Automated Extraction of Polymorphic Virus Signatures using Abstract Interpretation

Serge Chaumette, Olivier Ly, Renaud Tabary

LaBRI/CNRS/Région Aquitaine

NSS 2011 - Sept. 6-8, Milan

Outline

1. Context
   - Definition
   - Syntactic models
   - Structural and semantic models
   - Hybrid model
2. The mutation semantics
3. Implementation & preliminary results
4. Conclusion

Virus and language theory

Let $(M, V) \in \mathcal{TM} \times \wp(\Sigma^*)$ be a virus [Coh87]. In practice, M is known (processor, virtual machine, etc.). We can see V as the *language* of the virus.

Classification of viruses

A virus $(M, V) \in \mathcal{TM} \times \wp(\Sigma^*)$ can be:

- a non-mutating virus if $|V| = 1$ (singleton virus) or $|V| \in \mathbb{N}$ (oligomorphic virus);
- a mutating virus if $|V|$ is infinite (polymorphic and metamorphic viruses)
- mutating viruses make use of code obfuscation techniques [CTL97]

The problem of detection of a mutating virus is the recognition of its language V.

Polymorphic virus

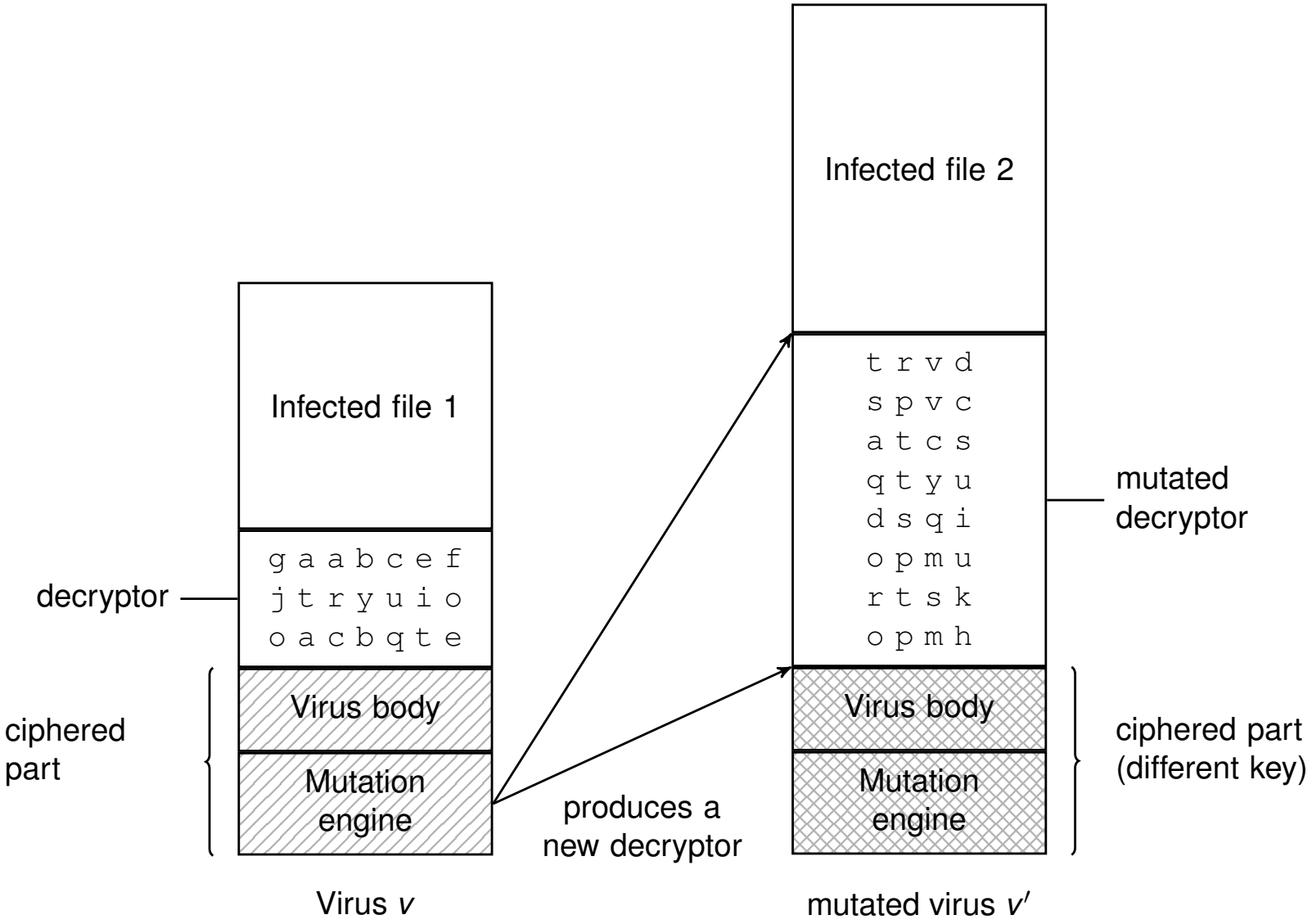


Figure: Spreading of a polymorphic virus

Schemes used in current antivirus

- **Model** of programs: sequence of bits
- **Signature**: regular expression
- **Detection**: word in program recognized by signature
- **Extraction** of the signature: **learning**-based
  - ... longest common pattern + **manual** analysis [KA94, SJHWG+03, SXCM04, NKS05]

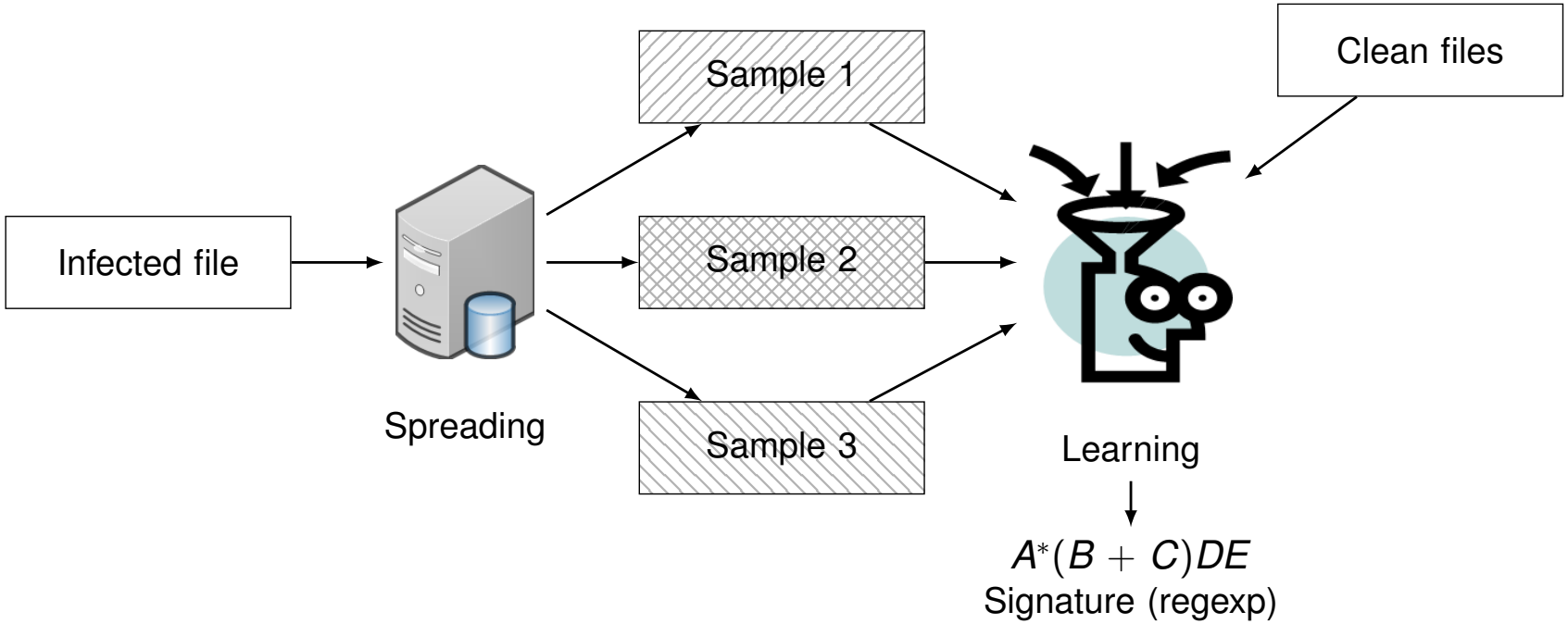


Figure: Signature extraction for industrial antivirus

Problematics of **static** detection

First threat : increasing number of new unique malwares/day

- $\simeq 10^4$ signatures in 2001 to $2.9 * 10^6$ signatures in 2009 [Sve11];
- $\simeq$ 8000 new signatures written each day.

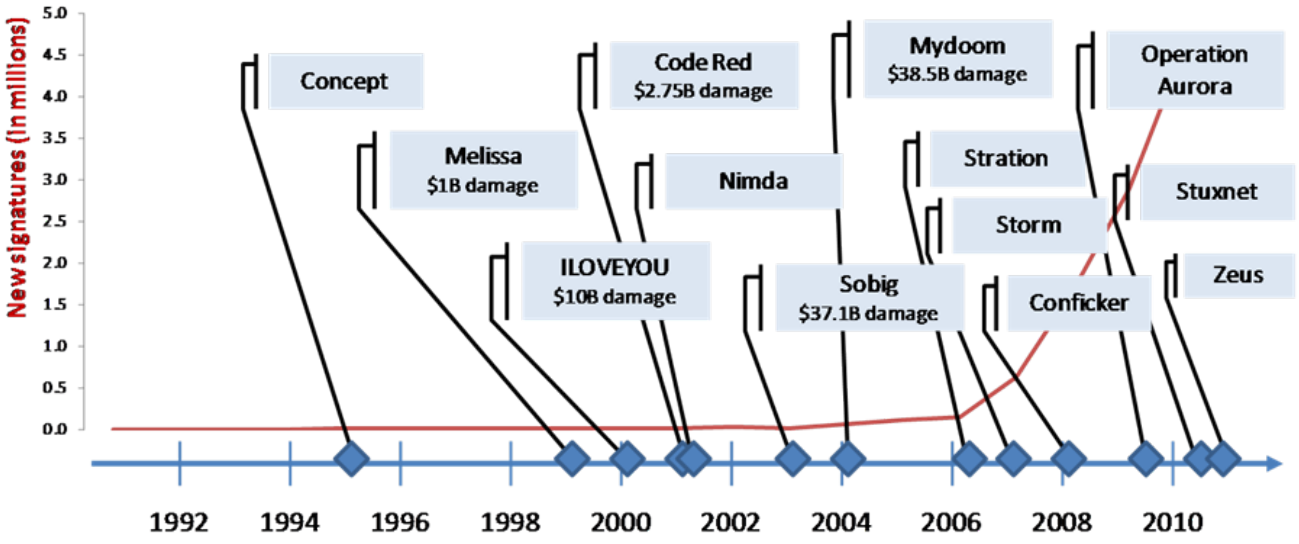


Figure: Number of signature in McAffee antivirus

Problematics (2)

Second threat : current signatures fail against mutating viruses

- slow polymorphism;
- current schemes fail to handle code obfuscation [CJ04];

Nom	Réf	Type	Détection[1]
W32/Evol	[Vir00a]	Méta	74,1%
W32/AOC	[Vir02a]	Poly	74,2%
W32/BlackBat	[Vir01]	Poly	70,5%
W32/Bolzano	[Vir06]	Poly	53,9%
W32/Hatred	[Vir00b]	Poly	48,3%
W32/Hezhi	[Vir02b]	Poly	68%
W32/Lexotan	[Vir09]	Méta	12,6%

Figure: Detection rate of some mutating viruses [LSM09]

[1] www.virustotal.com

Syntactic models

- Current antivirus
  - **model**: sequence of bits
  - **signature** are regular expressions
  - **extraction** of the signature: longest common pattern [KA94, SJHWG+03, SXCM04, NKS05]
  - semi-automatic learning (manual for highly mutating viruses)
- Statistical models on byte tuples
  - HMM [GSHcC09, YW09], naive Bayes [WDJL03], decision trees [WDJL03], neural networks [TKS96], data mining [SEZS01, YWLY07]
  - automatic learning
- Pros/cons
  - detection in $\Theta(n)$
  - automatic signature extraction for machine-learning based schemes
  - FP and FN not negligible
  - depends on the learning phase: slow polymorphism ?

Structural and semantic detection schemes

- Use **high-level** extraction/detection model that can:
  - cope with software obfuscation
  - recognize high-level language classes
- Based on:
  - the **structure** of the program (CFG)
  - the **semantics** of the program (functional equivalence)
- **Idea**:
  - mutation engines $\simeq$ syntactical rewriting
  - two instances $(v, v') \in V^2$ are functionally equivalent

Current trends

- **Structural** detection schemes (CFG)
  - Signature: simplified sub-*CFG* of an instance of the virus
    - ... when possible[KRVV04], IDAPro based
  - Detection: ismorphism [BKM07], equality up to variables renaming [CJ03]
- **Model checking**
  - Virus as Kripke Structure
  - Detection: CTPL formula [Kin05]
- **Abstract interpretation**
  - Define the abstract semanctics S^A of a program
  - Detection: $S^A(virus) \subseteq_A S^A(program)$ [LSM09]

Our idea: hybrid model

	Pros	Cons
Syntactic models	▸ detection complexity	▸ *FP* and *FN* ▸ learning ▸ signature expressiveness
Semantic models	▸ signature expressiveness ▸ automatic signature ▸ *FP* and *FN*	▸ detection complexity

Our idea: hybrid model

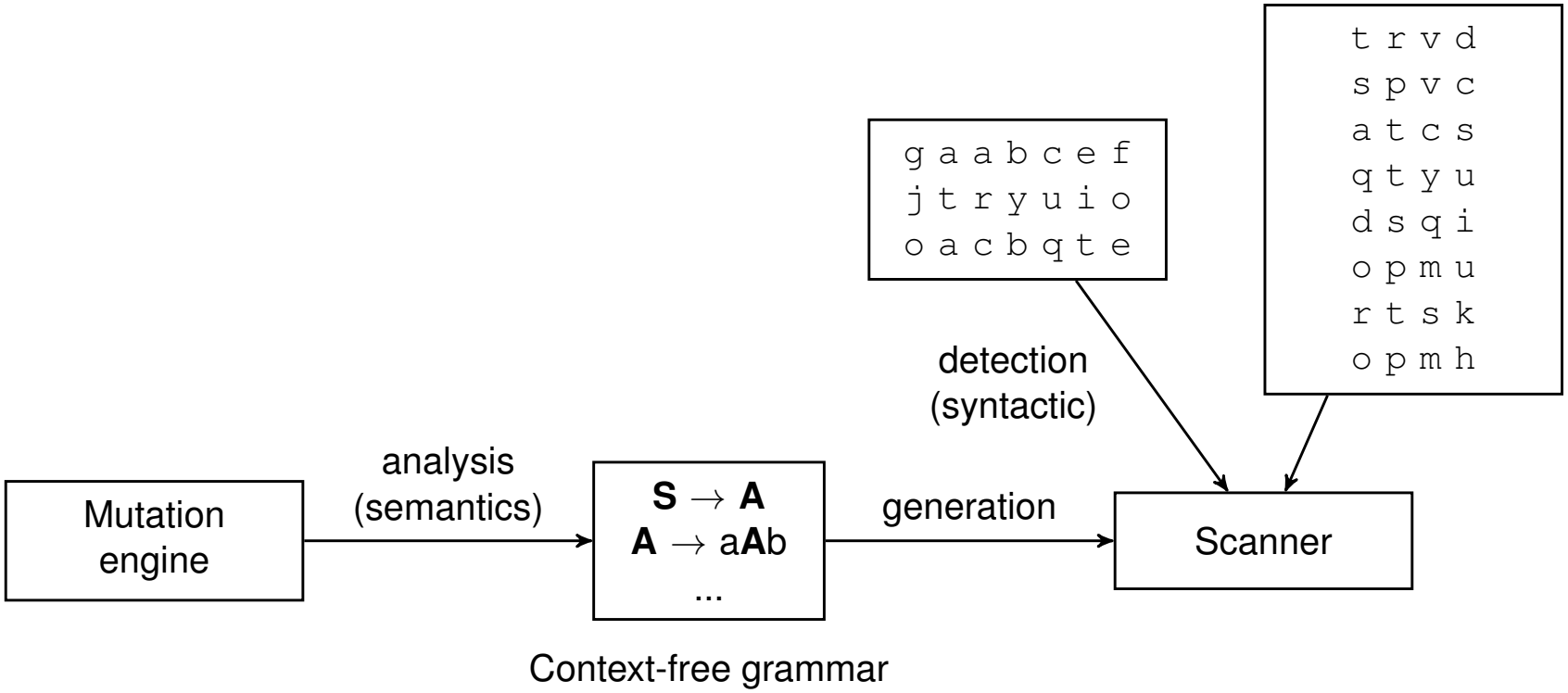
	Pros	Cons
Syntactic models	▸ detection complexity	▸ *FP* and *FN* ▸ learning ▸ signature expressiveness
Semantic models	▸ signature expressiveness ▸ automatic signature ▸ *FP* and *FN*	▸ detection complexity

Our idea: hybrid model

	Pros	Cons	
Syntactic models	▸ detection complexity	▸ *FP* and *FN* ▸ learning ▸ signature expressiveness	⇒ detection
Semantic models	▸ signature expressiveness ▸ automatic signature ▸ *FP* and *FN*	▸ detection complexity	⇒ extraction

Our solution: hybrid detection scheme

- Mutation engine does not vary functionally
- Mutation engine indirectly expresses virus grammar
- Mutation engine can be extracted easily (manual analysis)
- Outline:



Why context-free grammars ?

- Experience
  - mutation engines produce *almost* context-free grammars
  - ... they are a set of recursive functions
  - ... but sometimes use finite additional memory (e.g. `freereg`, array of unused registers)

```
1  void mut_mov(reg,value) {
2   if(random()) {
3    mut_push(value);
4    write(0x44|reg); //pop reg
5   else
6    write(0xB8|reg::value);
7  }
8
9  void mut_push(value) {
10  if(random()) {
11   mut_mov(freereg,value);
12   write(0x54|freereg);} //push
        reg
13  else
14   write(0x55::value);
15 }
```

$\mathbf{S} \rightarrow \mathbf{M}$

$\mathbf{M} \rightarrow 0xB8[0{:}5]\ \top[0:3]\ \top[0:32]$

$\mathbf{M} \rightarrow \mathbf{P}\ 0x44[0{:}5]\ \top[0:3]$

$\mathbf{P} \rightarrow 0x55[0{:}8]\ \top[0:32]$

$\mathbf{P} \rightarrow \mathbf{M}\ 0x54[0{:}5]\ \top[0:3]$

Outline

1. Context
2. The mutation semantics
   - Concrete semantics
   - The word semantics
   - Abstract word semantics
3. Implementation & preliminary results
4. Conclusion

Concrete semantics

- Analysis of **binary** programs
- Based on BINCOA [BHL+11] semantics
  - translation from ARM/x86 binary programs
  - framework for dataflow analysis/abstract interpretation

Microcode language

- Expressions
  $\mathbb{E} ::= n[n_1 : n_2[\mid e \text{ op } e \mid \text{MEM}[e] \mid e[n_1 : n_2[$
- Instructions $\mathbb{I} ::=$

`Assign` $\texttt{MEM}[x], y$	$x \in \mathbb{E}$, $y \in \mathbb{E}$
`Guard` $cond, l_{true}, l_{false}$	$cond \in \mathbb{E}$, $l_{true} \in \mathbb{N}$, $l_{false} \in \mathbb{N}$
`Call` l	$l \in \mathbb{N}$
`Return`	
`Halt`	

- Decoding: $decode : \mathbb{N} \rightarrow \mathbb{I}$

Word semantics

- We want to capture all possibly produced words

Word semantics

Let C^* be the set of concrete execution traces
Let $Out : \mathcal{C} \rightarrow \Sigma$ be the output oracle (ϵ if not output)
Let $Word : C^* \rightarrow \Sigma^*$ be:

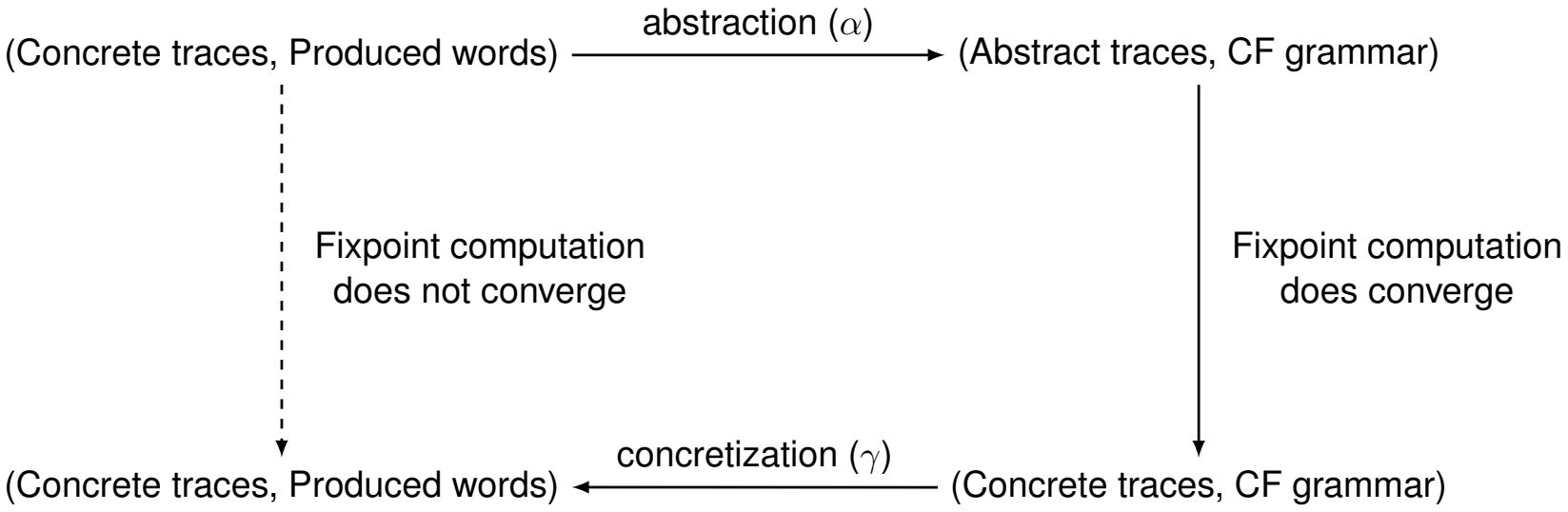
$$Word(\sigma = \sigma_0 :: ... :: \sigma_n) = Out(\sigma_0) :: ... :: Out(\sigma_n)$$

The word semantics is then:

$$S^W = \{\forall \sigma \in C^*, (\sigma, Word(\sigma))\}$$

- Cannot be computed !
  - Does not converge: infinite (number of) words
  - Based on concrete semantics
  - Initial configuration: $\perp_W == ?$

Abstract Interpretation



- Abstract domain: **sound** overapproximation [CC79]
  - semi-lattice
  - fixpoint function converges
  - widening to accelerate convergence

Context free grammar domain

- Set of words: does not converge $\rightarrow$ context free grammar
- Grammar tied to a program $P = (PP \subseteq \mathbb{N}, \iota : PP \rightarrow \mathbb{I})$
- Finite overapproximation of produced words

Grammar domain

$$(\mathcal{G}, \subseteq, \vee_G, \wedge_G, \top_G, \bot_G)$$

- $\mathcal{G} = (\{0,1\}, N, S, R)$ a context-free grammar
- $G \vee_G G' = (\{0,1\}, N \cup N' \cup \{S''\}, S'', R \cup R' \cup \{S'' \rightarrow S, S'' \rightarrow S'\}$
- $\top_G$ is one grammar producing $\{0,1\}*$
- $\bot_G = (\{0,1\}, PP, N_0, \emptyset)$

Abstract word semantics

Our abstract domain:

$$\mathcal{A} = (\langle \mathcal{G}, \mathcal{N} \rangle, \leq_G \wedge \leq_N, \vee_G \wedge \vee_N, \wedge_G \wedge \wedge_N, (\top_G, \top_N), (\bot_G \wedge \bot_N))$$

- $\mathcal{N}$ is an abstract domain of choice (interval,polyhedra,...)
  - used by the *Out* oracle to compute which word may be produced at each program point

Predicate transformer

Let $\sigma = \sigma_0 :: \sigma_n \in C^*$, $\sigma_n =< ip, m, s >\in C$ and $i = decode(ip) \in \mathbb{I}$:

$$F^A(\langle (T, N, S, R), n \rangle) = \bigcup^{\mathcal{A}}_{\forall \sigma \in \gamma_N(n)} \{$$

$$\begin{array}{ll}
i = \texttt{Assign MEM}[x], y & \langle (T, N, S, R \cup \{N_{ip} \rightarrow Out(i, \sigma_n) N_{ip+1}\}), F^N(n) \rangle \\
i = \texttt{Guard}\ cond, l_t, l_f & \langle (T, N, S, R \cup \{N_{ip} \rightarrow N_{l_t}, N_{ip} \rightarrow N_{l_f}\}), F^N(n) \rangle \\
i = \texttt{Call}\ l & \langle (T, N, S, R \cup \{N_{ip} \rightarrow N_l N_{ip+1}\}), F^N(n) \rangle \\
i = \texttt{Return} & \langle (T, N, S, R \cup \{N_{ip} \rightarrow \epsilon\}), F^N(n) \rangle \\
i = \texttt{Halt} & \langle (T, N, S, R \cup \{N_{ip} \rightarrow \epsilon\}), F^N(n) \rangle
\end{array}$$

$\}$

Example

```
1  Guard random(),9,2
2  Assign mem[OUT],'A'
3  Call 1
4  Guard random(),5,7
5  Assign mem[var],'B'
6  Guard True,8,8
7  Assign mem[var],'C'
8  Assign mem[OUT],var
9  Return
```

$\gamma_{\mathcal{N}} = \{\langle var = B \rangle\}$

$\gamma_{\mathcal{N}} = \{\langle var = C \rangle\}$

$\gamma_{\mathcal{N}} = \{\langle var = B \rangle, \langle var = C \rangle\}$

$$
\begin{aligned}
N_1 &\rightarrow N_9 \mid N_2 \\
N_2 &\rightarrow A\ N_3 \\
N_3 &\rightarrow N_1\ N_4 \\
N_4 &\rightarrow N_5 \mid N_7 \\
N_5 &\rightarrow N_6 \\
N_6 &\rightarrow N_8 \\
N_7 &\rightarrow N_8 \\
N_8 &\rightarrow B\ N_9 \mid C\ N_9 \\
N_9 &\rightarrow \epsilon
\end{aligned}
$$

Figure: Example on $A^n(B|C)^n$

Prerequisites

Prerequisites of our scheme:

1. Call/return identification
   - hard for binary code
   - overapproximations exist [Boc09]
   - ... but polymorphic engines are rarely obfuscated
2. Output identification
   - pattern matching
     - ★ ... usually a fixed register
   - abstract interpretation
3. Dynamic jumps overapproximation (besides rets)
   - CFG reconstruction [KVZ09]
4. Mutated code must be written *sequentially*
   - can be verified during analysis

Outline

1. Context
2. The mutation semantics
3. Implementation & preliminary results
   - Implementation
   - Results
4. Conclusion

Implementation

- Working prototype ($\simeq$ 40 kLOC)
- We use a **symbolic analysis** as our numerical domain
  - context-insensitive, compositional $\Rightarrow$ good performances
  - backtracking + function inlining when too much precision is lost
  - widening on length and depth of terms
- Bit-precise, always try to evaluate most of the values bits

  Rewriting rules:
  $x[0{:}a[\ \texttt{AND}\ (2^k - 1) \rightarrow x[0{:}k[\ ::\ 0[0{:}a - k[$
  $x[0{:}a[\ \texttt{OR}\ (y[0{:}b[\ ::\ 0[0{:}a - b[) \rightarrow (x[0{:}b[\ \texttt{OR}\ y[0{:}b[) ::\ x[b{:}a[$
  Result: $\texttt{58}[0{:}8[\ \texttt{OR}\ (\top[0{:}8[\ \texttt{AND}\ 7[0{:}8[) \rightarrow \top[0{:}3[\ ::\ \texttt{58}[3{:}8[$
- Tested on real-word polymorphic viruses

Preliminary results (cont'd)

Name	# instrs	Grmr size	Amb.	FP	FN
CRPE	359	20	41	0	0
Bolzano	578	17	15	0	0
NBKPE	697	68	115	0	0
VoltagePE	1044	152	221	0	0
OffensivePE	1053	173	1203	22%	0
ETMS	1100	240	2792	too long	too long

Figure: Scanning results

- **Note**:
  - proof of concept written in python
  - no disambiguation pass

CRPE grammar

A → U E8[0:8[⊤[0:32[K | U U E8[0:8[⊤[0:32[K
B → ⊤[0:8[⊤[0:8[79[0:8[⊤[0:8[R | U ⊤[0:8[⊤[0:8[79[0:8[⊤[0:8[R
C → 8B[0:8[⊤[0:8[Q | 85[0:8[⊤[0:8[| 3B[0:8[⊤[0:8[
D → ⊤[0:8[D | ⊤[0:8[N
E → 4[0:3[⊤[0:5[
F → 8B[0:8[⊤[0:6[3[0:2[
G → 3[0:8[⊤[0:6[3[0:2[
H → C3[0:8[| U C3[0:8[
I → D1[0:8[⊤[0:8[1B[0:8[⊤[0:8[⊤[0:8[75[0:8[⊤[0:8[⊤[0:8[75[0:8[⊤[0:8[8B[0:8[4[0:3[⊤[0:2[2[0:3[2024[0:16[O |
U D1[0:8[⊤[0:8[1B[0:8[⊤[0:8[⊤[0:8[75[0:8[⊤[0:8[⊤[0:8[75[0:8[⊤[0:8[8B[0:8[4[0:3[⊤[0:2[2[0:3[2024[0:16[O
J → U P | 60[0:8[⊤[0:2[2E[0:6[⊤[0:32[⊤[0:2[2E[0:6[⊤[0:32[M F T
K → ⊤[0:8[K | ⊤[0:8[J
L → 83[0:8[⊤[0:3[1[0:1[⊤[0:1[⊤[0:1[⊤[0:1[⊤[0:1[| 83[0:8[⊤[0:8[
M → 33[0:8[⊤[0:2[0[0:1[⊤[0:1[⊤[0:1[6[0:3[| 2B[0:8[⊤[0:2[0[0:1[⊤[0:1[⊤[0:1[6[0:3[
N → 44444444[0:32[| 83[0:8[4C4[0:16[
O → ⊤[0:3[17[0:5[⊤[0:32[G 31[0:8[⊤[0:8[33[0:8[⊤[0:8[B
P → 60[0:8[⊤[0:2[2E[0:6[⊤[0:32[⊤[0:2[2E[0:6[⊤[0:32[M F T
Q → 85[0:8[⊤[0:8[| E8[0:8[⊤[0:32[D | 3B[0:8[⊤[0:8[
R → U 61[0:8[H | 61[0:8[H
T → 81[0:8[⊤[0:4[F[0:4[⊤[0:32[32[0:8[E ⊤[0:8[FF[0:8[L 8[0:8[I | 1[0:1[⊤[0:1[1[0:1[⊤[0:1[3[0:2[⊤[0:1[⊤[0:1[
⊤[0:32[32[0:8[E ⊤[0:8[FF[0:8[L 8[0:8[I
U → E8[0:8[⊤[0:32[D | 8B[0:8[⊤[0:8[Q | 85[0:8[⊤[0:8[| 3B[0:8[⊤[0:8[| ⊤[0:3[A[0:5[⊤[0:8[C

Outline

1. Context
2. The mutation semantics
3. Implementation & preliminary results
4. Conclusion

Conclusion & future work

- **Hybrid detection scheme**
  - similar to [PGD+10] but CF grammar + implementation
  - semantics-aware grammar extraction
    - ★ no false negative
    - ★ nearly automated
    - ★ only one sample needed: no learning !
  - syntactic detection
- **Context-free grammars** ⇒ more precise
- But may be too time consuming
  - use it as extra verification tool
  - safe approximation by a regular language [Ned00]
- Use grammar to **classify** malwares ?

Sebastien Bardin, Philippe Herrmann, Jérôme Leroux, Aymeric Vincent, Olivier Ly, and Renaud Tabary.
The bincoa framework for binary code analysis.
In *Computer Aided Verification, 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011, Proceedings*, volume 6806 of *Lecture Notes in Computer Science*. Springer, 2011.

Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion.
Control flow graphs as malware signatures.
In Eric Filiol, Jean-Yves Marion, and Guillaume Bonfante, editors, *International Workshop on the Theory of Computer Viruses TCV'07*, Nancy France, 2007.

Davidson Rrodrigo Boccardo.
Context-Sensitive Analysis of x86 Obfuscated Executables.
PhD thesis, Universidade Estadual Paulista, 2009.

Patrick Cousot and Radhia Cousot.
Systematic design of program analysis frameworks.
In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

Mihai Christodorescu and Somesh Jha.
Static analysis of executables to detect malicious patterns.
In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

Mihai Christodorescu and Somesh Jha.
Testing malware detectors.
SIGSOFT Softw. Eng. Notes, 29:34–44, July 2004.

Fred Cohen.
Computer viruses : Theory and experiments.
Computers & Security, 6(1):22 – 35, 1987.

Christian Collberg, Clark Thomborson, and Douglas Low.
A taxonomy of obfuscating transformations.
Technical Report 148, Department of Computer Science, The University of Auckland, Jul 1997.
http://www.cs.auckland.ac.nz/ collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

Kent Griffin, Scott Schneider, Xin Hu, and Tzi cker Chiueh.
Automatic generation of string signatures for malware detection.
Technical report, Symantec, 2009.

Jeffrey O. Kephart and William C. Arnold.
Automatic extraction of computer virus signatures.
In Virus Bulletin Ltd., editor, *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, 1994.

Johannes Kinder.
Model checking malicious code.
Master's thesis, Technische Universite at Munchen, 2005.

Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna.
Static disassembly of obfuscated binaries.
In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

Johannes Kinder, Helmut Veith, and Florian Zuleger.
An abstract interpretation-based framework for control flow reconstruction from binaries.
Proceedings of the 10th International Conference on Verification,Model Checking, and Abstract Interpretation (VMCAI 2009), 5403:214–228, 2009.

F. Leder, B. Steinbock, and P. Martini.
Classification and detection of metamorphic malware using value set analysis.
In *Malicious and Unwanted Software (MALWARE), 2009 4th International*, pages 39 – 46, 2009.

Mark-Jan Nederhof.
Practical experiments with regular approximation of context-free languages.
Comput. Linguist., 26:17–44, March 2000.

James Newsome, Brad Karp, and Dawn Song.
Polygraph: Automatically generating signatures for polymorphic worms.
Security and Privacy, IEEE Symposium on, 0:226–241, 2005.

Mila Dalla Preda, Roberto Giacobazzi, Saumya Debray, Kevin Coogan, and Gregg M. Townsend.
Modelling metamorphism by abstract interpretation.
In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 218–235, Berlin, Heidelberg, 2010. Springer-Verlag.

Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo.
Data mining methods for detection of new malicious executables.
Security and Privacy, IEEE Symposium on, 0:0038, 2001.

Deng Peter Shaohua, Wang Jau-Hwang, Shieh Wen-Gong, Yen Chin-Pin, and Tung Cheng-Tan.
Intelligent automatic malicious code signatures extraction.
In *Security Technology, 2003. Proceedings. IEEE 37th Annual 2003 International Carnahan Conference*, pages 600–603, 2003.

Harry Sverdlove.
20 years of malware security, Feb 2011.
http://blog.bit9.com/bid/40340/What-s-the-Score-20-Years-of-Malware-Security.

A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala.
Static analyzer of vicious executables (save).
In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 326–334, Washington, DC, USA, 2004. IEEE Computer Society.

G.J. Tesauro, J.O. Kephart, and G.B. Sorkin.
Neural networks for computer virus recognition.
volume 11, pages 5 –6, aug 1996.

W32/evol, 2000.
www.openrce.org/articles/full_view/27.

W32/hatred, 2000.
http://www.securelist.com/en/descriptions/old20608.

W32/blackbat, 2001.
http://www.rohitab.com/blackbat-virus.

W32/aoc, 2002.
http://www.securelist.com/en/descriptions/old20299.

W32/hezhi, 2002.
www.symantec.com/security_response/writeup.jsp?docid=2003-032615-3916-99.

W32/bolzano, 2006.
http://www.avira.com/fr/support-threats-description/tid/6062/threat/W32.Bolzano.5572.

W32/lexotan, 2009.
vx.org.ua/29a/29a-6.html.

Jau-Hwang Wang, P.S. Deng, Yi-Shen Fanand Li-Jing Jaw, and Yu-Ching Liu.
Virus detection using data mining techinques.
In *Proceedings of the IEEE 37th Annual 2003 International Carnahan Conference on Security Technology*, 2003.

Wei Yan and Erik Wu.
Toward automatic discovery of malware signature for anti-virus cloud computing.
Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 4:724–728, 2009.

Yanfang Ye, Dingding Wang, Tao Li, and Dongyi Ye.
Imds: intelligent malware detection system.
In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 1043–1047, New York, NY, USA, 2007. ACM.