Static Analysis of Executables to Detect Malicious Patterns

[12th USENIX Security Symposium, 2003]

Mihai Christodorescu

Somesh Jha

CS @ University of Wisconsin, Madison

Presented by K. Vikram
Cornell University

Problem & Motivation…

- Malicious code is … malicious
- Categorize: Propagation Method & Goal
  - Viruses, worms, trojan horses, spyware, etc.
- Detect Malicious Code
  - In executables

The Classical Stuff

- Focus mostly on Viruses
  - Code to replicate itself + Malicious payload
  - Inserted into executables
- Look for *signatures*
- Not always enough
- Obfuscation-Deobfuscation Game

Common Obfuscation Techniques

- Encryption
- Dead Code insertion*
- Code transposition*
- Instruction Substitution*
- Register reassignment*
- Code Integration
- Entry Point Obscuring

Common Deobfuscation Techniques

- Regular Expressions
- Heuristic Analyses
- Emulation

Mostly Syntactic…

The Game

- Vanilla Virus
- Register Renaming
- Packing/Encryption
- Code Reordering
- Code Integration

- Signatures
- Regex Signatures
- Emulation/Heuristics
- ?
- ?

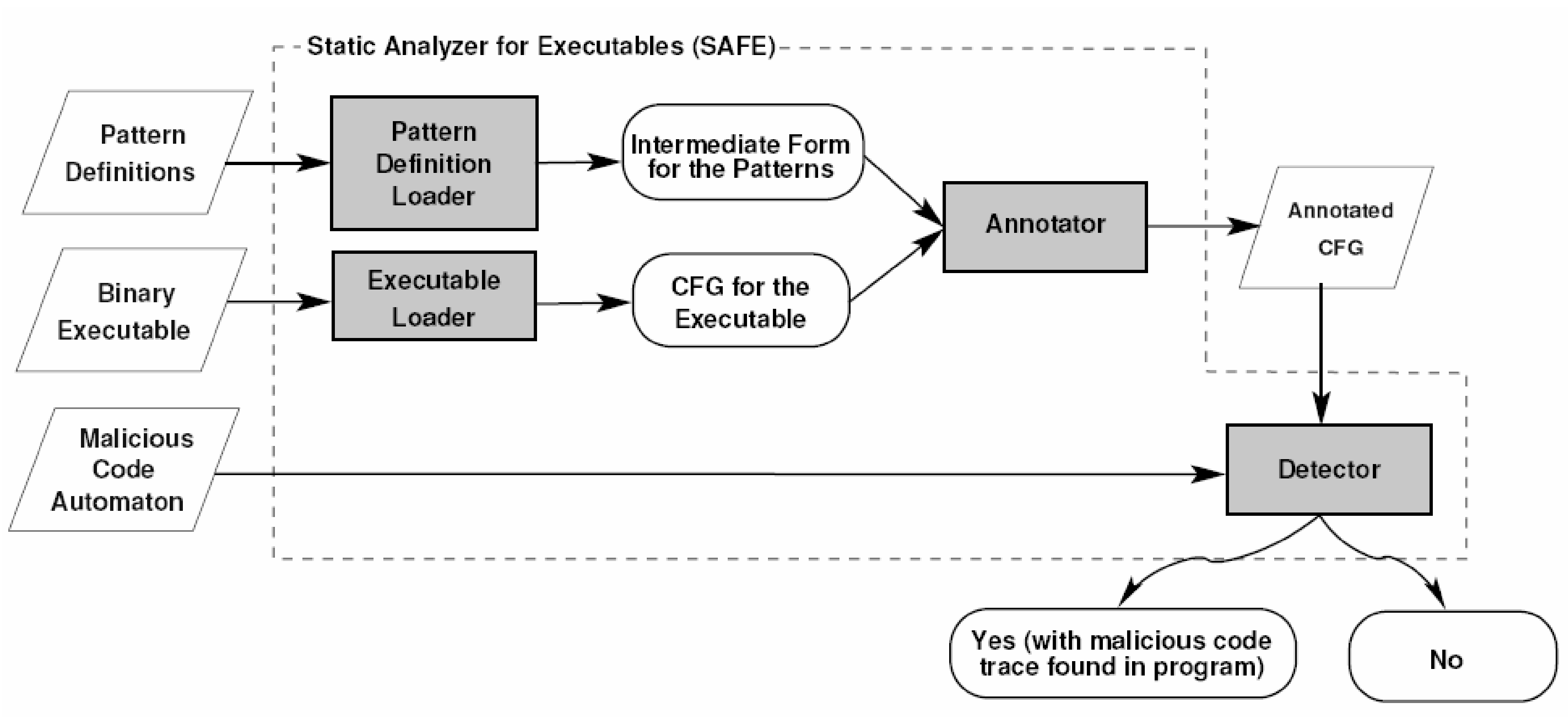
Current Technology

- Antivirus Software
  - Norton, McAfee, Command
- Brittle
  - Cannot detect simple obfuscations
  - nop-insertion, code transposition
- Chernobyl, z0mbie-6.b, f0sf0r0, Hare

Theoretical Limits

- Virus Detection is undecidable
- Some Static Analyses are undecidable
- But, Obfuscation is also hard

The SAFE* Methodology



Procedure

- Key Ideas:
  - Analyze program’s semantic structure
  - Use existing static analyses (extensible)
  - Use uninterpreted symbols
- Abstract Representation of Malicious Code
- Abstract Representation of Executable
  - Deobfuscation
- Detect presence of malicious code

The Annotator

- Inputs:
  - CFG of the executable
  - Library of Abstraction Patterns
- Outputs:
  - Annotated CFG

Some groundwork

- Instruction I : $\tau_1 \times \ldots \times \tau_k \rightarrow \tau$
- Program P : $\langle I_1, \ldots, I_N \rangle$
- Program counter/point
  - pc : $\{ I_1, \ldots, I_N \} \rightarrow [1,\ldots,N]$
  - $pc(I_j) = j, \forall\, 1 \leq j \leq N$
- Basic Block, Control Flow Graph*
- Static Analysis Predicates
- Types for data and instructions

Example Predicates

$Dominators(B)$	the set of basic blocks that dominate the basic block B
$PostDominators(B)$	the set of basic blocks that are dominated by the basic block B
$Pred(B)$	the set of basic blocks that immediately precede B
$Succ(B)$	the set of basic blocks that immediately follow B
$First(B)$	the first instruction of the basic block B
$Last(B)$	the last instruction of the basic block B
$Previous(I)$	$\begin{cases} \bigcup_{B' \in Pred(B_I)} Last(B') & \text{if } I = First(B_I) \\ I' & \text{if } B_I = \langle \ldots, I', I, \ldots \rangle \end{cases}$
$Next(I)$	$\begin{cases} \bigcup_{B' \in Succ(B_I)} First(B') & \text{if } I = Last(B_I) \\ I' & \text{if } B_I = \langle \ldots, I, I', \ldots \rangle \end{cases}$
$Kills(p, a)$	*true* if the instruction at program point p kills variable a
$Uses(p, a)$	*true* if the instruction at program point p uses variable a
$Alias(p, x, y)$	*true* if variable x is an alias for y at program point p
$LiveRangeStart(p, a)$	the set of program points that start the a's live range that includes p
$LiveRangeEnd(p, a)$	the set of program points that end the a's live range that includes p
$Delta(p, m, n)$	the difference between integer variables m and n at program point p
$Delta(m, p_1, p_2)$	the change in m's value between program points p_1 and p_2
$PointsTo(p, x, a)$	*true* if variable x points to location of a at program point p

Abstraction Patterns

- Abstraction pattern Γ : (V,O,C)
  - $V = \{ x_1 : \tau_1, \ldots, x_k : \tau_k \}$
  - $O = \langle I(v_1, \ldots, v_m) \mid I : \tau_1 \times \ldots \times \tau_m \rightarrow \tau \rangle$
  - C = boolean expression involving static analysis predicates and logical operators
- Represents a deobfuscation
- Predicate controls pattern application
- *Unify* patterns with sequence of instructions

Example of a pattern

$$
\begin{aligned}
\Gamma(\ X : int(0:1:31)\ ) = \\
(\quad & \{\ X : int(0:1:31)\ \}, \\
& \langle\ p_1 :\ \text{``}pop\ X\text{''}, \\
& \quad p_2 :\ \text{``}add\ X, \texttt{03AFh}\text{''}\ \rangle, \\
& p_1 \in LiveRangeStart(p_2, X)\ \ )
\end{aligned}
$$

Defeating Garbage Insertion

<instruction A>
<instruction B>

→

<instruction A>
add ebx, 1
sub ebx, 1
nop
<instruction B>

Pattern:

instr 1
…
instr N
Where
Delta(state pre 1, state post N) = 0

Defeating Code-reordering

Pattern:

jmp TARGET
where
Count (CFGPredecessors(TARGET)) = 1

The Annotator

- Given set of patterns $\Sigma = \{ \Gamma_1, \ldots, \Gamma_m \}$
- Given a node n for program point p
- Matches each pattern in Σ with
  $\langle \ldots, \mathit{Previous}^2(\mathrm{I}_p), \mathit{Previous}(\mathrm{I}_p), \mathrm{I}_p \rangle$
- Associates all patterns that match with n
- Also stores the bindings from unification

The Detector

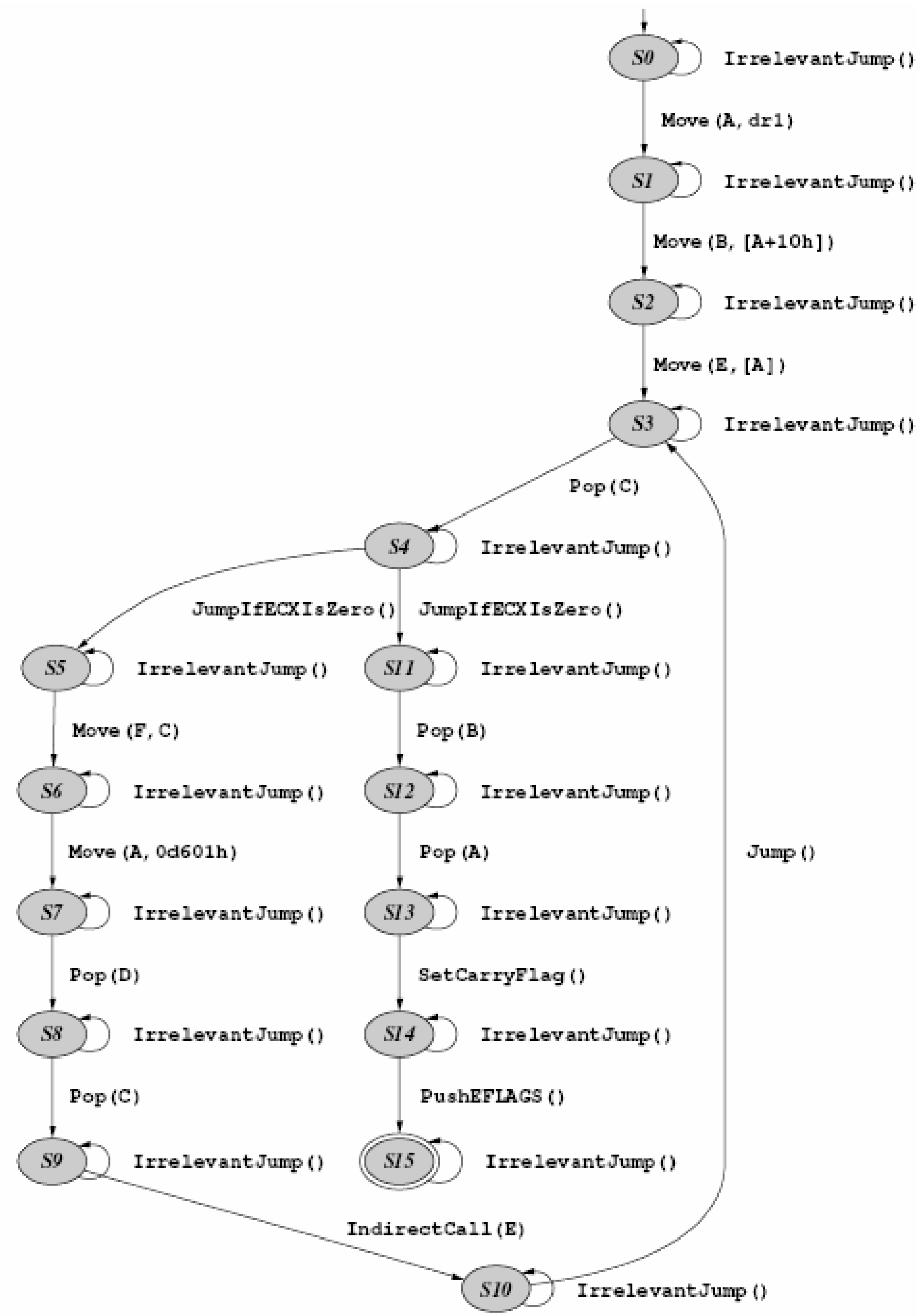
- Inputs:
  - Annotated CFG for a procedure
  - Malicious code *representation*
- Output:
  - Sequence of instructions exhibiting the malicious pattern

Malicious Code Automaton

- Abstraction of the vanilla virus
- 6-tuple $(V,\Sigma,S,\delta,S_0,F)$
  - $V = \{ v_1{:}\tau_1, \ldots, v_k{:}\tau_k \}$
  - $\Sigma = \{ \Gamma_1, \ldots, \Gamma_n \}$
  - S = finite set of states
  - $\delta : S \times \Sigma \to 2^S$ is a transition function
  - $S_0 \subseteq S$ is a non-empty set of *initial* states
  - $F \subseteq S$ is a non-empty set of *final* states

Malicious Code

```
WVCTF:
        mov     eax, dr1
        mov     ebx, [eax+10h]
        mov     edi, [eax]
LOWVCTF:
        pop     ecx
        jecxz   SFMM
        mov     esi, ecx
        mov     eax, 0d601h
        pop     edx
        pop     ecx
        call    edi
        jmp     LOWVCTF
SFMM:
        pop     ebx
        pop     eax
        stc
        pushf
```


S0
IrrelevantJump()
Move(A, dr1)
S1
IrrelevantJump()
Move(B, [A+10h])
S2
IrrelevantJump()
Move(E, [A])
S3
IrrelevantJump()
Pop(C)
S4
IrrelevantJump()
JumpIfECXIsZero()
JumpIfECXIsZero()
S5
IrrelevantJump()
Move(F, C)
S6
IrrelevantJump()
Move(A, 0d601h)
S7
IrrelevantJump()
Pop(D)
S8
IrrelevantJump()
Pop(C)
S9
IrrelevantJump()
IndirectCall(E)
S10
IrrelevantJump()
Jump()
S11
IrrelevantJump()
Pop(B)
S12
IrrelevantJump()
Pop(A)
S13
IrrelevantJump()
SetCarryFlag()
S14
IrrelevantJump()
PushEFLAGS()
S15
IrrelevantJump()

Detector Operation

- Inputs:
  - CFG P_Σ
  - $\mathcal{A} = (V,\Sigma,S,\delta,S_0,F)$
- Determines whether the same (malicious) pattern occurs both in $\mathcal{A}$ and Σ
- More formally, tests the emptiness of

$$L(P_\Sigma) \cap (\cup_{\mathcal{B} \in \mathcal{B}_{All}} L(\mathcal{B}(\mathcal{A})))$$

Detector Algorithm

- Dataflow-like Algorithm
- Maintain a *pre* and *post* list at each node of the CFG P_{Σ}
- List is of $[s, \mathcal{B}_s]$, s is a state in $\mathcal{A}$
- Join operation is union

Detector Algorithm

- Transfer Function:

$$
\begin{array}{l}
\textbf{foreach } [s, \mathcal{B}_s] \in L_n^{pre} \\
\quad \textbf{foreach } [\Gamma, \mathcal{B}] \in Annotation(n) \\
\quad \quad \quad \wedge\, Compatible(\mathcal{B}_s, \mathcal{B}) \\
\quad \quad \textbf{add } [\ \delta(s, \Gamma),\ \mathcal{B}_s \cup \mathcal{B}\ ] \textbf{ to } NewL_n^{post}
\end{array}
$$

- Return:

$$\exists\, n \in N \,.\, \exists\, [s, \mathcal{B}_s] \in L_n^{post} \,.\, s \in F$$

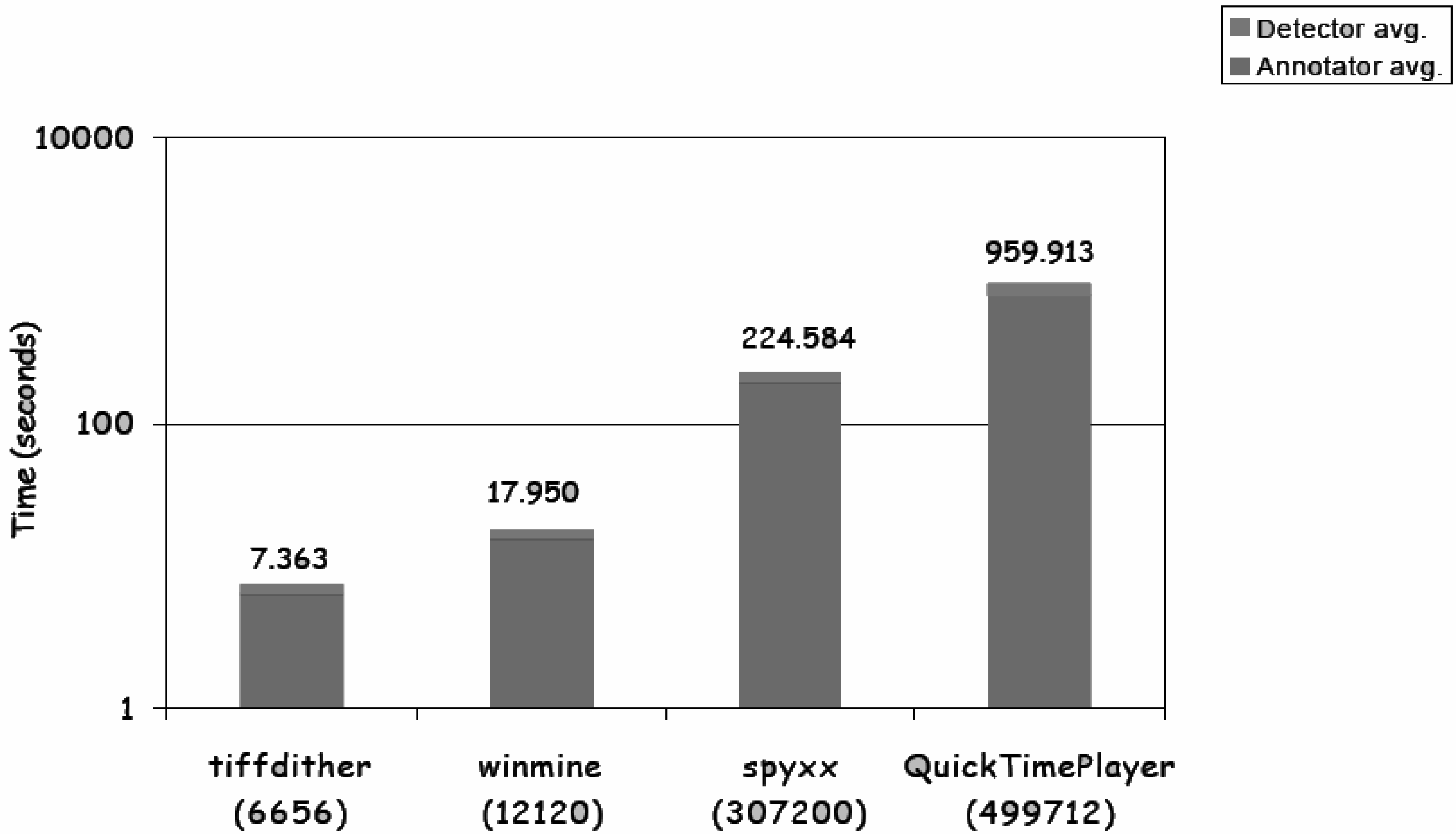
Defenses Against…

- Code Re-ordering
- Register Renaming
- Insertion of irrelevant code
  - nops*, code that modifies dead registers
  - Needs live-range and pointer analyses

Experimental Results

- False Positive Rate : 0
- False Negative Rate : 0
  - not all obfuscations are detected

Performance



Future Directions

- New languages
  - Scripts – VB, JavaScript, ASP
  - Multi-language malicious code
- Attack Diversity
  - worms, trojans too
- Irrelevant sequence detection
  - Theorem provers
- Use TAL/external type annotations

Pitfalls/Criticisms?

- Focus on viruses instead of worms
- Still fairly Ad-hoc
- Treatment of obfuscation is not formal enough
- Intractable techniques
  - Use of theorem provers to find irrelevant code
- Slow
- No downloadable code
- Not enough experimental evaluation