Language-Based Methods for Software Security

George Necula

EECS Department

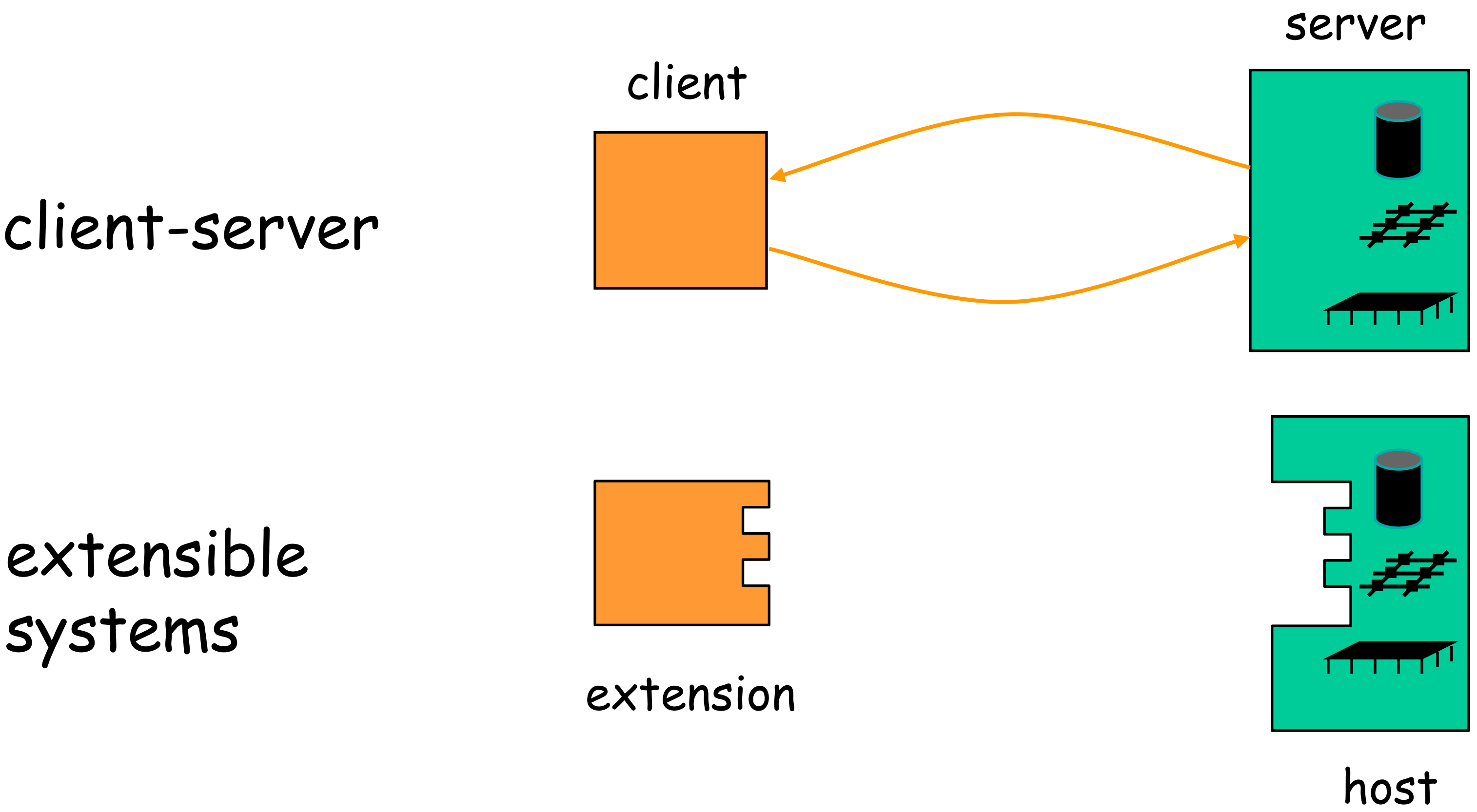
University of California, Berkeley

Roadmap

- Static checking vs. dynamic checking
- Dynamic: Enforcing memory safety for C programs
- Static: Proof-carrying code
  - Type checking Java bytecodes
  - Type checking assembly language
  - Proof-carrying code tools and techniques

Motivation

- Extensible systems can be more flexible and more efficient than client-server interaction



server

client

client-server

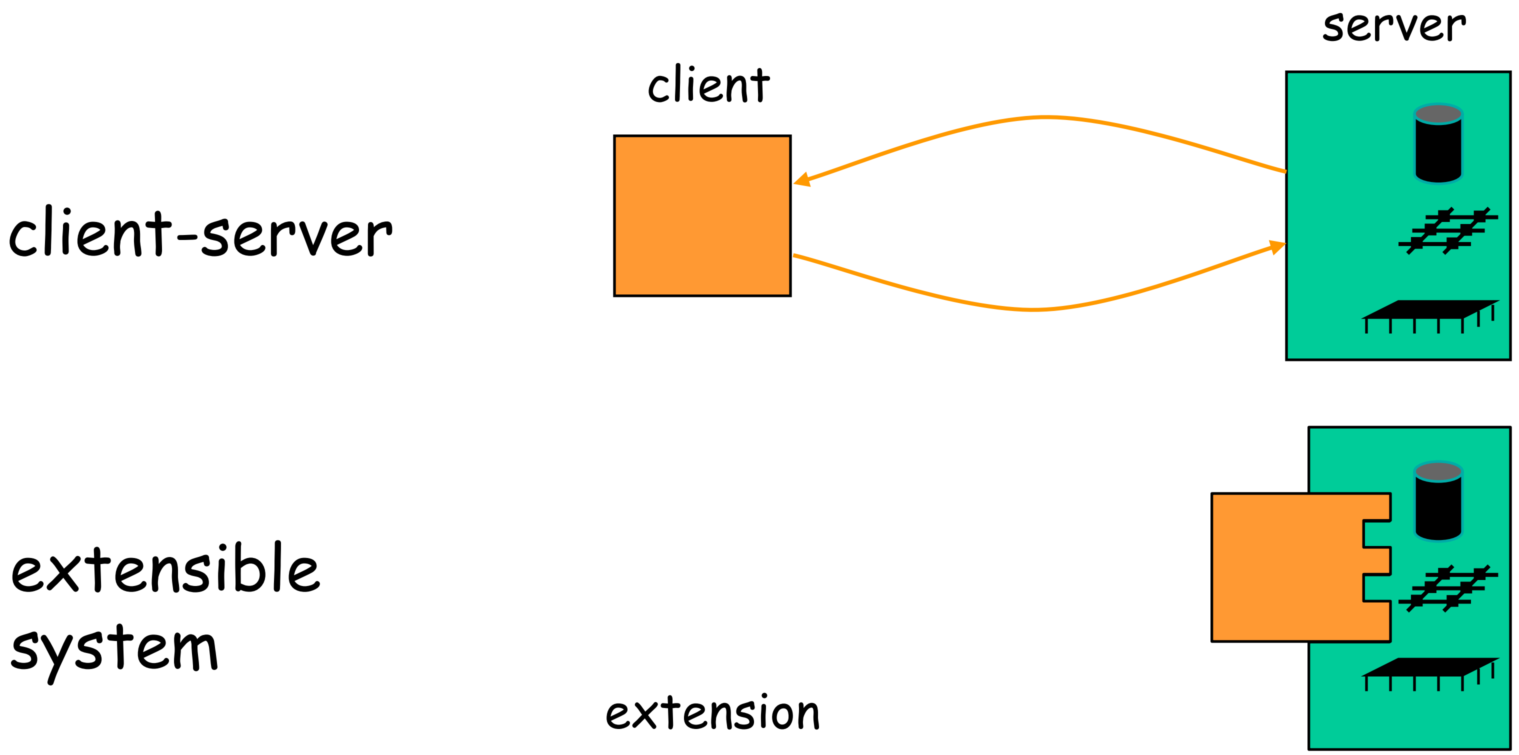
extensible systems

extension

host

Motivation

- Extensible systems can be more flexible and more efficient than client-server interaction



Must make sure extension does not bypass the interface

Examples of Extensible Systems

Device driver	Operating system
Applet	Web browser
Stored procedure	Database server
COM Component	COM host
…	

Concerns Regarding Extensibility

- Safety and reliability concerns

  è How to protect the host from the extensions ?

  Extensions of unknown origin ⇒ potentially malicious

  Extensions of known origin ⇒ potentially erroneous

- Complexity concerns

  è How can we do this without having to trust a complex infrastructure?

- Performance concerns

  è How can we do this without compromising performance?

- Other concerns (not addressed here)
  - How to ensure privacy and authenticity?
  - How to protect the component from the host?

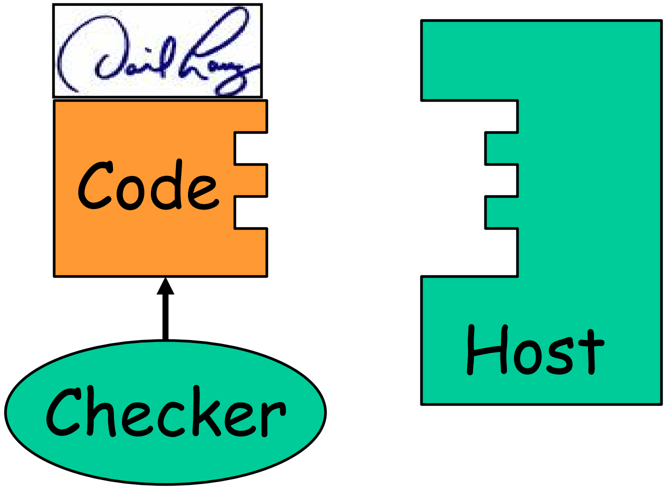
Existing Approaches to Component Safety

- Based on digital signatures

- Based on hardware protection

- Language-based mechanisms

Assurance Support: Digital Signatures



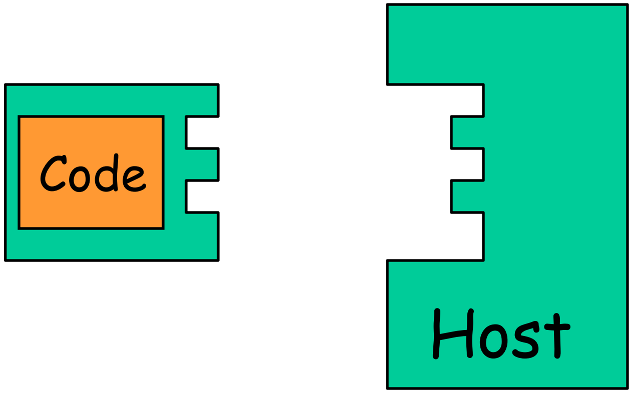
- Trust some code producers

- Ensures extrinsic properties (authorship, freshness)

L Not a behavioral assurance

L Does not scale well to many code producers

Run-Time Monitoring and Checking



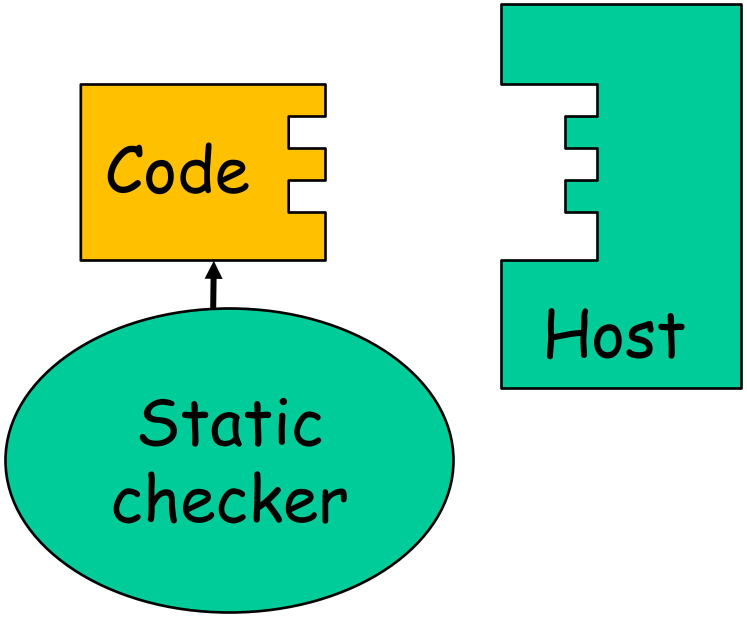
- A monitor detects attempts to violate the safety policy and stops the execution
  - Hardware-enforced memory protection
  - Software fault isolation (sandboxing)

J Simple, tried-out idea

Disadvantages of Run-Time Checking Alone

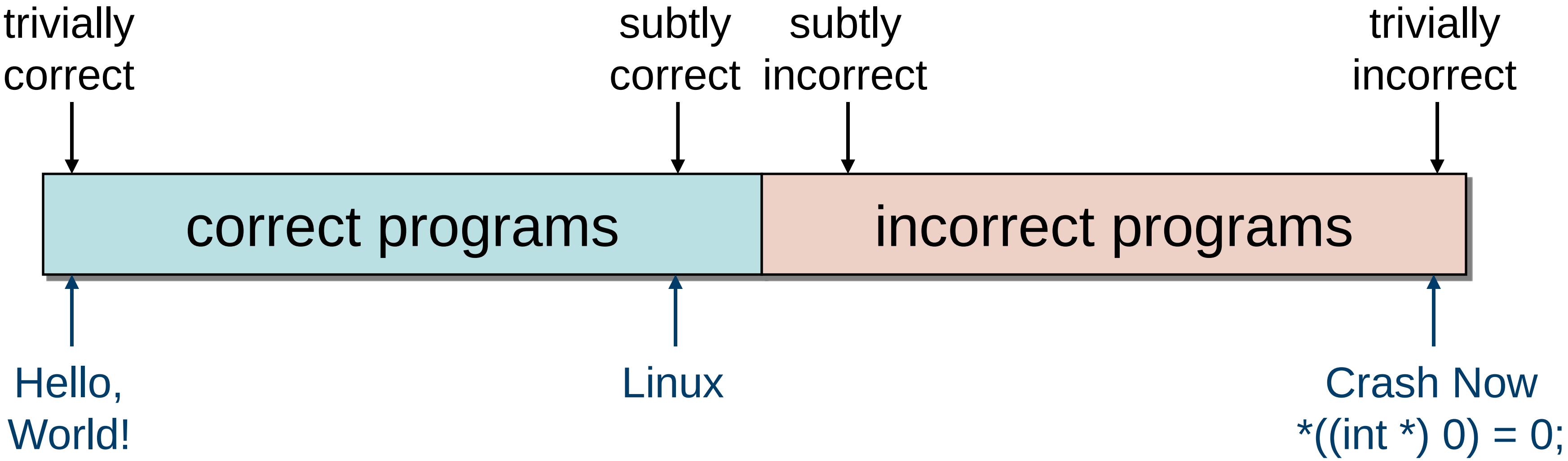
- High run-time cost
  - Crossing the protection boundary is expensive
- Sometimes it is hard to detect the "bad" event
  - "A pointer does not point to a NULL-terminated string"
  - "A pointer does not point to a file data structure"
  - Data abstraction is hard to check at run-time
- Sometimes stopping the execution is not a solution
  - We cannot (easily) stop a program that has acquired a critical resource
  - Time cannot be stopped
  - E.g., "code must shutdown the reactor in at most 500ms"

Static Checking



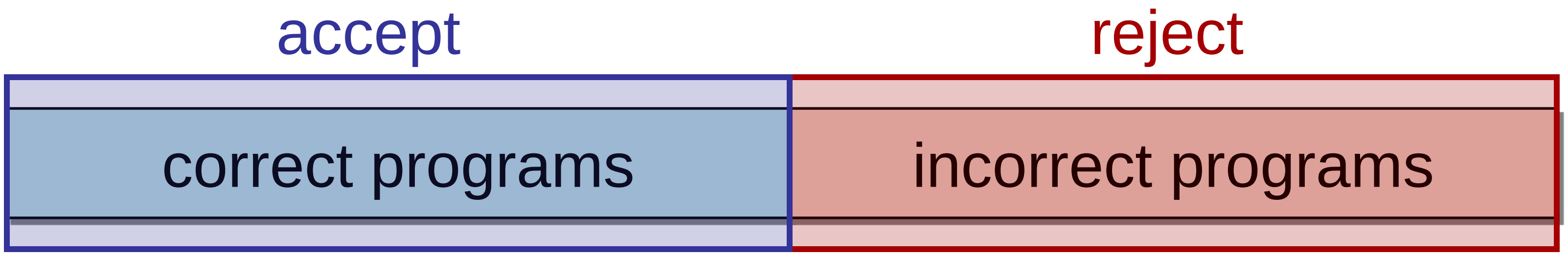
- Advantages:
  - No run-time cost
  - Can consider hard-to-test scenarios
- Disadvantages:
  - Must trust complex certification tools
  - Undecidable unless enough restrictions are placed

Static vs. Dynamic Checking



Static vs. Dynamic Checking

The Dynamic Checker



accept

reject

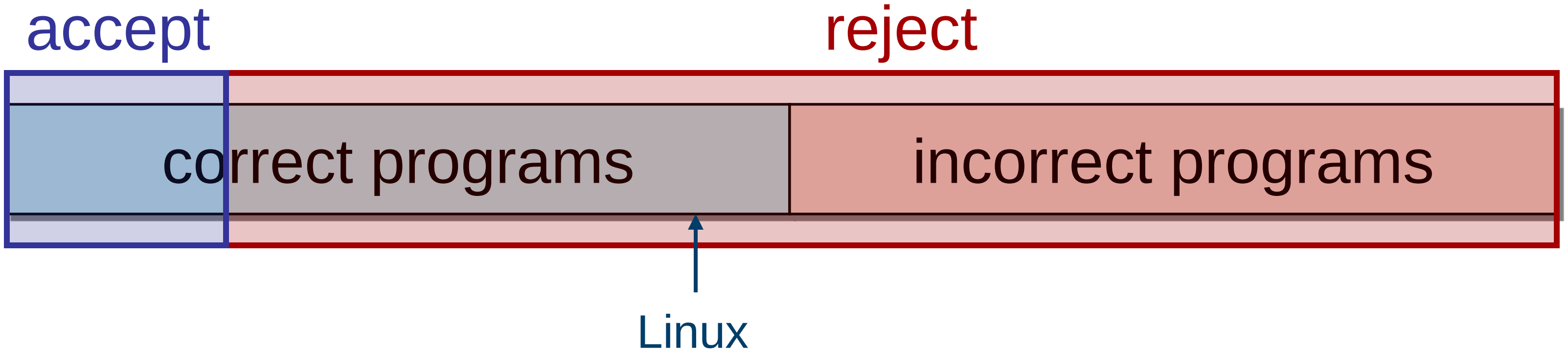
correct programs

incorrect programs

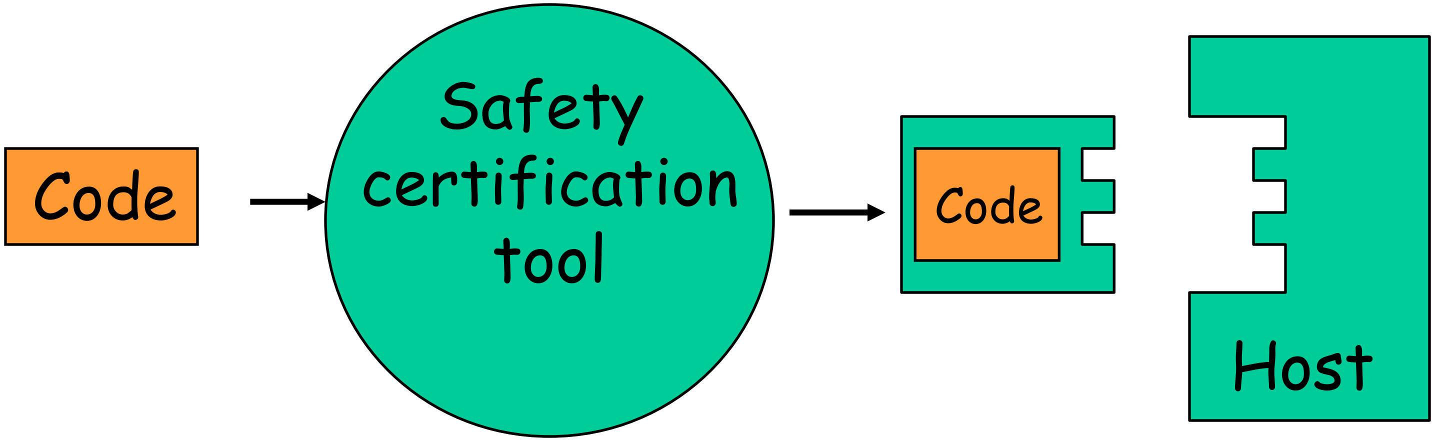
Static vs. Dynamic Checking

Purely static checking

+ No run-time checks
– Unsuitable for existing code



Hybrid Checking

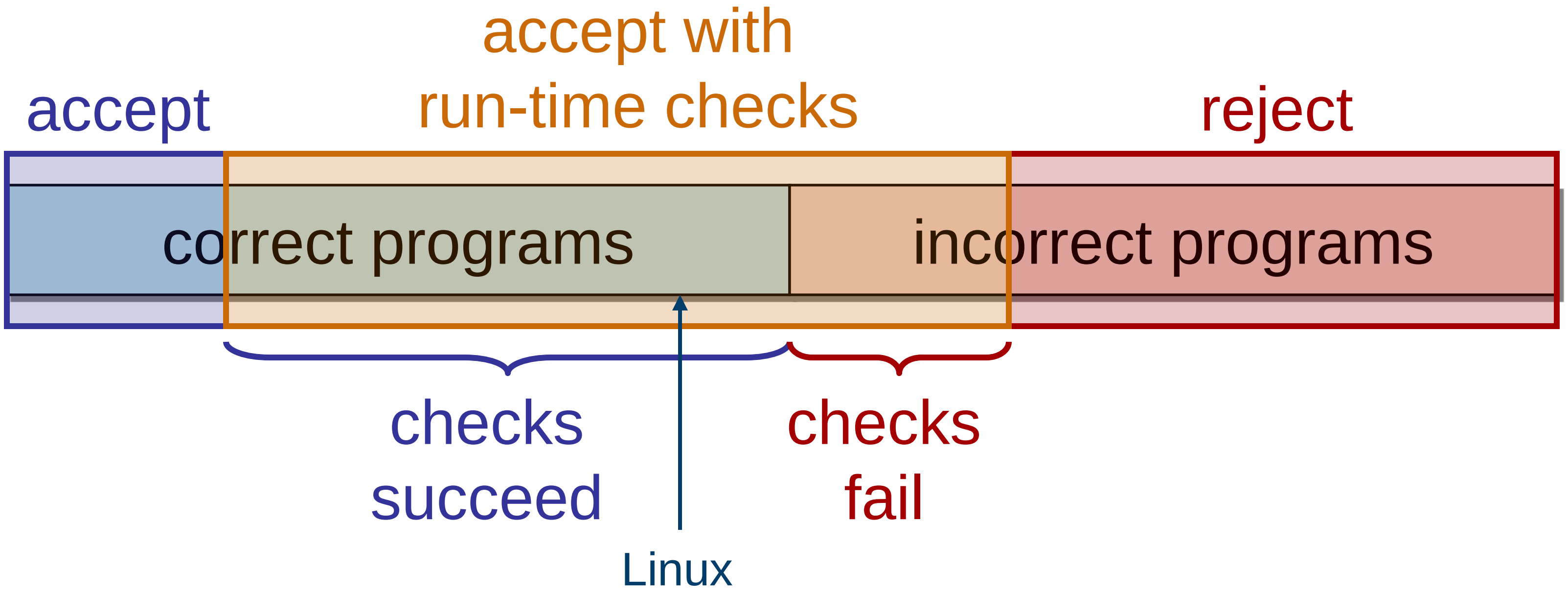


- Check statically, insert dynamic checks where necessary
- Advantages:
  - Reduced run-time cost
- Disadvantages:
  - Still some run-time checking
  - Complex tools ?

Static vs. Dynamic Checking

Hybrid Checking
(static + dynamic)

+ Suitable for existing code
– Some errors delayed



Roadmap

- Static checking vs. dynamic checking
- ➢ Dynamic: Enforcing memory safety for C programs
- Static: Proof-carrying code
  - Type checking Java bytecodes
  - Type checking assembly language
  - Proof-carrying code tools and techniques

Memory Safety

- Essential component of a security infrastructure
  - Isolates modules in extensible systems
  - 85% of Windows crashes caused by drivers
  - 50% of reported attacks are due to buffer overruns
    - 1988: Robert Morris's internet worm
    - 2000: Code Red, SQL Slammer
    - Recent exploitable bugs:

Quicktime
(1/5/07)

Java Runtime
(1/16/07)

Windows
(4/3/07)

- Software engineering advantages
  - Memory bugs are hard to find
  - Foundation for most other software analyses

Type and Memory Safety

Definition

Type Safety:
Run-time values correspond to compile-time types

Memory Safety:
No illegal or out-of-bounds memory accesses

Example Error

```
cheese c;
wine w = (wine) c;
drink(w);
```

```
int array[42];
array[100] = 0;
```

C and C++ does not enforce type and memory safety.

We can do better!

The Legacy of C

- Millions of lines of safety-critical C code
  - Huge investment!
- These systems are unsafe and unreliable due to C's lack of type and memory safety
- Need an incremental transition to safer and more reliable systems!

Deputy goals

- Modular, fine-grained safety and isolation enforces type and memory safety
    - Works on existing C programs (including Linux)
    - Dependent types enable modular approach

- Efficiency: 0-50% slowdown
    - vs. Purify or Valgrind 10+x slowdown

- More effective and efficient than Purify
    - Because it leverages existing type information in source

Enforcing Safety

Previous source-based approach (Cyclone, CCured, SafeC)

```
struct buffer {
    int *data;
    int len;
} b;
```

```
    int *data_b;   // lower bound (base)
    int *data_e;   // upper bound (end)
```

```
for (i = 0; i < b.len; i++) {
    // verify that b.data[i] is safe
    ... b.data[i] ...
}
```

```
    assert(data_b <= b.data + i < data_e);
```

Enforcing Safety

Deputy’s Approach

```
struct buffer {
    int * count(len) data;
    int len;
} b;

for (i = 0; i < b.len; i++) {
    assert(0 <= i < b.len);
    ... b.data[i] ...
}
```

Advantages:

1. No change in data layout
2. Easier to optimize
3. Contract is in the code!

Deputy

```
struct buffer {
    int * count(len) data;
    int len;
} b;
```

Key Insight:

Most pointers' bounds information is already present in the program in some form--just not in a form the compiler understands!

Deputy

```
struct buffer {
    int * count(len) data;
    int len;
} b;
```

Dependent Types:
Types whose meaning depends on the *run-time value* of a program expression.

Dependent types enable
modular checking!

Modularity

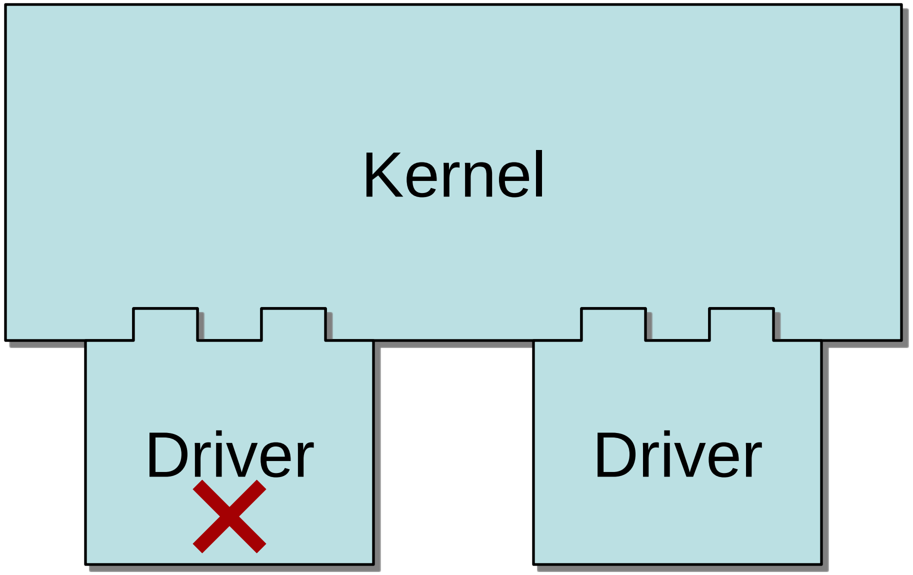
Alternative to whole-program analysis and instrumentation

- Source code unavailable
- Source code cannot be recompiled

Incremental improvements

- Improve program module by module
- Improve overall code quality gradually

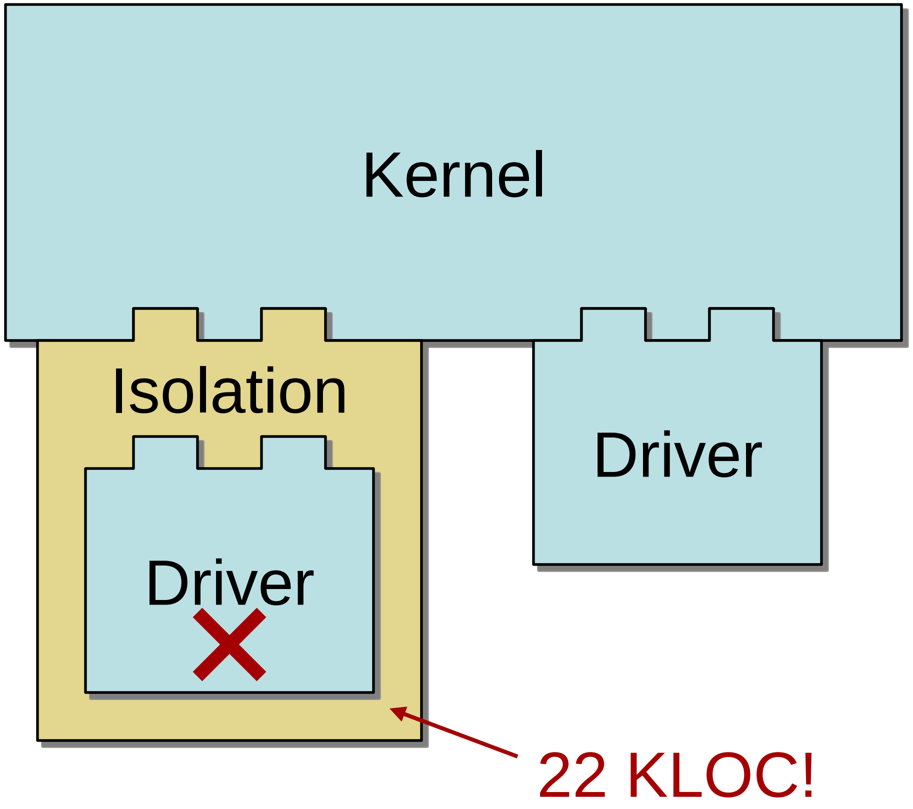
Isolating Extensions



Problems:

- Driver bug can corrupt kernel

Isolating Extensions



22 KLOC!

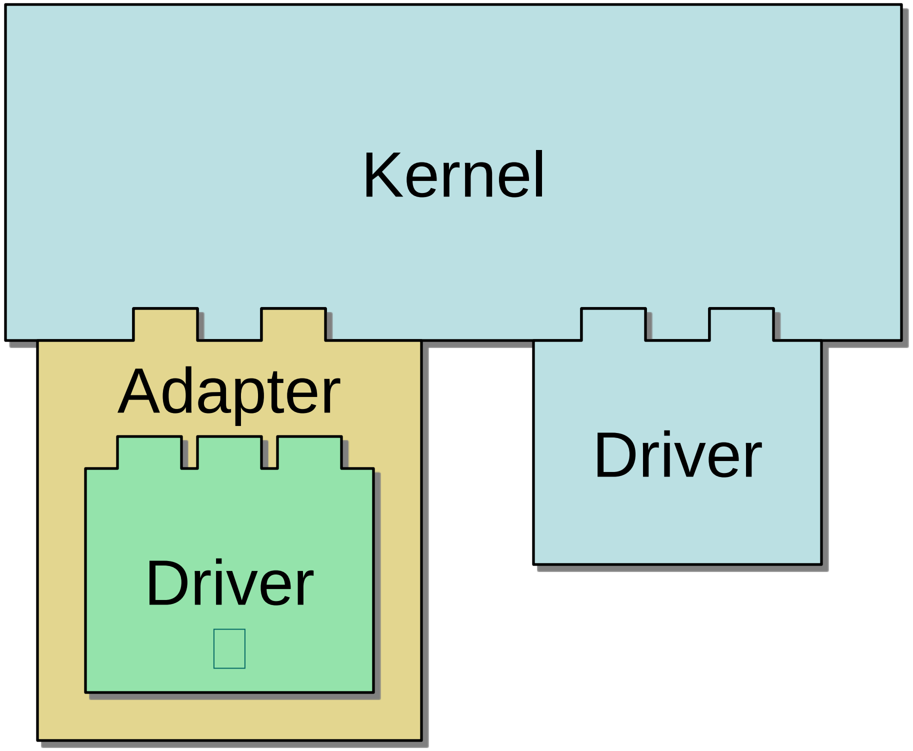
Problems:

✓ Driver bug can’t corrupt kernel

- Driver can still corrupt itself
- Isolation layer is complicated!

Nooks [Swift et al., SOSP 03]

Isolating Extensions



Problems:

✓ Driver bug can’t corrupt kernel

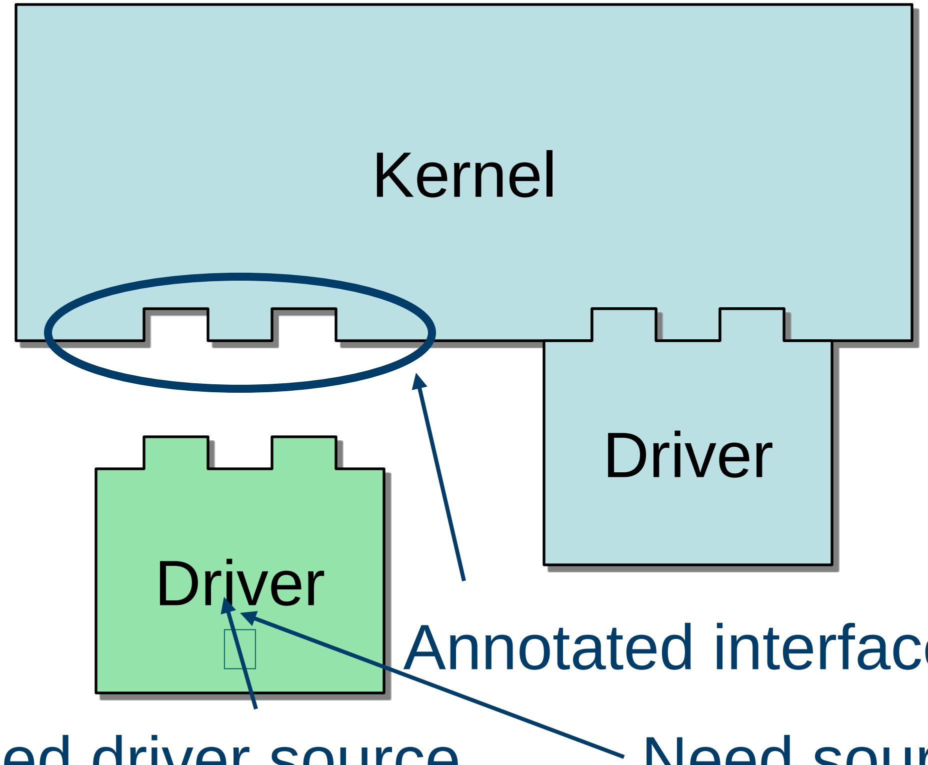
✓ Driver can’t corrupt itself

✓ Isolation layer not needed

- Adapter is complicated!

CCured [PLDI 03], Cyclone [Jim et al., USENIX 02]

Misbehaving Extensions



Annotated interface

Need driver source

Need source annotations

Problems:

✓ Driver bug can’t corrupt kernel

✓ Driver can’t corrupt itself

✓ No adapter required

Deputy [ESOP 07, OSDI 06]

Deputy Outline

✓ Overview
- Deputy
- Applications
- Related & Future Work

Why Dependent Types?

Used by many common idioms in C code

```
struct buffer {
    char * data;
    int len;
};
```

```
int strlcpy(char * dst,
            char * src,
            int n);
```

```
struct message {
    int tag;
    union {
        int num;
        char *str;
    } u;
};
```

Why Dependent Types?

Used by many common idioms in C code

If we annotate these idioms, we can check for correct use!

```
struct buffer {
    char * count(len) data;
    int len;
};
```

```
int strlcpy(char * nt count(n) dst,
            char * nt count(0) src,
            int n);
```

```
struct message {
    int tag;
    union {
        int num   when(tag == 1);
        char *str when(tag == 2);
    } u;
};
```

Challenges

Previous dependent type systems were not designed for use with existing code

- – Static checking is difficult
  - ⇒ Hybrid checking (i.e., with run-time checks)
- – Mutation is heavily used
  - ⇒ Use ideas from axiomatic semantics
- – Annotation burden is high
  - ⇒ Automatic dependencies & inference

Static vs. Hybrid Checking

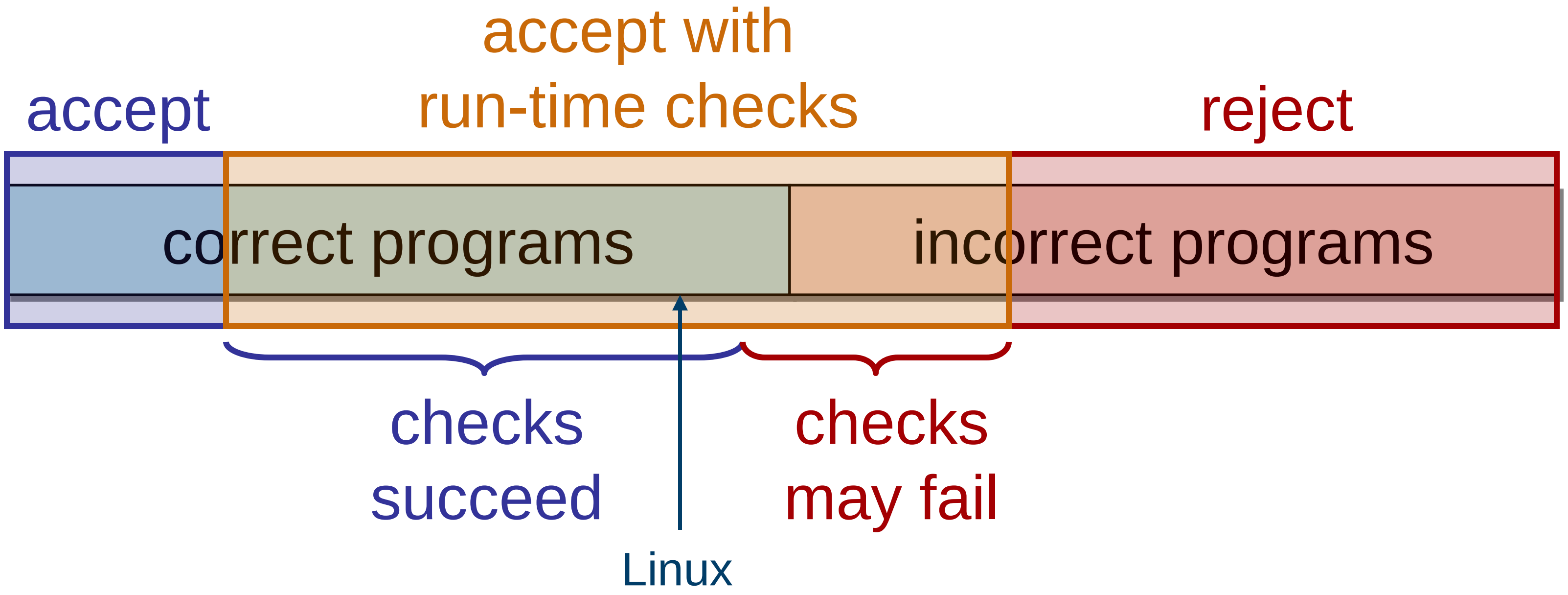
```
struct buffer {
    int * count(len) data;
    int len;
} b;

int limit = get_limit();
for (i = 0; i < limit; i++) {
    assert(0 <= i < b.len);
    ... b.data[i] ...
}
```

Hard to prove statically!

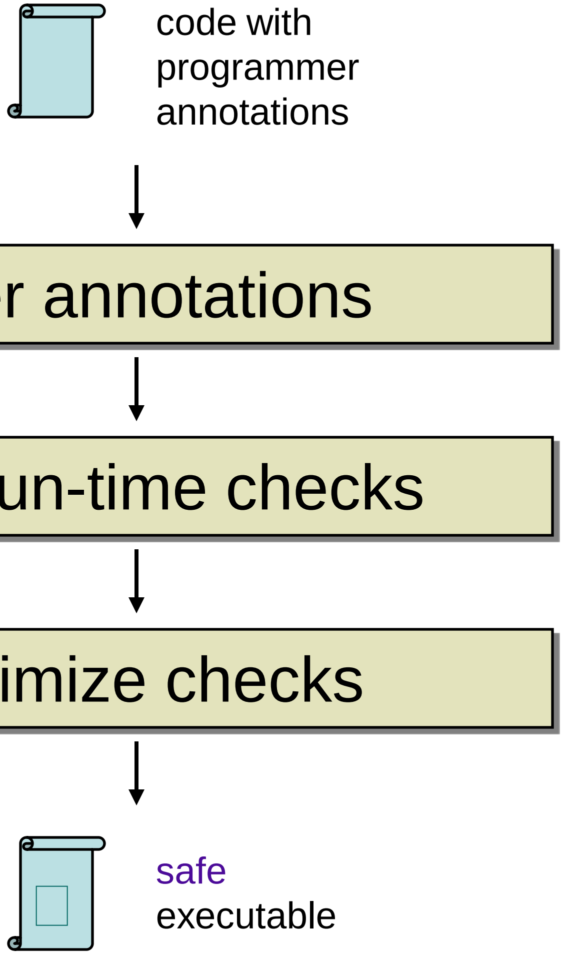
Deputy Checking

Hybrid Checking
(static + dynamic)

+ Suitable for existing code
– Some errors delayed



Compiler Overview



code with programmer annotations

infer annotations

add run-time checks

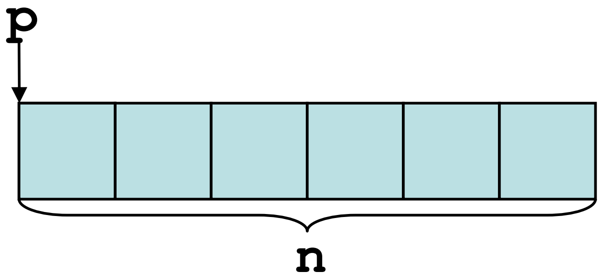
optimize checks

safe executable

Adding Checks

Dereference:

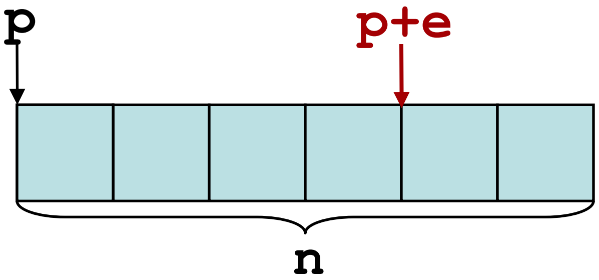
```
int * count(n) p;
assert(n > 0);
... *p ...
```



Adding Checks

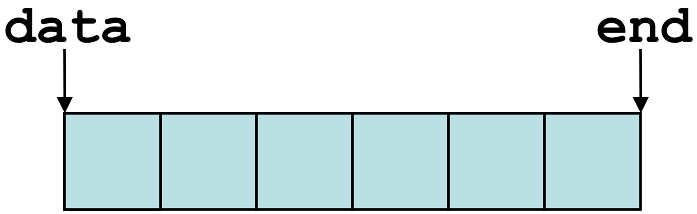
Arithmetic:

```
int * count(n) p;
assert(0 <= e <= n);
... p + e ...
```



Mutation

```
int * bound(end, end) end;
int * bound(data,end) data;
...
assert(data <= data + 1 <= end);
data = data + 1;
```



Local Expressions

Dependencies can refer to variables in the immediately enclosing scope

```
int * count(n + m)      data;
```

□

Memory references and function calls are disallowed

```
int * count(*len_ptr)   data;
```

✗

```
int * count(get_len()) data;
```

✗

Usability

Type checker expects every pointer to be annotated ⇒ inference required!

Three inference mechanisms:

- Automatic dependencies
- Pointer graph
- Assumptions

Automatic Dependencies

For unannotated locals, we can add annotations that use fresh variables

```
void foo(int * count(p_len) p, int p_len,
         int * count(q_len) q, int q_len) {
    int * x;
          count(x_len)
    if (...) { x = p; }
               x_len = p_len;
    else       { x = q; }
               x_len = q_len;
    assert(0 <= 42 < x??
                     x_len);
    ... x[42] ...
}
```

x_len is updated when **x** is updated

C Features

Deputy handles:

- Bounded pointers
- Null termination
- Tagged unions
- Polymorphic functions
- Allocations
- Calls to memset, memcpy

Deputy trusts:

- Deallocation & concurrency
- External library code
- User-specified trusted code

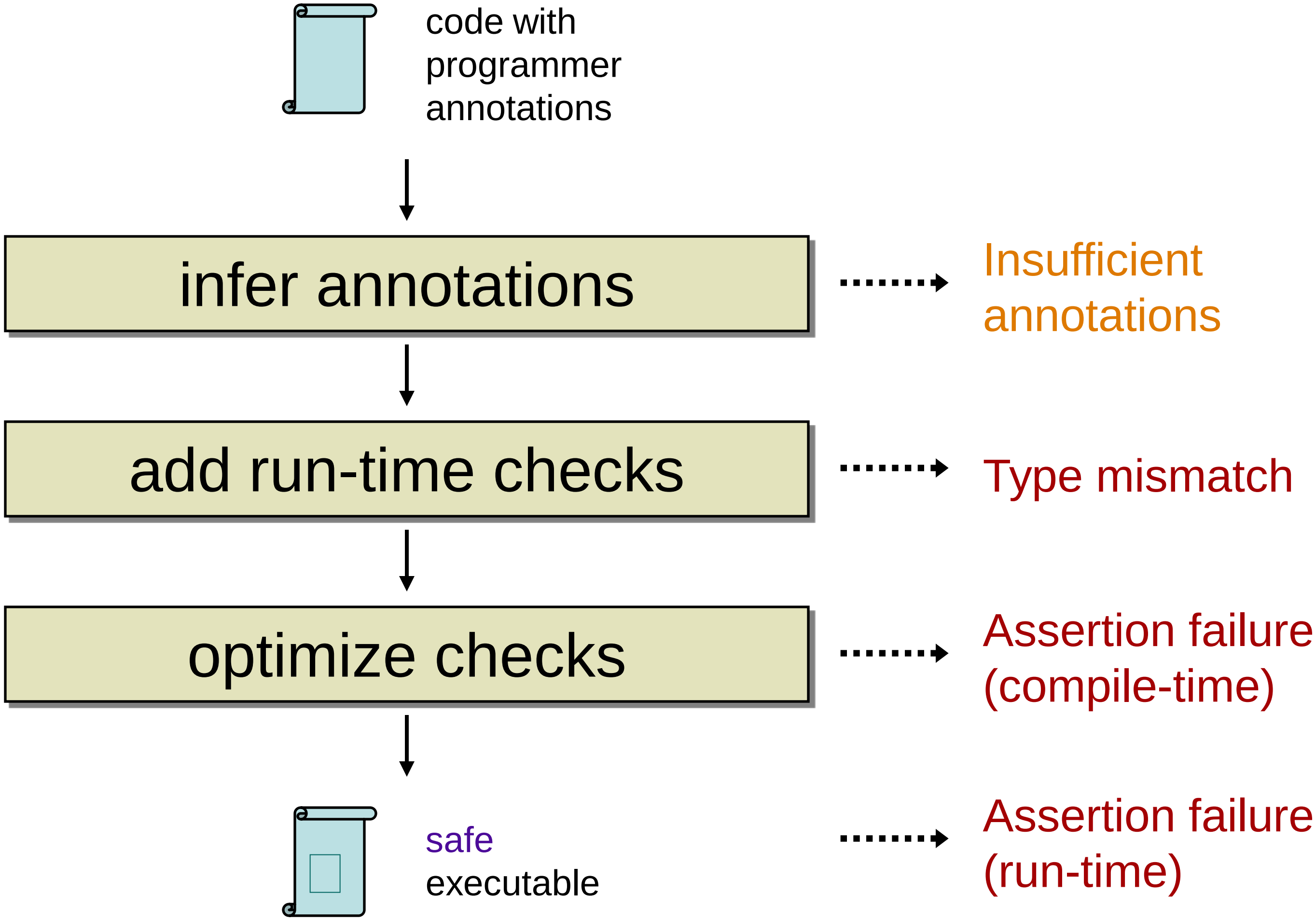
Roadmap

- Static checking vs. dynamic checking

➢ Dynamic: Enforcing memory safety for C programs

- Static: Proof-carrying code
  - Type checking Java bytecodes
  - Type checking assembly language
  - Proof-carrying code tools and techniques

The Deputy Compiler



code with programmer annotations

infer annotations

Insufficient annotations

add run-time checks

Type mismatch

optimize checks

Assertion failure (compile-time)

safe executable

Assertion failure (run-time)

Outline

- ✓ Overview
- ✓ Deputy
- Applications
- Related & Future Work

Deputy Applications

Three categories of applications

- Small programs (SPEC, Olden, Ptrdist)
- Linux device drivers (SafeDrive)
- Linux kernel

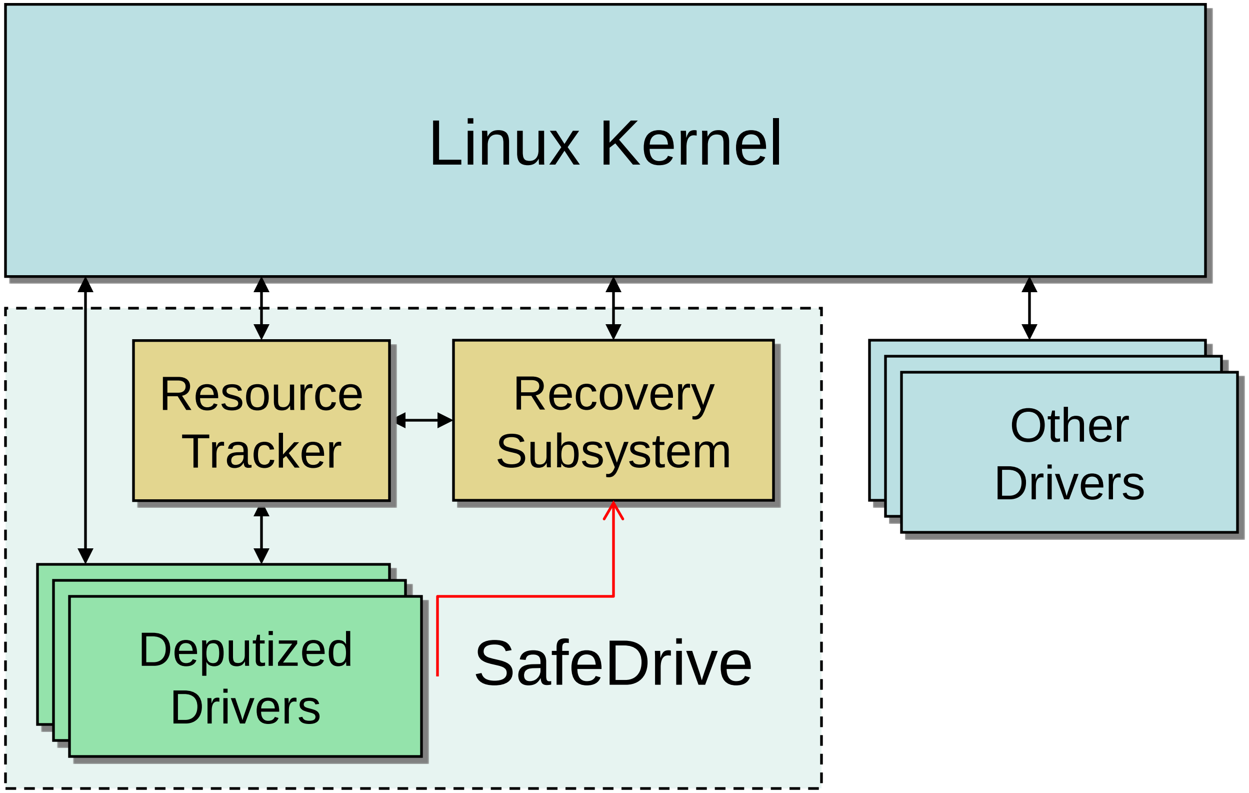
Evaluate Deputy on each application

- Annotation burden
- Performance impact

Small Programs (1)

	Benchmark	Total Lines	Lines Changed	Deputy Exec. Ratio	CCured Exec. Ratio
spec95	go	29339	0.6%	1.12	1.06
	gzip	8678	3.5%	1.12	-
	li	7431	9.1%	1.47	1.45
olden	bh	1907	30.0%	1.09	1.25
	bisort	679	13.8%	0.95	0.98
	em3d	358	19.0%	1.53	1.95
	health	605	4.5%	1.21	1.04
	mst	417	14.9%	1.31	1.00
	power	768	4.0%	1.02	2.03
	treeadd	127	11.0%	1.79	1.11
	tsp	565	1.8%	1.03	1.03

SafeDrive Architecture



Deputized Drivers

Used Deputy on Linux 2.6 drivers

- Network, sound, video, USB (10-20 KLOC each)

Approximately 1-4% of lines annotated

	Lines Changed	Bounds	Strings	Tagged Unions	Trusted Code
All 6 drivers	1544	379	83	2	390
Kernel headers	1866	187	260	8	140

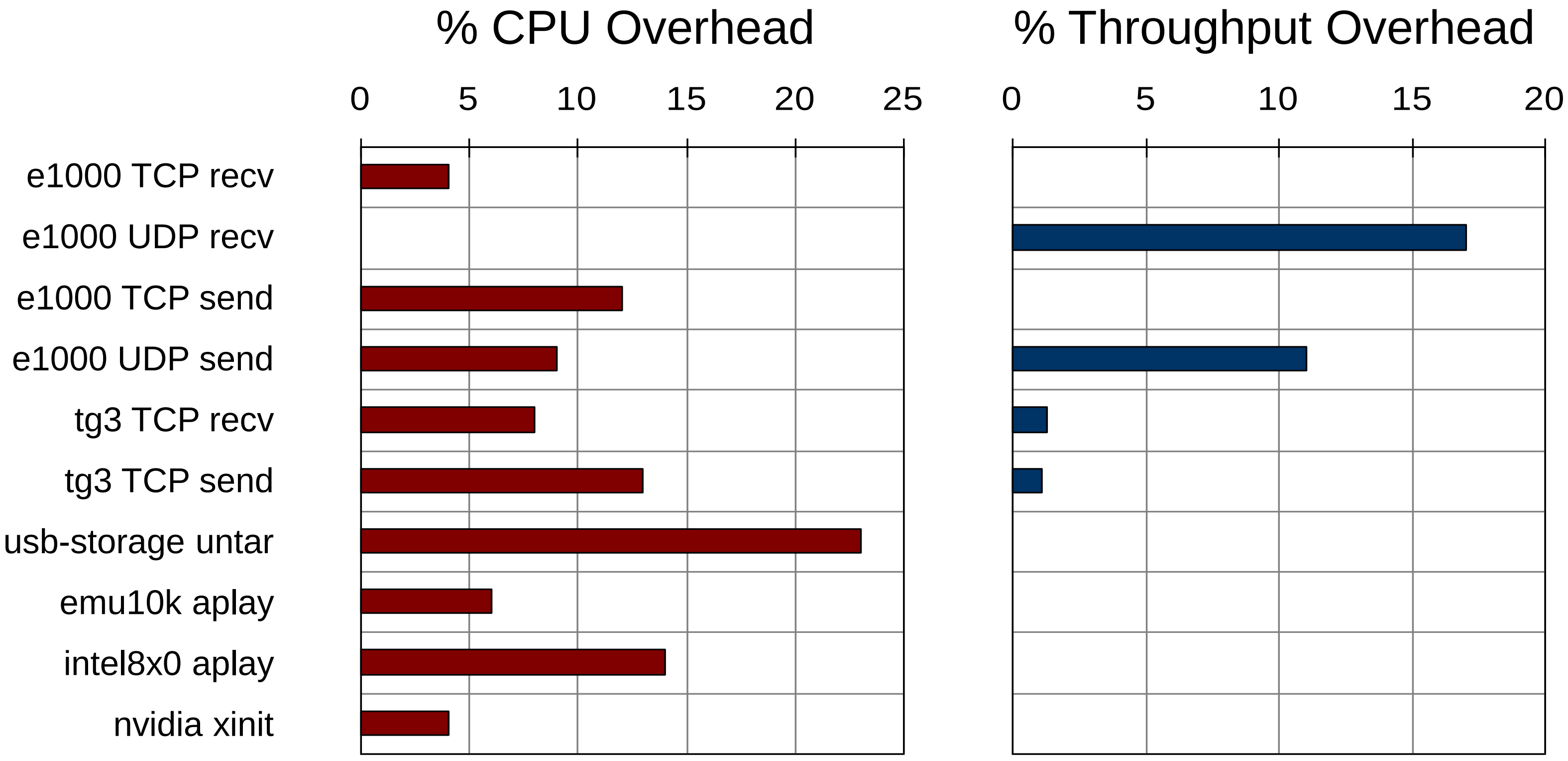
Evaluation: Recovery

Injected bugs at compile time:

- 140 tests over 7 different categories
- Corrupt parameter, off-by-one, etc.
- Run **`e1000`** driver with & without SafeDrive

- Without SafeDrive: With SafeDrive:
  - 44 crashes : 10 static err., 34 dyn. err.
  - 21 failure : 2 dyn. err., 19 no err
  - 75 test passes : 3 st. err, 5 dyn. err., 67 no err

Evaluation: Performance



Nooks CPU Overhead:	e1000 TCP recv:	**46% (vs. 4%)**
(Linux 2.4)	e1000 TCP send:	**111% (vs. 12%)**

The Language Advantage

Deputy & SafeDrive provide:

- Fine-grained safety checks
- Better performance

Next Step: The Kernel Itself!

Applied Deputy to a full kernel

- 435 KLOC configuration
- Memory, file systems, network, drivers

Manageable amount of work

- 2627 lines annotated (0.6%)
- 3273 lines trusted (0.8%)
- 7 person-weeks of effort required

Kernel Performance

Three categories of performance tests

- – Microbenchmarks: HBench-OS
- – End-to-end: Large compile
- – End-to-end: Web server performance

Test machine:

- – 2.33 GHz Intel Xeon processor
- – 1 GB RAM, 4 MB cache

Microbenchmarks

HBench-OS

kernel benchmarks

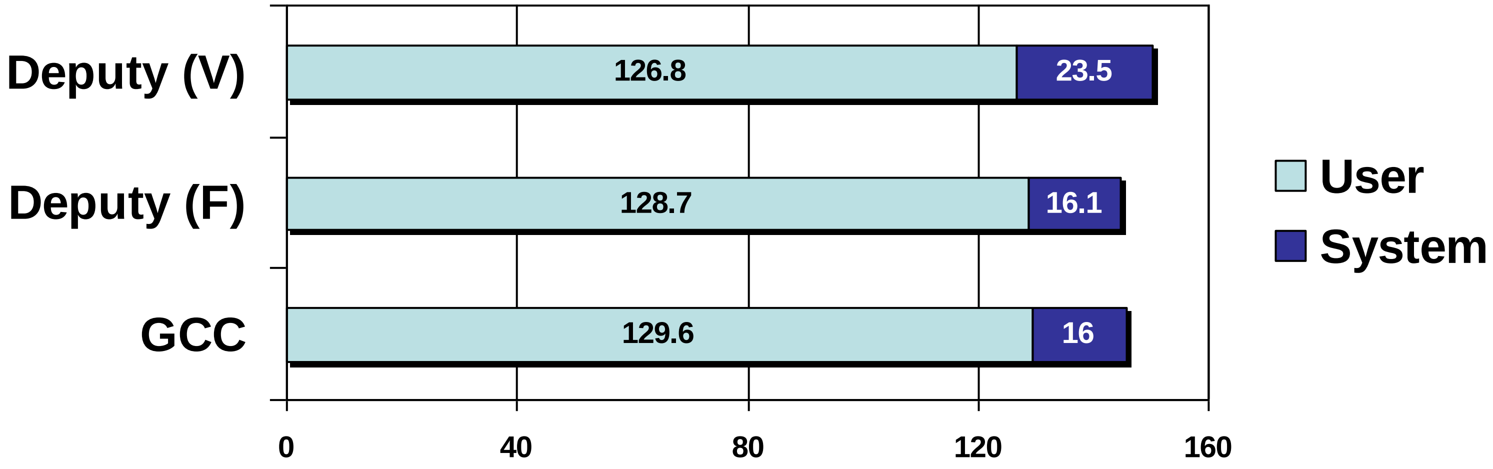
[Brown ‘97]

Bandwidth Tests	Ratio	Latency Tests	Ratio
bzero	0.99	connect	1.03
file_rd	0.98	ctx	1.08
mem_cp	0.98	ctx2	1.01
mem_rd	0.99	fs	1.17
mem_wr	0.99	fslayer	1.02
mmap_rd	0.87	mmap	1.51
pipe	0.98	pipe	1.16
tcp	0.92	proc	1.00
		rpc	1.27
		sig	1.33
		syscall	1.04
		tcp	1.20
		udp	1.29

Kernel Build Benchmark

Measure time to build a large system

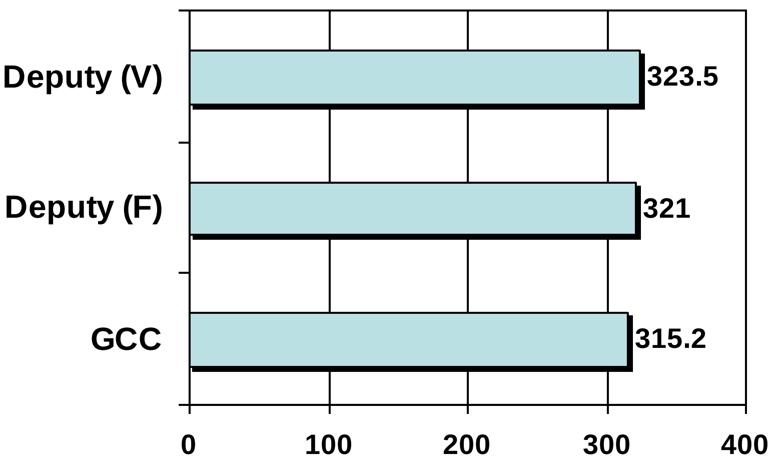
- – Test: Linux 2.6.15.5 built with GCC 4.1.3
- – Same test machine as before

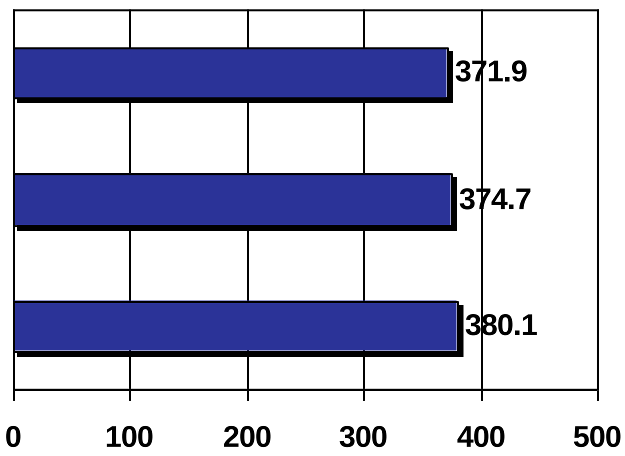


SPEC Web Benchmark

Measure HTTP bandwidth and latency

- Test: SPEC Web 99
- Same test machine as before


Latency (ms/op)


Bandwidth (Kbits/s)

Deputy Conclusions

- Many C programs are close to being memory safe

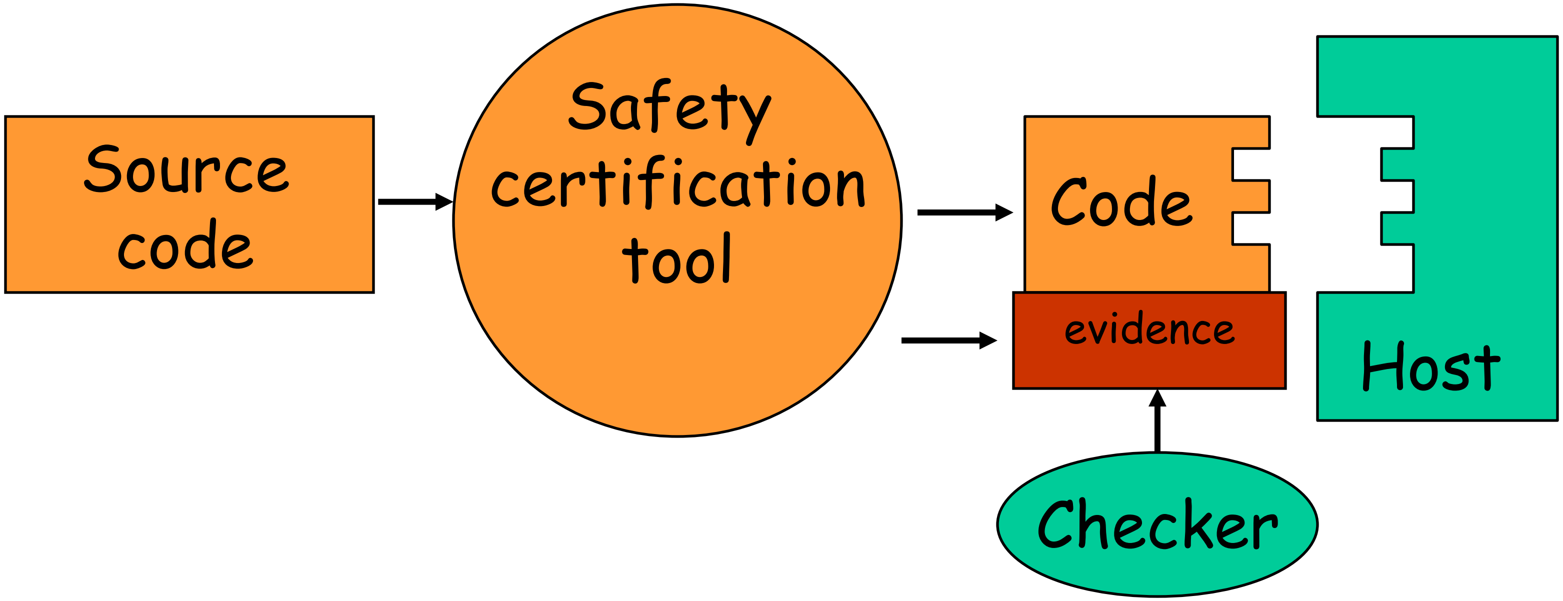
- With some compiler help and user annotations we can have efficient dynamic checking for memory safety

Roadmap

- Static checking vs. dynamic checking
- Dynamic: Enforcing memory safety for C programs
- ➢ Static: Proof-carrying code
  - Type checking Java bytecodes
  - Type checking assembly language
  - Proof-carrying code tools and techniques

Static Checking Made Easy

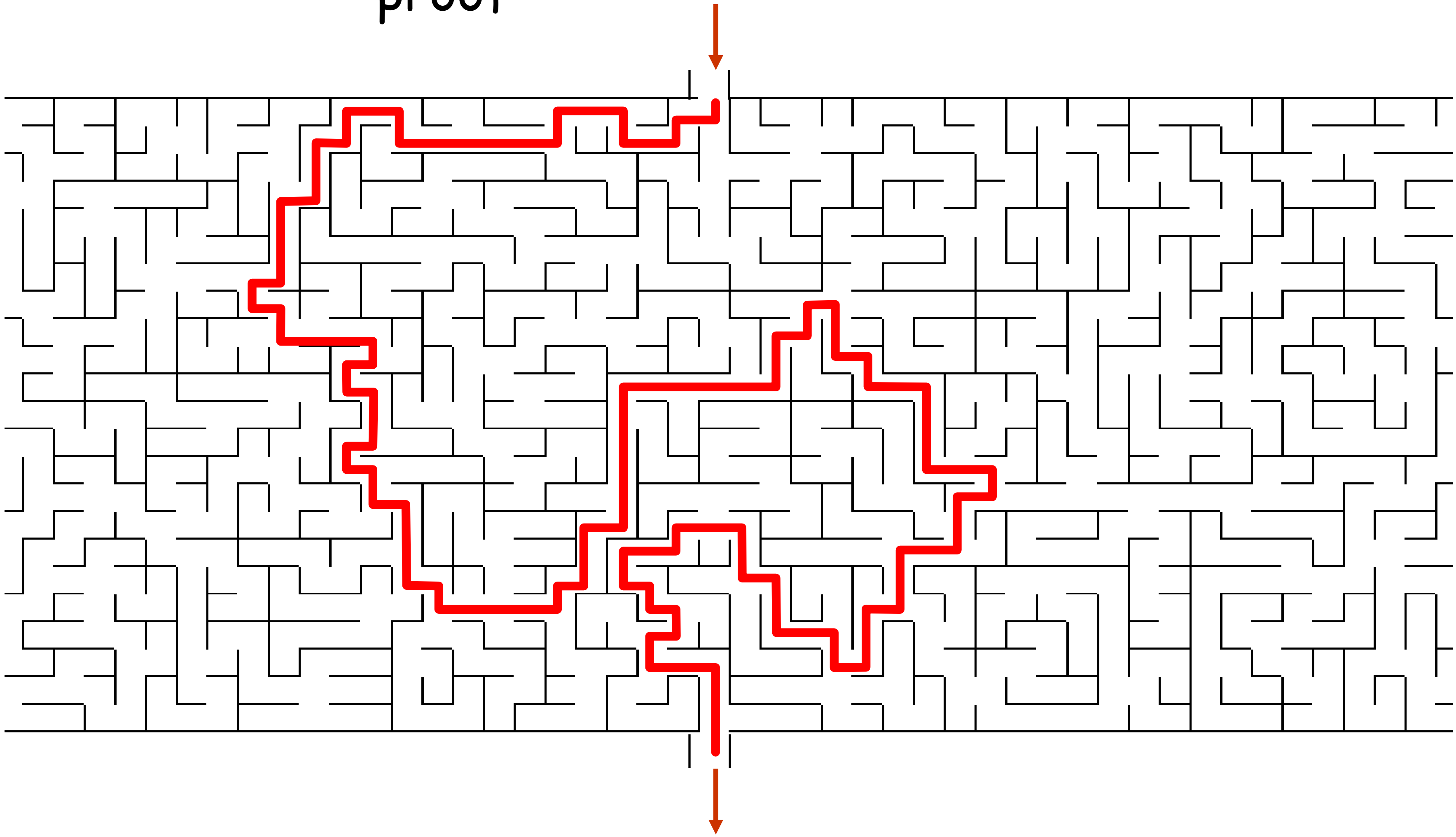
- Static checking is key to safety and performance
- Static checking is possible (and in fact easy) if the client supplies evidence attesting code safety
- For an important class of properties, the evidence can be produced by a client-side tool



Proof-Carrying Code: An Analogy

Legend: ┘ code

— proof



Good Things About PCC

1. Someone else does the really hard work
   - Hard to prove safety but easy to check a proof
2. Requires minimal trusted infrastructure
   - Trust proof checker but not the compiler
3. Agnostic to how the code and proof are produced
   - Hand-optimized code is Ok
4. Flexible and general
   - One checker for many policies
   - "if you can prove it PCC can check it!"
5. Coexists peacefully with cryptography
   - Signatures are a syntactic checksum
   - Proofs are a semantic checksum

What PCC Does Not Do

- PCC is useful when proving is hard
  - Because it requires human assistance
  - Because it requires a long time
  - Because it requires a complex tool
- … and checking is comparatively easy
  - With an automatic and simple proof checker
  - Think of the definition of NP

- PCC cannot be used to prove things about code

- PCC is a transport mechanism, to use after you proved something about your code

Roadmap

- Static checking vs. dynamic checking
- Dynamic: Enforcing memory safety for C programs
- Hybrid: Enforcing resource bounds usage
- Static: Proof-carrying code
  - ➢ Type checking Java bytecodes
  - – Type checking assembly language
  - – Proof-carrying code tools and techniques

Java Virtual Machine (JVM)

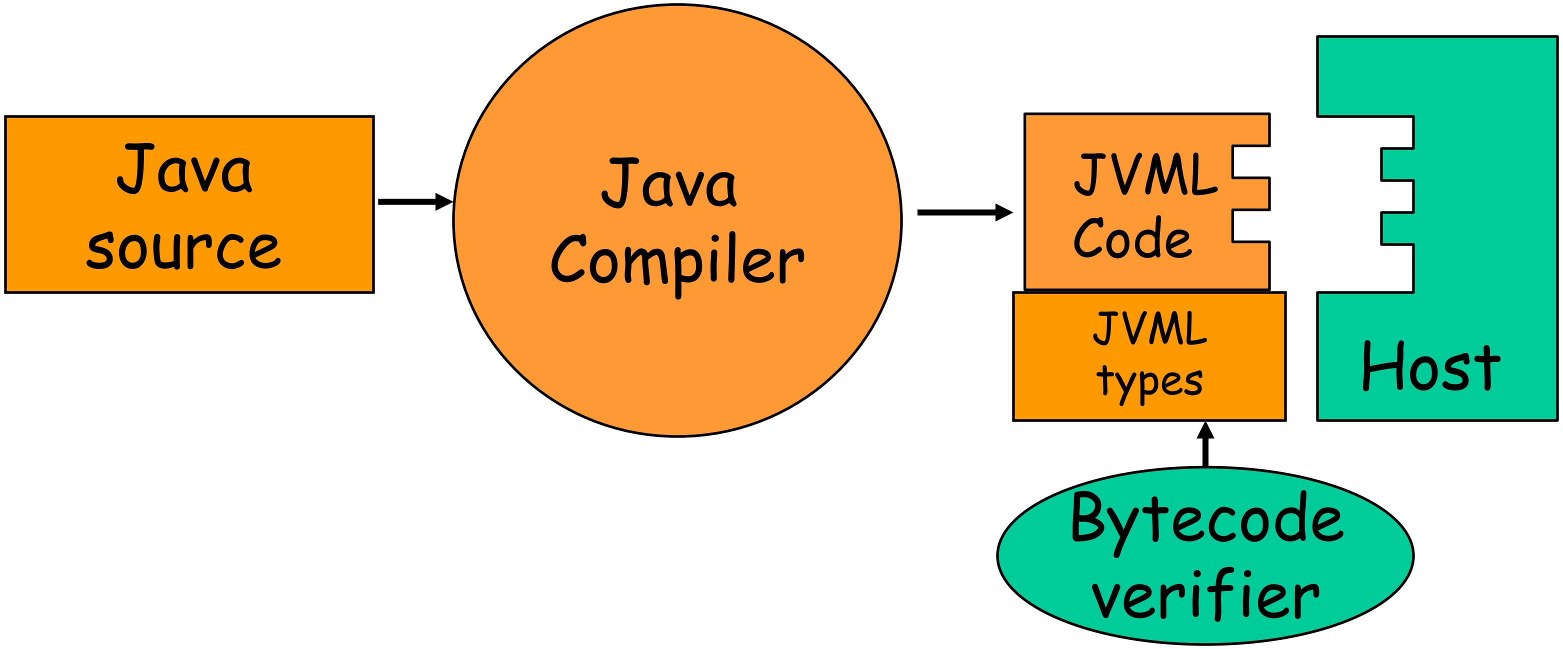
- The first successful attempt to bring type safety to a lower-level language
- Difficulties with low-level languages:
  - Variables (registers) not used consistently with same type
  - High-level operations are "unbundled"
    - allocation and initialization
    - array access and bounds checking
  - Must deal with concrete implementation details
    - stack allocation of locals, calling conventions
    - exception implementation
- JVM tackles some of the above and avoids others by not going too low level

Overview of the JVM

- JVML programs are in .class files
- A .class file contains the implementation of a class
  - Tables describing the class
    - name, attributes, superclass, interfaces, referenced classes
  - Tables describing the fields and methods
    - name, <u>type</u>, attributes (public, private, etc.)
  - The code for the methods in the form of bytecodes

- Before methods in a class are executed, a <u>bytecode verifier</u> checks the type safety of the code

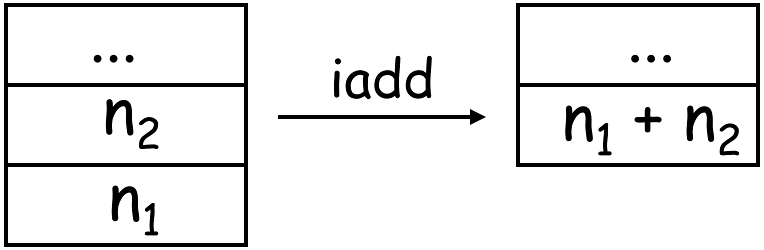
Java Bytecode Verification



- Theorem: if BV succeeds then the JVML code is (type) safe to execute

JVM Runtime Data Structures

- Java heap
  - Used for allocating objects, garbage collected
- Java stack
  - One per thread, used for method activation frames
  - Activation frames containing:
    - Local variables (a.k.a., registers)
    - An operand stack, used for operator arguments and results
  - Example: **iadd** adds two integers on the top of stack



Typed Instructions

- Most JVM instructions are typed !

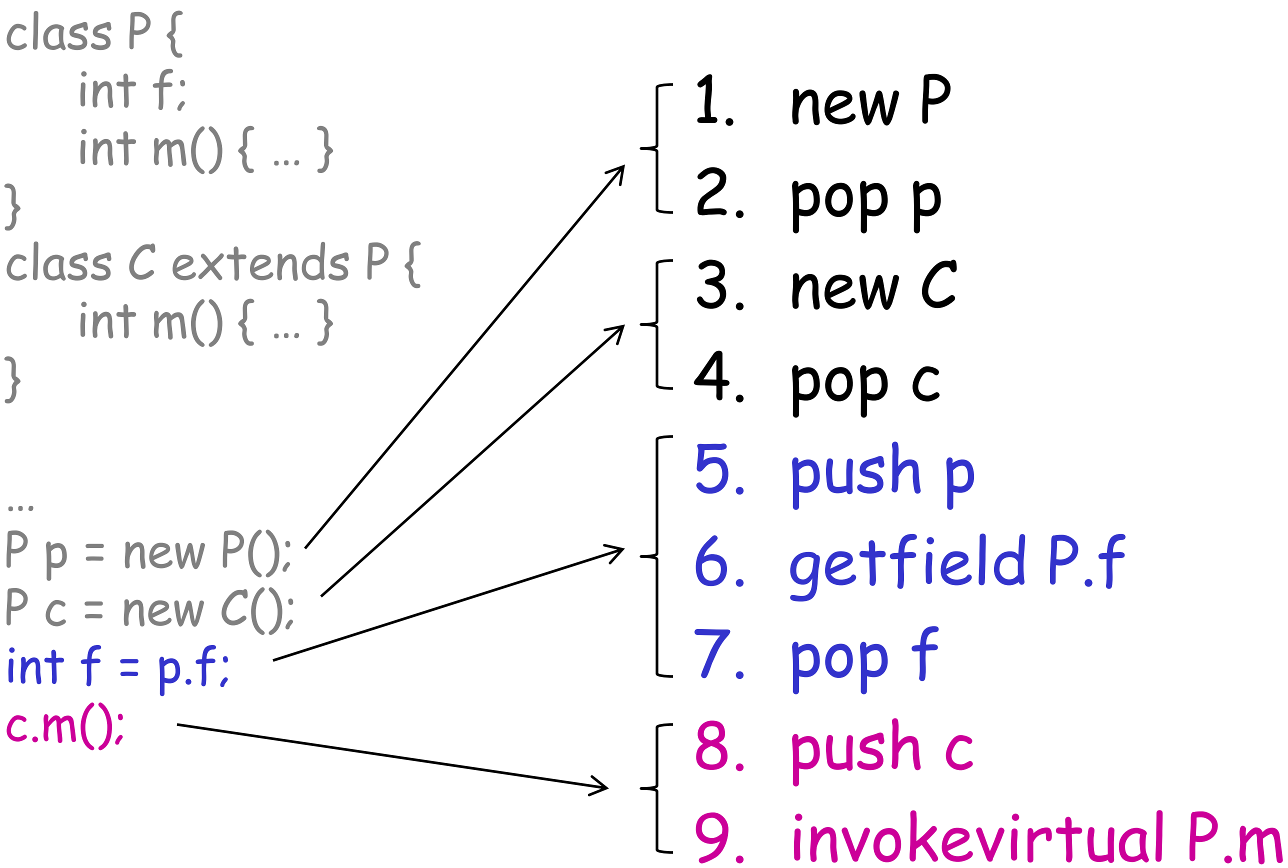
Example:

- "xload v" ($x \in \{a, i, l, f, d\}$)
  - Loads (i.e. pushes) a variable v on the stack
  - The prefix specifies the type
  - If x = l (long) or x = d (double) then two words are pushed
  - Otherwise, the type annotation is only for type checking

Built-In Support

- Objects
  - Code does not access objects directly
  - "getfield name" for reading fields
  - "invokevirtual name" for invoking methods
  - "invokeinterface name" for invoking methods in interfaces
  - "invokespecial name" for constructors
- Arrays
  - Bounds checking
  - Run-time type checking for aastore (due to covariance)
- Exceptions
  - JVM handles the stack unwinding
- This way JVM side-steps many difficult issues
  - But also kills many opportunities for optimization

Example of JVM Verification



```
class P {
    int f;
    int m() { … }
}
class C extends P {
    int m() { … }
}

…
P p = new P();
P c = new C();
int f = p.f;
c.m();
```

1. new P
2. pop p
3. new C
4. pop c
5. push p
6. getfield P.f
7. pop f
8. push c
9. invokevirtual P.m

Errors in JVML Programs

- We'd like to know that the JVML program is obtained by correct compilation from well-typed Java programs
- Instead verify that the JVML program is safe
- None of the following are allowed:
  - Type errors
  - Operand stack overflow or underflow
  - Access control violations (e.g., private fields and methods)
  - Reading of uninitialized variables
  - Use of uninitialized objects
  - Wild jumps
- How do we prevent all these?

The Java Bytecode Verifier

- Helps prevent errors by checking untrusted JVML code before execution

- Essentially a system for type inference for programs with unstructured control flow

JVML Verification Strategy

- Evaluate the program symbolically, remembering only the types of registers and stack slots
- Evaluation state:

  <pc, F, S>

  - where pc is the program counter
  - F is a mapping from register names to types
  - Types are the class names along with primitive types
  - S is a stack of types: Stack ::= empty | τ :: S
- Example:

  <1, [x:=int; f:=C], P :: C :: _ >

  - means: program counter is 1, x has type int, f has type C, the stack contains at least two elements of type P and C, respectively (P is on top of stack)

JVML Typechecking Rules

$$\frac{I(pc) = \text{new } P}{\langle pc, F, S\rangle \rightarrow \langle pc+1, F, P :: S\rangle}$$

$$\frac{I(pc) = \text{pop } x}{\langle pc, F, \tau :: S\rangle \rightarrow \langle pc+1, F[x:=\tau], S\rangle}$$

$$\frac{\begin{array}{l} I(pc) = \text{getfield } P.f \\ P' \text{ subtype of } P \\ P \text{ has field } f \text{ of type } \tau \end{array}}{\langle pc, F, P' :: S\rangle \rightarrow \langle pc+1, F, \tau :: S\rangle}$$

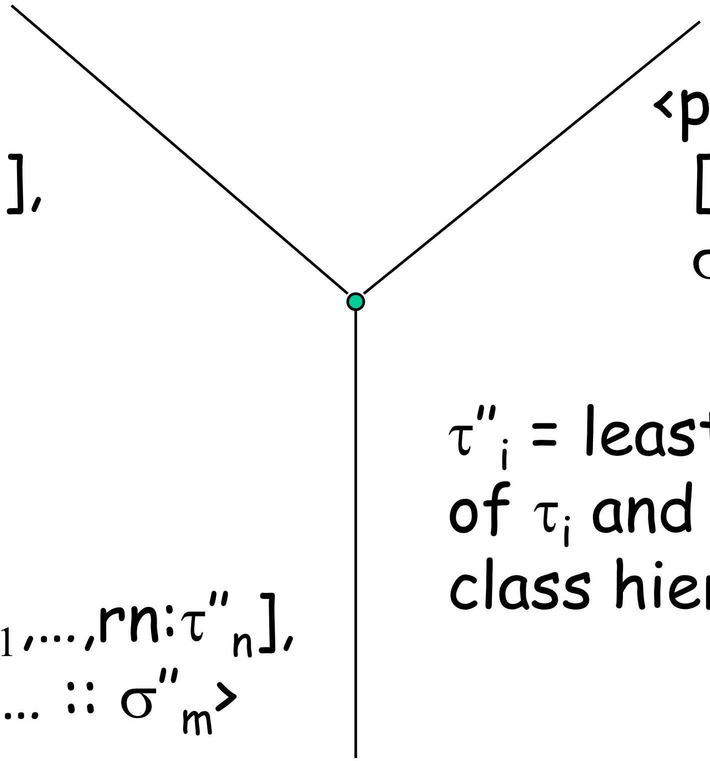
$$\frac{\begin{array}{l} I(pc) = \text{invokevirtual } P.m \\ P' \text{ subtype of } P \\ P \text{ has method } m \\ \qquad \text{of type } \tau_1 \times \ldots \times \tau_n \rightarrow \tau \\ S = \tau_1' :: \ldots :: \tau_n' :: S' \\ \text{For each } i, \tau_i' \text{ subtype of } \tau_i \end{array}}{\langle pc, F, P' :: S\rangle \rightarrow \langle pc+1, F, \tau :: S'\rangle}$$

Example of JVM Verification

```
class P {
    int f;
    int m() { … }
}
class C extends P {
    int m() { … }
}

…
P p = new P();
P c = new C();
int f = p.f;
c.m();
```

1. new P	<1, F, S>
2. pop p	<2, F, P :: S>
3. new C	<3, F[p:P], S>
4. pop c	<4, F[p:P], C :: S>
5. push p	<5, F[p:P,c:C], S>
6. getfield P.f	<6, F[p:P,c:C], P :: S>
7. pop f	<7, F[p:P,c:C], int ::S>
8. push c	<8, F[p:P,c:C,f:int], S>
9. invokevirtual P.m	<9, F[p:P,c:C,f:int], C :: S>

Join Points



<pc,
$[r1{:}\tau_1,\ldots,rn{:}\tau_n]$,
$\sigma_1 :: \ldots :: \sigma_m$>

<pc,
$[r1{:}\tau'_1,\ldots,rn{:}\tau'_n]$,
$\sigma'_1 :: \ldots :: \sigma'_m$>

<pc,
$[r1{:}\tau''_1,\ldots,rn{:}\tau''_n]$,
$\sigma''_1 :: \ldots :: \sigma''_m$>

τ''_i = least common ancestor of τ_i and τ'_i in the class hierarchy

- Continue checking unless result is the same as the previous state at the join point
  - Terminates because of finite class hierarchy

Join Points: Subtleties

- May need to verify some code fragments multiple times
  - An $O(n^2)$ complexity bound (some bad implementations even worse)
  - This is not true for Java, only Java bytecode !
  - KVM avoids this with type declarations
- Verification is sound and guaranteed to terminate
- Denial-of-service attack: an adversary sends you a worst-case bytecode program
  - Your browser will hang trying to verify the code (15 minutes on a 3GHz machine)

Java Exceptions

- Java has typed exceptions
- Exceptions can be handled with catch and/or finally

```
int test (int i) {
   try {
       if (i == 3) return foo ();
   } finally {
        bar ();
   }
   i ++;
   return i;
}
```

JVML Subroutines

- A simple solution is to duplicate the “finally” code
- To avoid this, the finally body is compiled into a *subroutine*
  - The subroutine is called from each escape point
  - A subroutine executes in the same activation frame as the host
  - Has access to, and can modify all local variables
- Typing challenges
  - Call points of subroutines need not agree on the type of all local variables; only the ones used in suroutine
  - Polymorphism is needed
  - Subroutines need not be LIFO

JVML Subroutines

Subroutines are the most difficult part of the verifier

- several bugs and inconsistencies in the implementation
- 14 of 26 proof invariants
- 50 of 120 lemmas
- 70 of 150 pages of proof

- Subroutines save space?
  - About 200 subroutines in 650 Klines of Java (mostly in JDK)
  - No subroutines calling other subroutines
  - Subroutines save 2427 bytes of 8.7 Mbytes (0.02%)!
  - Changing the name Java to Oak saves 13 times more space!
  - Latest version of javac does not use subroutines anymore

Roadmap

- Static checking vs. dynamic checking
- Dynamic: Enforcing memory safety for C programs
- Static: Proof-carrying code
  - Type checking Java bytecodes
  - ➢ Type checking assembly language
  - Proof-carrying code tools and techniques

Bytecode -> Assembly language

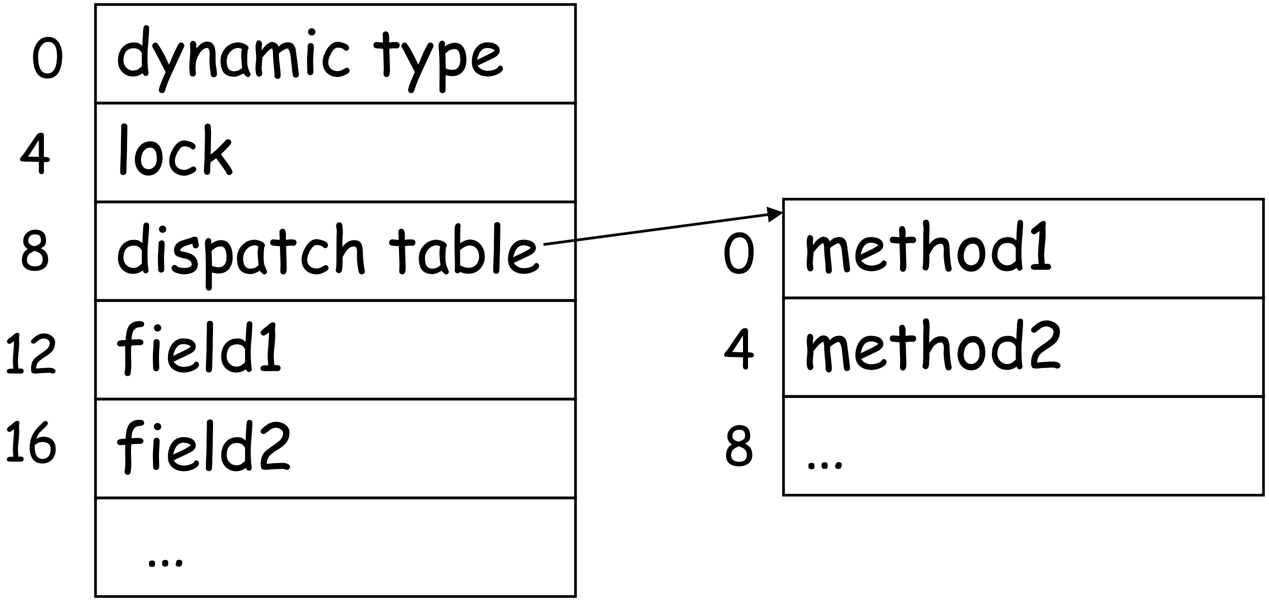
- Bytecode verification is quite powerful
  - Requires few annotations
  - Derives its simplicity from carefully crafted high-level bytecode language

- Can we apply similar ideas for the assembly language output of a just-in-time compiler?
  - Why is this interesting?

Compilation of JVML to Assembly

- We must work with the concrete object layout:

offset:



0	dynamic type
4	lock
8	dispatch table
12	field1
16	field2
	…

0	method1
4	method2
8	…

Checking Method Invocation

```
class P {
    int f;
    int m() { … }
}
class C extends P {
    int m() { … }
}

…
P p = new P();
P c = new C();
c.m();
…
```

```
push c
invokevirtual P.m()
```

```
push c
invokevirtual P.m()
```

branch (= r_c 0) L_{abort}

r_{tmp} := m[r_c + 8]

r_{tmp} := m[r_{tmp} + 12]

r_{arg0} := r_c

r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

Checking Method Invocation

- We must give types to intermediate results

- Idea: invent types for intermediate results
  - after doing the null check for an object of type P

    nonnull(P)
  - result of fetching dispatch table of object of type P

    disp(P)
  - result of fetching kth method from table of class P

    method(P, k)
  - pointer to a field of type F

    ptr(F)
  - …
- We write appropriate typing rules

Checking Method Invocation

```
…
P p = new P();
P c = new C();
c.m();
…
```

invokevirtual *P.m()*

Typing rule:

$$\frac{r : \text{nonnull } P}{m[r + 8] : \text{disp}(P)}$$

$\langle r_c : P, \ldots \rangle$

branch (= r_c 0) L_{abort}

$\langle r_c : \text{nonnull } P, \ldots \rangle$

r_{tmp} := m[r_c + 8]

$\langle r_{tmp} : \text{disp}(P), \ldots \rangle$

r_{tmp} := m[r_{tmp} + 12]

r_{arg0} := r_c

r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

Checking Method Invocation

```
…
P p = new P();
P c = new C();
c.m();
…
```

invokevirtual *P.m()*

$\langle r_c : P, \ldots \rangle$

branch (= r_c 0) L_{abort}

$\langle r_c : \text{nonnull } P, \ldots \rangle$

r_{tmp} := m[r_c + 8]

$\langle r_{tmp} : \text{disp}(P), \ldots \rangle$

r_{tmp} := m[r_{tmp} + 12]

$\langle r_{tmp} : \text{meth}(P,3), \ldots \rangle$

r_{arg0} := r_c

r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

Typing rule:

$$\frac{r : \text{disp}(P)}{m[r + 4k] : \text{meth}(P, k)}$$

Checking Method Invocation

```
…
P p = new P();
P c = new C();
c.m();
…
```

invokevirtual *P.m()*

$\langle r_c : P, \ldots \rangle$

branch (= r_c 0) L_{abort}

$\langle r_c : \text{nonnull } P, \ldots \rangle$

r_{tmp} := m[r_c + 8]

$\langle r_{tmp} : \text{disp}(P), \ldots \rangle$

r_{tmp} := m[r_{tmp} + 12]

$\langle r_{tmp} : \text{meth}(P,3), \ldots \rangle$

r_{arg0} := r_c

$\langle r_{arg0} : P, \ldots \rangle$

r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

$\langle r_{rv} : \text{int}, \ldots \rangle$

Typing rule:

r : meth(P, k)

k^{th} method in class P has arg. D and return R

r_{arg0}: P

r_{arg1}: D

r_{ra} : &L (next instr)

(Jump [r]; L:) ⇒ r_{rv} : R

Compiling Virtual Method Dispatch

- Regular compilation of c.m()

  pfunc = kth method in table of c

  call pfunc(c)

  - The called method needs to take the "host" object as argument
  - Or another object of the same dynamic type

- What if the compiler passes "p" as host argument?

Unsoundness

```
…
P p = new P();
P c = new C();
c.m();
…
```

invokevirtual *P.m()*

Typing rule:

~~r : meth(C, k)~~

~~k^th method in class C has arg. D and return R~~

~~r_{arg0}: C~~

~~r_{arg1}: D~~

~~r_{ra} : &L (next instr)~~

~~(Jump [r]; L:) r_{rv} : R~~

branch (= r_c 0) L_{abort}

r_{tmp} := m[r_c + 8]

r_{tmp} := m[r_{tmp} + 12]

r_{arg0} := **r_p**

r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

unsound

~~⟨r_c : P, …⟩~~

~~⟨r_c : nonnull P, …⟩~~

~~⟨r_{tmp} : disp(P), …⟩~~

~~⟨r_{tmp} : meth(P,3), …⟩~~

~~⟨r_{arg0} : **P**, …⟩~~

~~⟨r_{rv} : int, …⟩~~

More Challenges

```
class P {
      int f;
      int m() { … }
}
class C extends P {
      int m() { … }
}


…
P p = new P();
P c = new C();
int f = p.f;
p.m();
x = f + 1;
```

branch $(= r_p\ 0)\ L_{abort}$

$r_{tmp} := r_p + 12$

$r_f := m[r_{tmp}]$

$\langle r_{tmp} : ptr(int), \ldots \rangle$

branch $(= r_p\ 0)\ L_{abort}$

~~$r_{tmp} := m[r_p + 8]$~~

$r_{tmp} := m[r_{tmp} + 12]$

$r_{arg0} := r_p$

$r_{ra} := \&L_{ret}$

jump $[r_{tmp}]$

L_{ret}:

$r_x := r_f + 1$

reordering
and
optimization

More Challenges

```
class P {
    int f;
    int m() { … }
}
class C extends P {
    int m() { … }
}

…
P p = new P();
P c = new C();
int f = p.f;
p.m();
x = f + 1;
```

branch (= r_p 0) L_{abort}

r_{tmp} := r_p + 12

r_f := m[r_{tmp}]

r_x := r_f + 1

~~branch (= r_p 0) L_{abort}~~

r_{tmp} := m[r_{tmp} - 4]

r_{tmp} := m[r_{tmp} + 12]

r_{arg0} := r_p

r_{ra} := &L_{ret}

jump [r_{tmp}]

L_{ret}:

⟨r_{tmp} : ptr(int), … ⟩

“funny” pointer arithmetic

Low-level Type Checking

- We must keep track of dependencies
  - E.g., carry equality information
- We must deal with compiler optimizations
  - E.g., carry arithmetic equalities

- Solution: instead of simple types, use dependent types:

"register r_{tmp} contains the dispatch table of object in register r_c"

r_{tmp} : disp(r_c)

Summary: Typechecking Assembly Language

- We have a typechecker for assembly output of Java compiler
  - Same type safety as for JVML
  - But works at lower level and in presence of optimizations

  - We needed more care
  - We needed to extend types with dependencies
  - Type inference becomes more complicated

  - Same idea works for assembly output of other compilers

Overview of the Lectures

- ✓ Proof-carrying code: motivation and overview
- ✓ Type checking Java bytecodes
  - ✓ Type checking assembly language
- Proof-carrying code: design and implementation
  - Verification-condition generation based PCC
  - Foundational proof-carrying code
  - Open Verifier infrastructure for PCC

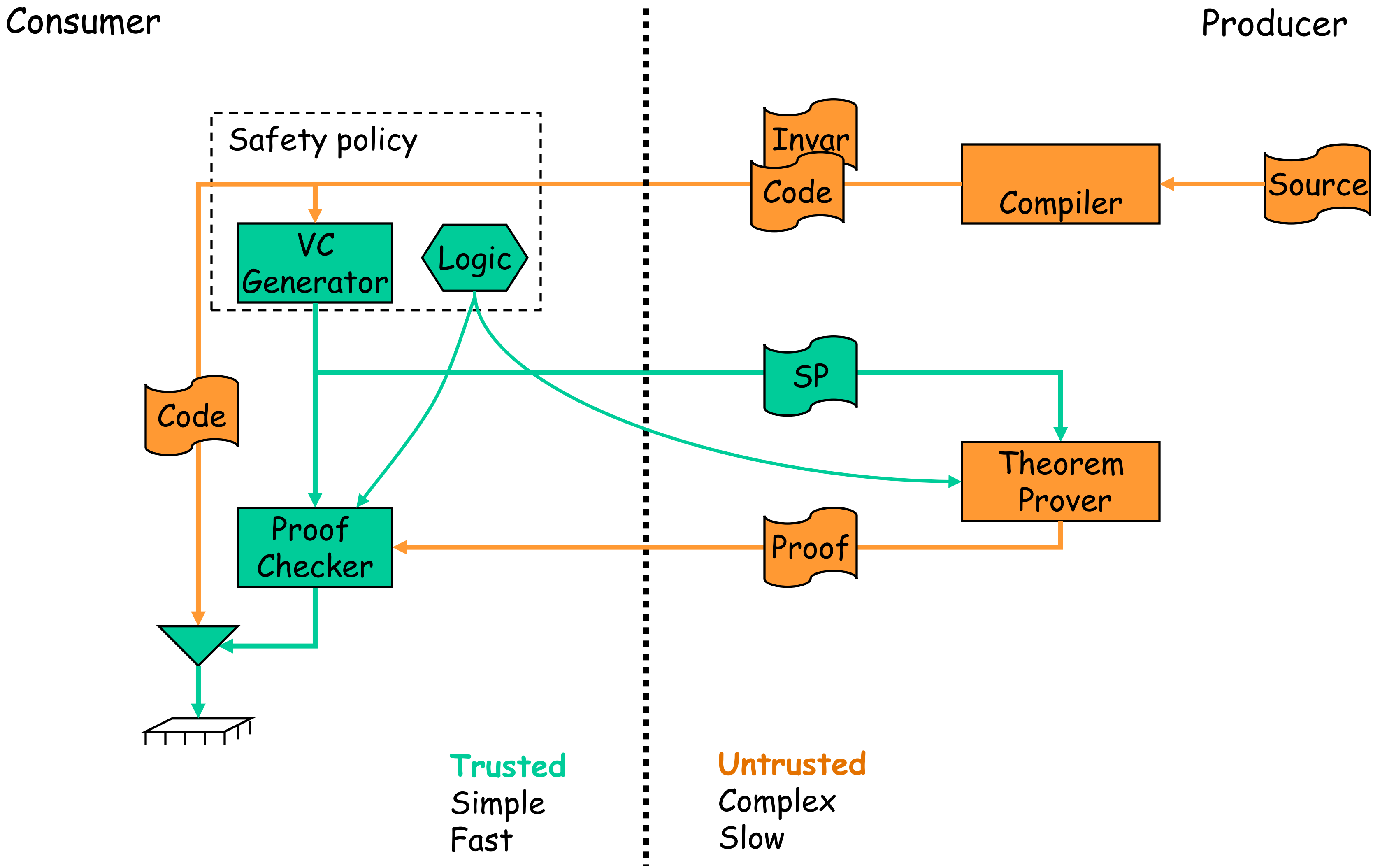
Limitations of Type Safety

- So far the annotations are just hints for type inference
  - Requires few annotations
  - Applicable only when type inference is decidable

- What if we want to allow complex optimizations (e.g., array bounds checking elimination)
  - Complex types and checking (keep track of inequalities)
  - Complex or impossible inference

- We need to:
  - Step beyond simple types (use logic)
  - Get more checking help through annotations (use proofs)

General Proof-Carrying Code



VC Generator: Overview

- Performs simple syntactic checks on the code
  - E.g., verifies that all jump targets are valid
- Produces the safety predicate (SP)
  - For each safety-related operation emits a verification condition (VC) that is provable only if the operation is safe to execute
  - The safety predicate is a "set" of verification conditions
- One pass through the code
  - Needs function specifications and loop invariants
- An old idea from program verification
  - e.g., Floyd, King, Hoare, Dijkstra, etc. ,

VCGen

- VCGen can be viewed as a symbolic evaluator:
  - This is not the traditional formulation of VCGen
  - Traditional view of VCGen is as a backward substitution constructing the weakest precondition

- The symbolic language (for a type-based policy):

$E := x \mid n \mid E_1 + E_2$

(expressions)

$P := E_1 = E_2 \mid E_1 \geq E_2 \mid P_1 \wedge P_2 \mid P_1 \Rightarrow P_2 \mid \forall x.\, P_1$ (formulas)

$\mid \mathrm{saferd}(E) \mid \mathrm{safewr}(E_1, E_2)$ (memory safety formulas)

$\mid E : T$ (typing formulas)

$T ::= \mathrm{int} \mid \mathrm{bool} \mid \mathrm{array}(T,E) \mid \mathrm{pointer}(T)$ (types)

VCGen: Memory Safety

- For a memory read at symbolic address E the verification condition is:

  $$\mathrm{saferd}(E)$$

- For a memory write of symbolic value E_2 at symbolic address E_1 is:

  $$\mathrm{safewr}(E_1, E_2)$$

- It is up to the safety policy to define the meaning of "saferd" and "safewr"
  - VCGen does not depend on a particular safety policy

VCGen: Function Call Safety

- Preconditions
  - Checked at call site and assumed at function start
  - Which registers contain the arguments ?
  - What are the relationships between the arguments ?
  - What can be assumed of the state of memory ?
  - When VCGen sees a function call it emits its precondition as a verification condition

- Postconditions
  - Checked at return and assumed at call site
  - Properties of the return value and the state of memory
  - When VCGen sees "ret" it emits the postcondition as a verification condition

A Simple Example

- Consider the following function:

```
// Compute a conjunction of the booleans from an array
            bool forall(bool a[]) {
                for(int i=0; i<a.length; i++) {
                    if (! a[i]) return false; }
                return true; }
```

- Safety policy:
  - Memory accesses are allowed between a and a + a.length - 1
    - Only reads are allowed from these addresses
  - If the function returns, it must return a boolean
  - 0 and 1 are the only representations of booleans

Safety Policy $\Rightarrow$ Axiomatization

$$\frac{A : \text{array}(T, L) \qquad I \geq 0 \qquad I < L}{\text{saferd}(A + I)}\ \text{rd} \qquad \frac{A : \text{array}(T, L) \qquad I \geq 0 \qquad I < L}{M[A + I] : T}\ \text{typerd}$$

$$\frac{}{0 : \text{bool}}\ \text{bool0} \qquad \frac{}{1 : \text{bool}}\ \text{bool1} \qquad \frac{}{E : \text{int}}\ \text{int}$$

$$\frac{}{E \geq E}\ \text{geqid} \qquad \xcancel{\frac{I \geq E}{I + 1 \geq E}\ \text{inc}} \qquad \frac{I \geq E \qquad I < L \qquad A : \text{array}(T, L)}{I + 1 \geq E}\ \text{inc}$$

An Example: Type-Based Memory Safety

```
        PRE $a : \text{array}(\text{bool}, n)$
        $r \leftarrow 0$
        $i \leftarrow 0$
$L_0$ : INV= $i : \text{int} \wedge i \geq 0$, REG = { m, a, n, r }
        if i >= n goto $L_1$
        $t \leftarrow a + i$
        $t \leftarrow M[t]$
        if not t goto $L_2$
        $i \leftarrow i + 1$
        goto $L_0$
$L_1$:    $r \leftarrow 1$
$L_2$:    return r
        POST $r : \text{bool}$
```

- Safety policy expressed as preconditions and postconditions

Verification Condition Generation

```
→
        PRE a : array(bool, n)
        r $\leftarrow$ 0
        i $\leftarrow$ 0
$L_0$ : INV= i : int $\wedge$ i $\geq$ 0, REG = { m, a, n, r }
        if i >= n goto $L_1$
        t $\leftarrow$ a + i
        t $\leftarrow$ M[t]
        if not t goto $L_2$
        i $\leftarrow$ i + 1
        goto $L_0$
$L_1$:  r $\leftarrow$ 1
$L_2$:  return r
        POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i0
r	r0
t	t0

Assumptions:

Verification Condition Generation

```
      PRE a : array(bool, n)
      r ← 0
      i ← 0
$L_0$ :  INV= i : int ∧ i ≥ 0, REG = { m, a, n, r }
      if i >= n goto $L_1$
      t ← a + i
      t ← M[t]
      if not t goto $L_2$
      i ← i + 1
      goto $L_0$
$L_1$:   r ← 1
$L_2$:   return r
      POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i0
r	r0
t	t0

Assumptions:

a0 : array(bool, n0)

Verification Condition Generation

```
      PRE a : array(bool, n)
      r ← 0
      i ← 0
L0 :  INV= i : int ∧ i ≥ 0, REG = { m, a, n, r }
      if i >= n goto L1
      t ← a + i
      t ← M[t]
      if not t goto L2
      i ← i + 1
      goto L0
L1:   r ← 1
L2:   return r
      POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i0
r	0
t	t0

Assumptions:

a0 : array(bool, n0)

Verification Condition Generation

```
       PRE a : array(bool, n)
       r ← 0
       i ← 0
$L_0$ : INV= i : int ∧ i ≥ 0, REG = { m, a, n, r }
       if i >= n goto $L_1$
       t ← a + i
       t ← M[t]
       if not t goto $L_2$
       i ← i + 1
       goto $L_0$
$L_1$:  r ← 1
$L_2$:  return r
       POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	0
r	0
t	t0

Assumptions:

a0 : array(bool, n0)

Check: 0 : int

$0 \geq 0$

Verification Condition Generation

```
        PRE a : array(bool, n)
        r ← 0
        i ← 0
$L_0$ :  INV= i : int ∧ i ≥ 0, REG = { m, a, n, r }
        if i >= n goto $L_1$
        t ← a + i
        t ← M[t]
        if not t goto $L_2$
        i ← i + 1
        goto $L_0$
$L_1$:   r ← 1
$L_2$:   return r
        POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1
r	0
t	t1

Assumptions:

a0 : array(bool, n0)

i1 : int

i1 ≥ 0

Verification Condition Generation

```
     PRE a : array(bool, n)
     r ← 0
     i ← 0
L0 : INV= i : int ∧ i ≥ 0, REG = { m, a, n, r }
     if i >= n goto L1
     t ← a + i
     t ← M[t]
     if not t goto L2
     i ← i + 1
     goto L0
L1:  r ← 1
L2:  return r
     POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1
r	0
t	t1

Assumptions:

$a0 : \text{array}(\text{bool}, n0)$

$i1 : \text{int}$

$i1 \geq 0$

$i1 < n0$

Verification Condition Generation

```
      PRE a : array(bool, n)
      $r \leftarrow 0$
      $i \leftarrow 0$
$L_0$ :  INV= i : int $\wedge$ i $\geq$ 0, REG = { m, a, n, r }
      if i >= n goto $L_1$
→     $t \leftarrow a + i$
      $t \leftarrow M[t]$
      if not t goto $L_2$
      $i \leftarrow i + 1$
      goto $L_0$
$L_1$:   $r \leftarrow 1$
$L_2$:   return r
      POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1
r	0
t	a0 + i1

Assumptions:

a0 : array(bool, n0)
i1 : int
$i1 \geq 0$
$i1 < n0$

Check: saferd(a0 + i1)

Verification Condition Generation

```
      PRE a : array(bool, n)
      r ← 0
      i ← 0
$L_0$ :  INV= i : int $\wedge$ i $\geq$ 0, REG = { m, a, n, r }
      if i >= n goto $L_1$
      t ← a + i
→     t ← M[t]
      if not t goto $L_2$
      i ← i + 1
      goto $L_0$
$L_1$:   r ← 1
$L_2$:   return r
      POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1
r	0
t	m0[a0 + i1]

Assumptions:

a0 : array(bool, n0)

i1 : int

i1 $\geq$ 0

i1 $<$ n0

Verification Condition Generation

```
      PRE a : array(bool, n)
      r ← 0
      i ← 0
L0 :  INV= i : int ∧ i ≥ 0, REG = { m, a, n, r }
      if i >= n goto L1
      t ← a + i
      t ← M[t]
      if not t goto L2
→     i ← i + 1
      goto L0
L1:   r ← 1
L2:   return r
      POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1
r	0
t	m0[a0 + i1]

Assumptions:

- $a0 : \text{array}(\text{bool}, n0)$
- $i1 : \text{int}$
- $i1 \geq 0$
- $i1 < n0$
- $m0[a0+i1] = \text{true}$

Verification Condition Generation

```
        PRE a : array(bool, n)
        r ← 0
        i ← 0
$L_0$ : INV= i : int $\wedge$ i $\geq$ 0, REG = { m, a, n, r }
        if i >= n goto $L_1$
        t ← a + i
        t ← M[t]
        if not t goto $L_2$
        i ← i + 1
→
        goto $L_0$
$L_1$:  r ← 1
$L_2$:  return r
        POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1 + 1
r	0
t	m0[a0 + i1]

Assumptions:

a0 : array(bool, n0)

i1 : int

i1 $\geq$ 0

i1 $<$ n0

m0[a0+i1] = true

Verification Condition Generation

```
     PRE a : array(bool, n)
     r ← 0
     i ← 0
L_0 : INV= i : int $\wedge$ i $\geq$ 0, REG = { m, a, n, r }
     if i >= n goto L_1
     t ← a + i
     t ← M[t]
     if not t goto L_2
     i ← i + 1
     goto L_0
L_1: r ← 1
L_2: return r
     POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1 + 1
r	0
t	m0[a0 + i1]

Assumptions:

a0 : array(bool, n0)
i1 : int
$i1 \geq 0$
$i1 < n0$
sel(m0,a0+i1) = true

Check: $i1 + 1 : int \wedge i1 + 1 \geq 0$

Verification Condition Generation (Backtrack)

```
     PRE a : array(bool, n)
     $r \leftarrow 0$
     $i \leftarrow 0$
L0 : INV= i : int $\wedge$ i $\geq$ 0, REG = { m, a, n, r }
     if i >= n goto L1
     $t \leftarrow a + i$
→    $t \leftarrow M[t]$
     if not t goto L2
     $i \leftarrow i + 1$
     goto L0
L1:  $r \leftarrow 1$
L2:  return r
     POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1
r	0
t	m0[a0 + i1]

Assumptions:

a0 : array(bool, n0)
i1 : int
$i1 \geq 0$
$i1 < n0$

Verification Condition Generation

```
       PRE a : array(bool, n)
       r ← 0
       i ← 0
$L_0$ :   INV= i : int $\wedge$ i $\geq$ 0, REG = { m, a, n, r }
       if i >= n goto $L_1$
       t ← a + i
       t ← M[t]
       if not t goto $L_2$
       i ← i + 1
       goto $L_0$
$L_1$:    r ← 1
→ $L_2$:  return r
       POST r : bool
```

Symbolic register file:

a	a0
n	n0
m	m0
i	i1
r	0
t	m0[a0 + i1]

Assumptions:

a0 : array(bool, n0)

i1 : int

i1 $\geq$ 0

i1 $<$ n0

m0[a0+i1] = false

Check: 0 : bool

The Safety Predicate

Assumptions

$a0 : \text{array}(\text{bool}, n0)$

$i1 : \text{int}$

$i1 \geq 0$

$i1 < n0$

$m0[a0+i1] = \text{true}$

$m0[a0+i1] = \text{false}$

$i1 \geq n0$

Verification conditions

$0 : \text{int} \wedge 0 \geq 0$ (INV$_0$)

$\text{saferd}(a0+i1)$ (READ)

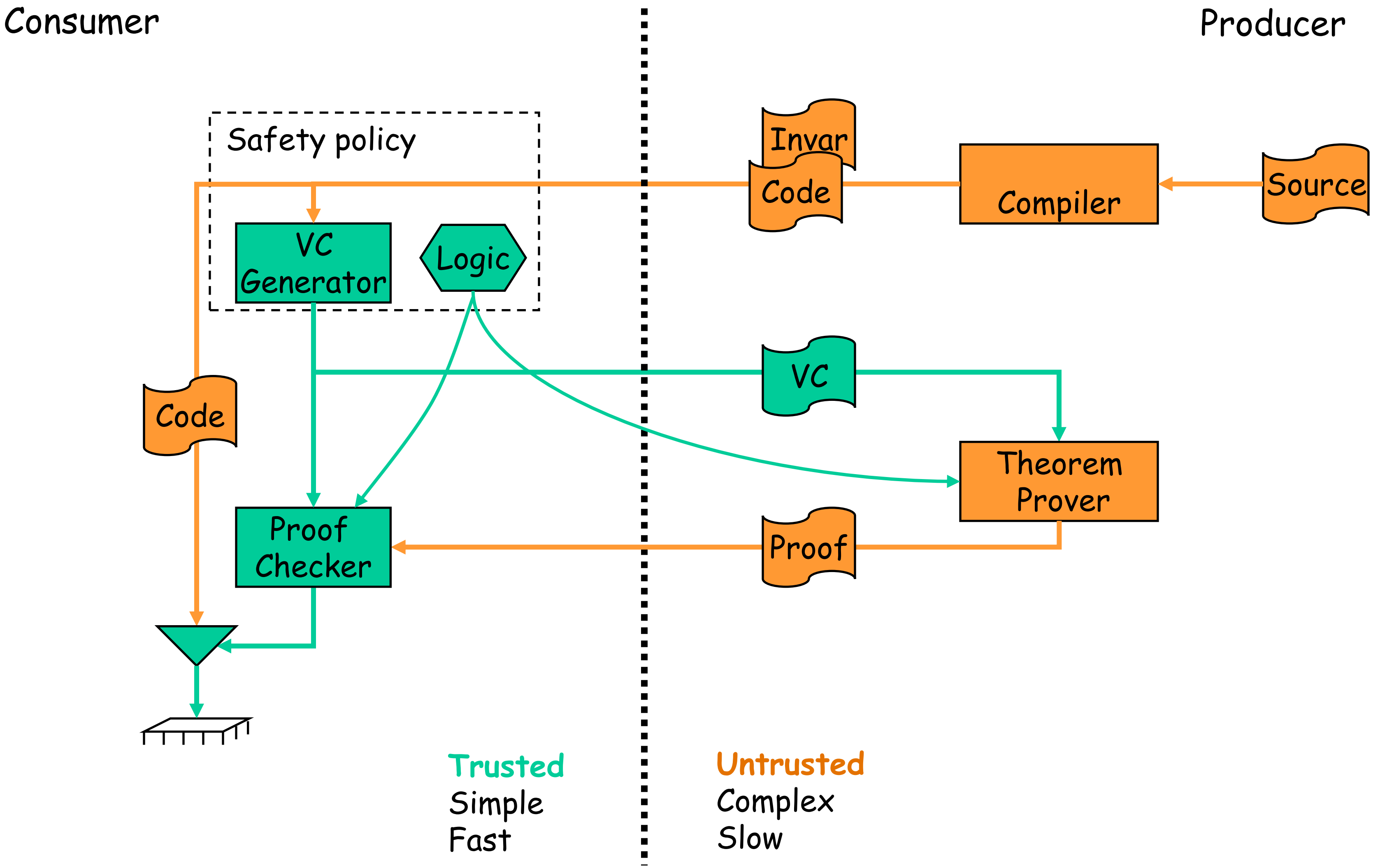
$i1 + 1 : \text{int} \wedge i1 + 1 \geq 0$ (INV$_1$)

$0 : \text{bool}$ (POST)

$1 : \text{bool}$ (POST)

PCC Client-Side Tools

Proof-Carrying Code. Design Details



Consumer

Producer

Safety policy

VC Generator

Logic

Code

Proof Checker

Invar

Code

Compiler

Source

VC

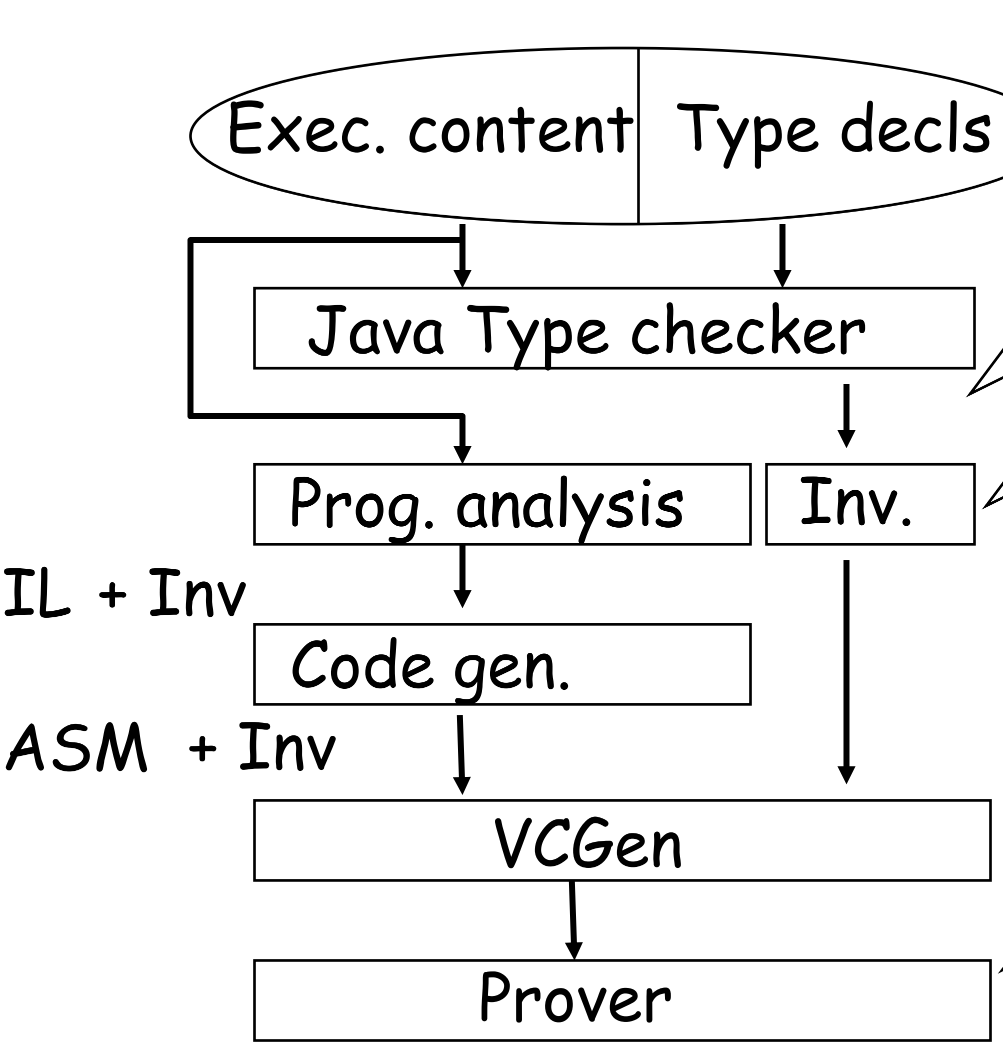
Theorem Prover

Proof

Trusted
Simple
Fast

Untrusted
Complex
Slow

A Certifier Compiler for Java



Don’t loose the types
Remember what was inferred

Only as smart as the type checker and analyzer

The Kettle Theorem Prover

- Automatic prover for
  - linear arithmetic, uninterpreted functions
  - quantifiers are handled with heuristics
  - Parameterized by typing rules (specific to type system)

$$\frac{e = \alpha + 8 \qquad \Gamma \vdash \alpha : \text{nonnull } C}{\Gamma \vdash e : \text{ptr}(\text{disp}(\alpha))}$$

- Constructs proofs upon success
  - In terms of natural deduction rules for FOL and typing rules

Proof Engineering

Proof Engineering

- Important for practical use of PCC
  - Must transport and check proofs
- Also important in other applications using explicit proof representations
  - Proof-generating theorem provers

Desired Characteristics

- General framework
  - Applicable to many logics
  - Allows high-level description of the logic

- Simple and fast proof checking
  - Parameterized by the logic (so we don't have to rewrite it over and over)

- Compact representations of proofs
  - Reduces bandwidth needed in Proof-Carrying Code
  - Reduces space required for storage of proofs
  - Speeds-up proof validation

Proof Representation Strategies

1. A proof is a proof script for a proof assistant
   - You get the checker for free, proofs are small
   - The checker is unnecessarily large and complex

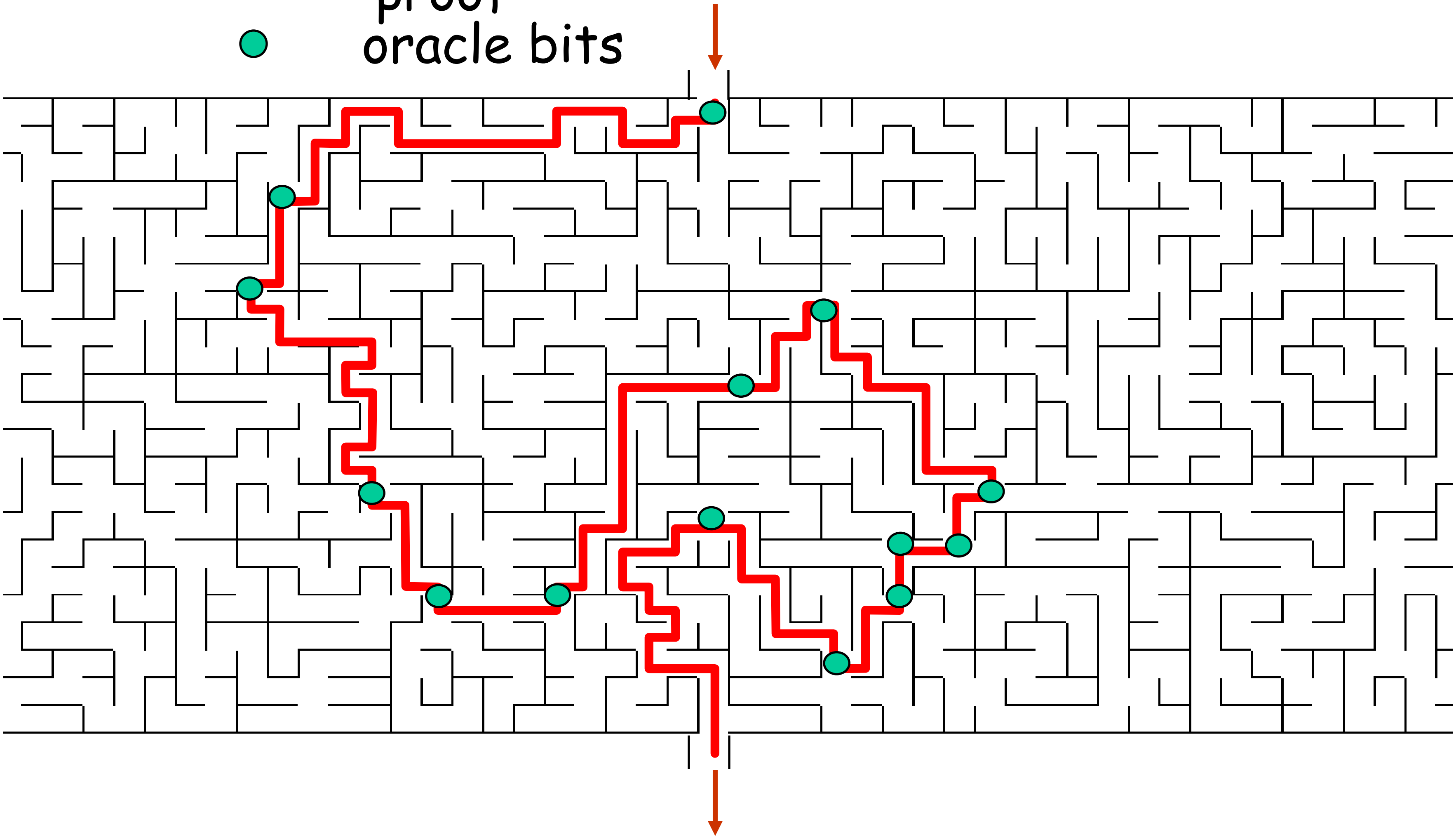
2. Or, design an ad-hoc proof representation language
   - Proofs are trees, nodes are labeled with proof rules, children correspond to premises of a rule
   - Must be careful with hypothetical judments
   - Proofs are small
   - Size of proof checker is linear in the # of proof rules

3. Oracle-based PCC

Legend: code

— proof

● oracle bits



Proof Representation. Conclusion

- There is a wide range of proof representation strategies

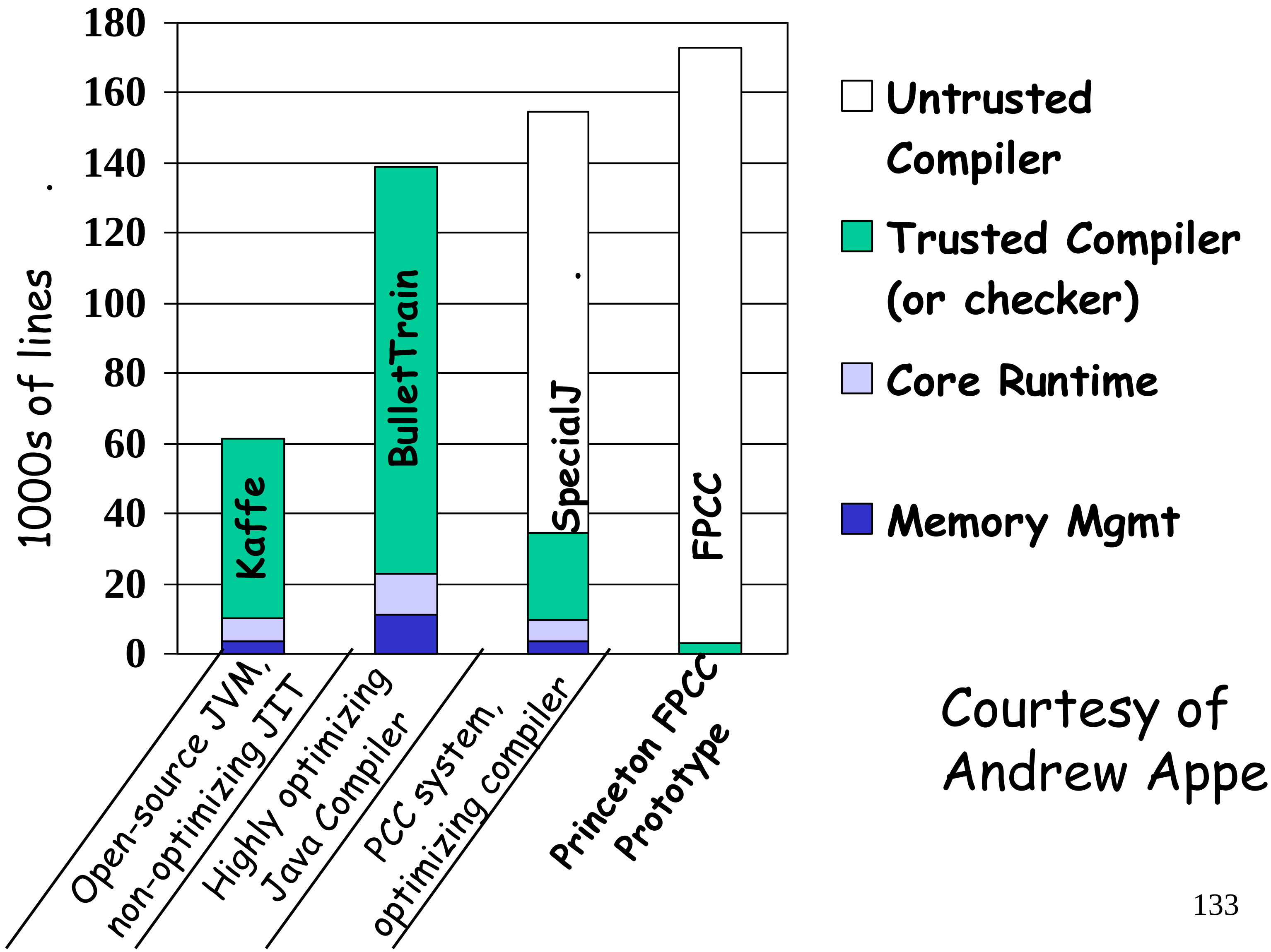
- Usually, the simpler the checker, the larger the proof must be
  - But there are some nice compromise points

- There are variants of PCC where the proof size does not matter that much

Overview of the Lectures

- ✓ Proof-carrying code: motivation and overview
- ✓ Type checking Java bytecodes
  - ✓ Type checking assembly language
- Proof-carrying code: design and implementation
  - ✓ Verification-condition generation based PCC
  - Foundational proof-carrying code
  - Open Verifier infrastructure for PCC

Foundational Proof Carrying Code



Courtesy of Andrew Appel

What About Proof Generation ?

- The focus so far has been on the infrastructure
  - Touchstone: scalable to large programs, but large TCB
  - FPCC: very small TCB, very difficult to produce proofs
  - Can we get the best of both ?

- Often overlooked detail:

  Must have proofs to have PCC !

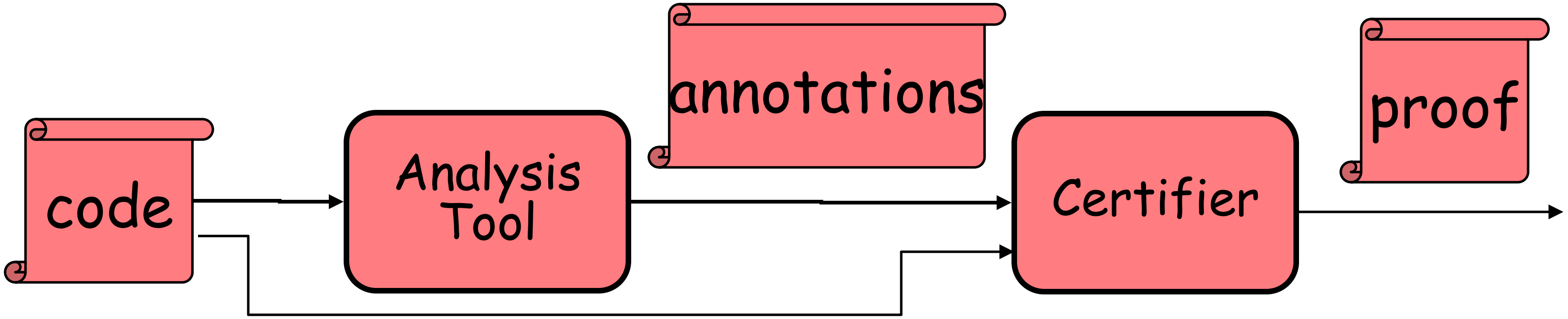
- Most of the cost of PCC is in proof generation
- Find low-cost strategies to generate the proofs

Common Safety Checking Tools

- Theorem proving
  - For complex properties on small codes
- Model checking
- Type checking, data-flow analysis and instrumentation
  - JVML, MSIL, TAL, CQual, Stackguard, Deputy, …
  - Includes virtually all PCC experiments to date

- Must be easy to obtain proofs from such tools

Certified Analysis Tools



- We separate the certification from analysis tool
  - Analysis tool emits annotations to help the certifier

- Examples:
  - Type inference + type checking
  - Model checker + invariant checking
  - Java compilation + bytecode verification

Certified Program Analysis Tools

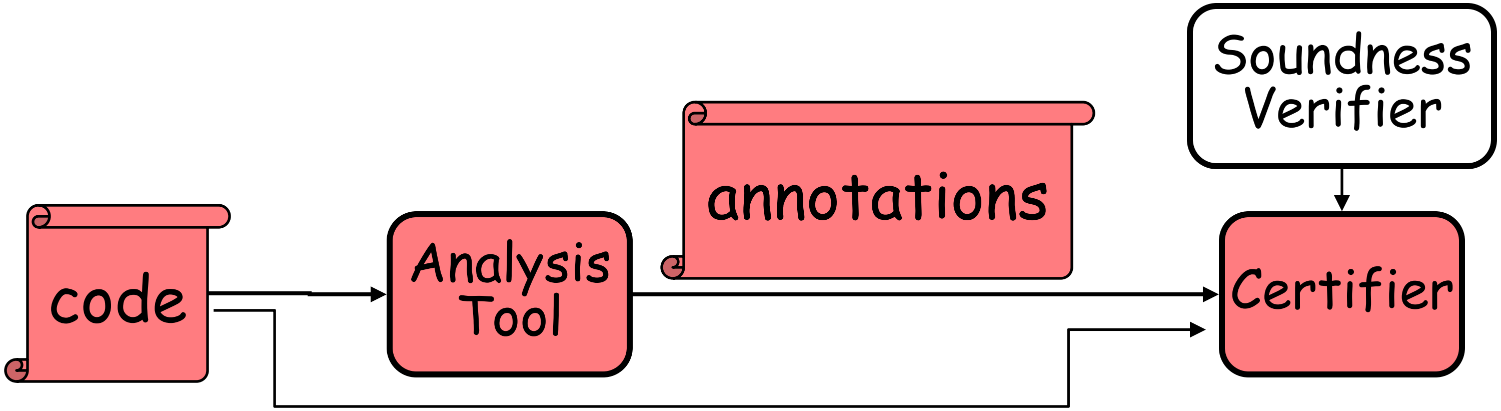
- Certifier and annotations customized for each analysis tool

- Advantages:
  - Easy debugging of analysis/instrumentation tools
  - Reduces soundness of tool to certifier soundness

- For PCC, we need proof-generating certifiers
  - We assume we know how to write certifiers
  - How to write proof-generating certifiers ?

Writing Certifiers

- Method 1: Proof-generating certifiers
  - Extend each certification step with proof generation
  - Glue together the proofs for individual steps
  - Experience: 2x code size increase, 25x slow-down
- Method 2: Verified certifiers
  - Prove statically the soundness of the certifier



Verified Certifiers

- How can we prove soundness of a certifier?
  - Harder than proving safety of each program
    - But needs to be done only once

- We can use a generic framework and tools for abstract-interpretation based certifiers
  - Write the certifier in Ocaml
  - Generate automatically a few Coq theorems to prove

"You write the type checker and we generate the soundness statement for the typing rules it uses"

A Flexible Variant of PCC

- We do not need proofs for each program
  - Send the certifier with its soundness proof
  - Then send annotations for each program
  - Certification is simpler and faster

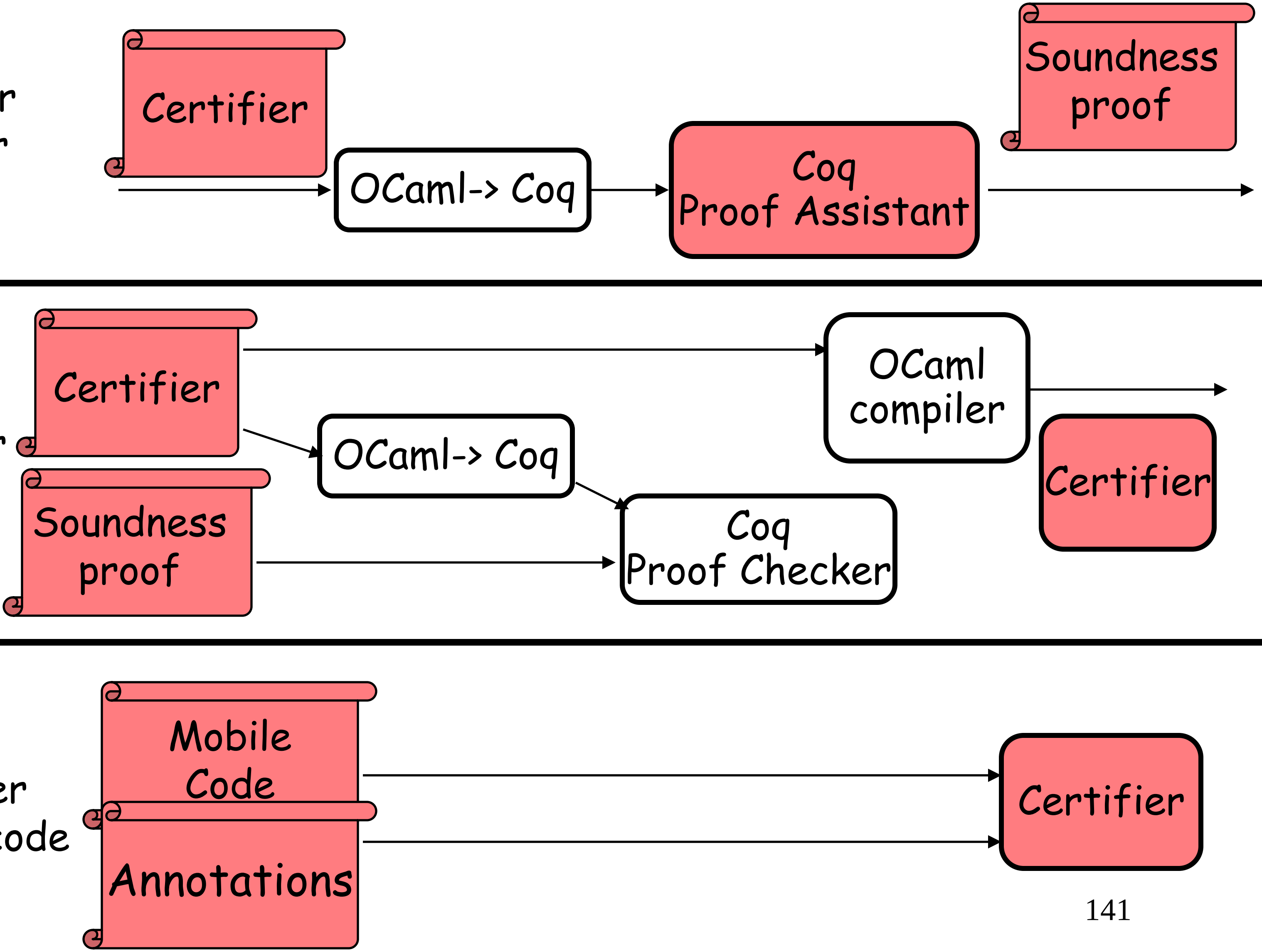
- Advantages:
  - No need to worry about building proofs, proof sizes, proof encoding, for individual programs
    - Speed up of 25x, code size reduction 2x
  - Subsumes old PCC: Annotations may contain proofs

Untrusted Certifiers Architecture

1. code producer proves certifier

2. code consumer downloads the certifier and its proof

3. code consumer downloads the code



Custom Verification

- Today’s VM have hard-coded verifiers
  - Force a type system, compilation strategy, even source language
  - Thus, fix the safety mechanism not just the policy
- Users should be able to
  - Pick source language, compilation strategy, and safety enforcement tool
  - Upload a certifier
  - Essentially, customize the verification
- Doable with the strategy outlined here

PCC Conclusions

- Software must be executable and checkable
  - Powerful safety checkers are kept simple by allowing them to consult proofs/oracles
- PCC is automatic and practical for type safety
  - More of less inference can be done at the receiver
- More research is needed before we can automate PCC beyond type safety
  - Type systems, specification logics and decision procedures

PCC Conclusions (II)

- Bridge the gap between PCC and source-level analysis tools

- Infrastructure must facilitate the interfacing to standard safety tools
  - Write custom untrusted certifiers

- Customizable verifiers
  - Maximum of flexibility for code producer, without loss of safety