A Type-Preserving Compiler Infrastructure

Christopher League
Yale University
17 May 2002

Advisor
Zhong Shao
Committee
Kim Bruce
Arvind Krishnamurthy
Carsten Schürmann
Acknowledgment
Valery Trifonov

Mobile code, pervasive networks
- Wireless handheld computers
- Remotely programmable devices
- Browser applets
- Widely distributed computation

Security is critical
- We might not completely trust the programs we receive and run
- Must ensure they do not misbehave:
  - crash the device
  - exhaust resources
  - interfere with other programs/data
- Correctness is hard—focus on safety

Security toolbox: digital signature
- Confirms identity of producer, not safety of code
- I might not trust Microsoft, but would still run the code—assuming it is harmless

Security toolbox: reference monitor
- Code runs in a sandbox
  - interactions with outside world mediated by the monitor
- Hardware mechanisms
  - expensive context switches
  - not available on all devices
- Software rewriting
  - frequent dynamic checks
Security toolbox: language features

- Array bounds checking
- Garbage collection
- Exceptions
- Encapsulation, access control
- Type systems

Many vulnerabilities are type errors

<table>
<thead>
<tr>
<th>Date</th>
<th>System</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/24/02</td>
<td>AOL ICQ</td>
<td>remotely exploitable buffer overflow</td>
</tr>
<tr>
<td>1/14/02</td>
<td>Solaris CDE</td>
<td>buffer overflow vulnerability</td>
</tr>
<tr>
<td>12/20/01</td>
<td>MS n-PNP</td>
<td>buffer overflow vulnerability</td>
</tr>
<tr>
<td>12/22/01</td>
<td>SysV 'login'</td>
<td>remotely exploitable buffer overflow</td>
</tr>
<tr>
<td>11/29/01</td>
<td>WU ftpd</td>
<td>format string vulnerability; free() on unallocated pointer</td>
</tr>
<tr>
<td>11/21/01</td>
<td>HP-UX lpd</td>
<td>remotely exploitable buffer overflow</td>
</tr>
<tr>
<td>12/12/01</td>
<td>Oracle9i AS</td>
<td>remotely exploitable buffer overflow</td>
</tr>
<tr>
<td>10/25/01</td>
<td>CDE ToolTalk</td>
<td>buffer overflow vulnerability</td>
</tr>
</tbody>
</table>

[CERT advisories]

The need for typed machine code

- Is it enough to program in type-safe languages? No.
- 1. Microsoft is unlikely to ship source code
- 2. Still must trust compiler

Trusted Computing Base
- system modules responsible for supporting security policy

Vision: high assurance systems with minimal TCB

- 1,000 LOC vs. 30,000+
- (We are not there yet!)
- "Foundational proof-carrying code" [Appel et al. 2001]

Key: type-preserving compiler

- 1,000 LOC vs. 30,000+
- (We are not there yet!)
- "Foundational proof-carrying code" [Appel et al. 2001]
Until now: functional languages or “safe C”
- TIL/ML [Tarditi, Morrisett et al. 1996]
- FLINT/ML [Shao 1997]
- Touchstone [Necula & Lee 1998]
- Popcorn
- Scheme— [Morrisett et al. 1999]

Real world: object-oriented
- Widely used (Java, C++)
- New, complex features
  - dynamic binding
  - dynamic casts
  - inheritance
  - interfaces
  - name-based types

Contributions
- Single typed IL that supports Java and ML
  - Not partial to either language
- Novel techniques for encoding most features of Java
  - Same efficiency as untyped
  - Preserves safety guarantees
- New high-level IL for Java
  - JVM primitives with functional flow
  - Easier to verify and optimize

Outline
- Mobile code security
- The need for typed machine code
- Dynamic dispatch security hole
- How our approach is different
- Safe and efficient object encoding
- Functional Java byte code
- A prototype compiler for Java and ML

Dynamic binding: essential to OO
- Inheritance without polymorphism is possible, but certainly not very useful.
- One can declare derived types, but the actual operation being called is always known at compile time.
  [Booch 1994]

Efficient dynamic dispatch
```java
public static void example ( Object x, Object y )
{ x.toString();
  // compiles to:
  (null check)
  let r1 = x.vtbl ;    // method suite
  let r2 = r1.toString ; // method pointer
  r2( x )   // "self application"
}
```
**Must ensure the call is safe**

```java
def example(Object x, Object y) {
    x.toString();
    // compiles to:
    let r1 = x.vtbl;
    let r2 = r1.toString;
    r2(x);
    // this is sound
    r2(y); // not!
}
```

Any unsoundness can be exploited.

**Cast arbitrary integer to pointer**

```java
def example(Object x, Object y) {
    let r1 = x.vtbl;
    let r2 = r1.toString;
    r2(y); // wrong!
}
```

**This is a major security hole in Cedilla Systems' PCC** [Colby et al. 2000]

Special J
- Java compiler; generates annotated x86
- Types retain Java abstractions
  - (object_ty C)
  - (method_ty C SIG)
- Faulty axiom; undetected for 3+ years

**Our approach is different**

- Start with a strong foundation
  - “off the shelf” type theory
  - Not specific to Java (or ML)
  - Simple soundness proof
- Design complex encodings of Java features that
  - Preserve type safety
  - Maintain efficiency

**Others have encoded objects in type theory**

- These are models of OOP
- Rather inefficient
  - Extra indirections
  - Extra function calls
- Assumed subsumption
  - A Circle is also a Shape

**Type theory toolbox: rows** [Rémy 1993]

- Intuition: the suffix or tail of a record type, starting at a given offset
Type theory toolbox: recursion

- Intuition: $\mu$ notation for recursive definitions
  - list = $\{\text{data: int, next: list}\}$
- Replace recursive ref. with a type variable:
  - list = $\mu a.\{\text{data: int, next: } a\}$

```
let x = fold y as list
let y : { data: int, next: list } = unfold x in y.f (y.z)
```

Type theory toolbox: existentials

- Intuition: hide a type from outsiders
  - enforces abstract data types
```
let x1 : $\mu a.\{z : a, f : a \rightarrow string\}$ = · b = int, {z=42, f=int2string} : { z : b \rightarrow string } in
```

The type of Object

```
ObjTy[Object] = 
\{fs::R8, ms::R4, 
\\mu self. ObjRcd[Object].fs ms self\}
ObjRcd[Object].fs ms self = 
\{vtbl : {toString : self \rightarrow string ; 
ms} ; hash : unsigned ; 
fs \}
```

```
Provably safe method invocation...

example (x, y : ObjTy[Object]) = 
open x as [fx, mx, x1 : self]. ObjRcd[Object].fx mx self x1;
let xz : ObjRcd[Object].fx mx rx = unfold xz;
let rz : (toString : rx \rightarrow string ; mx) = xz.vtbl;
let r2 : rx \rightarrow string = r1.toString;
```

`Techniques extend to most of Java`

- classes
- inheritance
- dynamic dispatch
- dynamic cast
- mutual recursion
- interfaces
- constructors
- super calls
- subroutines
- exceptions
- privacy

`Chapters 3–5 contain:
- Formal definition of source language (Featherweight Java)
- Formal definition of intermediate language (JFlint)
- Proofs that JFlint is sound and decidable
- Type-directed translation
- Proof that well-typed inputs yield well-typed outputs`
Act III
- Mobile code security
- The need for typed machine code
- Dynamic dispatch security hole
- How our approach is different
- Safe and efficient object encoding
- Functional Java byte code
- A prototype compiler for Java and ML

System building
- A prototype compiler for Java and ML
- Many practical problems must be solved
  - Efficient implementation of IL
  - Large semantic gap between Java and JFlint

Where to start?
- Java
  - x.println(y);
  - Java byte code
  - Many details are not explicit in Java source
- JVML byte code
  - 3 aload_0 # this
  - 4 getfield PrintStream C::x
  - 7 dload 2
  - 9 invokevirtual
    - void PrintStream::println(double)

Two sets of concerns
1. data & control flow, type inference
2. expanding Java primitives
- JVML byte code
  - 3 aload_0 # this
  - 4 getfield PrintStream C::x
  - 7 dload 2
  - 9 invokevirtual
    - void PrintStream::println(double)

A new IL to bridge the gap
- High-level Java primitives, types
- JVML
  - JVML
  - Functional control and data flow

A better Java byte code
- JVML
  - JVML
  - JFlint
  - Fully explicit
    - Supports all of JVML, yet is
    - Easier to verify and optimize
  - Nastiest parts of JVM become tractable
    - Object initialization
      - Freund & Mitchell 1999
    - Subroutines
      - Stata & Abadi 1998
  - Verification is just simple type checking
    - < 260 lines of ML code
      - Chapter 7
**Example: Factorial**

```java
public int fact(int n) {  
    int x = 1;  
    for (x = 1; n > 0; n--)  
        x *= n;  
    return x;  
}
```

```java
public int fact(int n) {  
    int x = 1;  
    goto TL;  
    iload_1 iload_0 imul  
    # x=x  
    n iinc 0  
    # n--  
    TL iload_0 ifgt LT  
    # n > 0  
    iload_1 ireturn  
}
```

**Subroutines are tricky**

- `jsr offset` — push return address, jump
- `ret var` — return to address in `var`
- Used to implement ‘finally’ blocks
  ```java
  try { }  
  catch (Error e) { B; throw e }  
  finally { C }  
  ```
- Can achieve complicated control flow

**Continuation-passing style**

- [Steele 1978] [Kranz et al. 1986]
- The higher-order answer to flexible control flow
  - Represent return address as a function

**1. ret need not obey stack discipline**

```java
void main(String[] args) {  
    try { }  
    finally { while(true) { try { } finally { break; } } }  
}
```

```java
M jsr A  
goto R  
A astore_1  
# return addr  
L jsr B goto L  
B pop  
ret  
R return
```

```java
letrec M = l( ). A(R)  
and A = l k1 : ( ) V . B(L)  
and B = l k2 : ( ) V . k1( )  
in L( )  
and R = l( ). return  
in M( )
```
2. Subroutine might update local var

letrec \( M \) = \( \lambda \). \( S(43, P) \)
and \( S = \lambda i, k : (\text{int} \rightarrow \text{int}) \. \text{let } j = i - 1 \; \text{in } k(j) \)
and \( P = \lambda i : \text{int} \. \text{invoke } \text{println}(i) \; \text{return } \)
in \( M() \)

M \( \text{ldc } 43 \text{ istore}_1 \)  \# \( i=43 \)
\( \text{goto } P \)
\( S \text{ astore}_2 \)  \# return addr
\( \text{ret } 2 \)

P \( \text{iload}_1 \)  \# print \( i \)
\( \text{invoke } \text{println} \)
\( \text{return } \)

3. Polymorphic over untouched vars

letrec \( S = \lambda k : (\text{int} \rightarrow \text{int}) \. k() \) in
letrec \( M = \lambda \). \( \text{let } x1 = 3.14 \; \text{in } \text{println}(x1) \; \text{return } \)
letrec \( I = \lambda \). \( \text{invoke } \text{println}(x1) \; \text{return } \)
in \( S(I) \)
in \( S() \) in \( M() \)

System overview

- Based on SML/NJ compiler 110.30
  - added a new type-preserving Java front end
  - interactively loads and runs Java classes
- Same back end and runtime system
- Front end \( 8k \) LOC; JFlint checker \( 1k \) LOC
- Runs CaffeineMark 3.0 (12 classes, \( 100k \))
- Compile time just 60% longer than gcj
- Does not load native code!

Synergy with ML front end

- The type system is impartial

<table>
<thead>
<tr>
<th>JFlint</th>
<th>Java</th>
<th>ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>inheritance</td>
<td>parametric poly.</td>
<td></td>
</tr>
<tr>
<td>object enc.</td>
<td>closures</td>
<td></td>
</tr>
<tr>
<td>( \mu )</td>
<td>rec. classes</td>
<td></td>
</tr>
<tr>
<td>tags</td>
<td>rec. datatypes</td>
<td></td>
</tr>
<tr>
<td>rows</td>
<td>dynamic cast</td>
<td></td>
</tr>
<tr>
<td>object enc.</td>
<td>exceptions</td>
<td></td>
</tr>
<tr>
<td>records</td>
<td>stable, objects</td>
<td></td>
</tr>
<tr>
<td>stable, objects</td>
<td>records, tuples</td>
<td></td>
</tr>
<tr>
<td>methods</td>
<td>functions</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary

- Mobile code security is critical
- High-assurance systems need minimal TCB
- Type-preserving compilers are the key
- With care, they scale to real languages
  - use type theory as the foundation;
  - focus on practical encodings
- A single typed IL can safely and efficiently support different kinds of source languages