

Verify Everything, Every Time

Michael Franz
University of California, Irvine

Armonk, New York, September 2004

Today's OS's: Highly Vulnerable

- the “Slammer” worm (January 2003)
 - consisted of just 376 bytes of code
 - infected nearly 90% of all vulnerable hosts worldwide
 - in under ten minutes
- astonishing facts:
 - the number of infected hosts doubled almost every eight seconds
 - attack exploited a buffer overflow vulnerability in Microsoft's SQL Server database product
 - this particular vulnerability had been made public six months prior to the attack, and a patch had been released even earlier
 - a number of Microsoft's own servers were affected by the attack
- “Slammer's” spread was stopped not by human intervention but by the network outages it had caused itself

Today's OSs: Rotten Foundations

- too large and too fluid to be audited properly
 - “old” software is usually more reliable than “new” software
 - but almost everything in a modern operating system is “new”
- device drivers, dynamic loading etc. thwart even static analysis techniques
- current approach to overcoming known vulnerabilities is “patching”, yet evidence shows that patching compliance is spotty
- unknown how many OS vulnerabilities are bona-fide programming errors and how many are actually “back doors” placed by agents of foreign nation states

Who Cares?

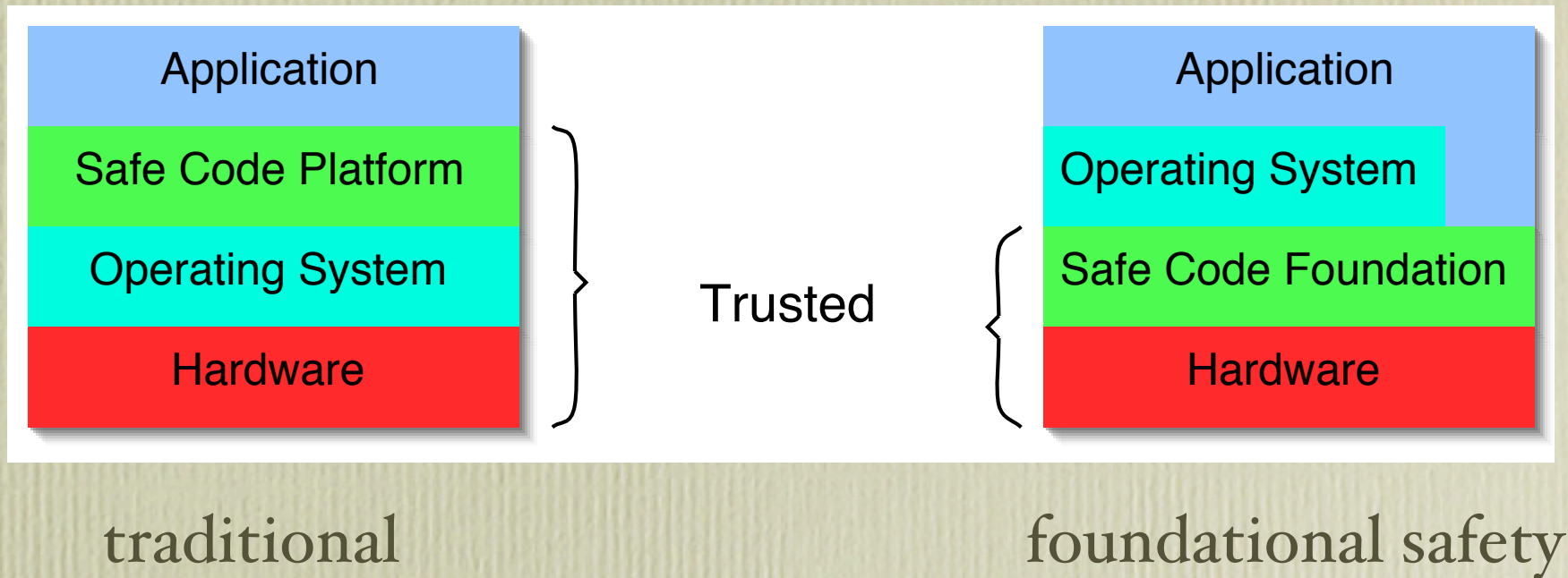
- from Microsoft's (Cisco's, Oracle's, ...) perspective, "an error is an error is an error"
- they don't care whether a foreign agent put it in or whether it is a genuine error (it is a liability/reputation issue)
- but some of Microsoft's (Cisco's, Oracle's, ...) products have become **essential facilities**, so we all should care!
- in the event of an international conflict, an adversary might be able to remotely tamper with our infrastructure in unexpected ways

Enter: This Community!

“the results of mobile code research are much broader than expected”

“apply these ideas to [operating] systems **safety in general**, not just for *mobile code*”

Solution: Don't Trust The OS!



- invert the traditional relationship between compiler and operating system
- use recent results from mobile-code domain to obtain safety everywhere

Foundational Safety

Application Code

———— Virtual Machine Abstraction Layer ————

Virtual Machine / Dynamic Translator

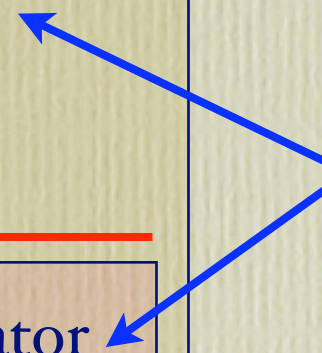
Operating System

———— Typed Hardware Abstraction Layer ————

Core Safe Code Verifier / Dynamic Translator

Hardware

Compilers!



Sorry, Java Won't Cut It...

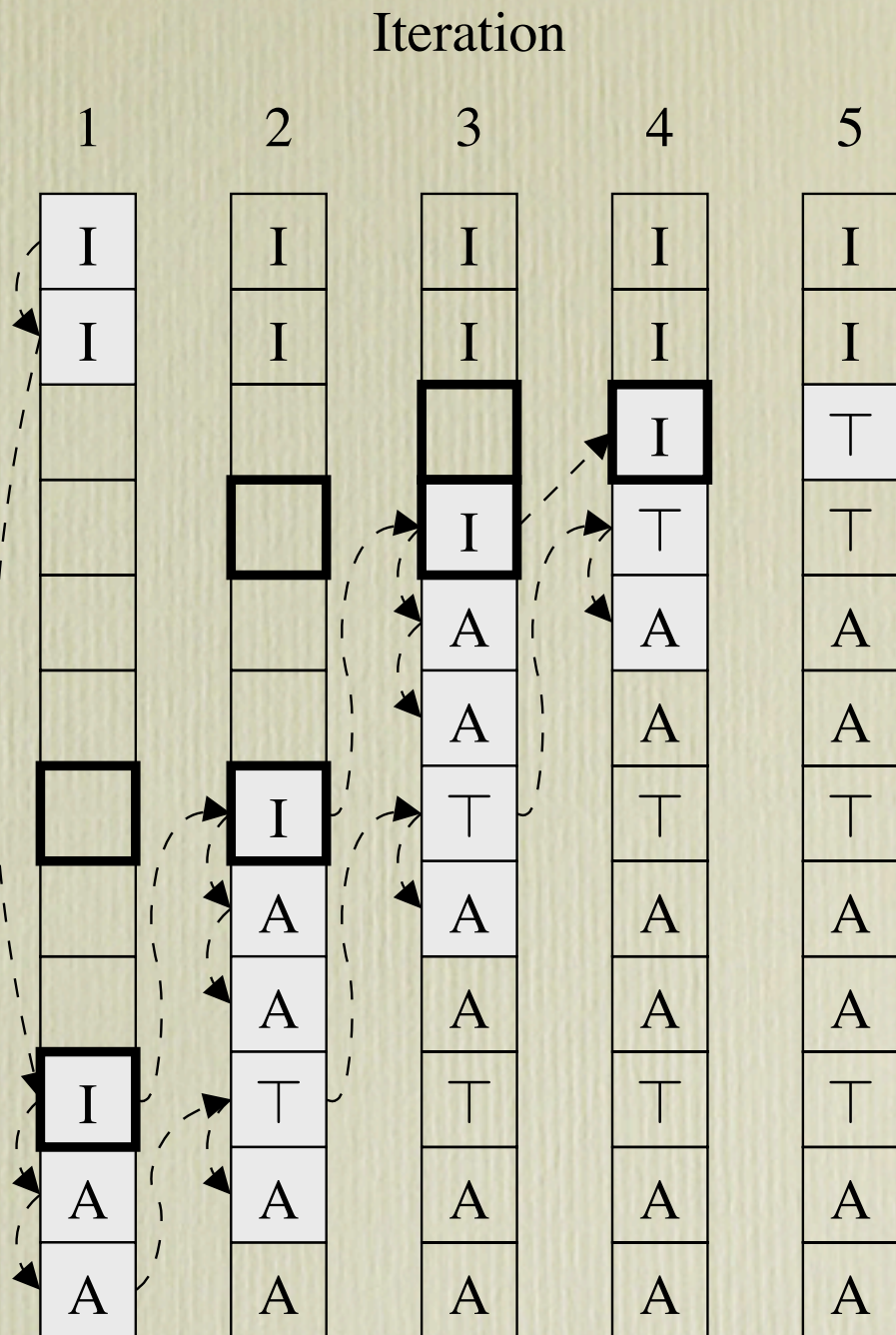
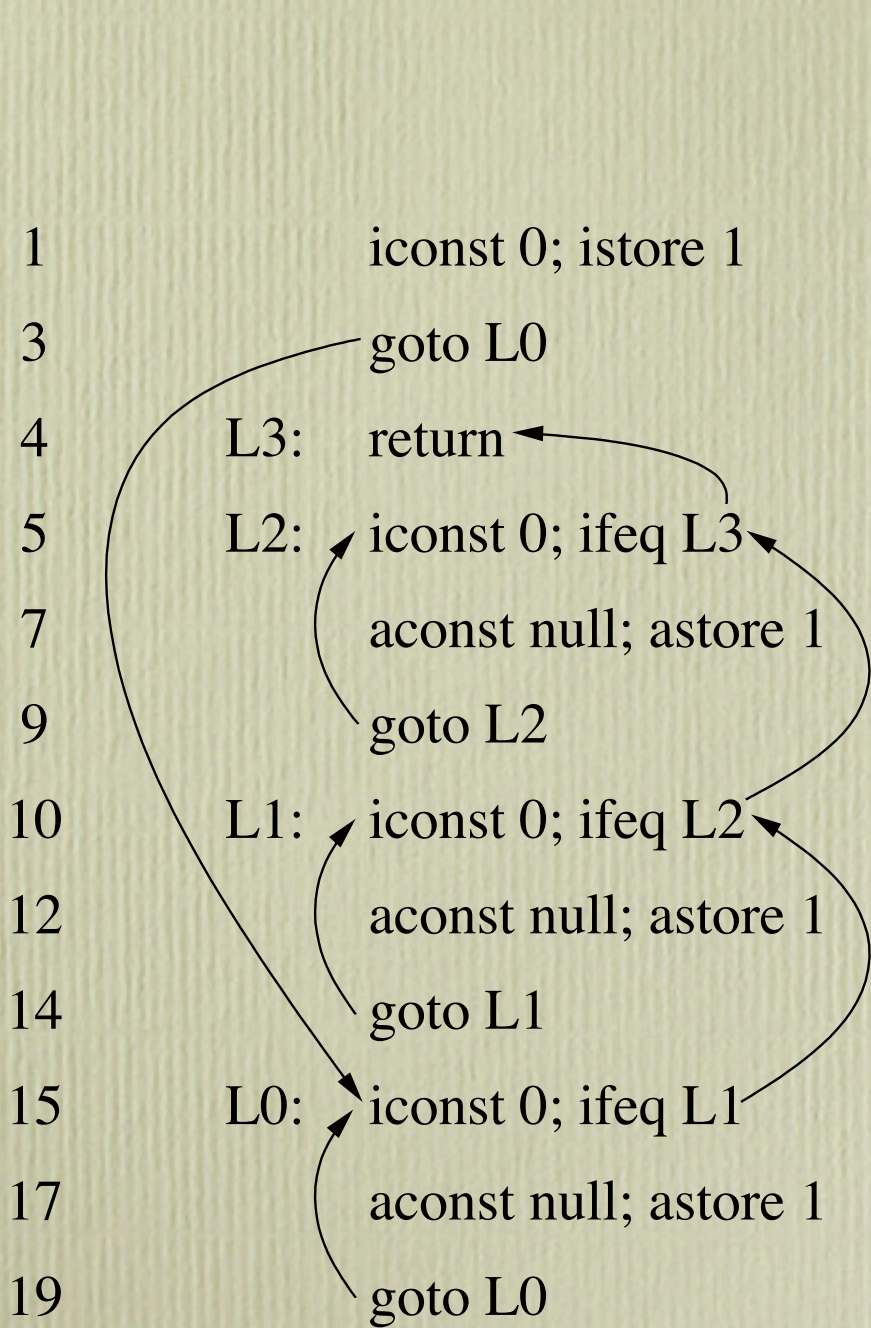
- Java just-in-time compilation has enjoyed spectacular success, but (unless Cliff Click pulls something insanely great out of the hat) won't support an OS on top...
- worse, need to verify everything in this model
- Java verification time essentially depends on the longest path of control flow edges along which type information has to flow, in the forward direction of the analysis
- could use the **verification cost itself** as a denial of service attack

Java Bytecode Verification

- the JVM uses an iterative data-flow algorithm to check each method for consistent typing
- the average case performance is good (1-5 iterations), but the worst case complexity is quadratic
 - some poor implementations are even $O(n^4)$

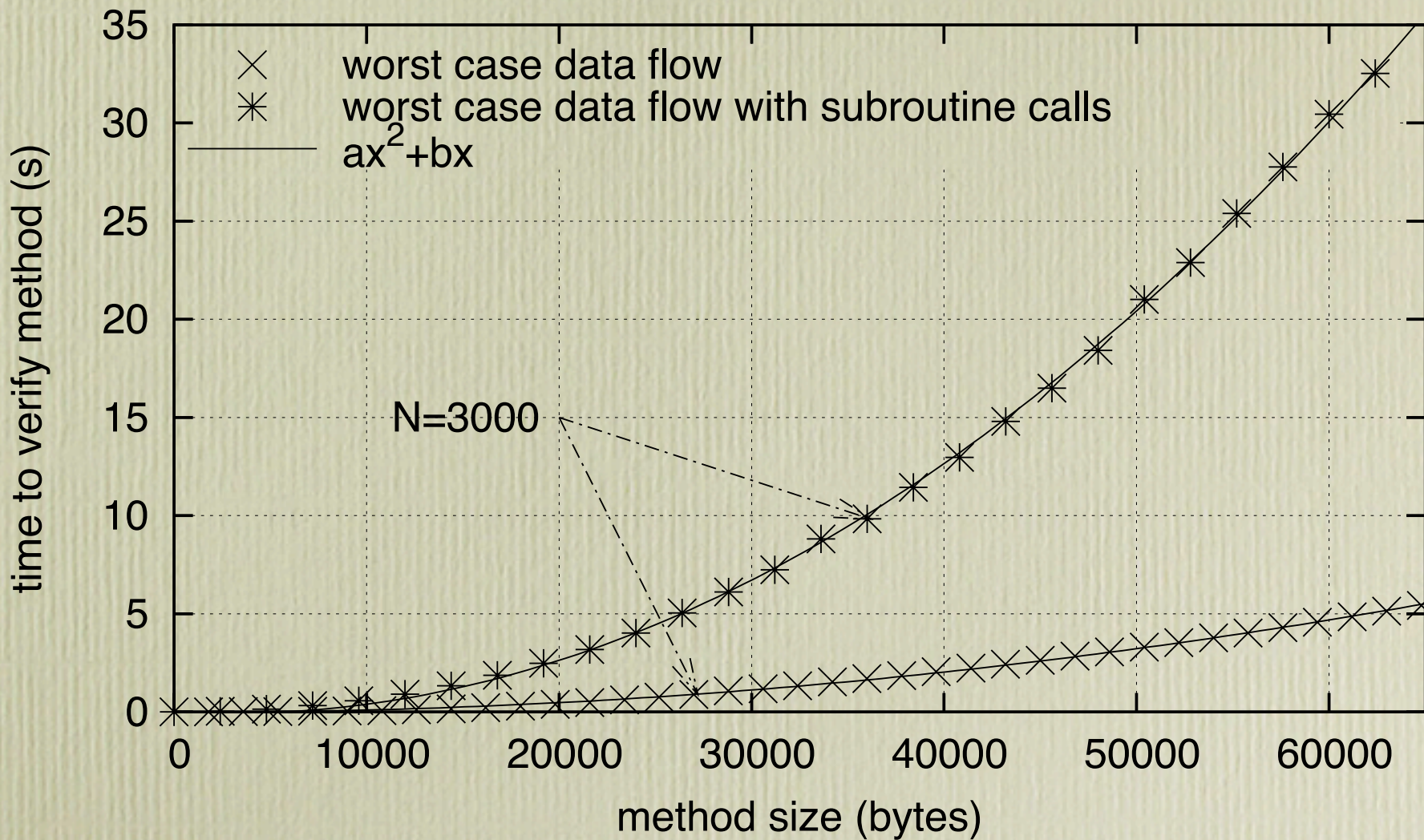
Pathologic Cases

- Java verification time essentially depends on the longest path of control flow edges along which type information has to flow, in the forward direction of the analysis
- all JVM verifiers we have seen simply process each method from beginning to end, and then repeat if anything has changed
 - every backward jump requires an additional iteration!
- can use this knowledge to construct a denial-of-service exploit
 - a JVM program that will verify eventually, but that will take an inordinate time to do so



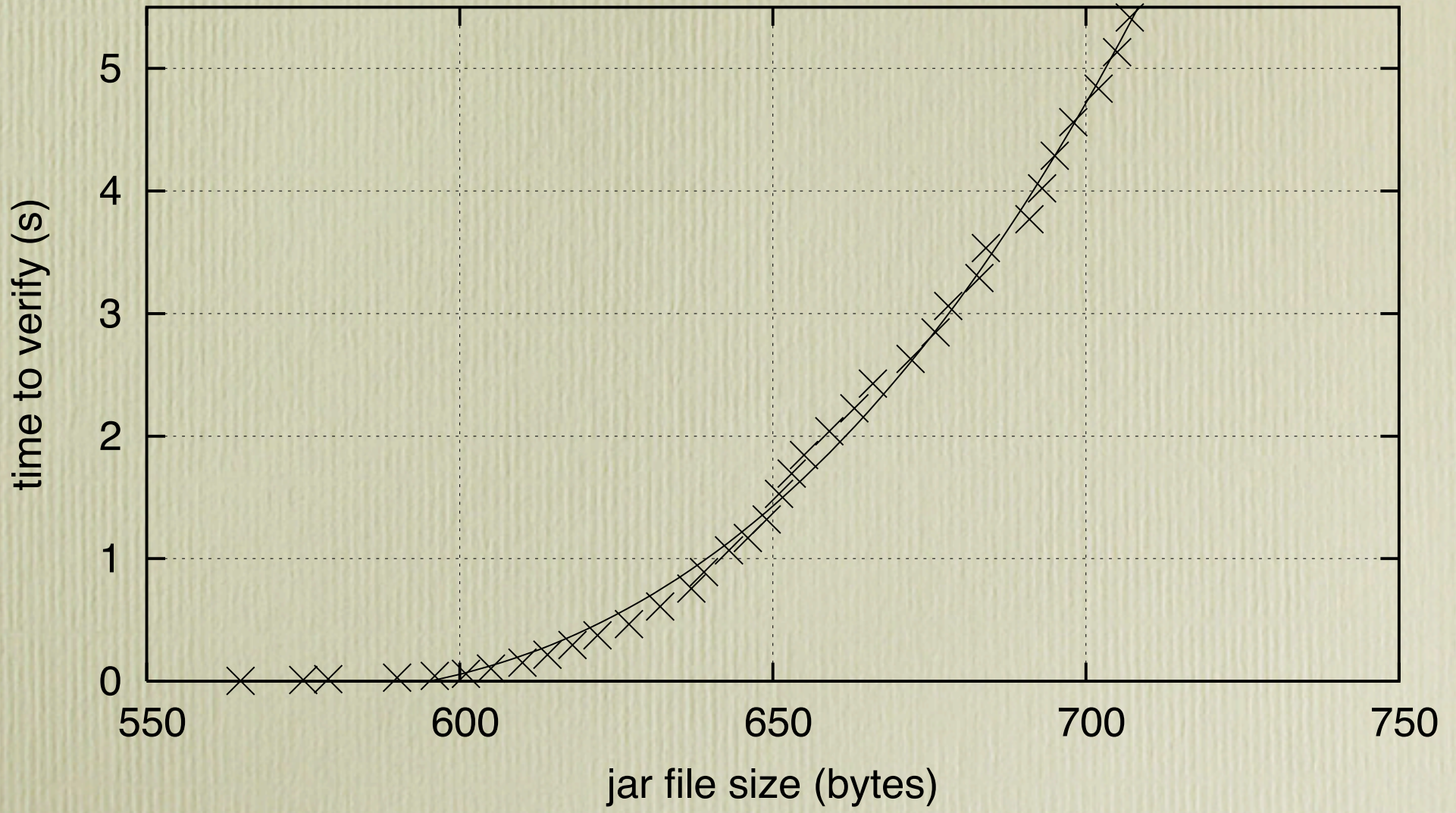
Verification Time

- verification time also depends on the specific implementation of the verifier
- Sun's verifier is very slow when dealing with subroutine invocations
- this is a constant overhead, but can be exploited to make the DoS attack worse
- on the next slide, we show the time to verify a single method with worst-case dataflow on a 2.53GHz P4 under Sun HotSpot VM 1.4.1



Obfuscating the Exploit

- to make matters worse, pathologic code as shown here can be compressed very well
- a 700-byte JAR file (one classfile with a single pathologic method) takes 5s to verify on a high-end P4 workstation (2.5 GHz)
- a 3500-byte JAR file (one classfile with many identical pathologic methods) takes 15 minutes to verify



demo:

<http://nil.ics.uci.edu/exploit>

Note: .NET CLR Does Better...

- since the stack is empty upon jumps, and there are no subroutines, the problem is less severe in .NET
- however, other complexity-based denial-of-service attacks may exist
 - e.g., knowing the implementation of the register allocator, I might be able to send it a particularly nasty graph-coloring puzzle...
 - we have a little research project going on, trying to find some more

Attacking the Compilation Pipeline

- why stop at the verifier?
- whenever the worst-case performance of an algorithm is known, a particular hard-to-solve puzzle can be constructed
 - verification
 - register allocation (graph coloring, known to be *NP*-complete)
 - instruction scheduling (topological sorting, *NP*-complete)
 - etc...

A New Security Paradigm

- currently, just-in-time compiler pipelines are optimized for the average case
- in mobile code environments, we should **optimize for the worst case**
- this applies to every step along the code pipeline
- apparently, nobody does it, (yet)

Open-Source Development

- this is the one situation where open source is actually bad
- there may be some truth to the claim that “thousand eyes spot a programming error faster than two” — **but this class of attacks is based on the algorithmic complexity of the underlying algorithm and not on any implementation error**
- errors are easily fixed but algorithms are not easily replaced
- for this type of attack, security by obscurity may be the best defense!

Foundational Safety

Application Code

———— Virtual Machine Abstraction Layer ————

Virtual Machine / Dynamic Translator

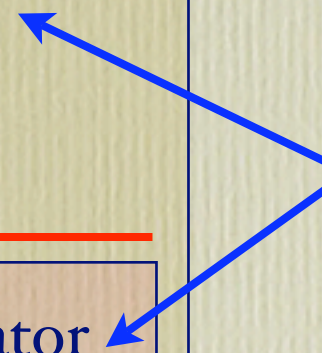
Operating System

———— Typed Hardware Abstraction Layer ————

Core Safe Code Verifier / Dynamic Translator

Hardware

Compilers!



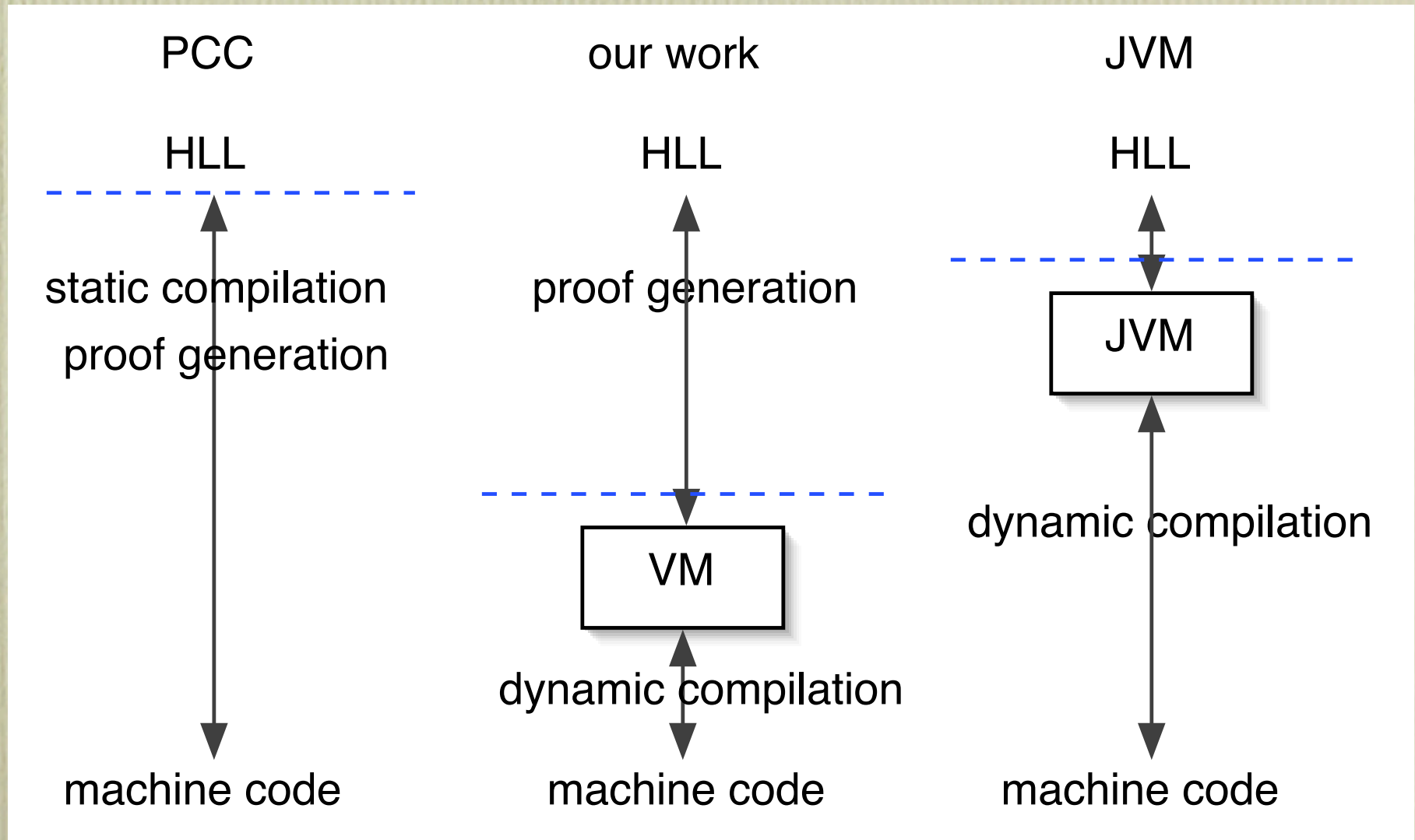
Foundational Safety

- need a small and efficient safe code foundation
 - small enough to be verified manually
 - efficient enough to support an OS running on top (!)
- current options all fall short
 - virtual machines / interpreters: too slow
 - virtual machines / just-in-time compilers: too big when good
 - proof carrying code: proofs are huge and code is non-portable
- solution: new hybrid approach using a software-defined architecture for PCC, optimized simultaneously for
 - minimizing the complexity of proofs
 - fast and good dynamic translation into the actual machine code
 - small size of the trusted code base

Research Question

- is there a middle ground such that
 - the architecture is concrete enough that most optimizations are possible
 - while semantics are high-level enough so that the complexity of proofs is reasonable

Hybrid Approach: PCC-VM



Motivating Example

- virtual machine needs to support dynamic typing with an unbounded number of user-level types.
- but, for efficient implementation we want it to have only a finite number of data types.
- proof-carrying code can be used to establish type safety (“cover the semantic distance between the high level and the low level”), but traditionally the corresponding proofs are very complex.
- need to “track” the type of a data item across the program to prove that it hasn’t changed.

Motivating Example

- the virtual machine supports a data type “tagged pointer” with a user-level tag field.
- reading and writing to the tag is done by two special instructions “tagread” and “tagwrite”.
- in particular, the tag field is not accessible using the normal “fieldread” and “fieldwrite” instructions.
- this simplifies proof greatly: any instruction sequence that doesn’t include “tagwrite” cannot have changed the high-level type of an object.

Splitting The Burden Of Proof

safety = type-safety + memory safety

scalar types

reference types

by construction
of the VM

using easily verifiable
proofs

by construction
of the VM

The Verification Problem

- first idea: self-certifying code (1996)
 - rather than trusting a digital certificate, examine the code itself before it is run
 - led us to Java and bytecode verification
- second idea: proof-carrying code (1997)
 - rather than performing a complete data-flow analysis, compute a “proof” at the code-producer side
 - having the proof shortcuts the verification
- can one do better yet?

The Verification Problem

- first idea: self-certifying code (1996)
 - rather than trusting a digital certificate, examine the code itself before it is run
 - led us to Java and bytecode verification
- second idea: proof-carrying code (1997)
 - rather than performing a complete data-flow analysis, compute a “proof” at the code-producer side
 - having the proof shortcuts the verification
- third idea: inherently safe code (2001)
 - how about inventing a “code language” in which “bad” programs cannot even be expressed in the first place
 - UCI research, patent pending

Inherently Safe Code

- at UCI, we have come up with a family of high-level code formats whose main idea is to prevent the transport of “illegal” programs in the first place, **by making it impossible to encode them in the transport format**
- this is similar to the idea described in Orwell’s classic *Nineteen Eighty-Four* in which the language Newspeak is derived from Oldspeak (English) by deleting all the words that could be used to express heretical thought
- the general question whether it is possible to “think the unthinkable” (existence of a Gödelization of thought) goes back to Wittgenstein’s *Tractatus Logico-Philosophicus*

Intrinsically Safe Code Formats

- a class of machine-independent code representations that can provably encode only “legal” programs
 - safety is obtained **by construction**
 - **the need for verification or checking disappears**
 - our approach can provide the identical safety guarantees as the Java Virtual Machine (including full Java binary compatibility semantics), *but it can express most of them statically as a well-formedness property of the encoding itself*
 - in our solution, an incoming mobile program may not do the intended task, but it will not do anything “bad” — **for any definition of “bad” that can be cast into a type system**
 - interestingly enough, such “intrinsically safe” code is also denser than virtual machine code, and permits to generate better object code, and faster

Inherently Safe Encodings

```
class Basic {...};  
class Extended extends Basic {...};
```

```
static void Sample1 {  
    Basic b1, b2; Extended x1 ,x2;  
  
    ...  
}
```

sample Java program

Inherently Safe Encodings

```
class Basic {...};  
class Extended extends Basic {...};
```

```
static void Sample1 {  
    Basic b1, b2; Extended x1 ,x2;
```

```
    ...  
}
```

possible
assignments
that can be written
for these Java
declarations

useful	illegal	pointless
b1 := b2	x1 := b1	b1 := b1
b1 := x1	x1 := b2	b2 := b2
b1 := x2	x2 := b1	x1 := x1
b2 := b1	x2 := b2	x2 := x2
b2 := x1		
b2 := x2		
x1 := x2		
x2 := x1		

Inherently Safe Encodings

```
class Basic {...};  
class Extended extends Basic {...};
```

```
static void Sample1 {  
    Basic b1, b2; Extended x1 ,x2;
```

```
    ...  
}
```

enumerate the legal
and useful
assignments and
transmit only these
indices

	useful	illegal	pointless
1	b1 := b2	x1 := b1	b1 := b1
2	b1 := x1	x1 := b2	b2 := b2
3	b1 := x2	x2 := b1	x1 := x1
4	b2 := b1	x2 := b2	x2 := x2
5	b2 := x1		
6	b2 := x2		
7	x1 := x2		
8	x2 := x1		

Inherently Safe Encodings

```
class Basic {...};  
class Extended extends Basic {...};
```

```
static void Sample1 {  
    Basic b1, b2; Extended x1 ,x2;
```

```
    ...  
}
```

beneficial side effect: compactness —
need to express only one out of 8 choices
rather than the original 16 choices

	useful
1	b1 := b2
2	b1 := x1
3	b1 := x2
4	b2 := b1
5	b2 := x1
6	b2 := x2
7	x1 := x2
8	x2 := x1

Inherently Safe Encodings

- the type-aware encoding is **inherently immune against malicious modifications** that would undermine type safety since programs that violate the assignment compatibility rule cannot be represented in the first place
- hence, unlike programs expressed in JVM language, which need to be verified upon arrival at the target machine, assignment compatibility in the example encoding **never needs to be verified** at all
- one of the goals of our current research is to extend this idea to type properties beyond “assignment compatibility”

Performance-Enhancing Information

- e.g. “Escape Analysis”
 - an object that doesn’t “escape” its defining scope can be allocated on the stack rather than on the heap
 - this optimization alone can often raise performance significantly
- the analysis itself is very difficult to do, but the results of the analysis are easy to verify
 - augment the type system by “captured/other”
 - **make this part of the encoding scheme itself**
 - e.g., => a captured reference cannot be assigned to a variable that might escape

Safe Annotations

```
static void Sample2 {  
    captured Object cap1, cap2;  
    Object o1, o2;  
    ...  
}
```

sample program in an
extended Java language
that enforces “capturedness”
in the type system

Safe Annotations

```
static void Sample2 {  
    captured Object cap1, cap2;  
    Object o1, o2;  
  
    ...  
}
```

a captured reference
cannot be assigned
to a variable that
isn't also captured

allowed

cap1 := cap2

cap1 := o1

cap1 := o2

cap2 := cap1

cap2 := o1

cap2 := o2

o1 := o2

o2 := o1

disallowed

o1 := cap1

o1 := cap2

o2 := cap1

o2 := cap2

pointless

cap1 := cap1

cap2 := cap2

o1 := o1

o2 := o2

Safe Annotations

```
static void Sample2 {  
    captured Object cap1, cap2;  
    Object o1, o2;  
  
    ...  
}
```

again, enumerate
the legal and useful
assignments and
transmit only these
indices

allowed
cap1 := cap2
cap1 := o1
cap1 := o2
cap2 := cap1
cap2 := o1
cap2 := o2
o1 := o2
o2 := o1

~~disallowed pointless
o1 := cap1 cap1 := cap1
o1 := cap2 cap2 := cap2
o2 := cap1 o1 := o1
o2 := cap2 o2 := o2~~

Annotation Precision

- we don't annotate allocation sites (values), but *variables* that hold only captured values
- hence, we “lose” some captured objects that temporarily are pointed to by variables that cannot be marked as captured
 - “splitting” such variables might be a solution
- still good enough to be worthwhile:
 - on average, 66% of captured allocation sites are covered by our algorithm, relative to Whaley and Rinard's algorithm
 - unlike other algorithm, ours can support dynamic loading trivially (without repeating the analysis), in which case we still cover 56% of Whaley and Rinard's captured allocation sites

Inherently Safe Encodings

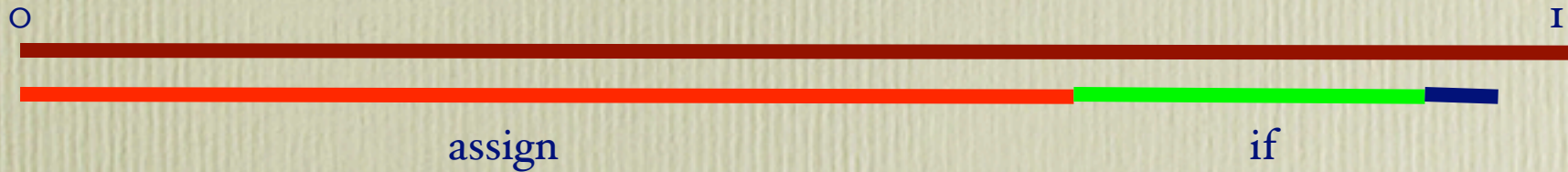
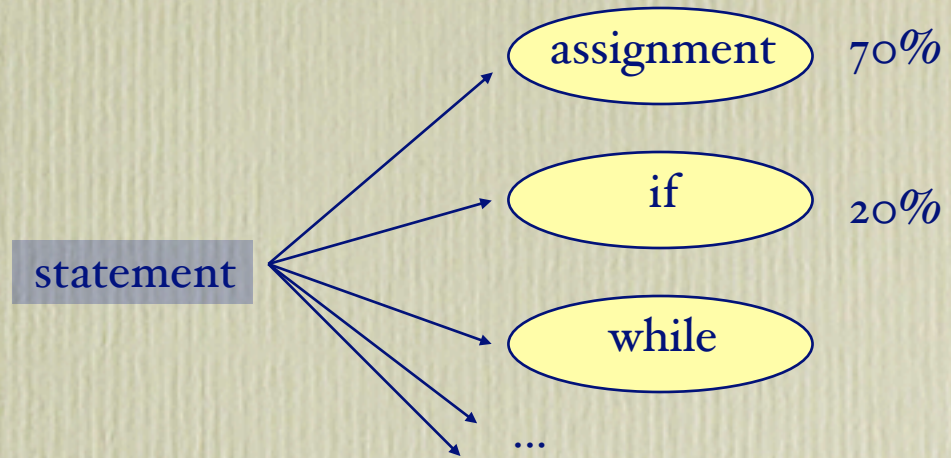
- a mapping between “legal programs” and “bit sequences of arbitrary length exceeding a certain threshold” that is invertible, surjective, but not necessarily injective
- several different bit patterns might map back to a single original program, but every bit pattern of sufficient length is guaranteed to correspond to some program that is legal in the original domain
- bit patterns that aren't long enough to represent a legal program can be rejected trivially

Probabilistic Arithmetic Coding

- encode the program based on an abstract grammar augmented by a continuously adapted probabilistic model
- illegal or useless constructs are given zero probability in this model
- map the program to a sub-interval within $[0, 1)$
- during encoding/decoding, this interval is successively refined
- every real number that falls inside the refined sub-interval is considered to represent the program

Example of Interval Coding

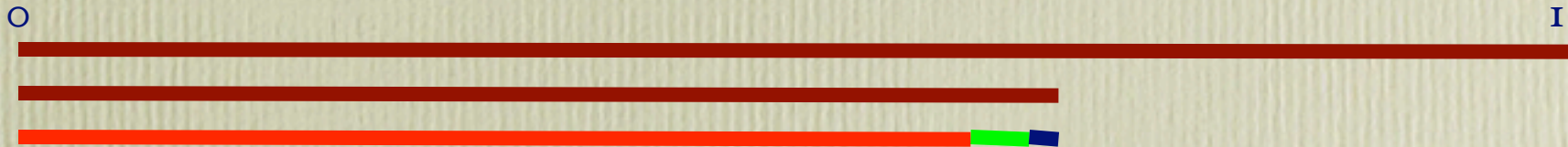
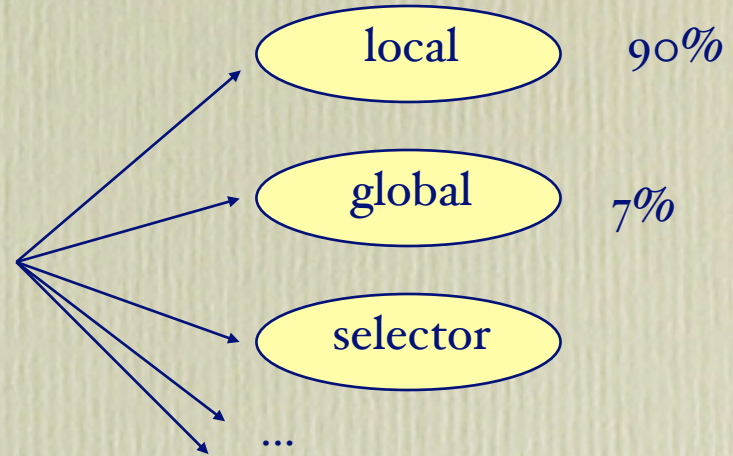
```
int i, j, k; float x;  
{ j = i; ...
```



Example of Interval Coding

```
int i, j, k; float x;  
{ j = i; ...
```

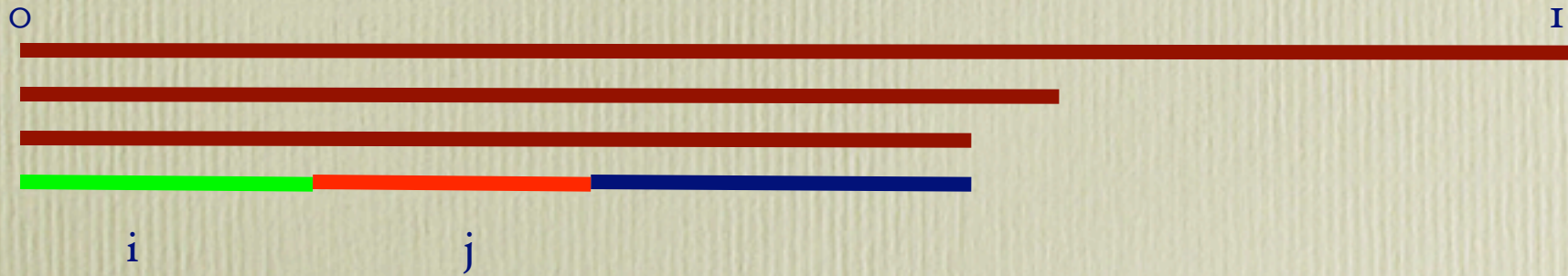
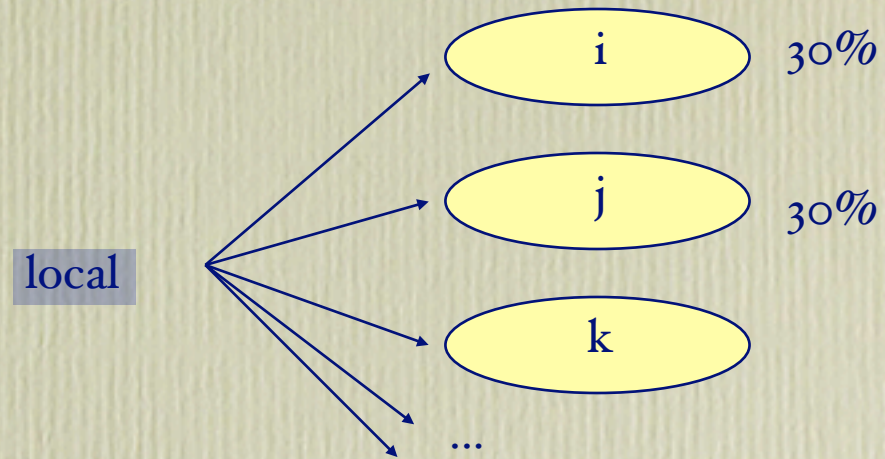
assignment target



local variable

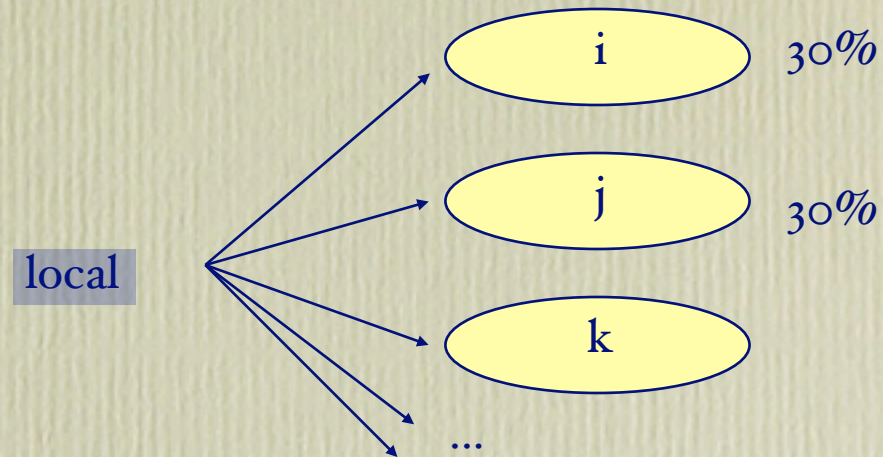
Example of Interval Coding

```
int i, j, k; float x;  
{ j = i; ...
```



Example of Interval Coding

```
int i, j, k; float x;  
{ j = i; ...
```



Example of Interval Coding

```
int i, j, k; float x;  
{ j = i; ...
```

final interval enclosed in $[\text{.19}, \text{.38}) < 0.5$
the first bit of the real-number representation
of this interval must be zero
=> output a zero and re-normalize



Example of Interval Coding

```
int i, j, k; float x;  
{ j = i; ...
```

.0

o



I

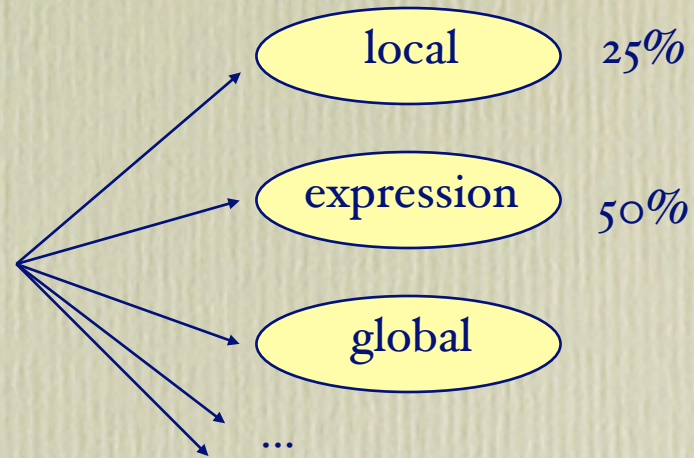
j:=

[.38, .76)

Example of Interval Coding

```
int i, j, k; float x;  
{ j = i; ...
```

assignment source



o



I

[.38, .48)

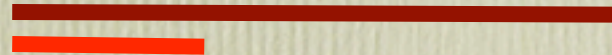
Example of Interval Coding

```
int i, j, k; float x;  
{ j = i; ...
```

.OOI

output two more digits and scale by 4

o



I

[.38, .48)

Inherently Safe Encodings

- elements that are illegal at any given point have zero probability and hence cannot be represented
- extremely compact representation in which every possible bit-stream of sufficient length represents a “legal” program in the original encoding domain
 - your favorite GIF of the Mona Lisa corresponds to a valid program — we don’t know what it does, but it doesn’t do anything “bad”
 - **we can make the above sentence correct for any definition of “bad” that can be cast into a type system**
 - our research endeavors to find broader definitions of “bad” and formalize them

Insights So Far

- abstract syntax trees viable as a mobile code format
- lead the way to a genuine improvement over virtual machine transportation formats
 - safety without need for validation
 - tamper-proof performance-improving information
- can be highly compressed
 - Java archives by factor of 3-8
 - better than best published Java bytecode-specific compression, but completely generic

Closing Comments & Viewpoint

- existing “trusted computing” projects are mostly not much more than yet another stopgap measure
 - code signing, digital rights management are the main focus
 - but tamper-proof hardware is an important first step
- real breakthrough will only come with true verification of the code itself
 - but I am confident that this will happen sooner than many expect

Summary & Outlook

- a hot topic: many “trusted systems” projects
 - Trusted BSD
 - Trusted Linux (Debian)
 - Security-Enhanced Linux (the-agency-that-is-not-to-be-named)
- confluence of factors
 - a real threat
 - recent research results from mobile-code community
 - hardware technology to sustain dynamic translation
 - politics: nobody but Microsoft has anything to lose
- hence, “trusted foundations” will be the next big thing
 - look forward to the biggest platform war ever, defining the next 25 years of computing

Thank You