one aspect of

Security on JIT VMs

and more

~~Richie++~~ / richie / gera@corest.com

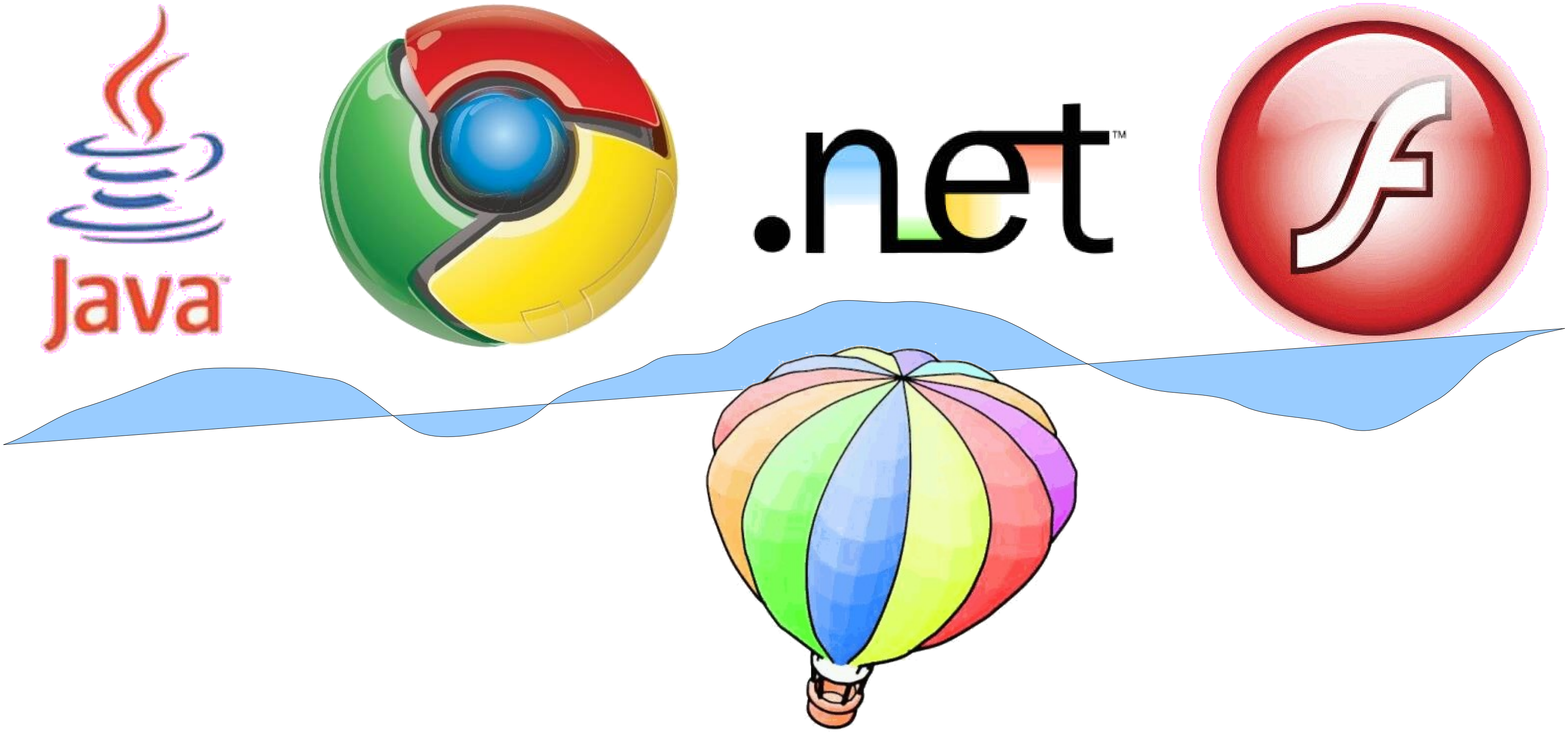
Security?

- Does you application need security?
- Do you do anything for its security?

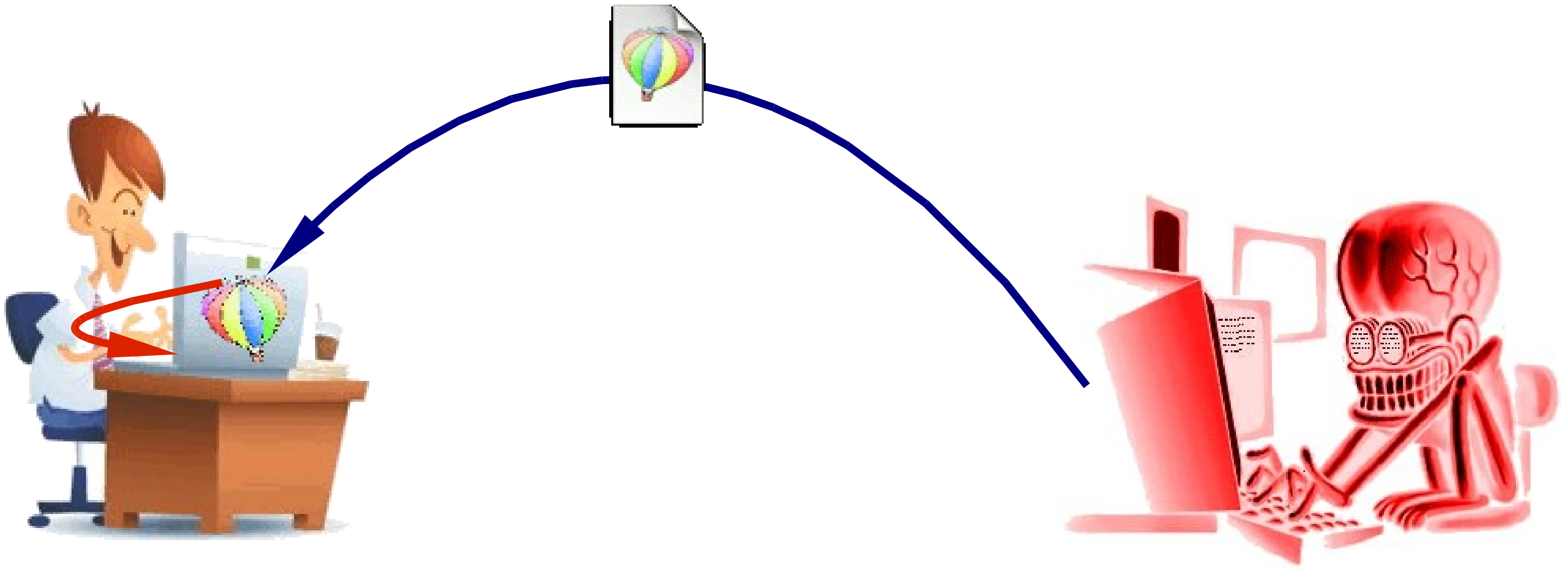
one aspect of

Security on JIT VMs

and more

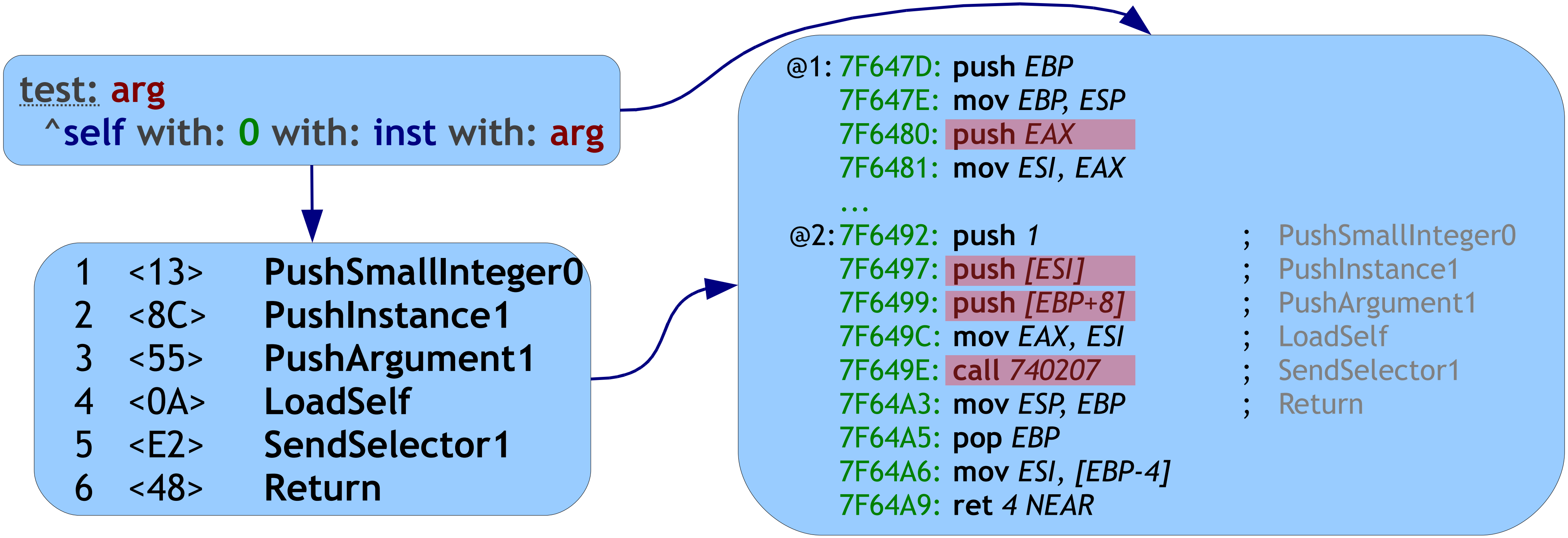


Scenario



- VM installed on user's machine (your application)
- Attacker provides content (behavior)
- Attacker escapes VM restrictions (if any)
- Attacker accesses private information

nativizing VMs
Digitalk VS



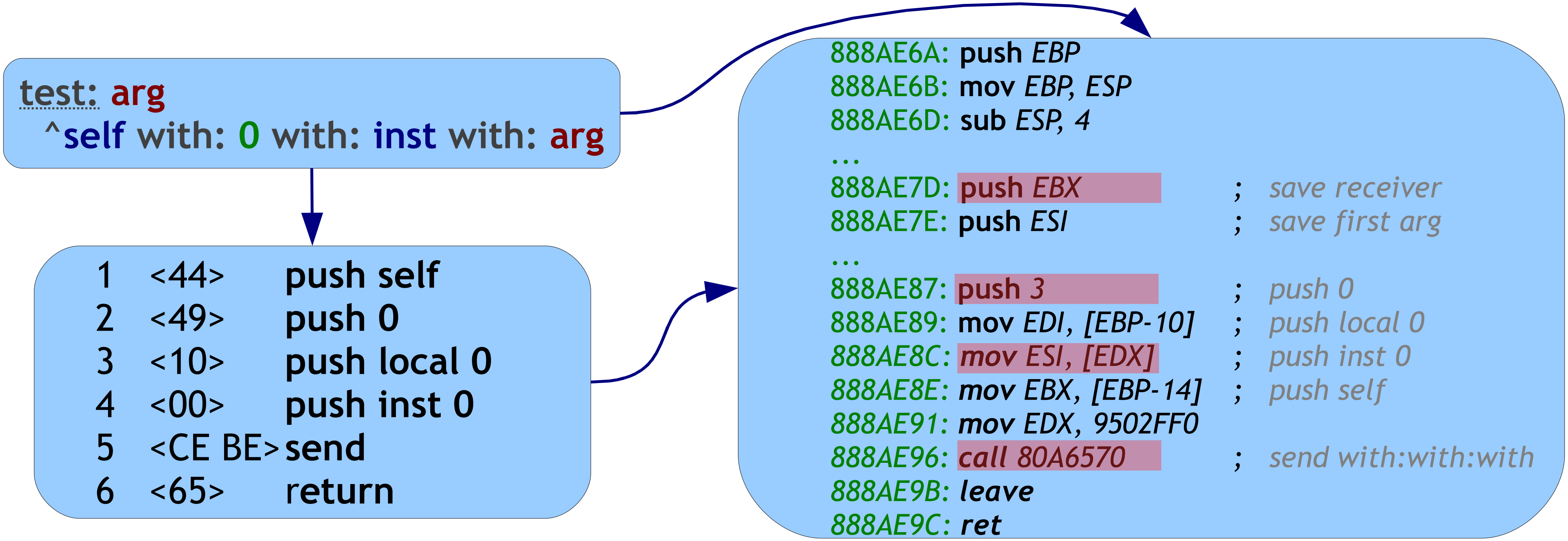
```
test: arg
  ^self with: 0 with: inst with: arg
```

```
1   <13>     PushSmallInteger0
2   <8C>     PushInstance1
3   <55>     PushArgument1
4   <0A>     LoadSelf
5   <E2>     SendSelector1
6   <48>     Return
```

```
@1: 7F647D: push EBP
    7F647E: mov EBP, ESP
    7F6480: push EAX
    7F6481: mov ESI, EAX
    ...
@2: 7F6492: push 1                  ;  PushSmallInteger0
    7F6497: push [ESI]              ;  PushInstance1
    7F6499: push [EBP+8]            ;  PushArgument1
    7F649C: mov EAX, ESI            ;  LoadSelf
    7F649E: call 740207             ;  SendSelector1
    7F64A3: mov ESP, EBP            ;  Return
    7F64A5: pop EBP
    7F64A6: mov ESI, [EBP-4]
    7F64A9: ret 4 NEAR
```

- Smalltalk stack == native stack
- Instance variables are accessed directly
- Contexts are stored in native stack

nativizing VMs
Cincom VW



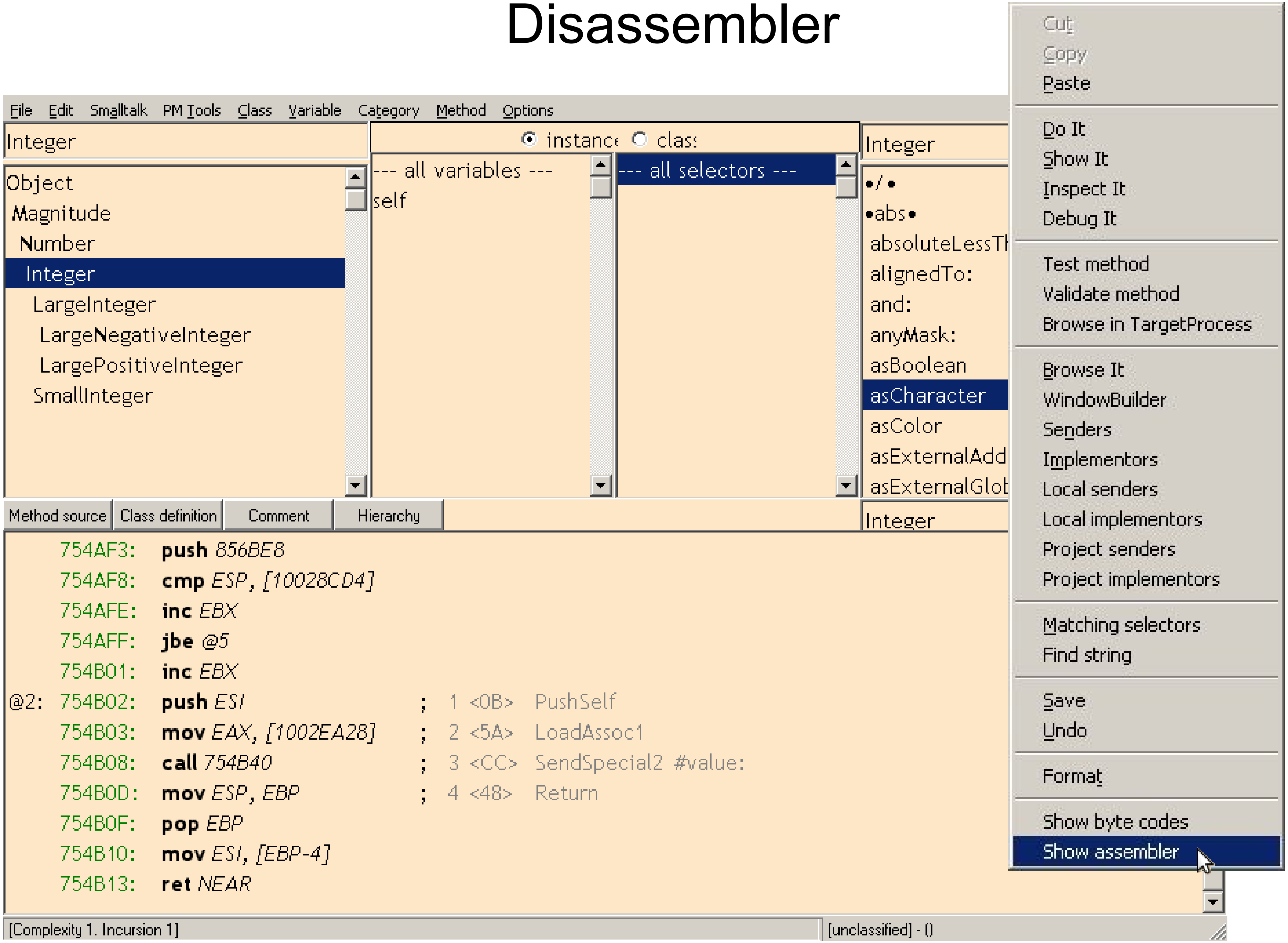
```
test: arg
  ^self with: 0 with: inst with: arg
```

```
1   <44>     push self
2   <49>     push 0
3   <10>     push local 0
4   <00>     push inst 0
5   <CE BE>  send
6   <65>     return
```

```
888AE6A: push EBP
888AE6B: mov EBP, ESP
888AE6D: sub ESP, 4
...
888AE7D: push EBX                  ;  save receiver
888AE7E: push ESI                  ;  save first arg
...
888AE87: push 3                    ;  push 0
888AE89: mov EDI, [EBP-10]         ;  push local 0
888AE8C: mov ESI, [EDX]            ;  push inst 0
888AE8E: mov EBX, [EBP-14]         ;  push self
888AE91: mov EDX, 9502FF0
888AE96: call 80A6570              ;  send with:with:with
888AE9B: leave
888AE9C: ret
```

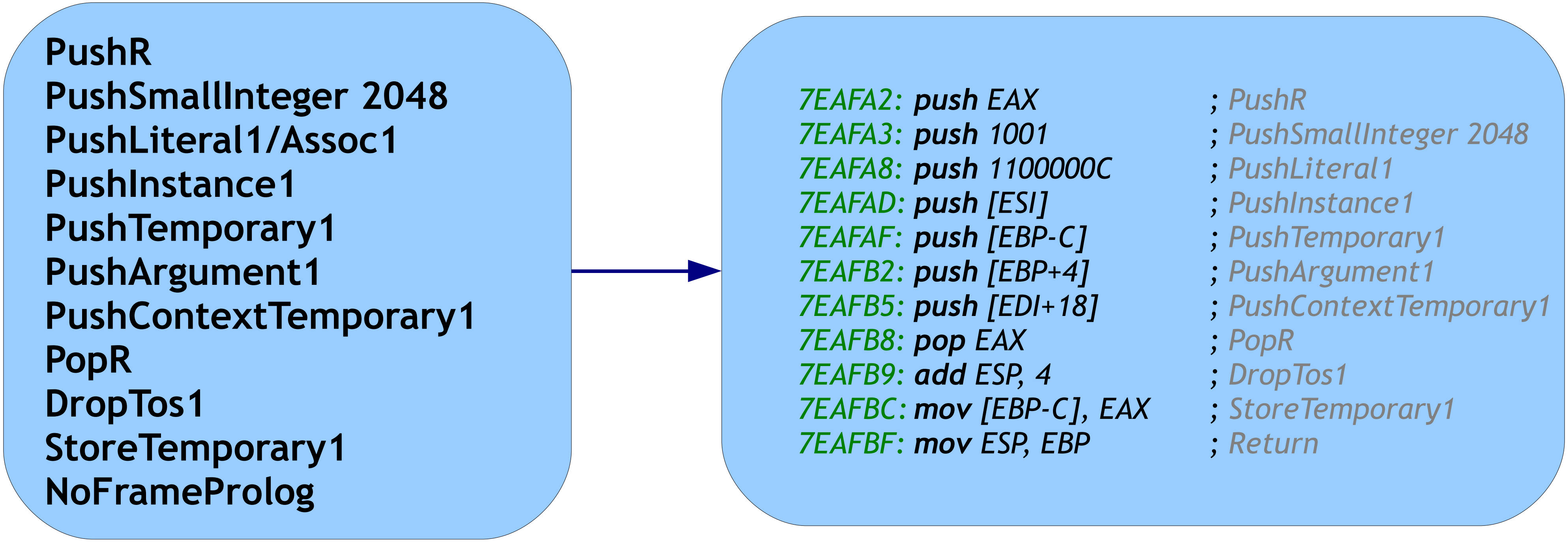
- Smalltalk stack =~= native stack
  - First arguments go in registers
- Instance variables are accessed “directly” (some)
- Contexts are stored in native stack

Disassembler



```
    754AF3:   push 856BE8
    754AF8:   cmp ESP, [10028CD4]
    754AFE:   inc EBX
    754AFF:   jbe @5
    754B01:   inc EBX
@2: 754B02:   push ESI                    ;  1 <0B>  PushSelf
    754B03:   mov EAX, [1002EA28]         ;  2 <5A>  LoadAssoc1
    754B08:   call 754B40                 ;  3 <CC>  SendSpecial2 #value:
    754B0D:   mov ESP, EBP                ;  4 <48>  Return
    754B0F:   pop EBP
    754B10:   mov ESI, [EBP-4]
    754B13:   ret NEAR
```

Conding in assembly stack operations

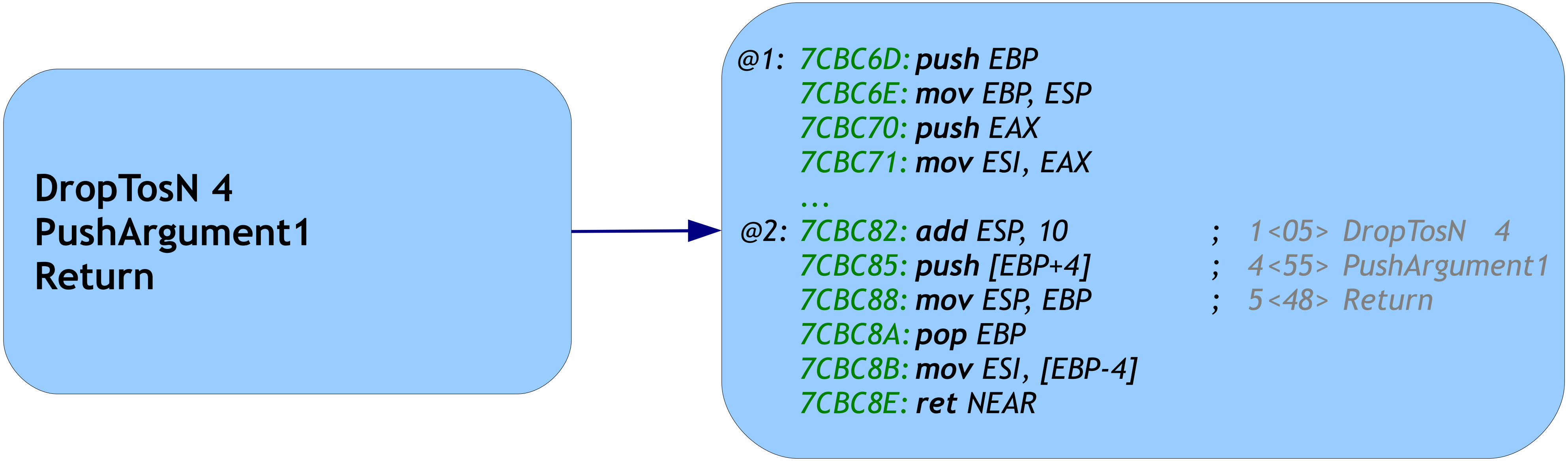


```
PushR
PushSmallInteger 2048
PushLiteral1/Assoc1
PushInstance1
PushTemporary1
PushArgument1
PushContextTemporary1
PopR
DropTos1
StoreTemporary1
NoFrameProlog
```

```
7EAFA2: push EAX               ; PushR
7EAFA3: push 1001              ; PushSmallInteger 2048
7EAFA8: push 1100000C          ; PushLiteral1
7EAFAD: push [ESI]             ; PushInstance1
7EAFAF: push [EBP-C]           ; PushTemporary1
7EAFB2: push [EBP+4]           ; PushArgument1
7EAFB5: push [EDI+18]          ; PushContextTemporary1
7EAFB8: pop EAX                ; PopR
7EAFB9: add ESP, 4             ; DropTos1
7EAFBC: mov [EBP-C], EAX       ; StoreTemporary1
7EAFBF: mov ESP, EBP           ; Return
```

- **Push**: add arbitrary things to the stack
- **StoreTemporary**: random access to negative offsets
- **PopR, DropTos**: arbitrarily move the stack pointer
- **NoFrameProlog**: skips saving/restoring the FP and SP

Escaping Digitalk VM

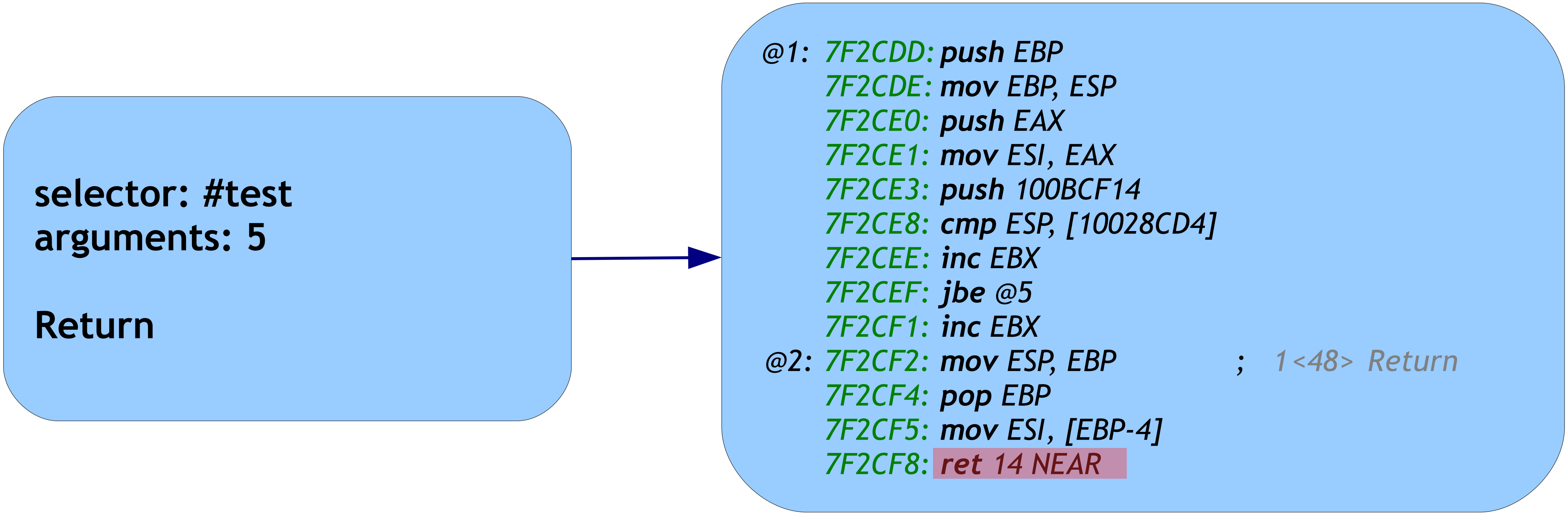


DropTosN 4
PushArgument1
Return

```
@1:  7CBC6D: push EBP
     7CBC6E: mov EBP, ESP
     7CBC70: push EAX
     7CBC71: mov ESI, EAX
     ...
@2:  7CBC82: add ESP, 10           ;  1<05> DropTosN    4
     7CBC85: push [EBP+4]          ;  4<55> PushArgument1
     7CBC88: mov ESP, EBP          ;  5<48> Return
     7CBC8A: pop EBP
     7CBC8B: mov ESI, [EBP-4]
     7CBC8E: ret NEAR
```

- Drop top of stack
- Overwrite return address with argument
- Return

Unbalancing the stack



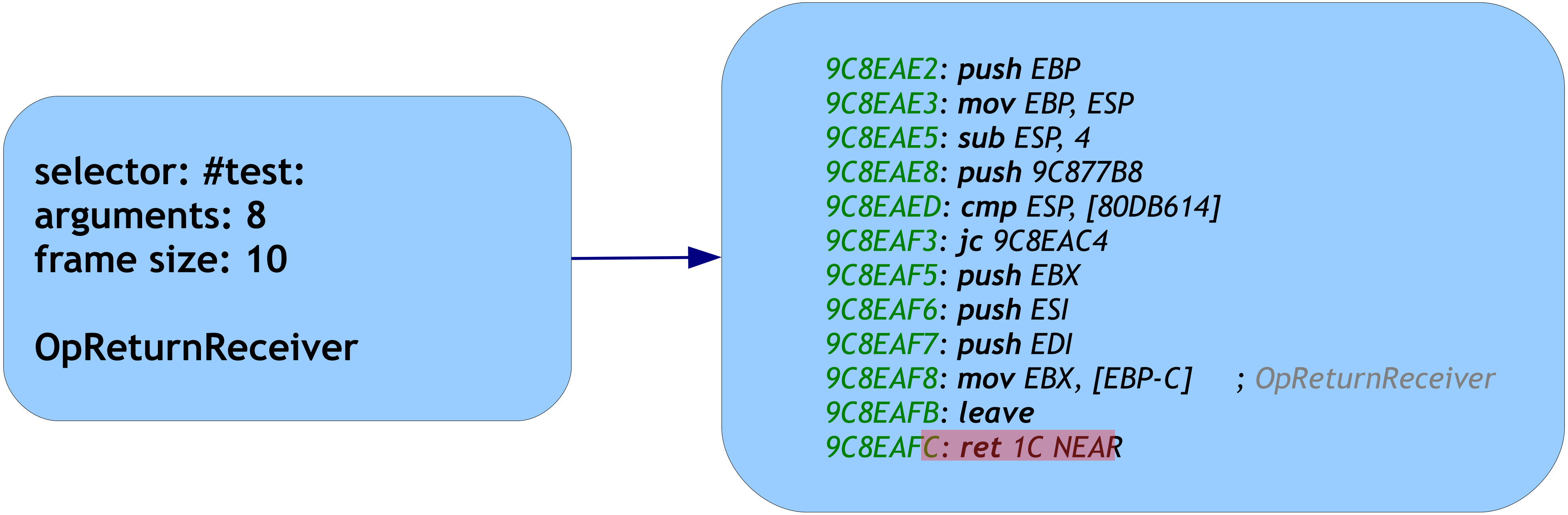
selector: #test
arguments: 5

Return

```
@1:  7F2CDD: push EBP
     7F2CDE: mov EBP, ESP
     7F2CE0: push EAX
     7F2CE1: mov ESI, EAX
     7F2CE3: push 100BCF14
     7F2CE8: cmp ESP, [10028CD4]
     7F2CEE: inc EBX
     7F2CEF: jbe @5
     7F2CF1: inc EBX
@2:  7F2CF2: mov ESP, EBP          ;   1<48> Return
     7F2CF4: pop EBP
     7F2CF5: mov ESI, [EBP-4]
     7F2CF8: ret 14 NEAR
```

- Caller pushes no arguments
- Callee cleans 5 arguments, unbalances stack
- Caller can modify *protected* values (return address)

Escaping Cincom VM



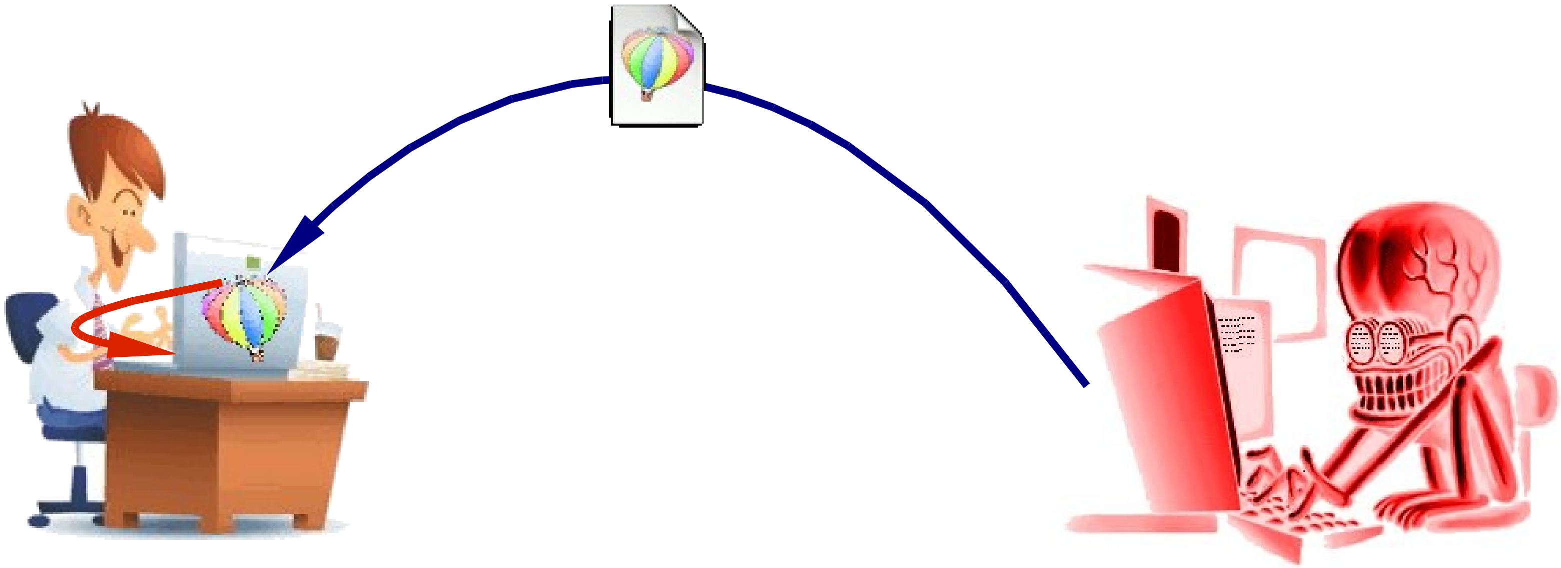
selector: #test:
arguments: 8
frame size: 10

OpReturnReceiver

```
9C8EAE2: push EBP
9C8EAE3: mov EBP, ESP
9C8EAE5: sub ESP, 4
9C8EAE8: push 9C877B8
9C8EAED: cmp ESP, [80DB614]
9C8EAF3: jc 9C8EAC4
9C8EAF5: push EBX
9C8EAF6: push ESI
9C8EAF7: push EDI
9C8EAF8: mov EBX, [EBP-C]     ; OpReturnReceiver
9C8EAFB: leave
9C8EAFC: ret 1C NEAR
```

- Caller pushes 1 argument
- Callee cleans 7 arguments, unbalances stack
- Caller can modify *protected* values (return address)

Attack layout



- Attacker transfers a CompiledMethod and activates it
- Attacker escapes the VM, and accesses the OS
- OS does not provide Application storage isolation
- Attacker gets stored passwords and credit cards

Securing Smalltalk
(some ideas)

✗

```
fileMeta := 0 class class class allInstances
    detect: [:metaClass |
      metaClass instanceClass name = 'File'].
fileMeta instanceClass openReadOnly: '...\savedPasswords'
```

- Reachability

✓

```
File pathNameReadOnly: 'temporaryFile.dat'
```

✗

```
File pathNameReadOnly: '...\savedPasswords'
```

- Sandboxing

✓

```
PushR, PushR, DropTos2, Return
```

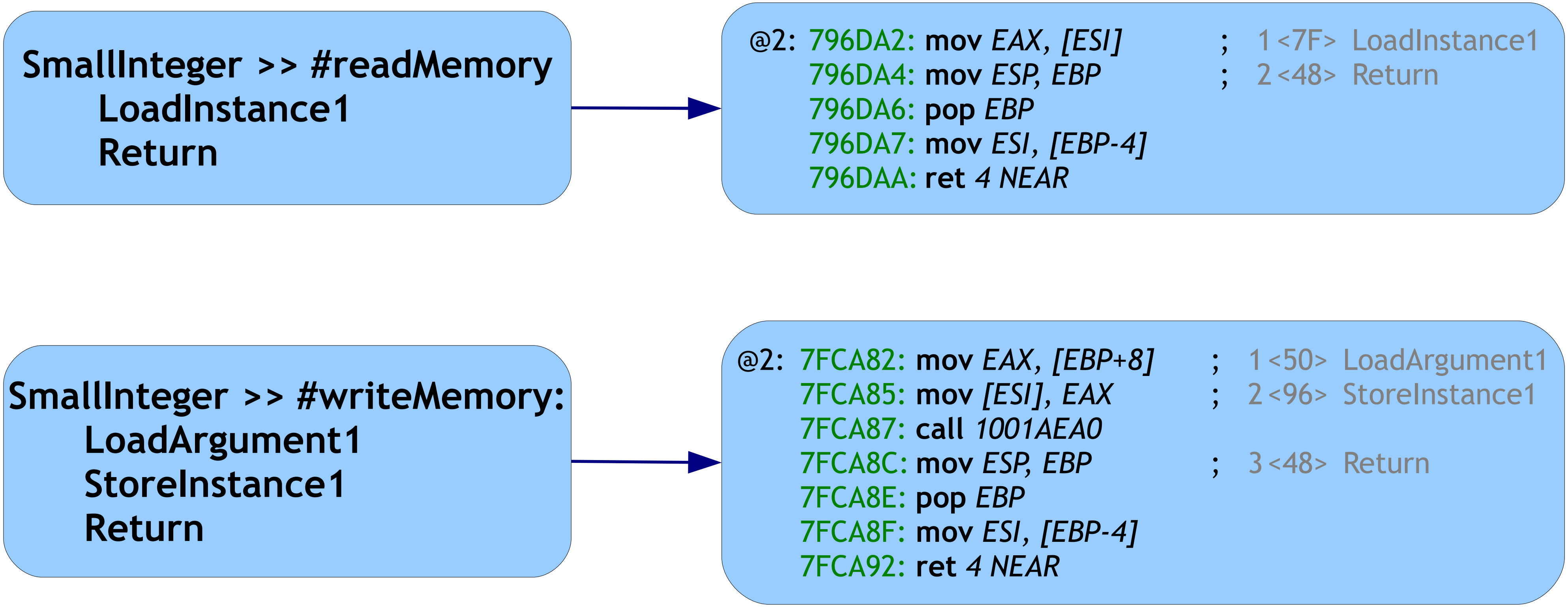
✗

```
PushR, PushR, Return
```

- Verifier

- Are all three necessary?

Escaping Digitalk VM



```
SmallInteger >> #readMemory
     LoadInstance1
     Return
```

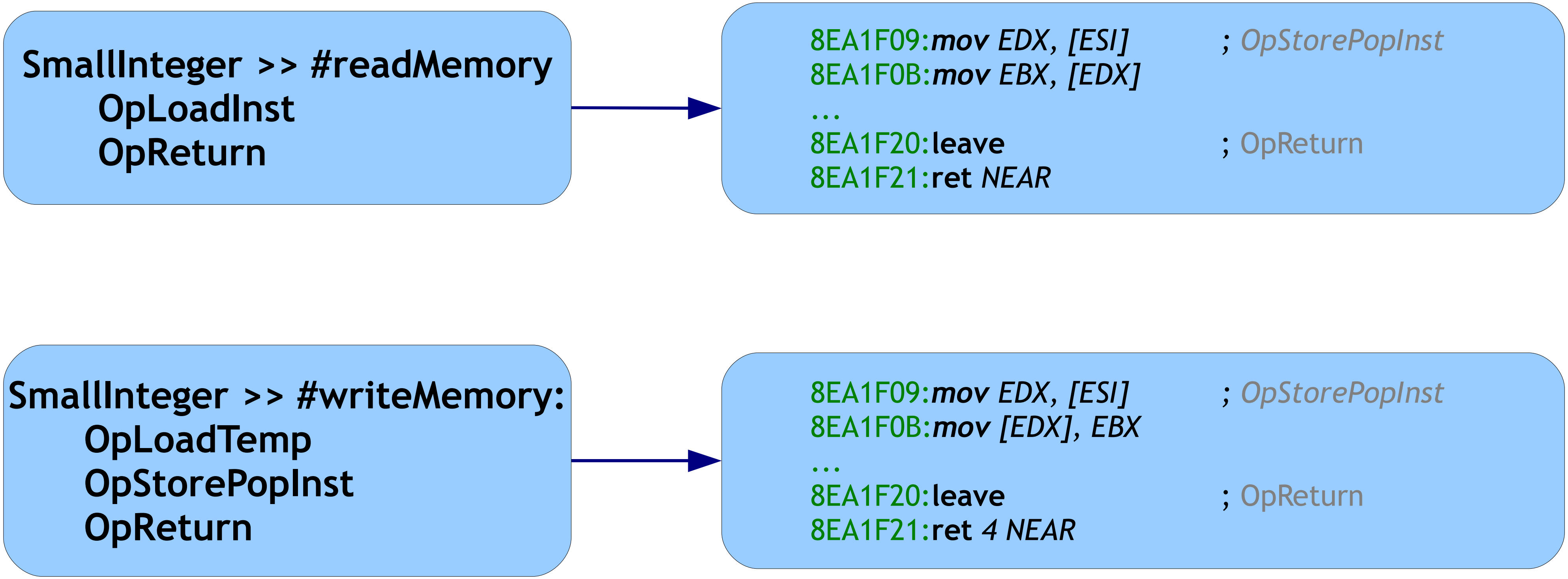
```
@2: 796DA2: mov EAX, [ESI]           ;  1 <7F>  LoadInstance1
    796DA4: mov ESP, EBP             ;  2 <48>  Return
    796DA6: pop EBP
    796DA7: mov ESI, [EBP-4]
    796DAA: ret 4 NEAR
```

```
SmallInteger >> #writeMemory:
     LoadArgument1
     StoreInstance1
     Return
```

```
@2: 7FCA82: mov EAX, [EBP+8]         ;  1 <50>  LoadArgument1
    7FCA85: mov [ESI], EAX           ;  2 <96>  StoreInstance1
    7FCA87: call 1001AEA0
    7FCA8C: mov ESP, EBP             ;  3 <48>  Return
    7FCA8E: pop EBP
    7FCA8F: mov ESI, [EBP-4]
    7FCA92: ret 4 NEAR
```

- Arbitrary memory read
- Arbitrary memory write

Escaping Cincom VM



```
SmallInteger >> #readMemory
      OpLoadInst
      OpReturn
```

```
8EA1F09:mov EDX, [ESI]         ; OpStorePopInst
8EA1F0B:mov EBX, [EDX]
...
8EA1F20:leave                  ; OpReturn
8EA1F21:ret NEAR
```

```
SmallInteger >> #writeMemory:
      OpLoadTemp
      OpStorePopInst
      OpReturn
```

```
8EA1F09:mov EDX, [ESI]         ; OpStorePopInst
8EA1F0B:mov [EDX], EBX
...
8EA1F20:leave                  ; OpReturn
8EA1F21:ret 4 NEAR
```

- Arbitrary memory read

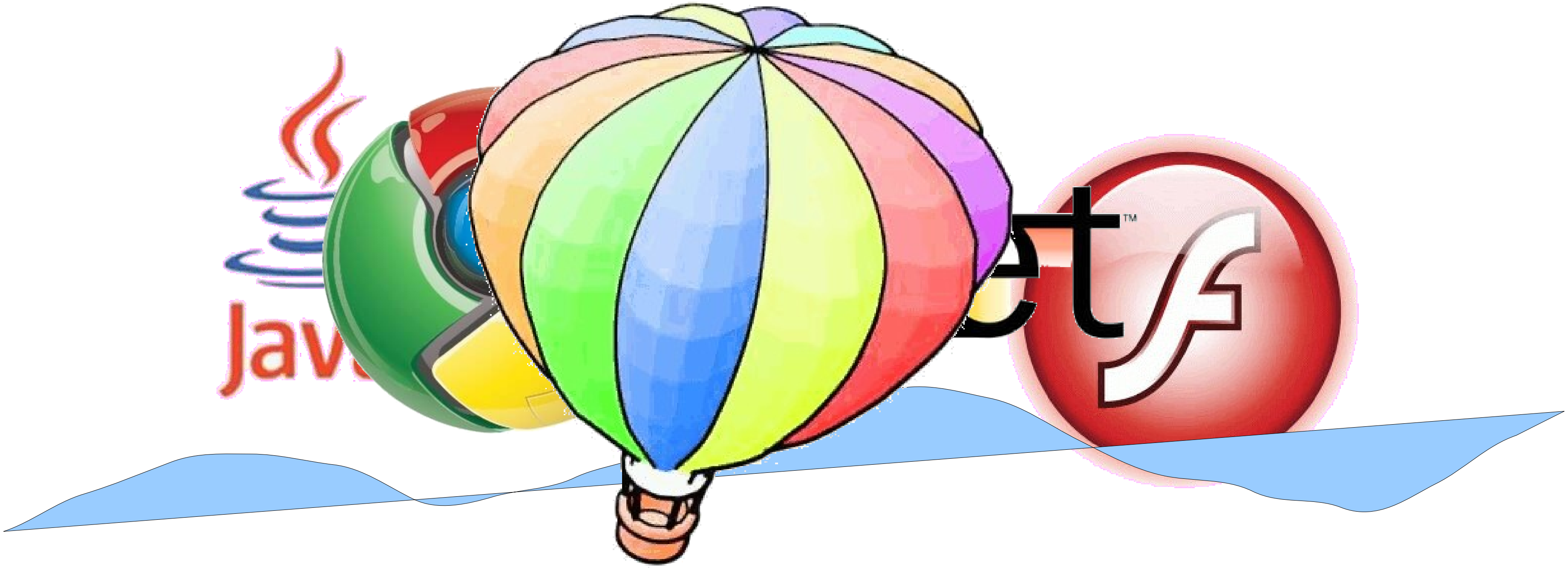
(#[0 0 0 16r78 16r56 16r34 16r12] copyToHeap asInteger / 4) readMemory

- Arbitrary memory write

```
[Audience  hasQuestions] whileTrue: [
    self answer: Audience nextQuestion]
```

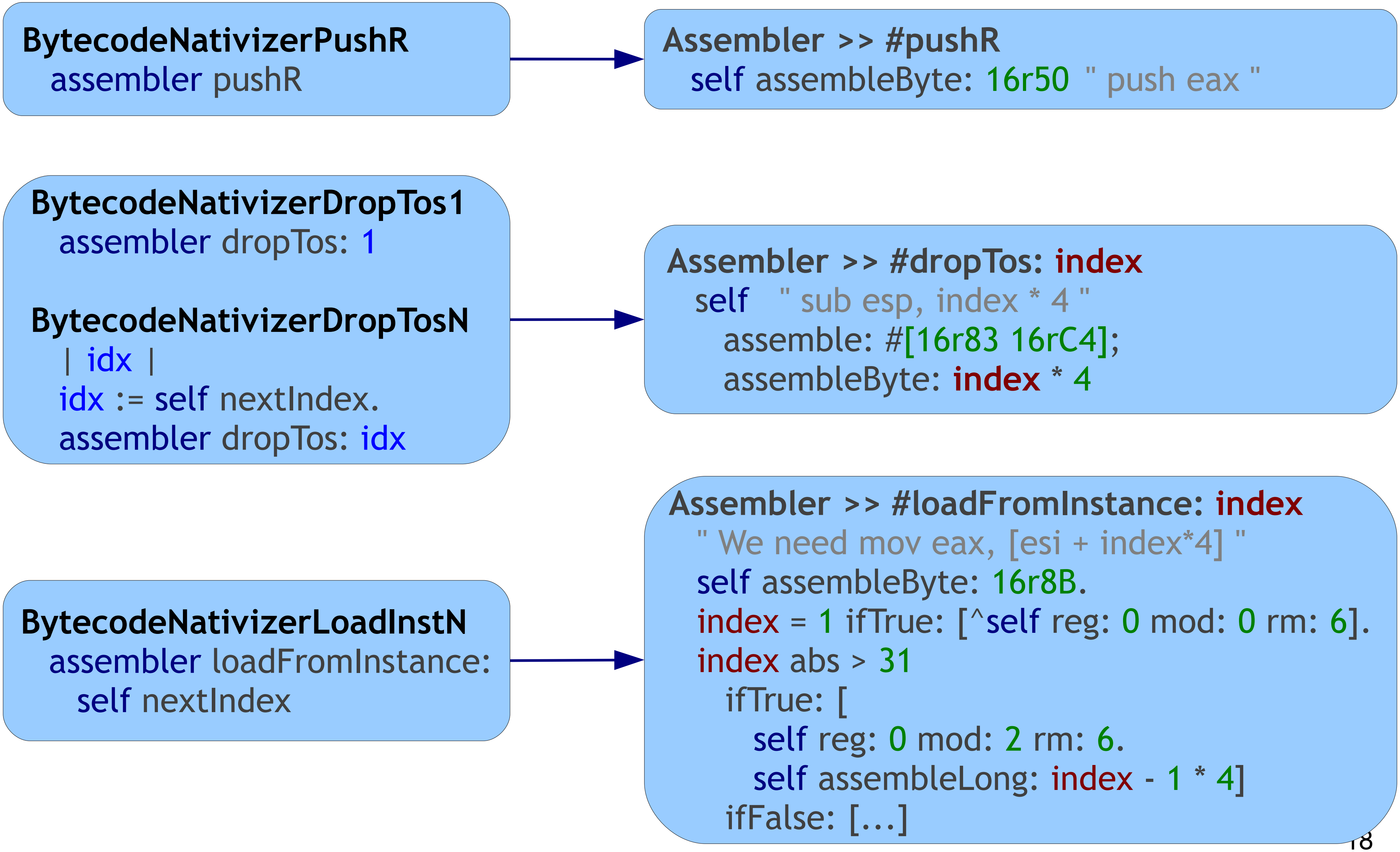
Further understanding

better assessment



PushArgument2

Documenting Bytecodes



```
BytecodeNativizerPushR
  assembler pushR
```

```
Assembler >> #pushR
  self assembleByte: 16r50  " push eax "
```

```
BytecodeNativizerDropTos1
  assembler dropTos: 1

BytecodeNativizerDropTosN
  | idx |
  idx := self nextIndex.
  assembler dropTos: idx
```

```
Assembler >> #dropTos: index
  self   " sub esp, index * 4 "
    assemble: #[16r83 16rC4];
    assembleByte: index * 4
```

```
BytecodeNativizerLoadInstN
  assembler loadFromInstance:
    self nextIndex
```

```
Assembler >> #loadFromInstance: index
  " We need mov eax, [esi + index*4] "
  self assembleByte: 16r8B.
  index = 1 ifTrue: [^self reg: 0 mod: 0 rm: 6].
  index abs > 31
    ifTrue: [
      self reg: 0 mod: 2 rm: 6.
      self assembleLong: index - 1 * 4]
    ifFalse: [...]
```

Testing Bytecodes

Templates

```
loadInstance1
  "
  1   <7F>  LoadInstance1
  2   <48>  Return
  "
     | ~~a~~ |
     ^testSelector
```

```
loadInstanceNoProlog1
  "
  1    <02>  NoFrameProlog
  2   <7F>  LoadInstance1
  3   <48>  Return
  "
  ^testSelector
```

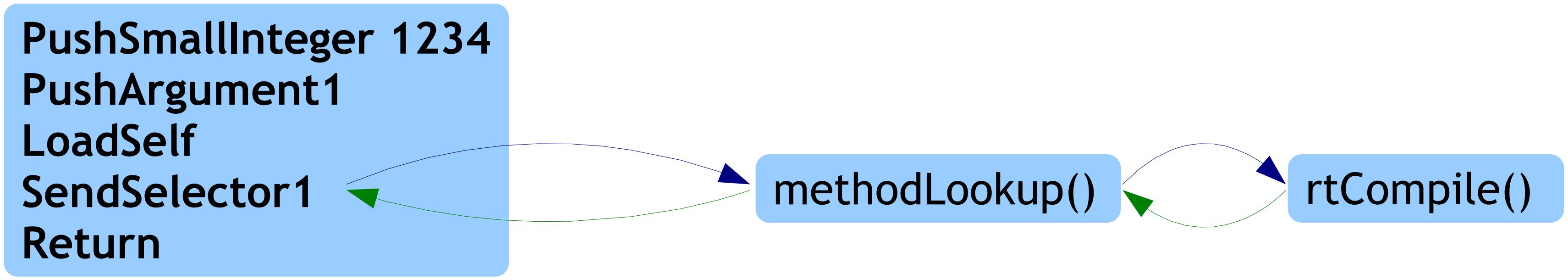
Test

```
testSameAsOriginal: cm
   | original documentation |
  original := CompiledMethodNativizer
       originalNativize: cm.
  documentation := CompiledMethodNativizer
       nativize: cm.
  self assert: original == documentation
```

```
shortForwardTestJumpFalse
  "
1   <0E>  LoadTrue
2   <1B>  TestJumpFalse   6
5   <14>  LoadSmallInteger1
6   <49>  ReturnSelf
  "
      true ifTrue: [1].
```

So far so good... but does the generated code work?

Two worlds unite

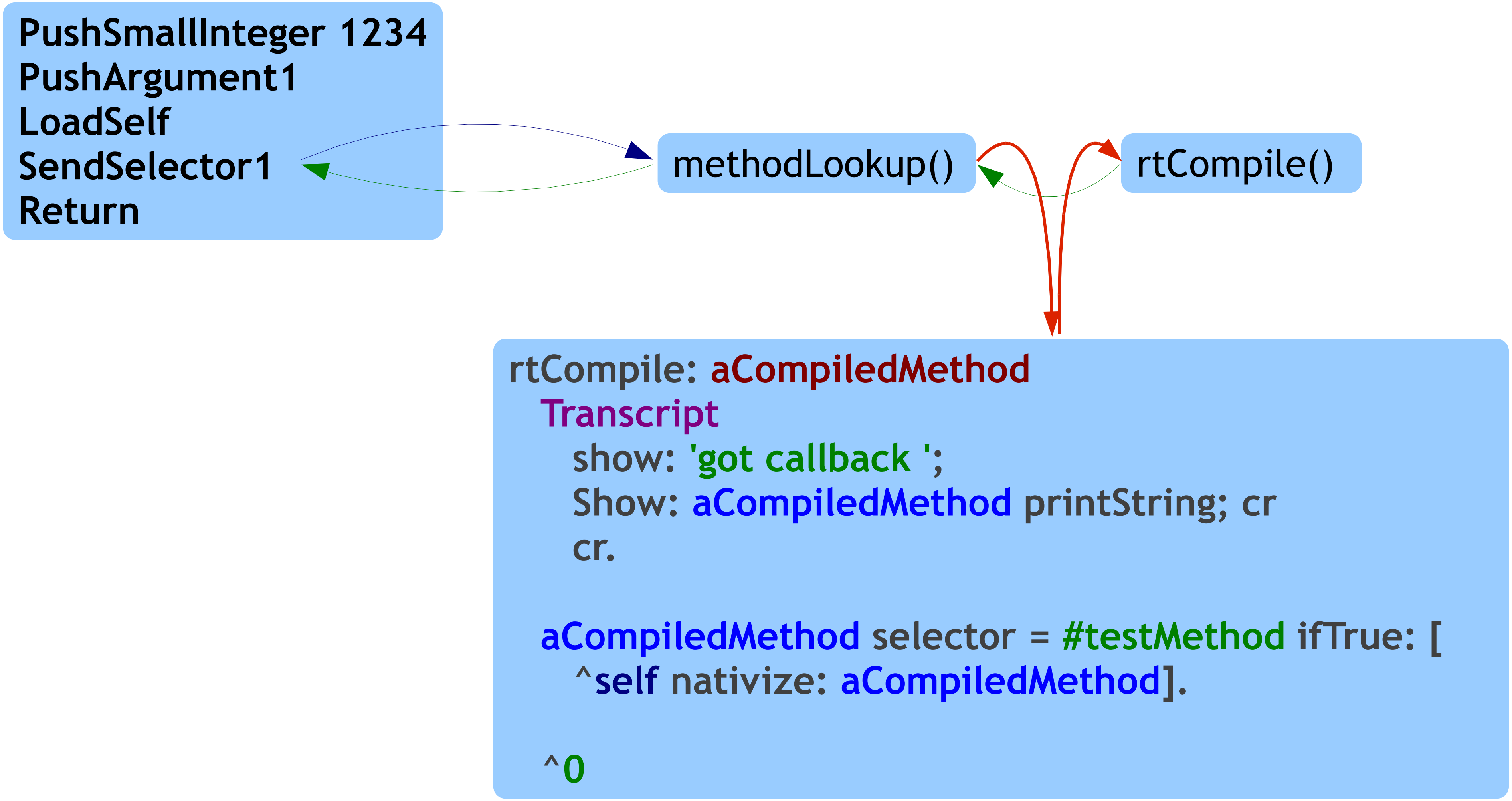


```
PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return
```

methodLookup()

rtCompile()

Two worlds unite



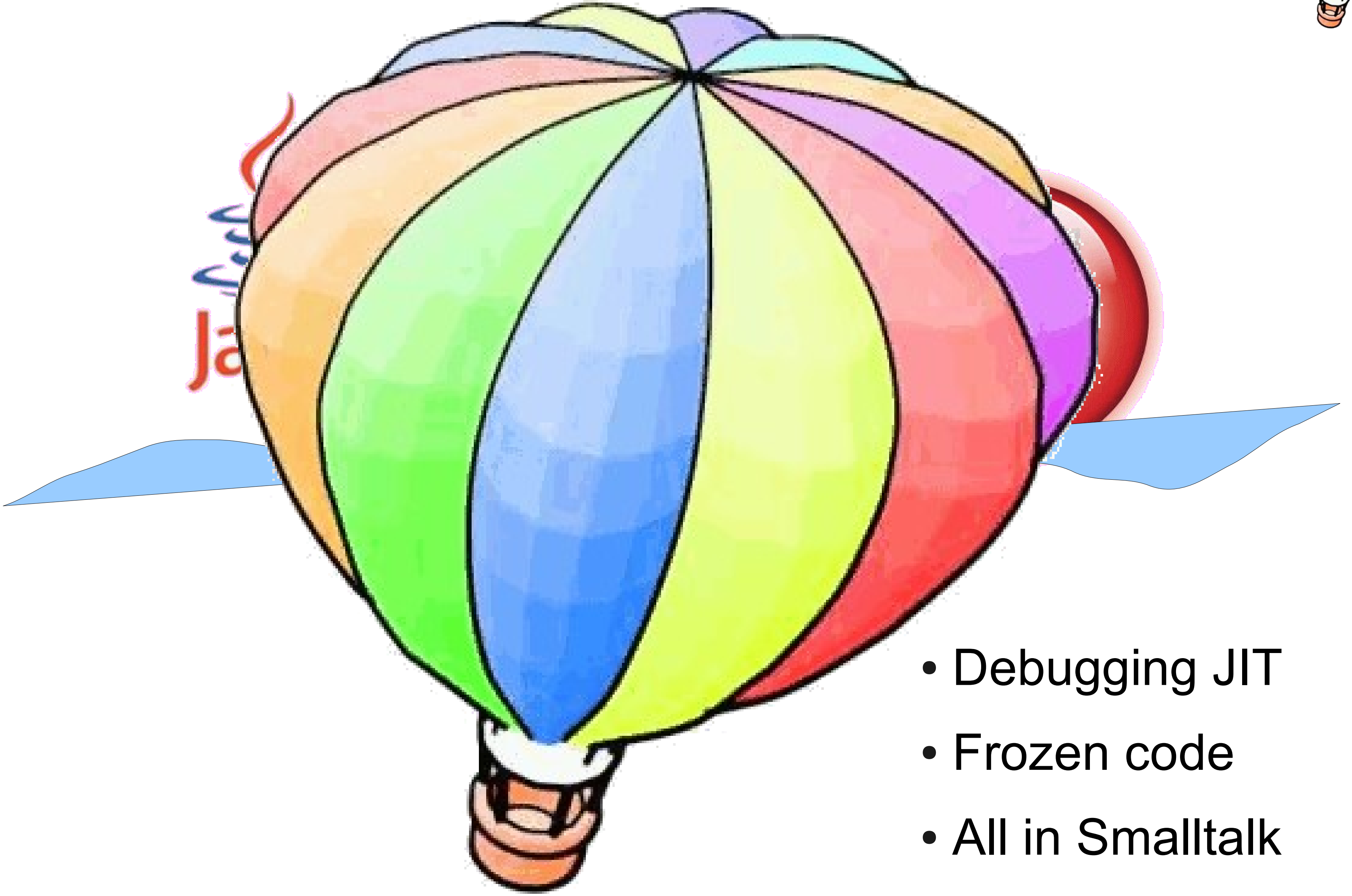
```
PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return
```

methodLookup()

rtCompile()

```
rtCompile: aCompiledMethod
  Transcript
    show: 'got callback ';
    Show: aCompiledMethod printString; cr
    cr.

  aCompiledMethod selector = #testMethod ifTrue: [
    ^self nativize: aCompiledMethod].

  ^0
```

What's next?



- Debugging JIT
- Frozen code
- All in Smalltalk

```
[Audience hasQuestions] whileTrue: [
   self answer: Audience nextQuestion].

Audience do: [:you | self thank: you].

self returnTo: Audience
```

one aspect of

Security on JIT VMs

and more

~~Richie++~~ / richie / gera@corest.com

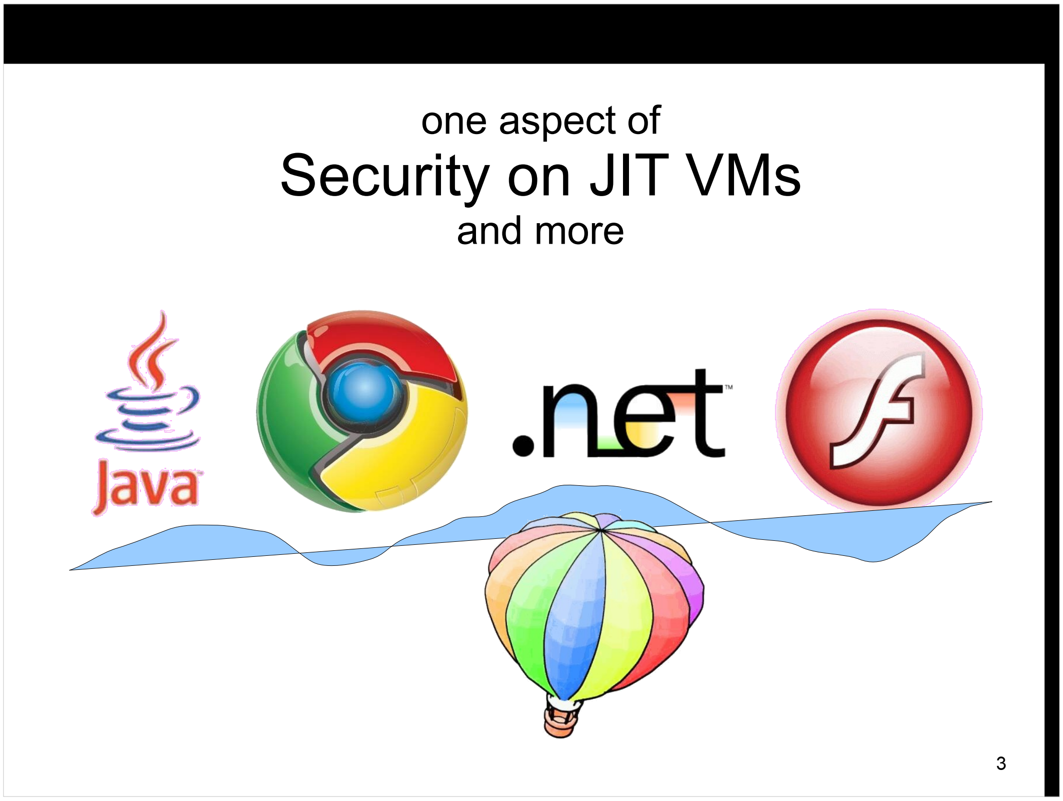
- We understand why it's important that all this VMs have security: We use most of them every day in some way or another, and through them we extend our trust to untrusted mobile code applications from we download from unknown sources.
- They have all gone, directly or indirectly, through some security audits and, one way or another, their developers today care about security issues.
- Smalltalk has grown as the very open "socialist" environment we all love, were we just trust everybody. The VMs developers community has not really payed much attention to security (not at least from a mobile code perspective)
- Time has come for mobile code to also reach Smalltalk (browser plugins, Croquet objects with their own behavior, Scratch/EToys projects, seaside hosting)

Security?

- Does you application need security?
- Do you do anything for its security?

- Any application installed on a computer can open the door to an attacker, in our scenario we are assuming mobile code (maybe embedded in other type of content)
- The VM is supposed to provide some sandboxing, if the sandboxing can be broken the trust chain can be abused:

  the user trusts the VM
  the VM trusts/verifies the mobile code
  the mobile code fools the VM

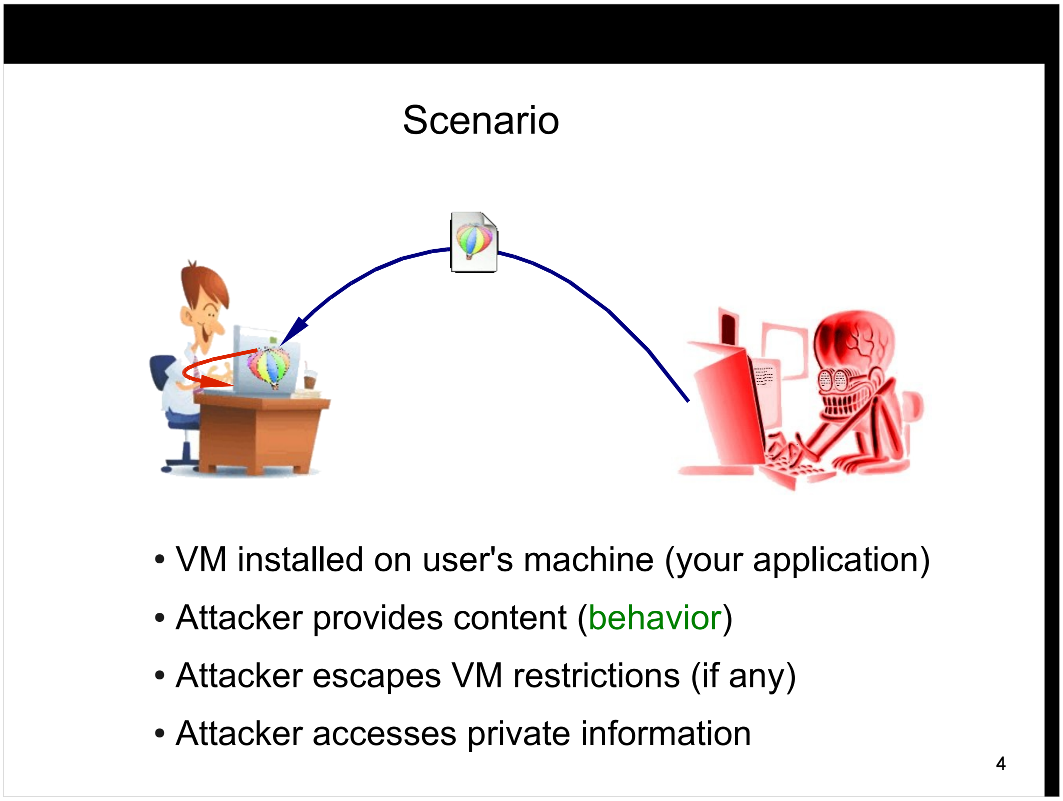


one aspect of

Security on JIT VMs

and more

- We understand why it's important that all this VMs have security: We use most of them every day in some way or another, and through them we extend our trust to untrusted mobile code applications from we download from unknown sources.
- They have all gone, directly or indirectly, through some security audits and, one way or another, their developers today care about security issues.
- Smalltalk has grown as the very open "socialist" environment we all love, were we just trust everybody. The VMs developers community has not really payed much attention to security (not at least from a mobile code perspective)
- Time has come for mobile code to also reach Smalltalk (browser plugins, Croquet objects with their own behavior, Scratch/EToys projects, seaside hosting)

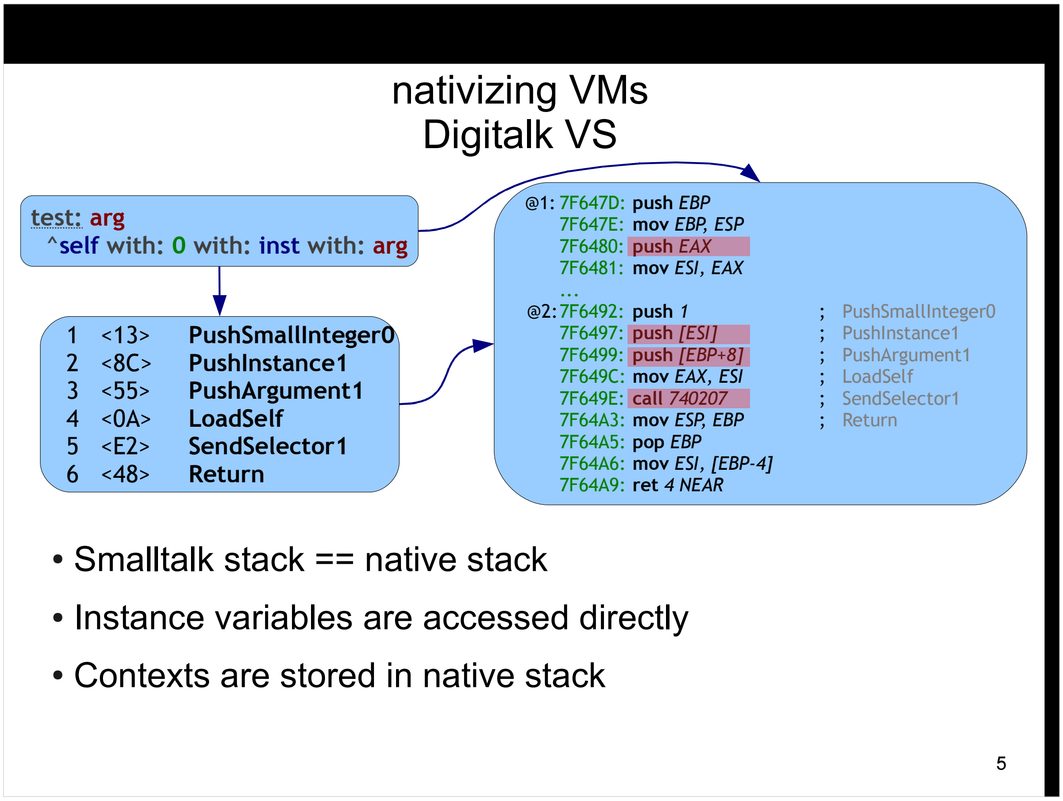


Scenario

- VM installed on user's machine (your application)
- Attacker provides content (behavior)
- Attacker escapes VM restrictions (if any)
- Attacker accesses private information

- Any application installed on a computer can open the door to an attacker, in our scenario we are assuming mobile code (maybe embedded in other type of content)
- The VM is supposed to provide some sandboxing, if the sandboxing can be broken the trust chain can be abused:

  the user trusts the VM
  the VM trusts/verifies the mobile code
  the mobile code fools the VM



nativizing VMs
Digitalk VS

```
test: arg
  ^self with: 0 with: inst with: arg
```

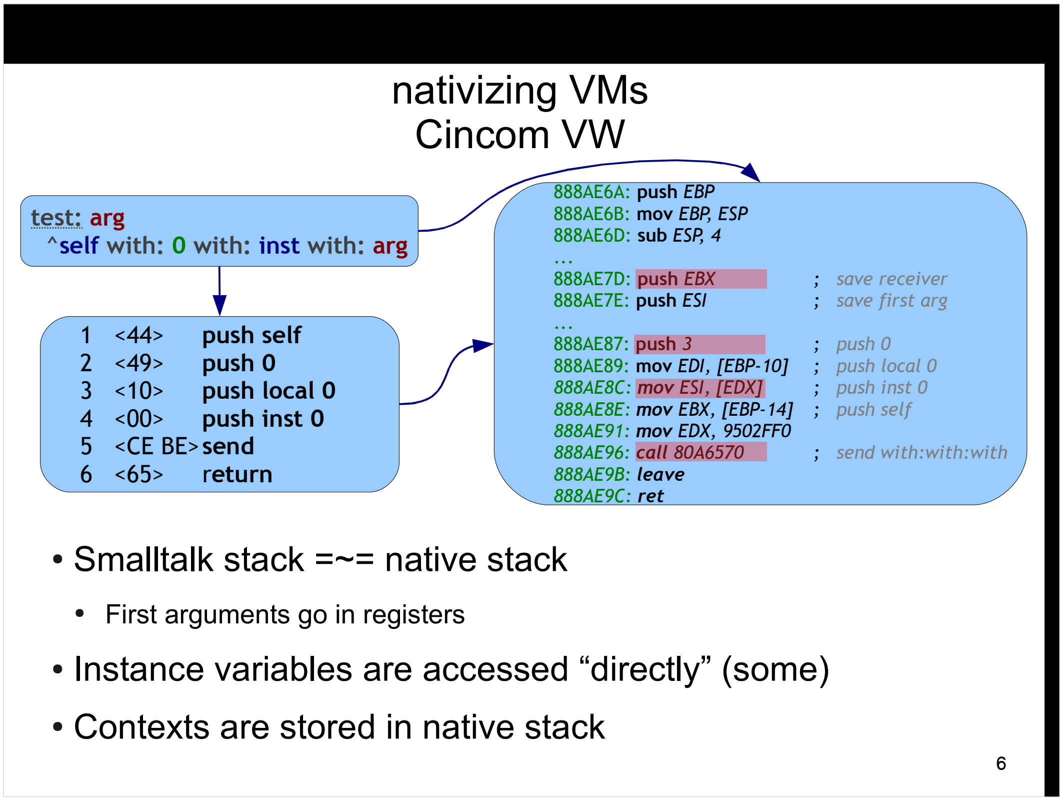
```
1  <13>   PushSmallInteger0
2  <8C>   PushInstance1
3  <55>   PushArgument1
4  <0A>   LoadSelf
5  <E2>   SendSelector1
6  <48>   Return
```

```
@1: 7F647D: push EBP
    7F647E: mov EBP, ESP
    7F6480: push EAX
    7F6481: mov ESI, EAX
    ...
@2: 7F6492: push 1                ;  PushSmallInteger0
    7F6497: push [ESI]            ;  PushInstance1
    7F6499: push [EBP+8]          ;  PushArgument1
    7F649C: mov EAX, ESI          ;  LoadSelf
    7F649E: call 740207           ;  SendSelector1
    7F64A3: mov ESP, EBP          ;  Return
    7F64A5: pop EBP
    7F64A6: mov ESI, [EBP-4]
    7F64A9: ret 4 NEAR
```

- Smalltalk stack == native stack
- Instance variables are accessed directly
- Contexts are stored in native stack

5

- Smalltalk is compiled to Bytecode
- Bytecode is nativized
- Smalltalk is directly compiled to Assembly
- Here we can see
    - Smalltalk stack is kept in the native stack (push nativized to push)
    - Contexts are normal native stack frames
        - return addresses managed through native call/ret
        - Arguments are accessed through the native frame pointer
        - Local variables are accessed through the native frame pointer
        - The receiver is saved in a local
    - Instance variables are accessed with direct memory accesses without any checks
- If we could arbitrarily access any indexed argument we could corrupt the return address or saved receiver (from other frames)



nativizing VMs
Cincom VW

```
test: arg
  ^self with: 0 with: inst with: arg
```

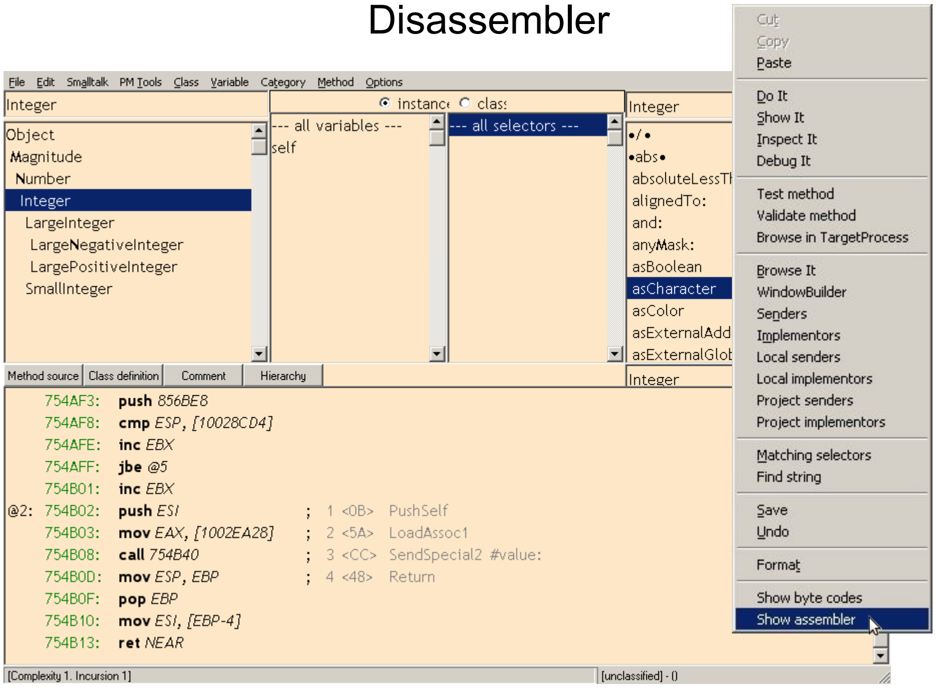
```
1  <44>     push self
2  <49>     push 0
3  <10>     push local 0
4  <00>     push inst 0
5  <CE BE>  send
6  <65>     return
```

```
888AE6A: push EBP
888AE6B: mov EBP, ESP
888AE6D: sub ESP, 4
...
888AE7D: push EBX              ;  save receiver
888AE7E: push ESI              ;  save first arg
...
888AE87: push 3                ;  push 0
888AE89: mov EDI, [EBP-10]     ;  push local 0
888AE8C: mov ESI, [EDX]        ;  push inst 0
888AE8E: mov EBX, [EBP-14]     ;  push self
888AE91: mov EDX, 9502FF0
888AE96: call 80A6570          ;  send with:with:with
888AE9B: leave
888AE9C: ret
```

- Smalltalk stack =~= native stack
  - First arguments go in registers
- Instance variables are accessed “directly” (some)
- Contexts are stored in native stack

- Here we can see
  - Smalltalk stack is kept in the native stack (push nativized to push)
  - Contexts are normal native stack frames
    - return addresses managed through native call/ret
    - after 3 args they accessed through the native frame pointer
    - Local variables are accessed through the native frame pointer
    - The receiver is saved in a local
  - Some instance variables are accessed with direct memory accesses without any checks, some are accessed through a routing that MAY do checks.
- If we could arbitrarily write any indexed argument we could corrupt the return address, saved receiver or locals (from other frames)
- If we could arbitrarily manipulate any indexed instance variable we could corrupt the object space

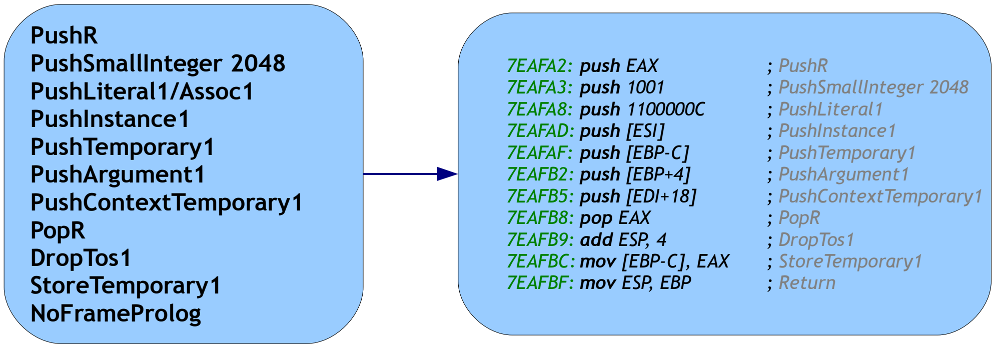
Disassembler



```
     754AF3:   push 856BE8
     754AF8:   cmp ESP, [10028CD4]
     754AFE:   inc EBX
     754AFF:   jbe @5
     754B01:   inc EBX
@2:  754B02:   push ESI                 ;  1 <0B>  PushSelf
     754B03:   mov EAX, [1002EA28]      ;  2 <5A>  LoadAssoc1
     754B08:   call 754B40              ;  3 <CC>  SendSpecial2 #value:
     754B0D:   mov ESP, EBP             ;  4 <48>  Return
     754B0F:   pop EBP
     754B10:   mov ESI, [EBP-4]
     754B13:   ret NEAR
```

[Complexity 1. Incursion 1]   [unclassified] - ()

Conding in assembly
stack operations

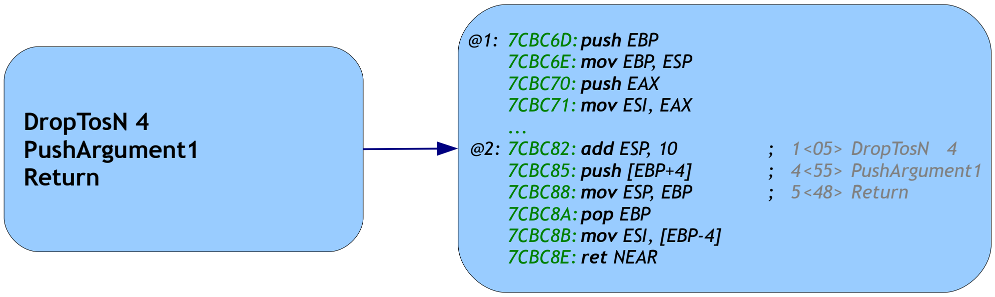


```
PushR
PushSmallInteger 2048
PushLiteral1/Assoc1
PushInstance1
PushTemporary1
PushArgument1
PushContextTemporary1
PopR
DropTos1
StoreTemporary1
NoFrameProlog
```

```
7EAFA2: push EAX              ; PushR
7EAFA3: push 1001             ; PushSmallInteger 2048
7EAFA8: push 1100000C         ; PushLiteral1
7EAFAD: push [ESI]            ; PushInstance1
7EAFAF: push [EBP-C]          ; PushTemporary1
7EAFB2: push [EBP+4]          ; PushArgument1
7EAFB5: push [EDI+18]         ; PushContextTemporary1
7EAFB8: pop EAX               ; PopR
7EAFB9: add ESP, 4            ; DropTos1
7EAFBC: mov [EBP-C], EAX      ; StoreTemporary1
7EAFBF: mov ESP, EBP          ; Return
```

- **Push**: add arbitrary things to the stack
- **StoreTemporary**: random access to negative offsets
- **PopR, DropTos**: arbitrarily move the stack pointer
- **NoFrameProlog**: skips saving/restoring the FP and SP

Escaping Digitalk VM

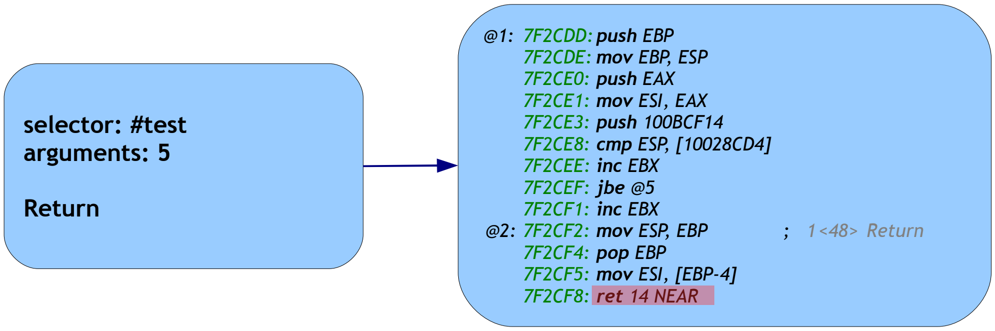


```
DropTosN 4
PushArgument1
Return
```

```
@1:  7CBC6D: push EBP
     7CBC6E: mov EBP, ESP
     7CBC70: push EAX
     7CBC71: mov ESI, EAX
     ...
@2:  7CBC82: add ESP, 10          ;  1<05> DropTosN   4
     7CBC85: push [EBP+4]         ;  4<55> PushArgument1
     7CBC88: mov ESP, EBP         ;  5<48> Return
     7CBC8A: pop EBP
     7CBC8B: mov ESI, [EBP-4]
     7CBC8E: ret NEAR
```

- Drop top of stack
- Overwrite return address with argument
- Return

Unbalancing the stack



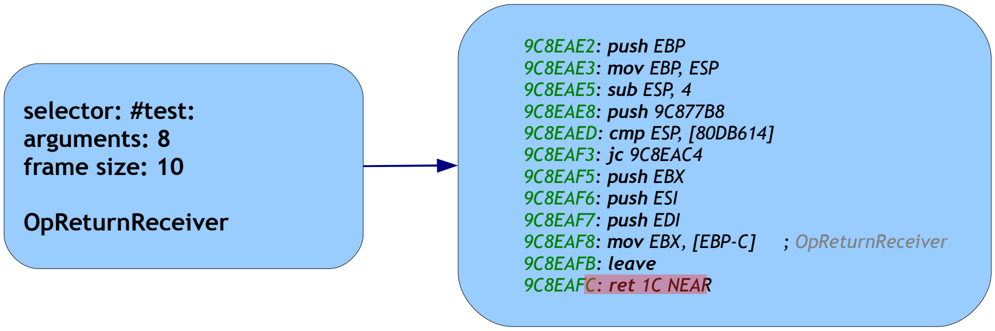
selector: #test
arguments: 5

Return

```
@1:  7F2CDD: push EBP
     7F2CDE: mov EBP, ESP
     7F2CE0: push EAX
     7F2CE1: mov ESI, EAX
     7F2CE3: push 100BCF14
     7F2CE8: cmp ESP, [10028CD4]
     7F2CEE: inc EBX
     7F2CEF: jbe @5
     7F2CF1: inc EBX
@2:  7F2CF2: mov ESP, EBP            ;   1<48> Return
     7F2CF4: pop EBP
     7F2CF5: mov ESI, [EBP-4]
     7F2CF8: ret 14 NEAR
```

- Caller pushes no arguments
- Callee cleans 5 arguments, unbalances stack
- Caller can modify *protected* values (return address)

Escaping Cincom VM



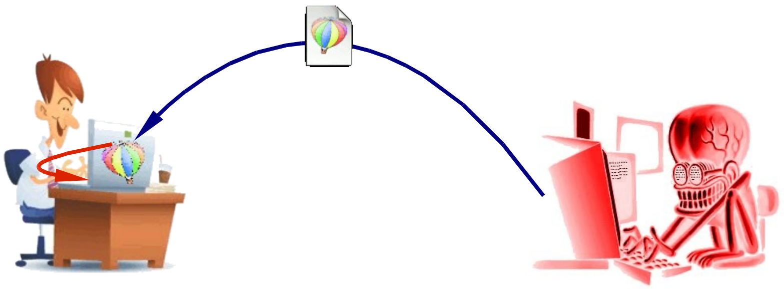
selector: #test:
arguments: 8
frame size: 10

OpReturnReceiver

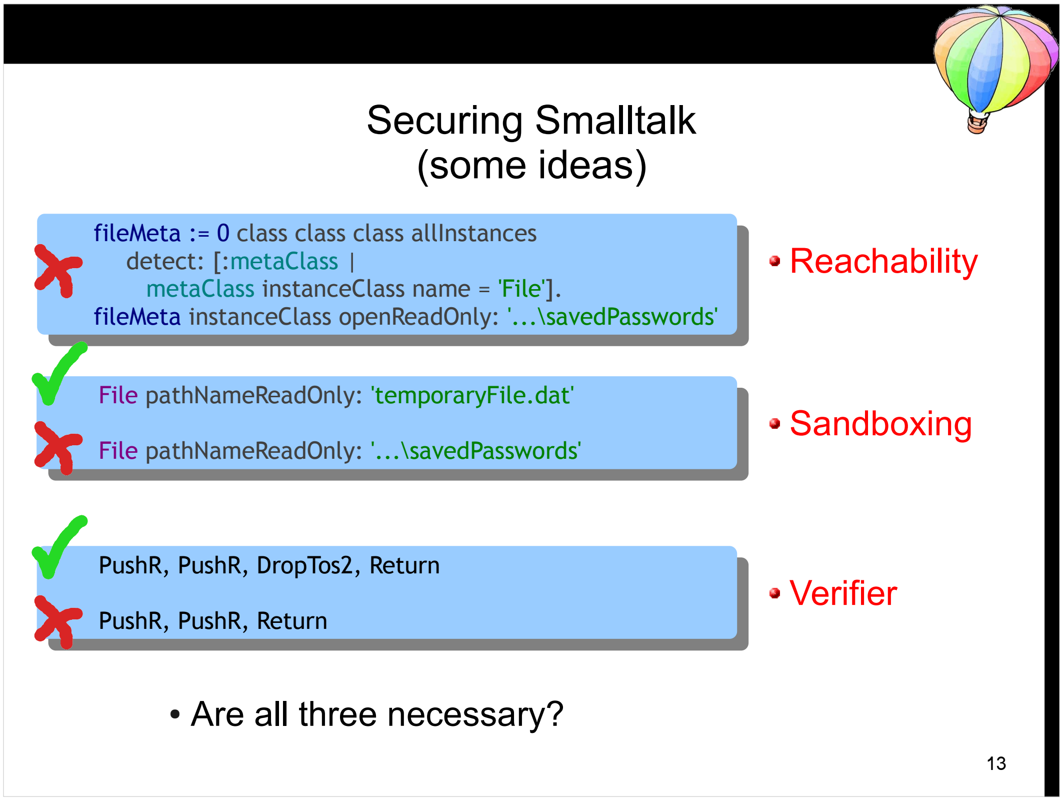
```
9C8EAE2: push EBP
9C8EAE3: mov EBP, ESP
9C8EAE5: sub ESP, 4
9C8EAE8: push 9C877B8
9C8EAED: cmp ESP, [80DB614]
9C8EAF3: jc 9C8EAC4
9C8EAF5: push EBX
9C8EAF6: push ESI
9C8EAF7: push EDI
9C8EAF8: mov EBX, [EBP-C]     ; OpReturnReceiver
9C8EAFB: leave
9C8EAFC: ret 1C NEAR
```

- Caller pushes 1 argument
- Callee cleans 7 arguments, unbalances stack
- Caller can modify *protected* values (return address)

Attack layout



- Attacker transfers a CompiledMethod and activates it
- Attacker escapes the VM, and accesses the OS
- OS does not provide Application storage isolation
- Attacker gets stored passwords and credit cards



Securing Smalltalk
(some ideas)

✗
```
fileMeta := 0 class class class allInstances
    detect: [:metaClass |
       metaClass instanceClass name = 'File'].
fileMeta instanceClass openReadOnly: '...\savedPasswords'
```

- Reachability

✓
```
File pathNameReadOnly: 'temporaryFile.dat'
```
✗
```
File pathNameReadOnly: '...\savedPasswords'
```

- Sandboxing

✓
```
PushR, PushR, DropTos2, Return
```
✗
```
PushR, PushR, Return
```

- Verifier

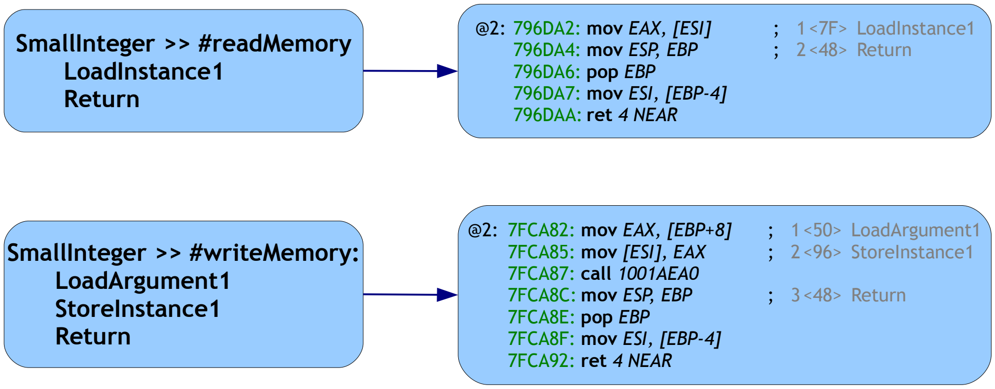
- Are all three necessary?

- In a fully dynamic and reflective system reachability is hard to constrain

- Sandboxing, ACL or permissions must be implemented in the VM itself. They could be bypassed, most likely, if implemented in the image.

- A bytecode verifier is mandatory in any kind of nativizing VM, and strict checks are also required in an interpreter.

Escaping Digitalk VM



```
SmallInteger >> #readMemory
      LoadInstance1
      Return
```

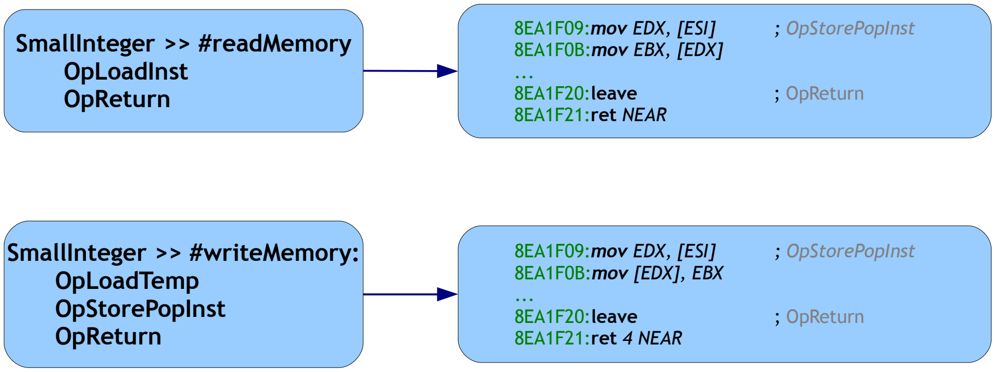
```
@2: 796DA2: mov EAX, [ESI]          ;  1 <7F> LoadInstance1
    796DA4: mov ESP, EBP            ;  2 <48> Return
    796DA6: pop EBP
    796DA7: mov ESI, [EBP-4]
    796DAA: ret 4 NEAR
```

```
SmallInteger >> #writeMemory:
      LoadArgument1
      StoreInstance1
      Return
```

```
@2: 7FCA82: mov EAX, [EBP+8]        ;  1 <50> LoadArgument1
    7FCA85: mov [ESI], EAX          ;  2 <96> StoreInstance1
    7FCA87: call 1001AEA0
    7FCA8C: mov ESP, EBP            ;  3 <48> Return
    7FCA8E: pop EBP
    7FCA8F: mov ESI, [EBP-4]
    7FCA92: ret 4 NEAR
```

- Arbitrary memory read
- Arbitrary memory write

Escaping Cincom VM



```
SmallInteger >> #readMemory
      OpLoadInst
      OpReturn
```

```
8EA1F09:mov EDX, [ESI]          ; OpStorePopInst
8EA1F0B:mov EBX, [EDX]
...
8EA1F20:leave                   ; OpReturn
8EA1F21:ret NEAR
```

```
SmallInteger >> #writeMemory:
      OpLoadTemp
      OpStorePopInst
      OpReturn
```

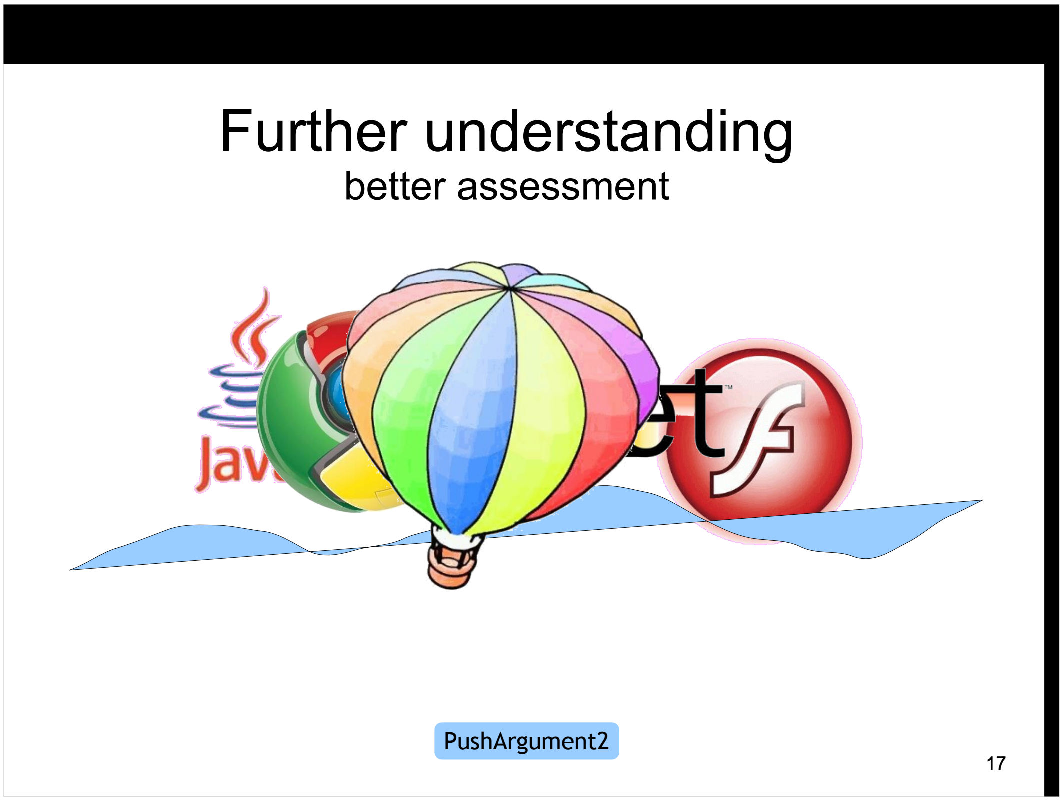
```
8EA1F09:mov EDX, [ESI]          ; OpStorePopInst
8EA1F0B:mov [EDX], EBX
...
8EA1F20:leave                   ; OpReturn
8EA1F21:ret 4 NEAR
```

- Arbitrary memory read

(#[0 0 0 16r78 16r56 16r34 16r12] copyToHeap asInteger / 4) readMemory

- Arbitrary memory write

```
[Audience  hasQuestions] whileTrue: [
   self answer: Audience nextQuestion]
```



Further understanding

better assessment

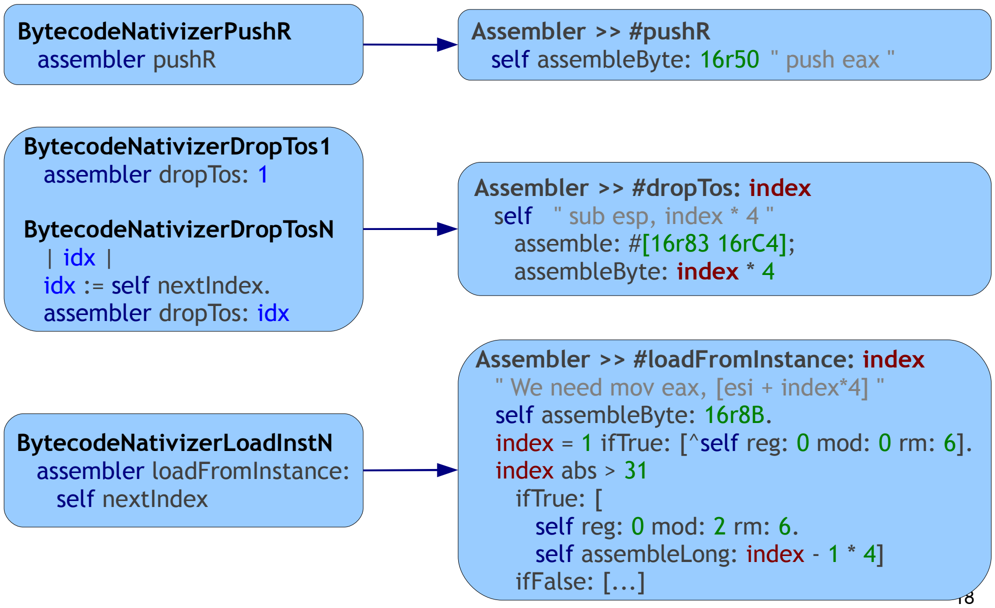
PushArgument2

17

- Just randomly looking how each bytecode is nativized is not enough to asses the security of the system, we need to understand all possible variations for each bytecode.

- So we need to understand and document the workings of the JIT nativizer, and what better documentation that something which can be debugged. So, as we all learned long time ago with the original specification of the Smalltalk VM, we started documenting not in .doc, but in .st

Documenting Bytecodes



```
BytecodeNativizerPushR
  assembler pushR
```

```
Assembler >> #pushR
  self assembleByte: 16r50  " push eax "
```

```
BytecodeNativizerDropTos1
  assembler dropTos: 1

BytecodeNativizerDropTosN
  | idx |
  idx := self nextIndex.
  assembler dropTos: idx
```

```
Assembler >> #dropTos: index
  self   " sub esp, index * 4 "
    assemble: #[16r83 16rC4];
    assembleByte: index * 4
```

```
BytecodeNativizerLoadInstN
  assembler loadFromInstance:
    self nextIndex
```

```
Assembler >> #loadFromInstance: index
  " We need mov eax, [esi + index*4] "
  self assembleByte: 16r8B.
  index = 1 ifTrue: [^self reg: 0 mod: 0 rm: 6].
  index abs > 31
    ifTrue: [
      self reg: 0 mod: 2 rm: 6.
      self assembleLong: index - 1 * 4]
    ifFalse: [...]
```

Testing Bytecodes

Templates

```
loadInstance1
  "
  1   <7F>  LoadInstance1
  2   <48>  Return
  "
     | a |
     ^testSelector
```

```
loadInstanceNoProlog1
  "
  1    <02> NoFrameProlog
  2   <7F>  LoadInstance1
  3   <48>  Return
  "
  ^testSelector
```

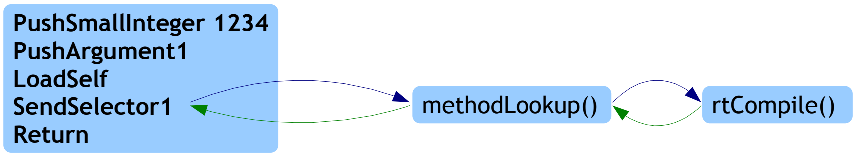
Test

```
testSameAsOriginal: cm
   | original documentation |
  original := CompiledMethodNativizer
       originalNativize: cm.
  documentation := CompiledMethodNativizer
       nativize: cm.
  self assert: original == documentation
```

```
shortForwardTestJumpFalse
  "
1   <0E>  LoadTrue
2   <1B>  TestJumpFalse   6
5   <14>  LoadSmallInteger1
6   <49>  ReturnSelf
  "
     true ifTrue: [1].
```

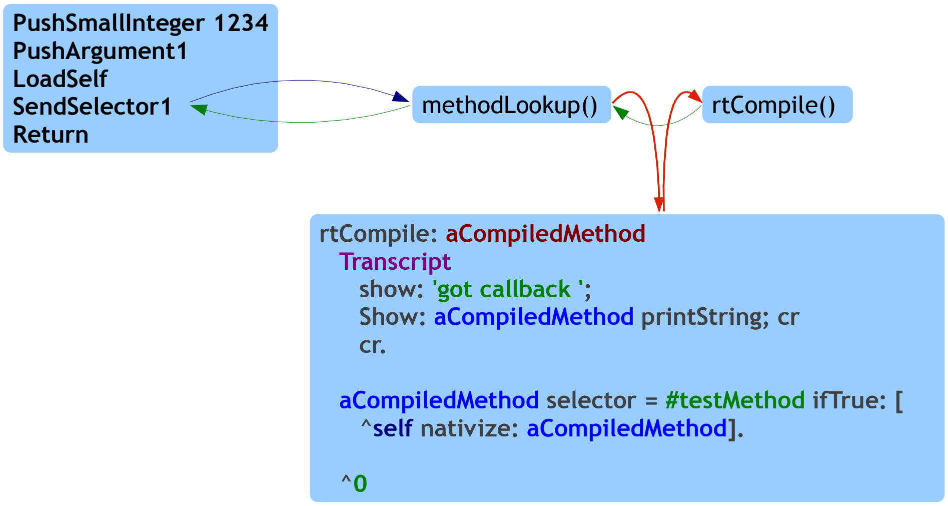
So far so good... but does the generated code work?

Two worlds unite



```
PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return
```

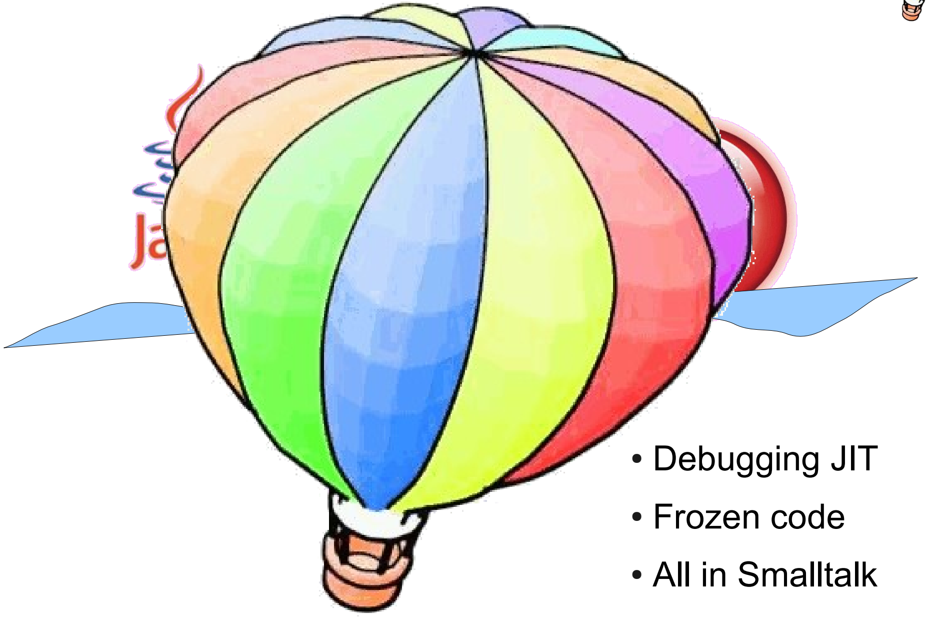
Two worlds unite



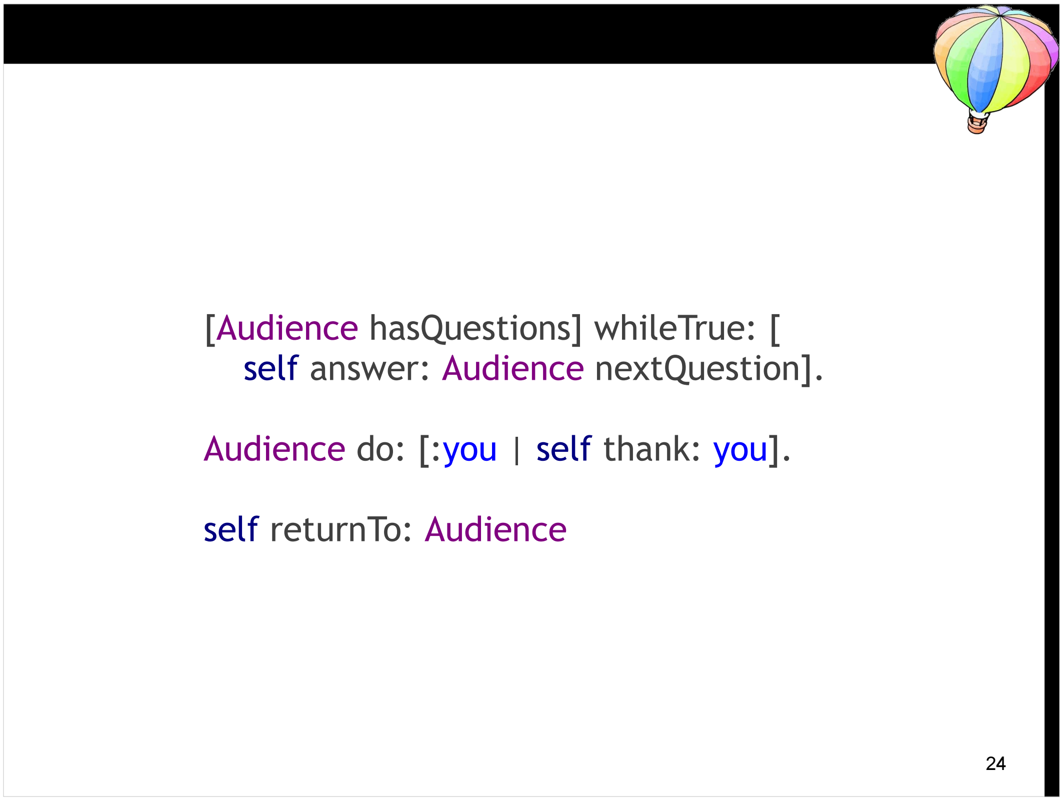
```
PushSmallInteger 1234
PushArgument1
LoadSelf
SendSelector1
Return
```

```
rtCompile: aCompiledMethod
  Transcript
    show: 'got callback ';
    Show: aCompiledMethod printString; cr
    cr.

  aCompiledMethod selector = #testMethod ifTrue: [
    ^self nativize: aCompiledMethod].

  ^0
```

What's next?



- Debugging JIT
- Frozen code
- All in Smalltalk



```
[Audience hasQuestions] whileTrue: [
   self answer: Audience nextQuestion].

Audience do: [:you | self thank: you].

self returnTo: Audience
```