General Purpose Binary Rewriting

Mihai Christodorescu

mihai@cs.wisc.edu

WiSA http://www.cs.wisc.edu/wisa

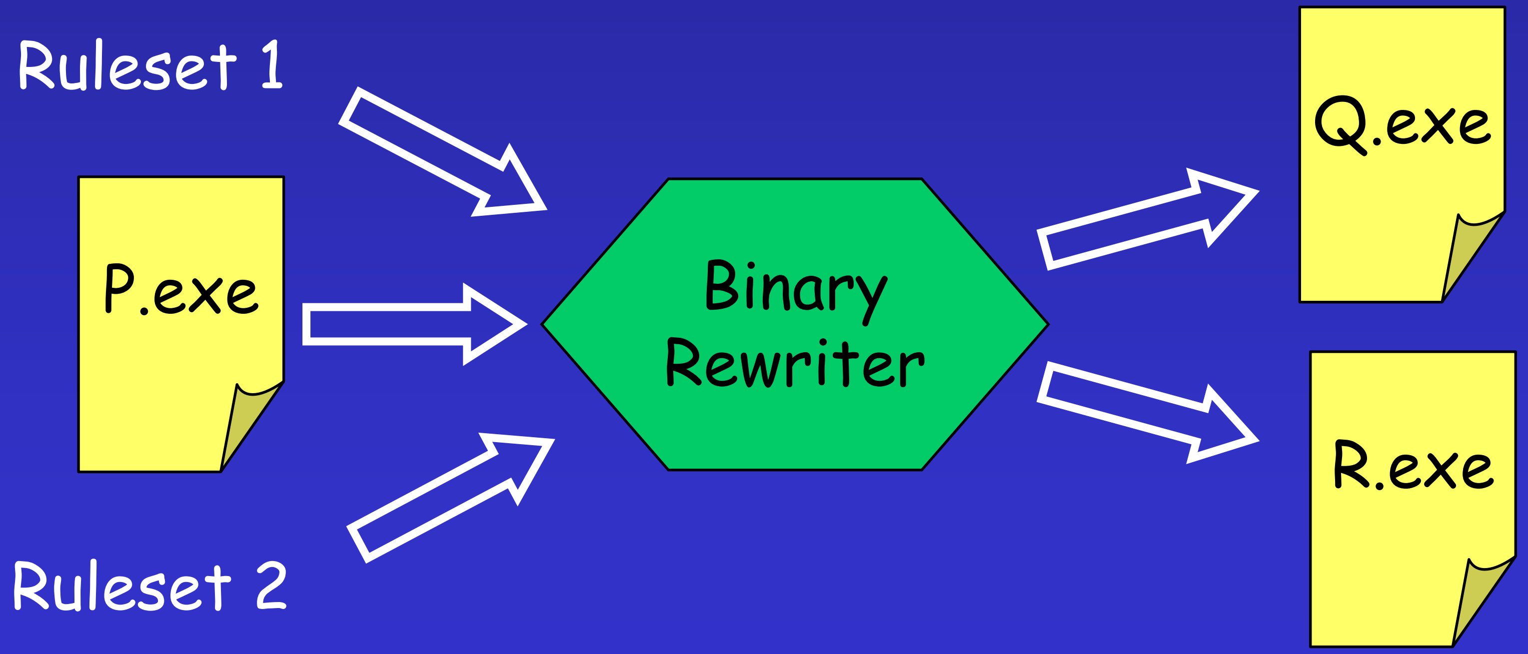
University of Wisconsin, Madison

Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting

Binary Rewriting

- Transform a binary executable based on input rules
  - Add / remove / edit code



Binary Rewriting

- No source code is needed
  - Commercial component software
  - Independent of programming language
    - Treats multi-language systems consistently

- Complete access to the binary
  - Not affected by optimizations

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Binary Rewriting

- Works regardless of program design
  - Similar to AOP
  - Apply transforms across the program

- Extremely powerful

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Binary Rewriting Applications

- Historically:
  - Program optimization
    - Based on profiling data
  - Performance instrumentation
  - Virus / worm infection
    - Using obfuscation

Program Obfuscation

- Used by viruses
  - Add code to make detection harder
  - No change in program behavior
- Competition:
  - Antivirus tools try to deobfuscate
  - Virus writers try to obfuscate

Program Obfuscation

- Semantic NOP Insertion
  - Add code fragment that does not modify program behavior
- Variable / Register Renaming
  - Change the name of program variables
- Instruction Reordering
  - Change the order of instructions without modifying program behavior
- Encode Program

Obfuscation Example for x86

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jecxz   SFModMark
        mov     esi, ecx
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        call    edi
        jmp     Loop
```

Morphed Virus Code
(based on Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        nop
        jecxz   SFModMark
        xor     ebx, ebx
        beqz    N1
N1:     mov     esi, ecx
        nop
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        nop
        call    edi
        xor     ebx, ebx
        beqz    N2
N2:     jmp     Loop
```


Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Obfuscation Example for x86

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jecxz   SFModMark
        mov     esi, ecx
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        call    edi
        jmp     Loop
```

Morphed Virus Code
(based on Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        nop
        jecxz   SFModMark
        xor     ebx, ebx
        beqz    N1
N1:     mov     esi, ecx
        nop
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        nop
        call    edi
        xor     ebx, ebx
        beqz    N2
N2:     jmp     Loop
```

Obfuscation Example for x86

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jecxz   SFModMark
        mov     esi, ecx
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        call    edi
        jmp     Loop
```

Morphed Virus Code
(based on Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jmp     L1
L4:     xor     ebx, ebx
        beqz    N2
N2:     jmp     Loop
L1:     nop
        jecxz   SFModMark
        xor     ebx, ebx
        jmp     L2
L3:     pop     ecx
        nop
        call    edi
        jmp     L4
L2:     beqz    N1
N1:     mov     esi, ecx
        nop
        mov     eax, 0d601h
        pop     edx
        jmp     L3
```


Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Obfuscation Example for x86

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jecxz   SFModMark
        mov     esi, ecx
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        call    edi
        jmp     Loop
```

Morphed Virus Code
(based on Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jmp     L1
L4:     xor     ebx, ebx
        beqz    N2
N2:     jmp     Loop
L1:     nop
        jecxz   SFModMark
        xor     ebx, ebx
        jmp     L2
L3:     pop     ecx
        nop
        call    edi
        jmp     L4
L2:     beqz    N1
N1:     mov     esi, ecx
        nop
        mov     eax, 0d601h
        pop     edx
        jmp     L3
```


Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Obfuscation Example for x86

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jecxz   SFModMark
        mov     esi, ecx
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        call    edi
        jmp     Loop
```

Morphed Virus Code
(based on Chernobyl CIH 1.4):

```
Loop:
        pop     eax
        jmp     L1
L4:     xor     ebx, ebx
        beqz    N2
N2:     jmp     Loop
L1:     nop
        jecxz   SFModMark
        xor     ebx, ebx
        jmp     L2
L3:     pop     eax
        nop
        call    edi
        jmp     L4
L2:     beqz    N1
N1:     mov     esi, eax
        nop
        mov     ecx, 0d601h
        pop     edx
        jmp     L3
```


Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Obfuscation Example for x86

Virus Code
(from Chernobyl CIH 1.4):

```
Loop:
        pop     ecx
        jecxz   SFModMark
        mov     esi, ecx
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        call    edi
        jmp     Loop
```

Morphed Virus Code
(based on Chernobyl CIH 1.4):

```
Loop:
        pop     eax
        jmp     L1
L4:     xor     ebx, ebx
        beqz    N2
N2:     jmp     Loop
L1:     nop
        jecxz   SFModMark
        xor     ebx, ebx
        jmp     L2
L3:     pop     eax
        nop
        call    edi
        jmp     L4
L2:     beqz    N1
N1:     mov     esi, eax
        nop
        mov     ecx, 0d601h
        pop     edx
        jmp     L3
```

Obfuscation Example for VB

Worm Code
(based on AnnaKournikova worm):

```
On Error Resume Next
...
Set outlookObj = CreateObject("Outlook.Application")
...
For Each addressObj In addressBookObj
        ...
        newMsgObj.Send
        ...
Next
...
'Vbswg 1.50b
```

Obfuscation Example for VB

Worm Code
(based on AnnaKournikova worm):

```
Execute "On Error Resume Next\n… \nSet outlookObj
    = CreateObject("Outlook.Application")\n… \n
    For Each addressObj In addressBookObj\n…
    \nnewMsgObj.Send\n… \nNext\n… \n'Vbswg
    1.50b"
```

Obfuscation Example for VB

Worm Code
(based on AnnaKournikova worm):

```
Execute F( "X)udQ0Vpgjn... udiy3^Q70d2" )

Function F(S)
        For I = 1 To Len(S) Step 2
            X= Mid(S, I, 1)
            ...
            If Asc(X) = 15 Then
                        ...
            End If
            ...
        Next
End Function
```

Program Obfuscation

- Obfuscation is a technique used by virus writers

=> Virus detection tools have to handle obfuscations

- Then, to test our virus detection tool, we need to rewrite infected binaries to add obfuscations

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Obfuscating…

Commercial antivirus tools vs. morphed versions of known viruses

	Norton AntiVirus	Command Software Systems	McAfee VirusScan
Chernobyl-1.4	× Not detected	× Not detected	× Not detected
f0sf0r0	× Not detected	× Not detected	× Not detected
Hare	× Not detected	× Not detected	× Not detected
z0mbie-6.b	× Not detected	× Not detected	× Not detected

Program Obfuscation

- The obfuscation war is not lost
  - We can effectively deobfuscate many idioms
  - More later

- Benefit of obfuscation:
  - IP protection (harder to reverse engineer)

Bounds Checking

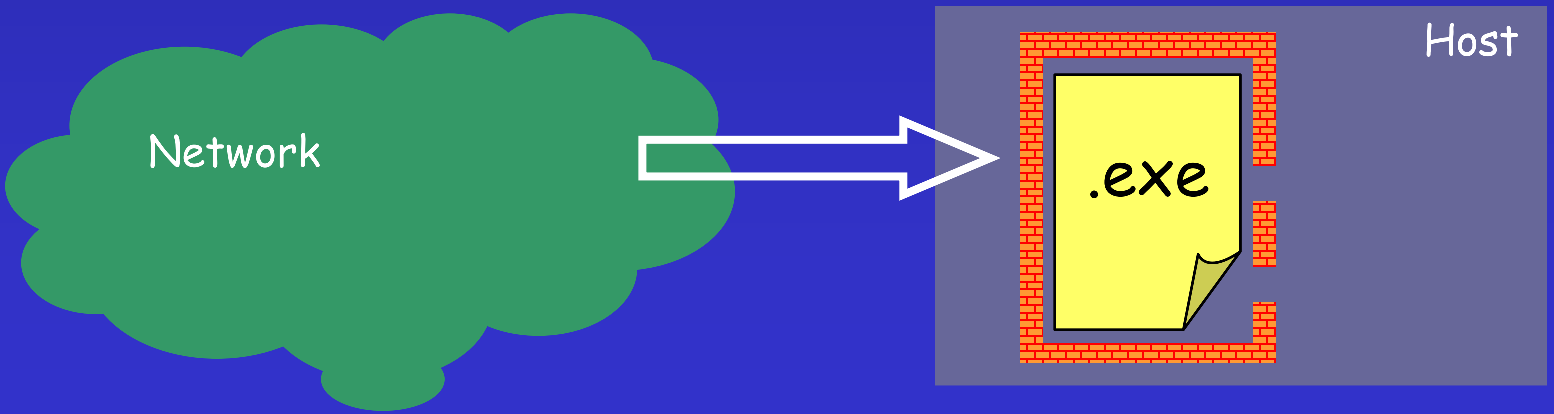
- C, C++ do not perform bounds checking on array accesses
  - Possible buffer overflow due to programming errors
- Patch all string buffer and array accesses to check for length

=> No more buffer overflows

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Sandboxing

- Restrict untrusted program's access to OS interface:
  - Contain disk access & memory usage
  - Allow / deny network connections
  - No OS modification



Security Policy Enforcement

- Similar to Engler's metacompilation work

- Enforce rules not supported by the standard OS security mechanisms:
  - Sanitize untrusted input
  - Do not release sensitive data to users
  - Check custom permissions before doing operation X

Security Policy Enforcement

- Powerful to check at runtime:
  - Programming errors that can lead to security flaws
  - Security violations

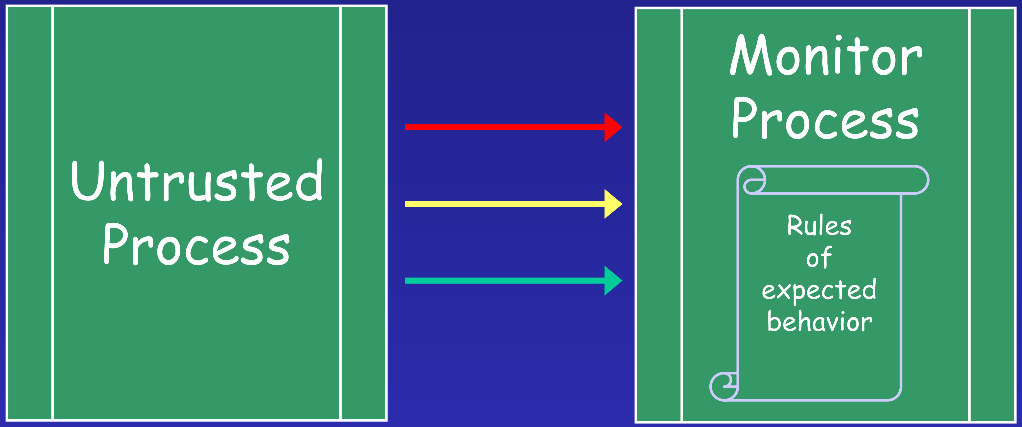
- Technique:
  - Add code that at runtime enforces desired rules

Program Monitoring

- Monitor a running program to prevent malicious modification
- A *monitor* process will:
  - Trace the events produced by a running program
  - Make sure the events match what is expected

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Program Monitoring



- Monitoring can be remote or local
- Flexible policy rules

Program Monitoring

- Problem:

Certain event sequences are ambiguous

- Solution:

Modify program to eliminate ambiguity (as much as possible)

  - Insert code to send special events
  - See Jon Giffin's work on IDS

Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting

Existing Binary Rewriters

- Limited in functionality or scope
- Some still in prototype mode

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Existing Binary Rewriters

- EEL (Executable Editing Library) (U. Wisconsin)
  - Works only on SPARC binaries

- Etch Binary Rewriter (U. Washington)
  - Works only on x86 Windows
  - Not available as a separate library

Existing Binary Rewriters

- Byte Code Engineering Library
  (Open source, Apache Foundation)
  - Java specific

- OM / ATOM
  (DEC WRL/Compaq WRL/HP?)
  - proprietary

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Existing Binary Rewriters

- DynInst
  (U. Wisconsin, U. Maryland)
  - Geared towards instrumenting running programs
  + Can handle multiple types of binaries

The Problem

- WiSA project relies on several tools:
  - EEL, IDA Pro, CodeSurfer + custom code
- Incomplete solution:
  - Platform specific (not cross platform)
  - Missing features, yet not easily extensible

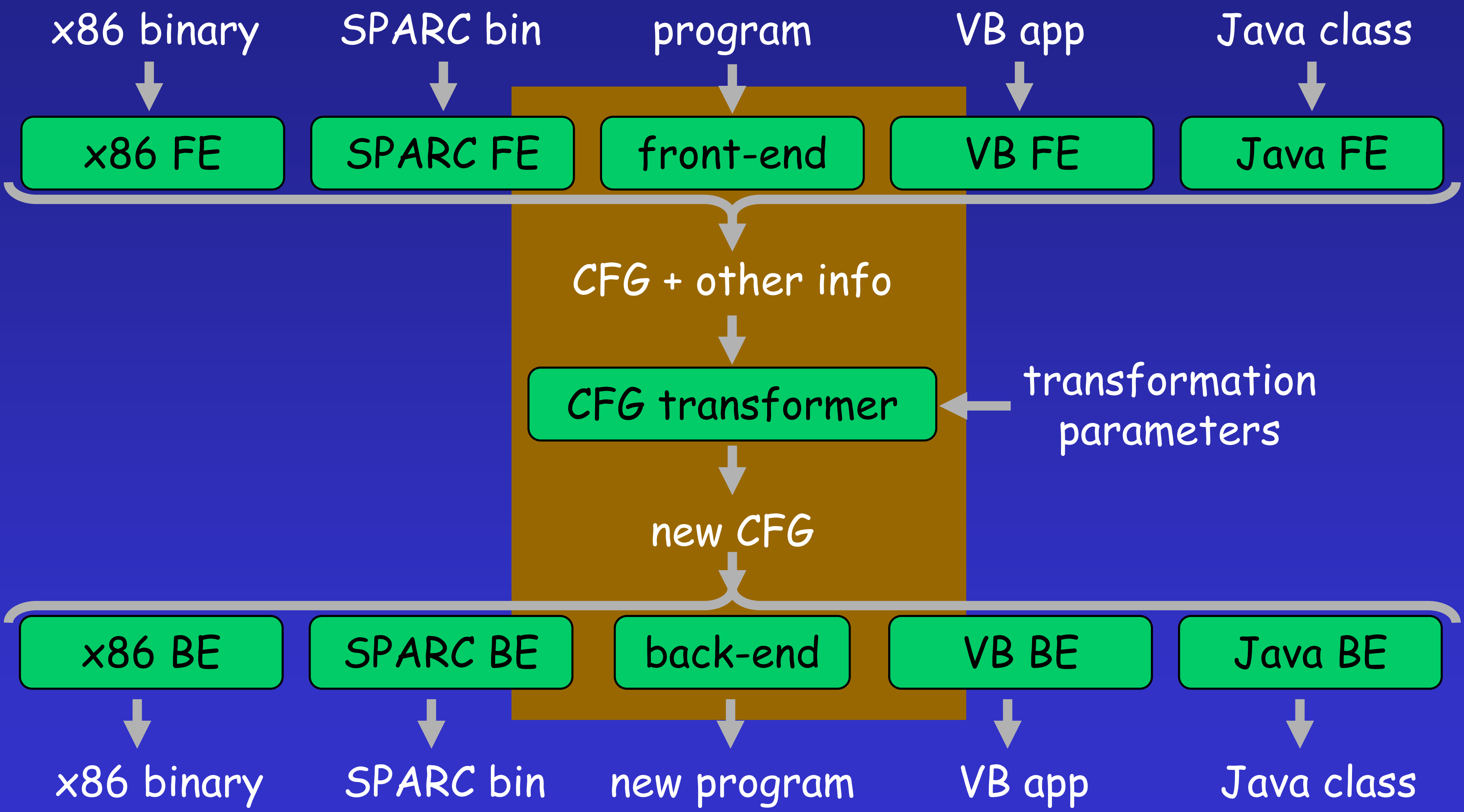
Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting

Goal

- General purpose binary rewriter
  - Cross platform
    - Windows, Linux, Solaris, ...
  - Multiple architectures
    - IA-32 (x86), IA-64, SPARC, ...
  - Extensible
  - Flexible
  - Useful

Binary Rewriter Architecture



Seems easy enough...

- Different architectures
- Different execution environments
- Different languages

=> Bringing all of them into one data structure is a challenge!

Seems easy…

- Each architecture has features non-existent elsewhere
  - SPARC has *register windows*
  - IA-32 (x86) has a *hardware stack*
  - …

- Did we mention it has to be flexible, customizable, and extensible?

Binary Rewriter Design

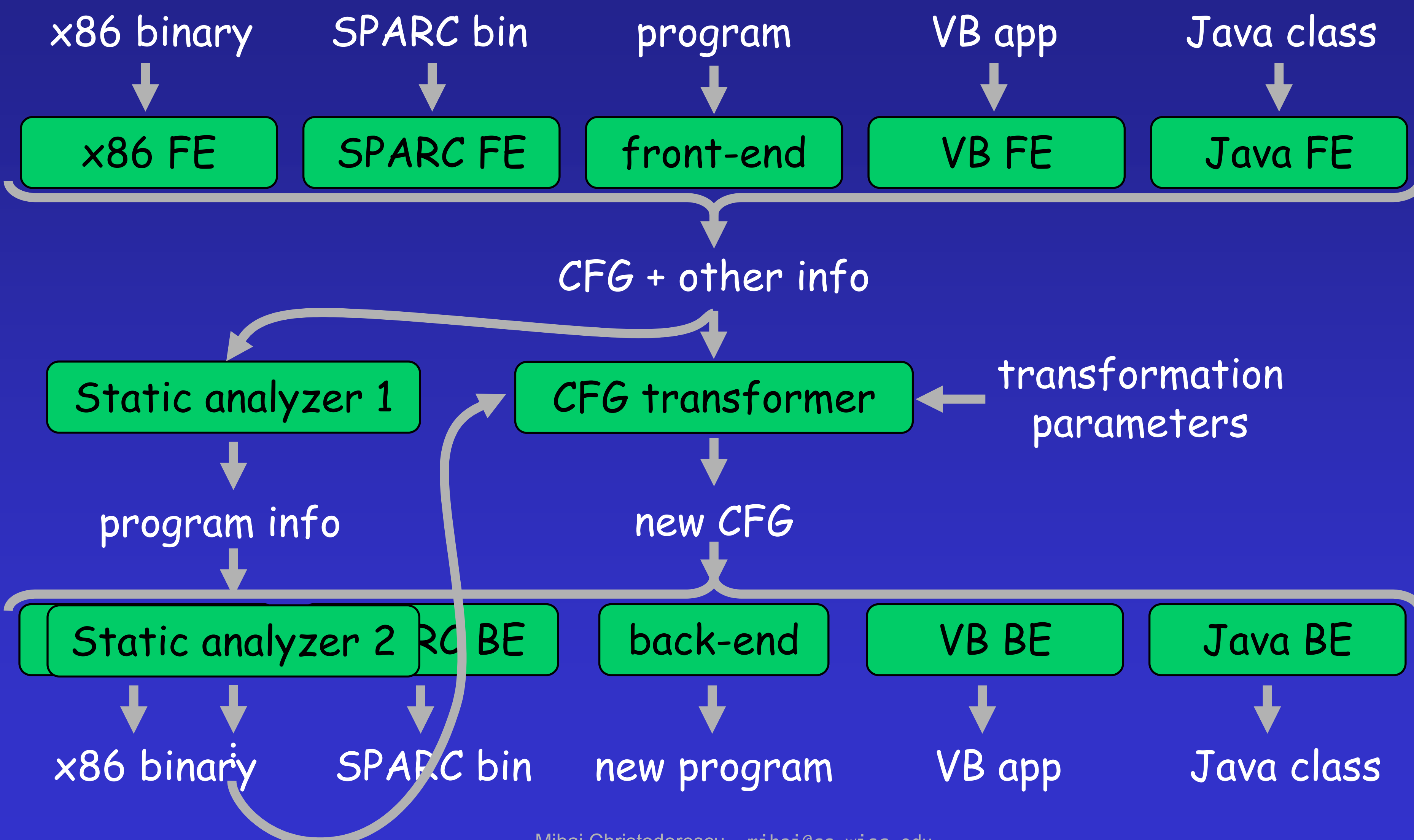
- Binary rewriter interface — multiple levels of abstraction:
  - Hide architecture-specific differences when needed
  - Provide low level details when necessary

Binary Rewriter

- A project worth undertaking:
  - The benefits are extraordinary

- The infrastructure can be reused…

Binary Rewriter Architecture



Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. *Architecture for rewriting*
5. Implementation status
6. Future work
7. Protection against malicious rewriting

Status

- Early stage
  - Gathering requirements
    - From WiSA subprojects
    - From external sources
  - Assessing tools used by WiSA
    - Some tools to be integrated in the new infrastructure
    - Some tools have good interfaces

Status

- Current tools
  - IDA Pro
    - Supports multiple architectures
    - Can act as a front-end only
    - Front-end for x86 – good progress
  - CodeSurfer
    - Multiple, complex static analyses
  - EEL
    - Good interface
    - Support for SPARC rewriting

Status

- CFG transformer
  - Some transformations have ad-hoc implementations
  - Specification and design in progress

=> The work done up to now can be integrated and reused

Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting

Steps Forward

1. Architecture and design
   - Documented & reviewed
2. Define interfaces
   - Based on existing tools
   - Focus on integration with existing tools
3. Prototype implemented
   - Support several key architectures
   - Test and get feedback on interface design issues

More Steps Forward

4. Review design
   - Based on internal feedback
5. Implementation
   - Complete front-ends and back-ends
   - Several transformations ported to the new infrastructure
   - Static analyses ported to the new infrastructure
6. Release

Future Work

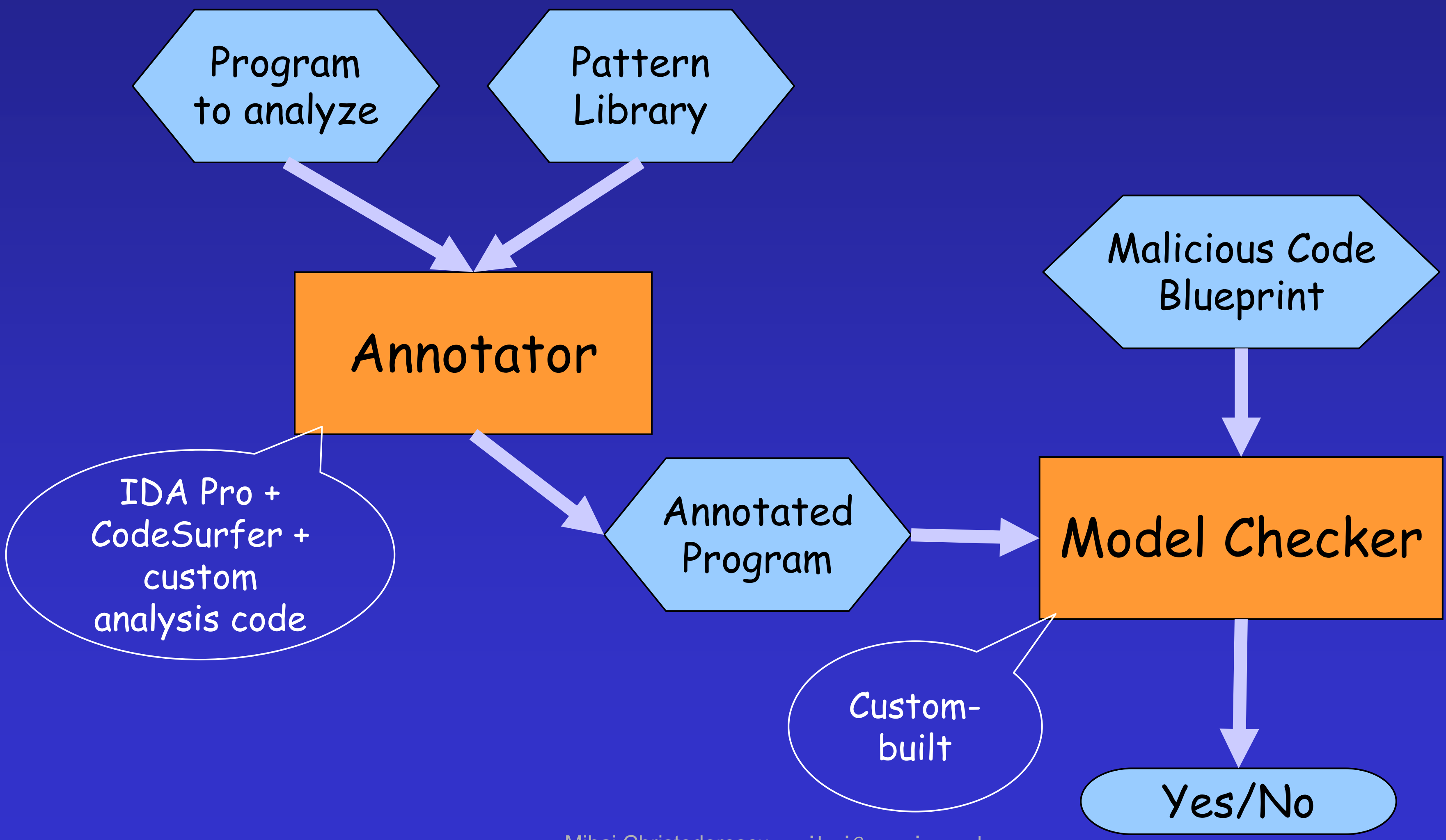
- Support many static analyses
  - Incremental precision gains
  - Enhance infrastructure info
- Add transformations of increasing complexity
- Add more architectures / languages

Overview

1. Introduction to binary rewriting
2. Benefits and applications of binary rewriting
3. Existing tools
4. Architecture for rewriting
5. Implementation status
6. Future work
7. Protection against malicious rewriting

Seeing Through the Obfuscations

Smart Virus Scanner



Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

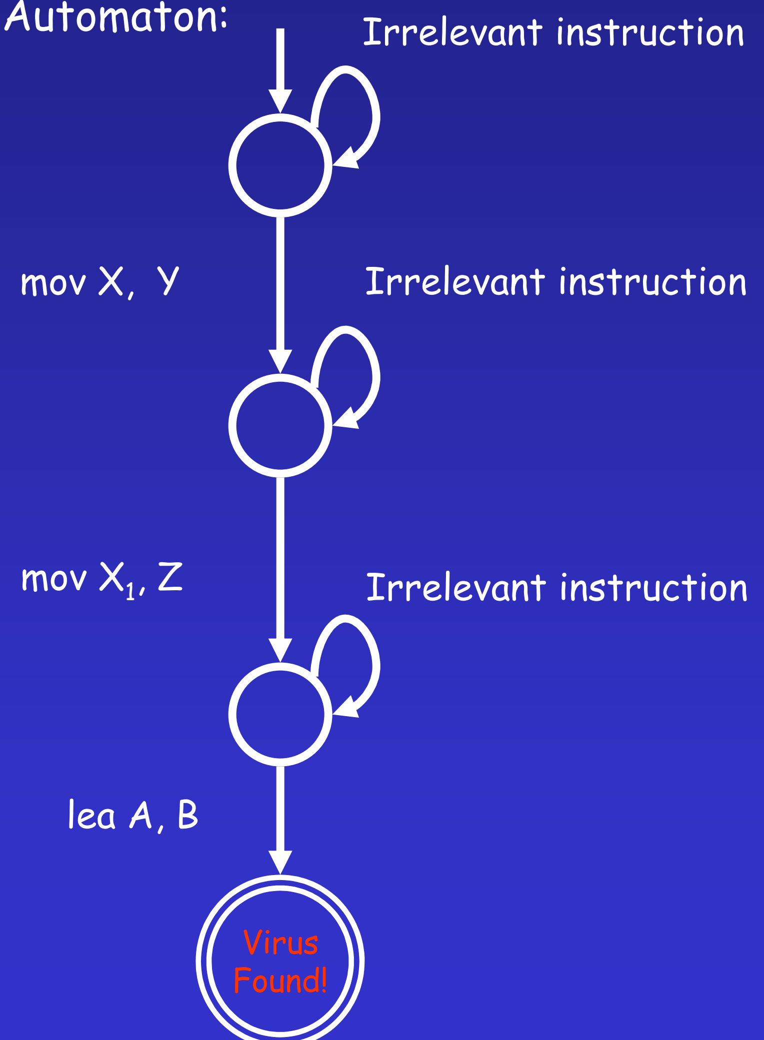
Detection Example

Virus Code:

```
push    eax
sidt    [esp-02h]
pop     ebx
add     ebx, HookNo * 08h + 04h
cli
mov     ebp, [ebx]
mov     bp, [ebx-04h]
lea     esi, MyHook - @1[ecx]
push    esi
mov     [ebx-04h], si
shr     esi, 16
mov     [ebx+02h], si
pop     esi
```

(from Chernobyl CIH 1.4 virus)

Virus Automaton:



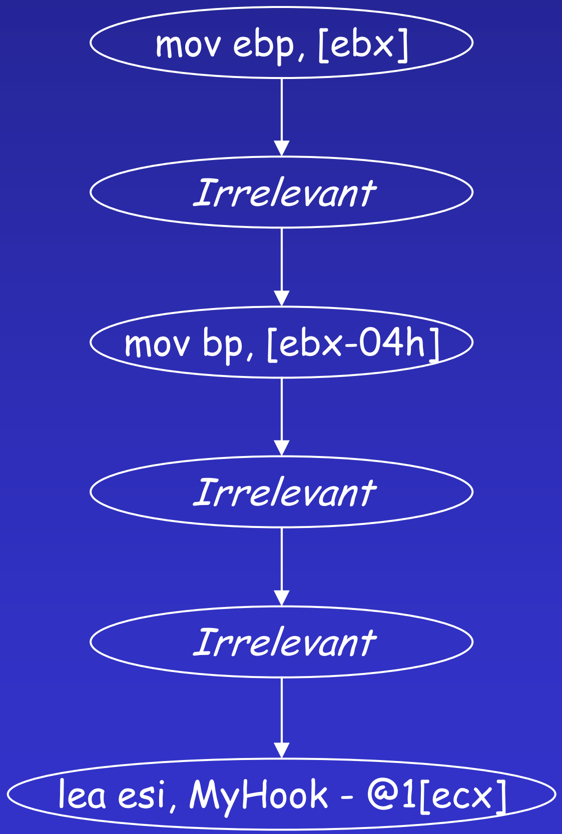
Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Detection Example

Annotated program:

Program to be checked:

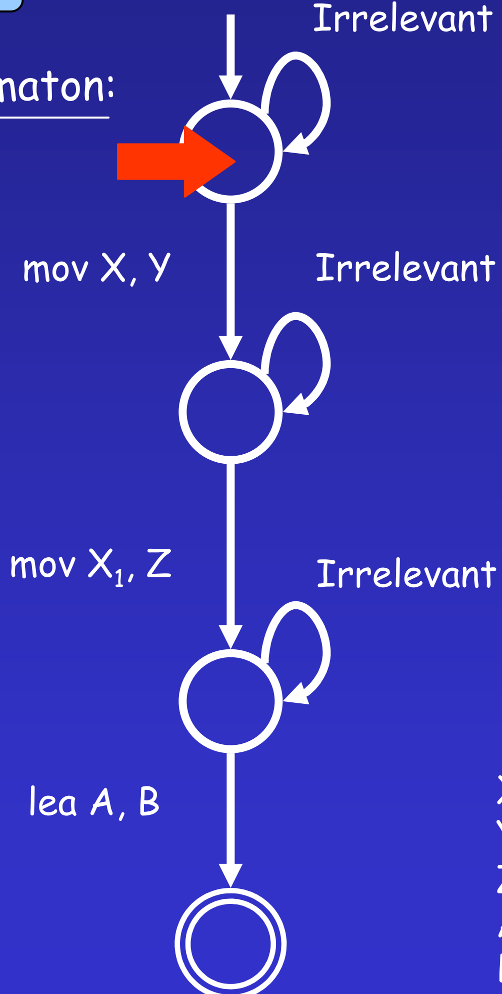
```
            mov ebp, [ebx]
            nop
            mov bp, [ebx-04h]
            test ebx
            beqz next
next:       lea esi, MyHook - @1[ecx]
```



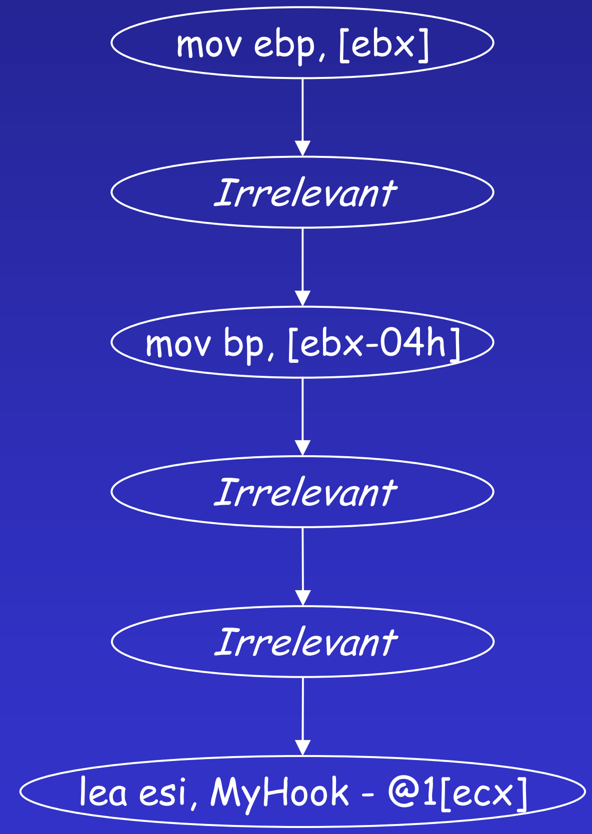
Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Detection Example

Virus Automaton:



Program model (annotated program):



X = ebp
Y = [ebx]
Z = [ebx – 04h]
A = esi
B = MyHook - @1[ecx]

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Smart Virus Scanner

- What are *irrelevant instructions*?
  - NOPs
  - Control flow instructions that do not change the control flow
    - e.g.: jumps/branches to the next instructions
  - Instructions that modify dead registers
  - Sequences of instructions that do not modify architectural state
    - e.g.:

```
add ebx, 1
sub ebx, 1
```


Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Uninterpreted Symbols

- What happens when the registers are changed?

Program 1:

```
        mov ebp, [ebx]
        nop
        mov bp, [ebx-04h]
        test ebx
        beqz next
next:   lea esi, MyHook - @1[ecx]
```

Program 2:

```
        mov eax, [ecx]
        nop
        mov ax, [ecx-04h]
        test edx
        beqz next
next:   lea ebi, MyHook - @1[ebx]
```

Virus Spec:

```
        mov ebp, [ebx]
```

=> No match with Program 2

Virus Spec with *Uninterpreted Symbols*:

```
        mov X, Y
```

=> Matches both Programs 1 and 2

Mihai Christodorescu mihai@cs.wisc.edu
WiSA Project http://www.cs.wisc.edu/wisa

Program Obfuscations

- Semantic NOPs
- Instruction Reordering
- Variable Renaming
  - Handled through static analysis

- Encoded Program Fragment
  - Partial evaluation

General Purpose Binary Rewriting

Mihai Christodorescu
mihai@cs.wisc.edu

WiSA http://www.cs.wisc.edu/wisa
University of Wisconsin, Madison