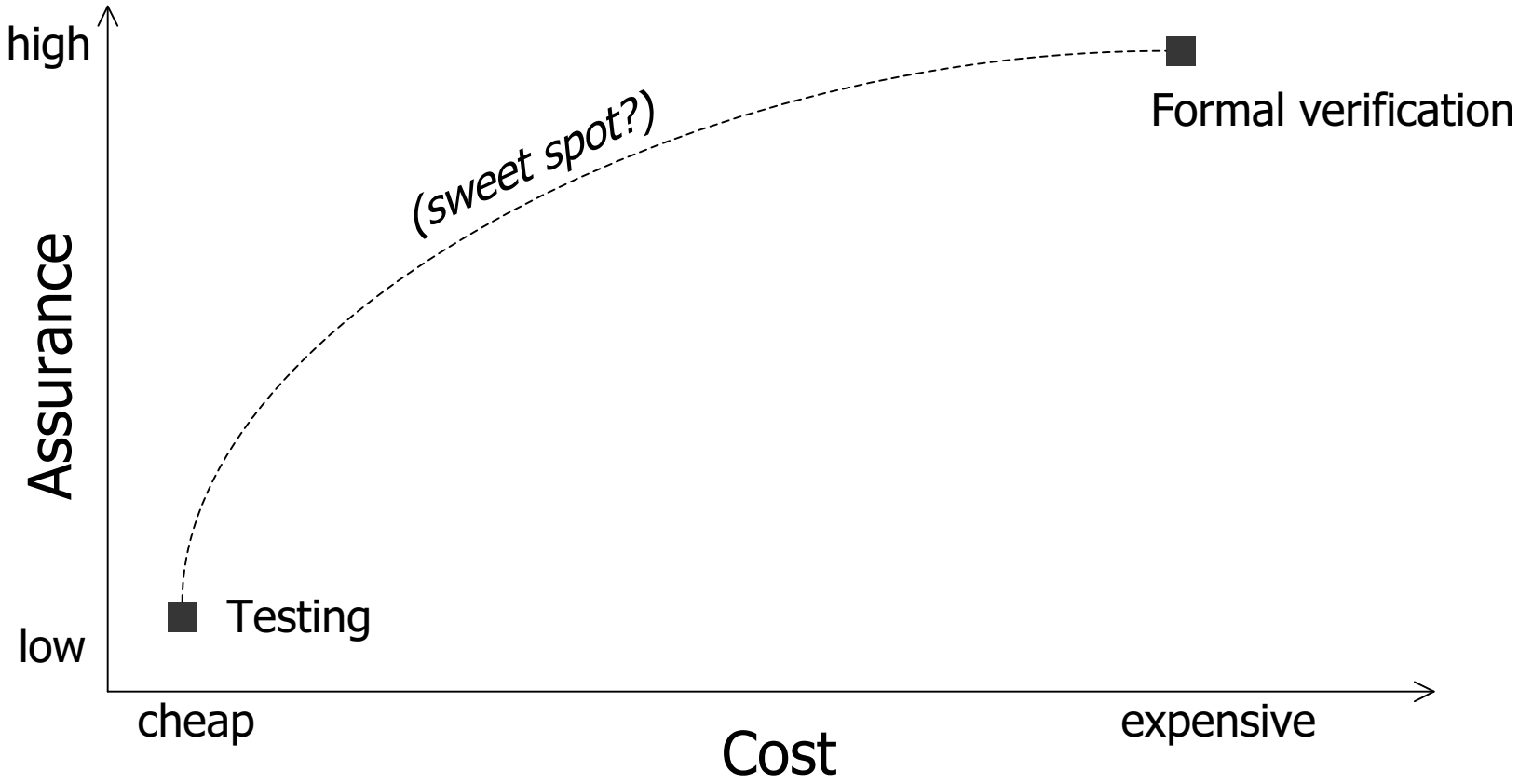
Existing Paradigms

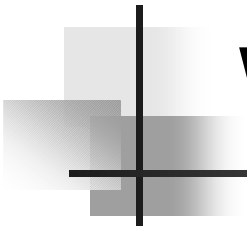


Existing Paradigms

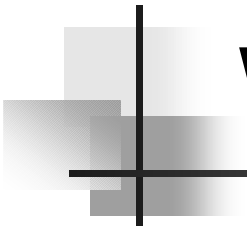


Existing Paradigms



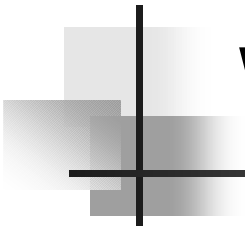


What Makes Security Hard?



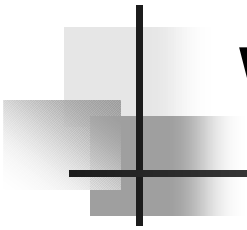
What Makes Security Hard?

- Security is hard because of…



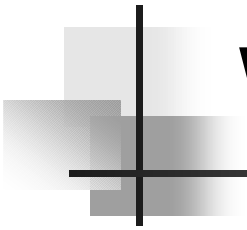
What Makes Security Hard?

- Security is hard because of…
  - language traps (buffer overruns)



What Makes Security Hard?

- Security is hard because of…
  - language traps (buffer overruns)
  - privilege pitfalls



What Makes Security Hard?

- Security is hard because of…
  - language traps (buffer overruns)
  - privilege pitfalls
  - untrusted data

What Makes Security Hard?

- Security is hard because of…
  - language traps (buffer overruns)
  - privilege pitfalls
  - untrusted data

… and many others that I won't consider in this talk

Plan of the Talk

- Security is hard because of…
  - language traps (buffer overruns)
  - privilege pitfalls
  - untrusted data

… and many others that I won't consider in this talk

Buffer Overruns

- An example bug:

```
char buf[80];

hp = gethostbyaddr(...);
strcpy(buf, hp->hp_hname);
```

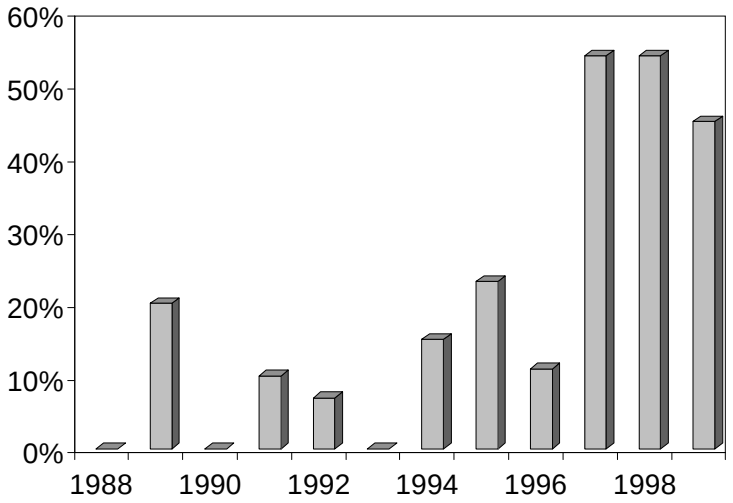
Buffer Overruns

- An example bug:

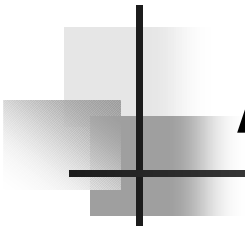
```
char buf[80];

hp = gethostbyaddr(...);
strcpy(buf, hp->hp_hname);
```

- Accounts for 50% of recent vulnerabilities

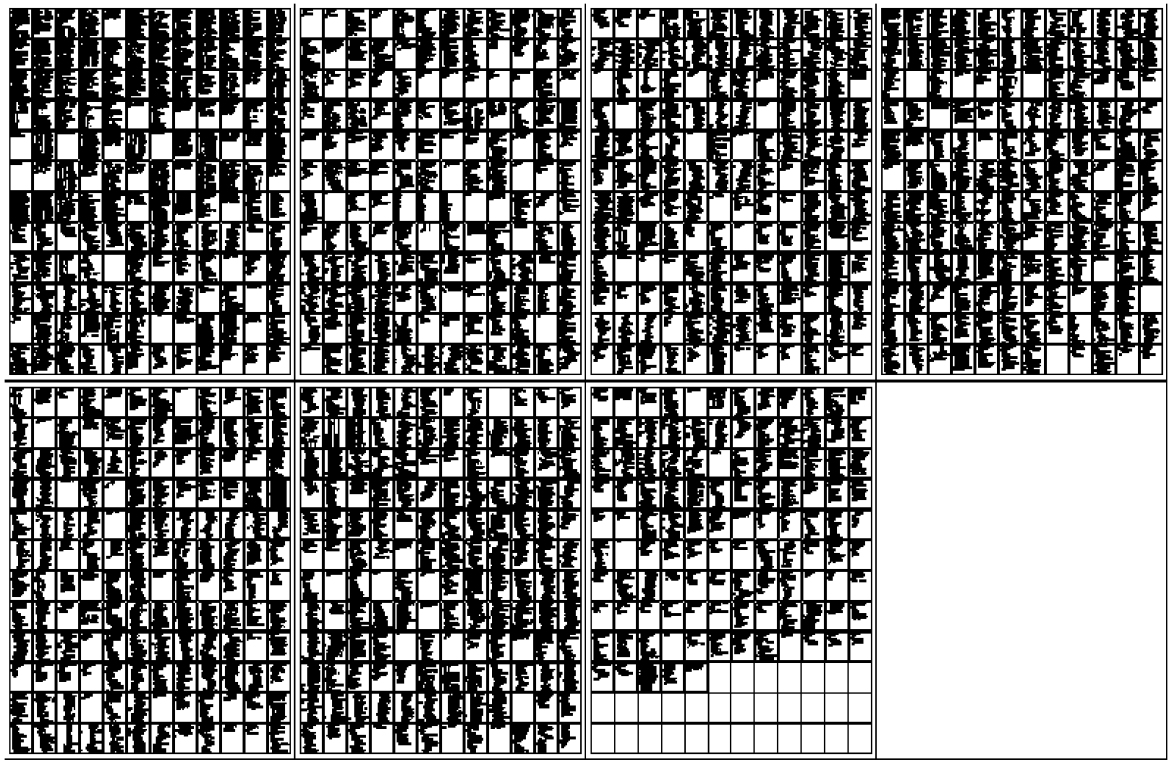


Percentage of CERT advisories due to buffer overruns each year



A Puzzle: Find the Overrun

A Puzzle: Find the Overrun



Static Detection of Overruns

- Introduce implicit variables:
  - alloc(buf) = # bytes allocated for buf
  - len(buf) = # bytes stored in buf
  - Safety condition: len(buf) ≤ alloc(buf)

Static Detection of Overruns

- Introduce implicit variables:
  - alloc(buf) = # bytes allocated for buf
  - len(buf) = # bytes stored in buf
  - Safety condition: len(buf) ≤ alloc(buf)
- Check safety using range analysis
  - Generate range constraints, and solve them

    `y := x+5;` $\Longrightarrow$ $X + 5 \subseteq Y$
  - New algorithm for solving range constraints

    $\mathrm{E} ::= n \mid \mathrm{E} + n\,V \qquad \mathrm{C} ::= \mathrm{E} \subseteq V \qquad n \in \mathrm{Z}, V \in \mathrm{Vars}$
  - Warn user of all potential violations

Current Status

Current Status

- Experimental results
  - Found new bugs in sendmail (30k LOC), others
  - Analysis is fast, but many false alarms (1/kLOC)

see also Dor, Rodeh, Sagiv

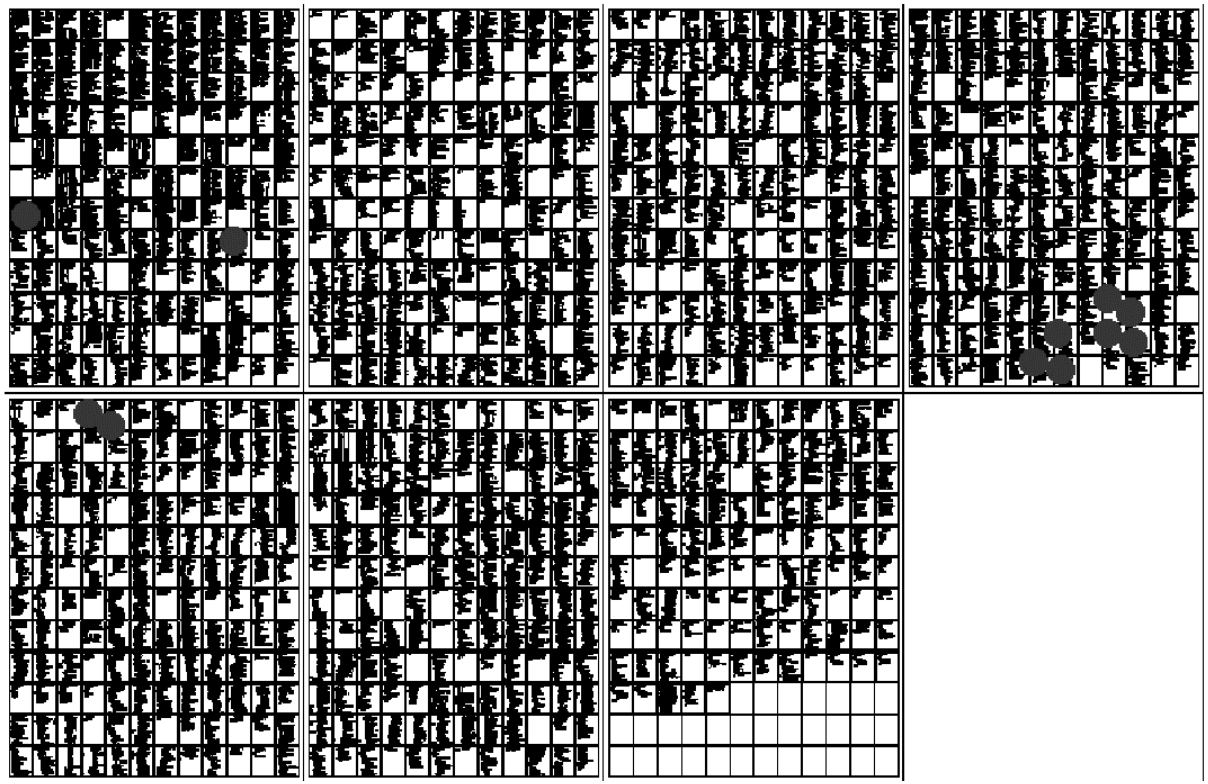
Current Status

- Experimental results
  - Found new bugs in sendmail (30k LOC), others
  - Analysis is fast, but many false alarms (1/kLOC)

see also Dor, Rodeh, Sagiv

- Research challenges
  - Pointer analysis (support strong updates)
  - Integer analysis (infer linear relations, flow-sensitivity)
  - Soundness, scalability, real-world programs

Solution to the Puzzle



Plan of the Talk

- Security is hard because of…
  - language traps (buffer overruns)
  - privilege pitfalls
  - untrusted data

… and many others that I won't consider in this talk

Pitfalls of Privileges

- Spot the bug:

```
setuid(0);
rv = bind(...);
if (rv < 0)
    return rv;
seteuid(getuid());
```

Pitfalls of Privileges

- Spot the bug:

enablePriv()

```
setuid(0);
rv = bind(...);
if (rv < 0)
    return rv;
seteuid(getuid());
```

Pitfalls of Privileges

- Spot the bug:

```
setuid(0);
rv = bind(...);
if (rv < 0)
    return rv;
seteuid(getuid());
```

checkPriv()

Pitfalls of Privileges

- Spot the bug:

```
setuid(0);
rv = bind(...);
if (rv < 0)
    return rv;
seteuid(getuid());
```

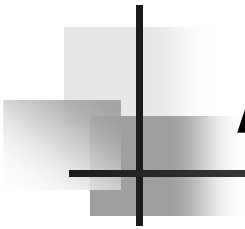
disablePriv()

Pitfalls of Privileges

- Spot the bug:

```
setuid(0);
rv = bind(...);
if (rv < 0)
    return rv;
seteuid(getuid());
```

Bug! Leaks privilege



A Common Language

A Common Language

- Abstracting the operations on privileges
  - S ::= call f() | S; S | S◊S (statements)
    | enablePriv(p) | disablePriv(p) | checkPriv(p)
    P ::= fun f = S | P P (programs)

A Common Language

- Abstracting the operations on privileges
  - S ::= call f() | S; S | S◊S (statements)
    | enablePriv(p) | disablePriv(p) | checkPriv(p)
    P ::= fun f = S | P P (programs)

- Various interpretations are possible
  - C: enablePriv(p) lasts until next disablePriv(p)
  - Java: … or until containing stack frame is popped
  - checkPriv(p) throws fatal error if p not enabled

Static Privilege Analysis

- Some problems in privilege analysis:
  - *Privilege inference* (auditing, bug-finding)
    - Find all privileges reaching a given program point
  - *Enforcing privilege-safety* (cleanliness of new code)
    - Verify statically that no checkPriv() operation can fail
    - … or that program behaves same under C & Java styles

One Possible Approach

One Possible Approach

- Privilege inference/enforcement in cubic time:

One Possible Approach

- Privilege inference/enforcement in cubic time:
  - Build a pushdown automaton

One Possible Approach

- Privilege inference/enforcement in cubic time:
  - Build a pushdown automaton

Γ = ProgPts $\times$ 2^{Privs} (stack symbols)

(t,e)::s $\rightarrow$ (f,e)::(t′,e)::s (call f())

(t,e)::s $\rightarrow$ s (return)

(t,e)::s $\rightarrow$ (t′,e $\cup$ {p})::s (enablePriv(p))

(t,e)::s $\rightarrow$ (t′,e)::s if p $\in$ e (checkPriv(p))

(t,e)::s $\rightarrow$ Wrong if p $\notin$ e (checkPriv(p))

(t,e)::s $\rightarrow$ (t′,e \ {p})::s (disablePriv(p))

One Possible Approach

- Privilege inference/enforcement in cubic time:
  - Build a pushdown automaton

Γ = ProgPts $\times$ 2^{Privs} (stack symbols)

(t,e)::s $\rightarrow$ (f,e)::(t′,e)::s (call f())

(t,e)::s $\rightarrow$ s (return)

(t,e)::s $\rightarrow$ (t′,e $\cup$ {p})::s (enablePriv(p))

(t,e)::s $\rightarrow$ (t′,e)::s if p $\in$ e (checkPriv(p))

(t,e)::s $\rightarrow$ Wrong if p $\notin$ e (checkPriv(p))

(t,e)::s $\rightarrow$ (t′,e \ {p})::s (disablePriv(p))

  - **Model-check** this PDA

One Possible Approach

- Privilege inference/enforcement in cubic time:
  - Build a pushdown automaton

Γ = ProgPts $\times$ 2^{Privs} (stack symbols)

(t,e)::s $\rightarrow$ (f,e)::(t′,e)::s (call f())

(t,e)::s $\rightarrow$ s (return)

(t,e)::s $\rightarrow$ (t′,e $\cup$ {p})::s (enablePriv(p))

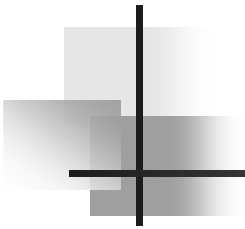
(t,e)::s $\rightarrow$ (t′,e)::s if p $\in$ e (checkPriv(p))

(t,e)::s $\rightarrow$ Wrong if p $\notin$ e (checkPriv(p))

(t,e)::s $\rightarrow$ (t′,e \ {p})::s (disablePriv(p))

  - **Model-check** this PDA

see also Pottier, Skalka, Smith



Future Directions

- Research challenges
  - Experimental studies on real programs
  - Handling data-directed privilege properties
  - Other access control models

Plan of the Talk

- Security is hard because of…
  - language traps (buffer overruns)
  - privilege pitfalls
  - untrusted data

… and many others that I won't consider in this talk

Manipulating Untrusted Data

- Spot the bug:

```
hp = gethostbyaddr(...);
printf(hp->hp_hname);
```

Manipulating Untrusted Data

- Spot the bug:

untrusted source of data

```
hp = gethostbyaddr(...);
printf(hp->hp_hname);
```

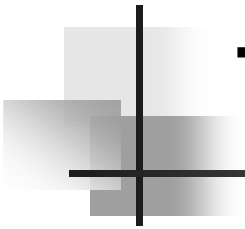
Manipulating Untrusted Data

- Spot the bug:

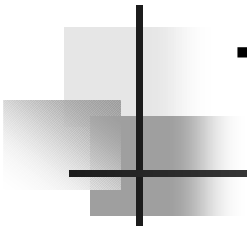
```
hp = gethostbyaddr(...);
printf(hp->hp_hname);
```

untrusted source of data

Bug! printf() trusts its first argument



Trust Analysis



Trust Analysis

- Security involves much mental "bookkeeping"
  - Problem: Help programmer keep track of which values can be trusted

Trust Analysis

- Security involves much mental "bookkeeping"
  - Problem: Help programmer keep track of which values can be trusted
- One approach: static *taint analysis*
  - Extend the C type system
  - Qualified types express annotations: e.g., `tainted char *` is an untrusted string
  - Typechecking enforces safe usage
  - Type inference reduces annotation burden

A Tiny Example

```
void printf(untainted char *, ...);
tainted char * read_from_network(void);
```

```
char *s = read_from_network();
printf(s);
```

… where untainted T ≤ tainted T

A Tiny Example

a trust annotation

```
void printf(untainted char *, ...);
tainted char * read_from_network(void);


char *s = read_from_network();
printf(s);
```

… where untainted T ≤ tainted T

After Type Inference…

```
void printf(untainted char *, ...);
tainted char * read_from_network(void);
```

```
tainted char *s = read_from_network();
printf(s);
```

… where untainted T ≤ tainted T

After Type Inference…

```
void printf(untainted char *, ...);
tainted char * read_from_network(void);
```

```
tainted char *s = read_from_network();
printf(s);
```

… where untainted T ≤ tainted T

After Type Inference…

```
void printf(untainted char *, ...);
tainted char * read_from_network(void);
```

an inferred type

```
tainted char *s = read_from_network();
printf(s);
```

… where untainted T ≤ tainted T

After Type Inference…

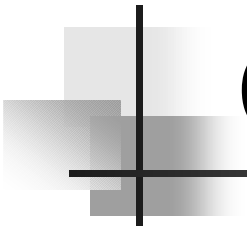
```
void printf(untainted char *, ...);
tainted char * read_from_network(void);
```

```
tainted char *s = read_from_network();
printf(s);
```

Doesn't type-check!
Indicates vulnerability

… where untainted T ≤ tainted T

Current Status



Current Status

- Experimental results
  - Successful on real programs
    - Able to find many previously-known format string bugs
    - Cost: 10-15 minutes per application
  - Type theory seems useful for security engineering

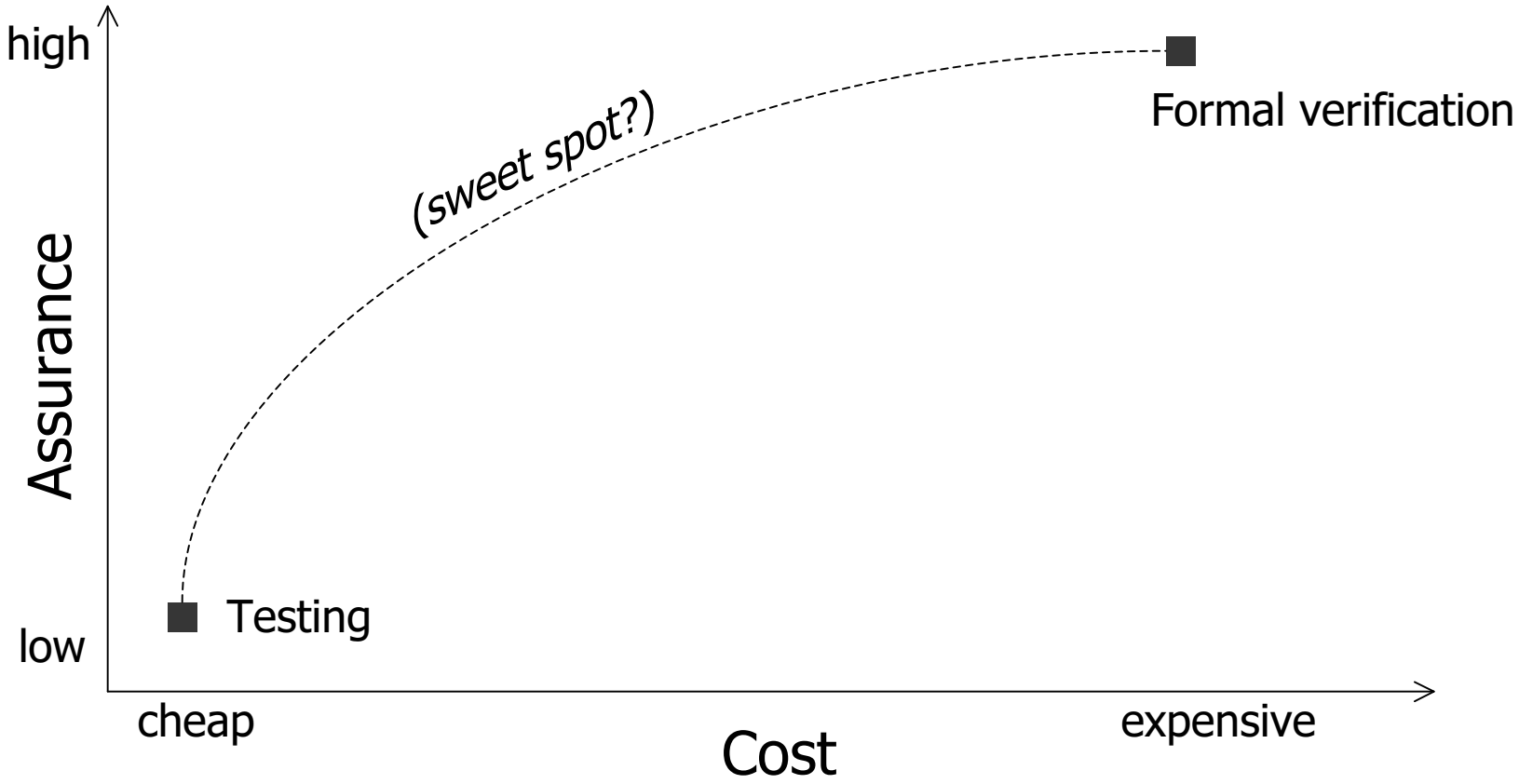
Current Status

- Experimental results
  - Successful on real programs
    - Able to find many previously-known format string bugs
    - Cost: 10-15 minutes per application
  - Type theory seems useful for security engineering

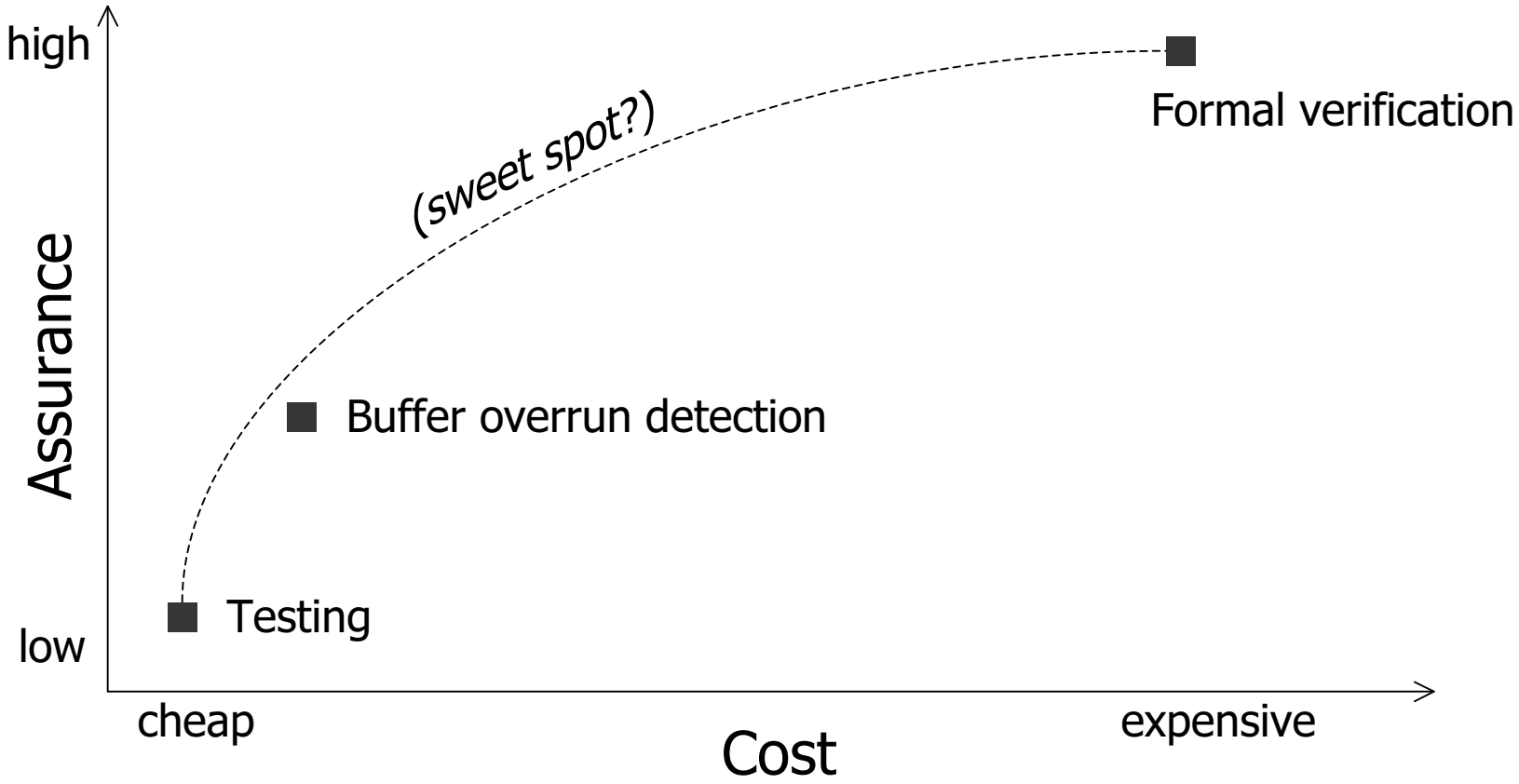
- Research challenges
  - Richer theory to support real programming idioms
  - More broadly-applicable discipline of good coding
  - Finer-grained notions of trust

see also Myers et. al

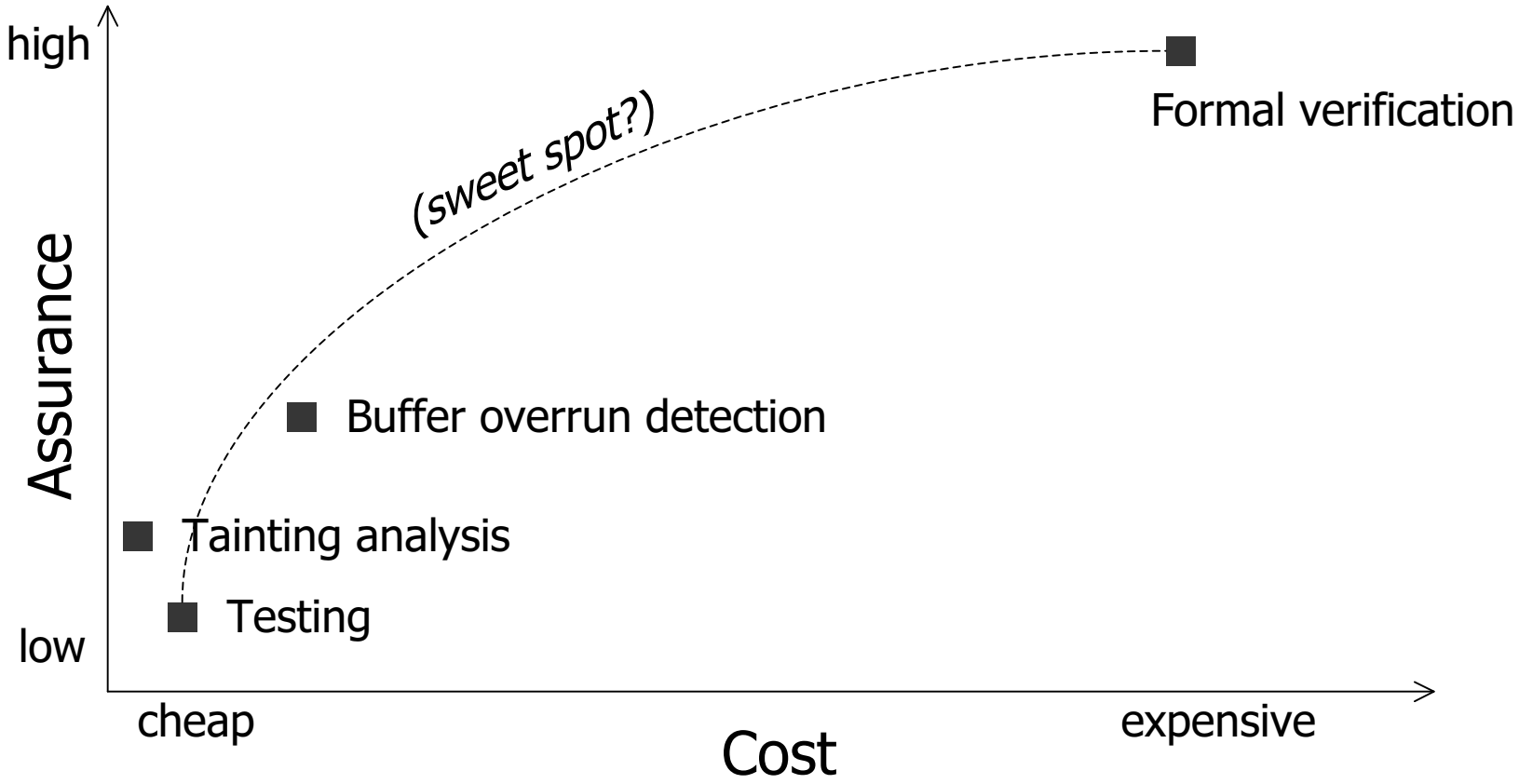
Summary


high
low
Assurance
(sweet spot?)
Formal verification
Testing
cheap
expensive
Cost

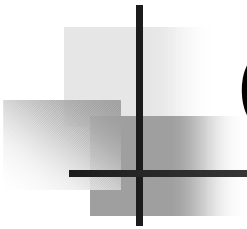
Summary


high
low
Assurance
(sweet spot?)
Formal verification
Buffer overrun detection
Testing
cheap
expensive
Cost

Summary


high
low
Assurance
(sweet spot?)
Formal verification
Buffer overrun detection
Tainting analysis
Testing
cheap
expensive
Cost

Concluding Remarks



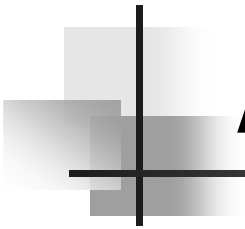
Concluding Remarks

- Static analysis can help secure our software
  - Buffer overruns, privilege bugs, format string bugs
  - Hits a sweet spot: cheap and proactive

Concluding Remarks

- Static analysis can help secure our software
  - Buffer overruns, privilege bugs, format string bugs
  - Hits a sweet spot: cheap and proactive
- Security as a source of interesting problems?
  - Motivations for better pointer, integer analysis
  - New problems: privilege analysis, trust analysis

Backup Slides



A Role for Static Analysis

A Role for Static Analysis

- Strong points of static analysis:
  - Can detect vulnerabilities proactively
  - Can focus on a few key properties with big payoffs
  - Can reuse security specifications across programs

A Role for Static Analysis

- Strong points of static analysis:
  - Can detect vulnerabilities proactively
  - Can focus on a few key properties with big payoffs
  - Can reuse security specifications across programs

- Application domains:
  - *Inference* (program understanding, bug-finding)
    - Appropriate for legacy code
  - *Enforcement* (proving absence of bugs)
    - Most useful when building new systems