POLITECNICO DI MILANO





Behavior-based Methods for Automated, Scalable Malware Analysis

Stefano Zanero, PhD

Assistant Professor, Politecnico di Milano

NECST laboratory

Knowing the enemy = key for success

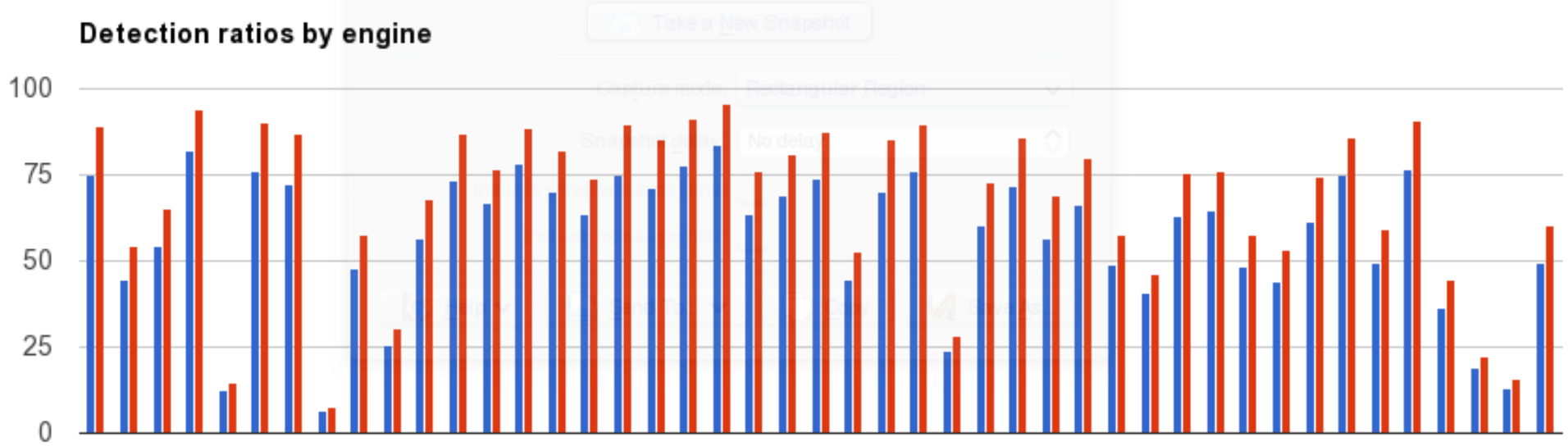
"He will win who knows when to fight and when not to fight... He will win who, prepared himself, waits to take the enemy unprepared. Hence the saying: If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle." [Sun-Tsu]

The malware problem

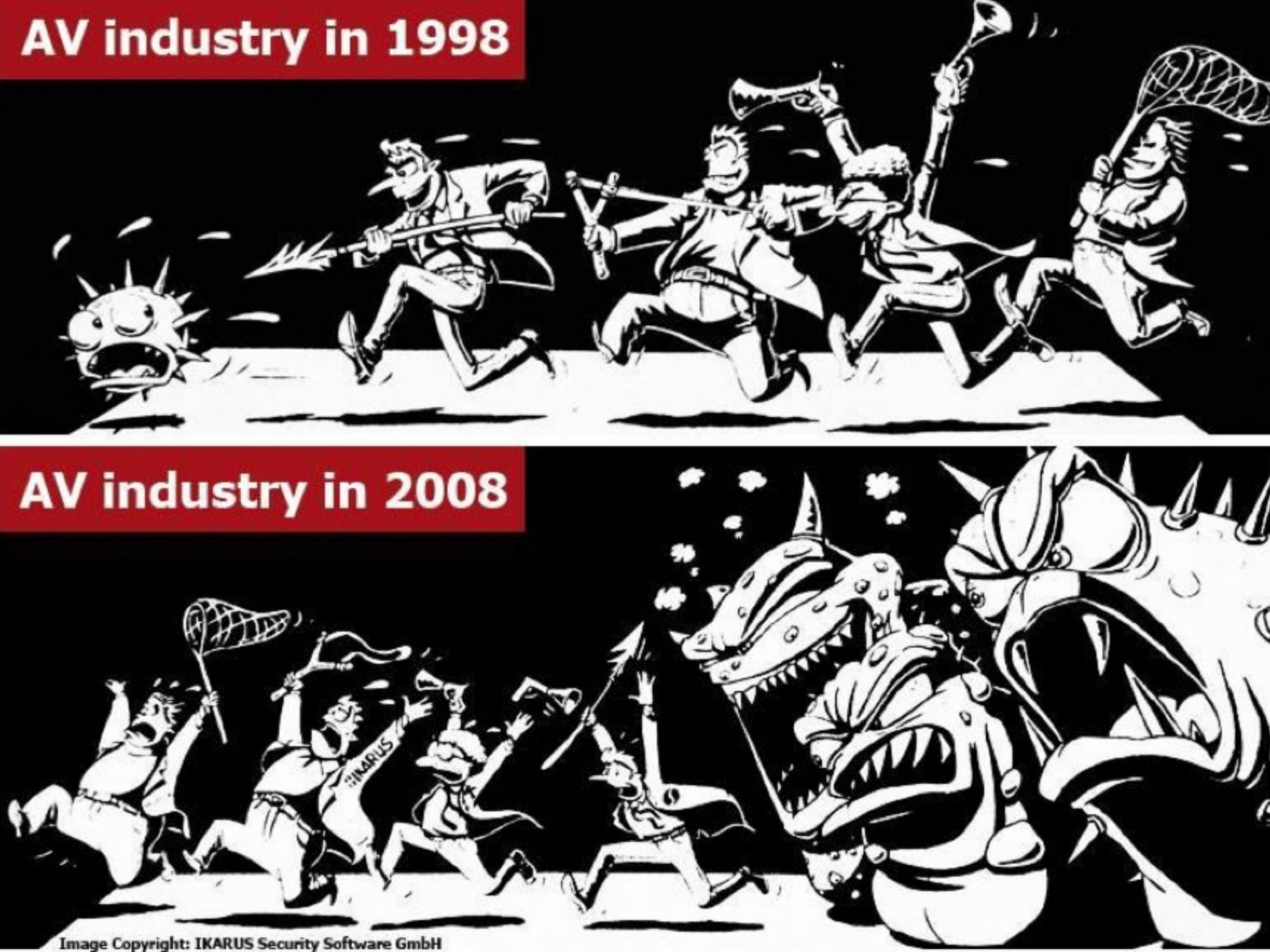
Malware at the root of many internet security problems

- Tens of thousands of new samples each day
- developed with creation kits = rapid evolution of multiple variations
- Underground economy fuelling malware creation
- 1990s: explosive diffusion of **identical** malware
- 2010s: stealthy diffusion of **variants** of malware **designed** to be difficult to identify, trace and analyze

Antivirus detection ratio...



Names censored to protect the culprits...
Thanks to VirusTotal (www.virustotal.com)



The analysis issue

Analysts are way too few, code is way too much

Need better ways to

- Automatically analyze/reverse engineer malware
- Automatically classify/cluster malware, e.g. in families

For both, we have two approaches with symmetric issues

Static vs. dynamic approaches

Static approaches	Dynamic approaches
+ Complete analysis	- “Dormant” code
- Difficult to extract semantics	+ Easy to see “behaviors”
- Obfuscation / packing	+ Malware unpacks itself

Our Approach

Turn weakness into strength, and strength into weakness, as Sun-Tzu would suggest: **leverage code reuse between malware samples to our advantage**

- Automatically generate semantic-aware models of code implementing a given malicious behavior
- Use these models to statically detect the malicious functionality in samples that do not perform that behavior during dynamic analysis
- Use a variation of this technique to study malware evolution over time

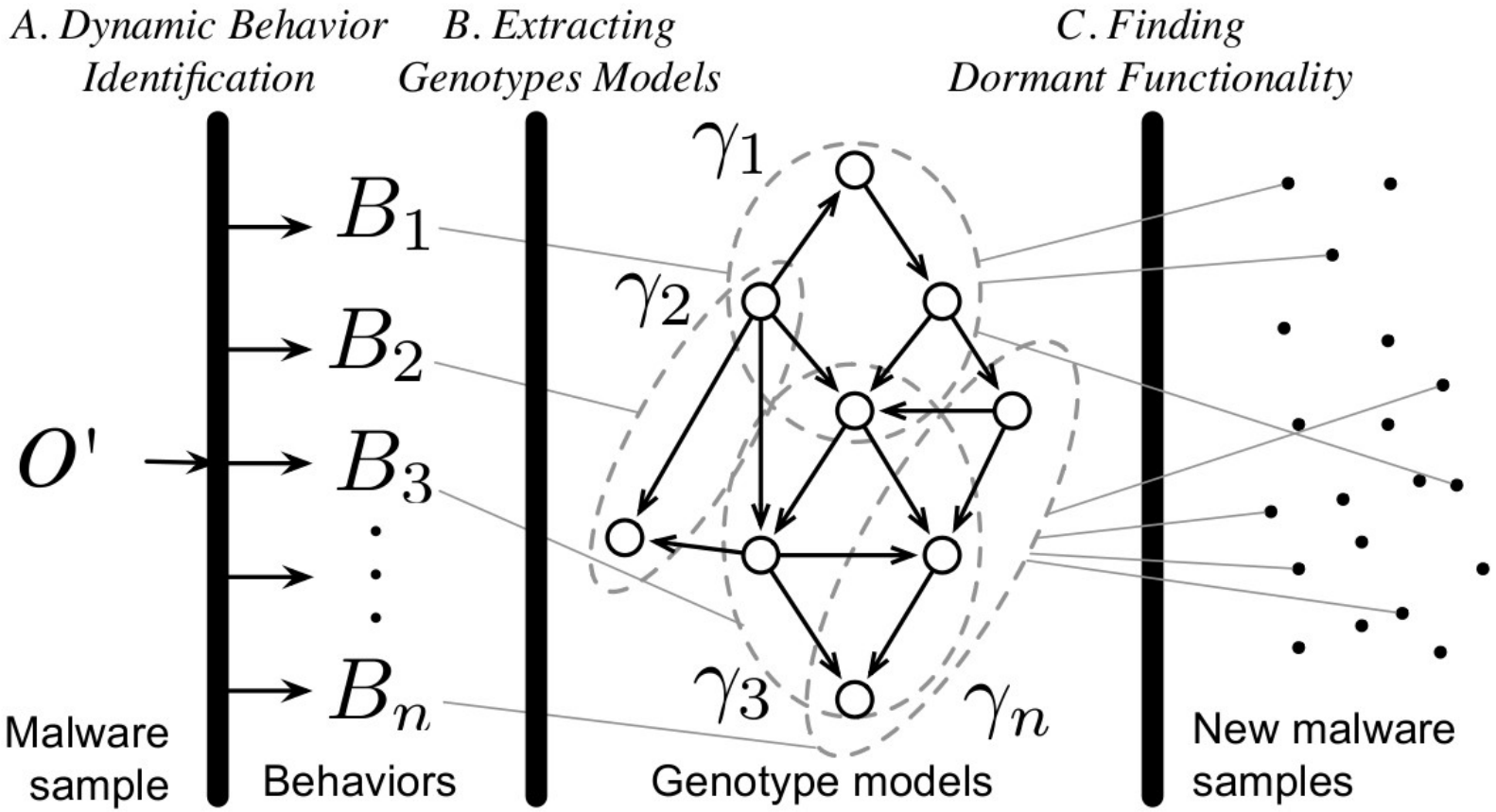
REANIMATOR

Run malware in monitored environment and detect a malicious behavior *(phenotype)*

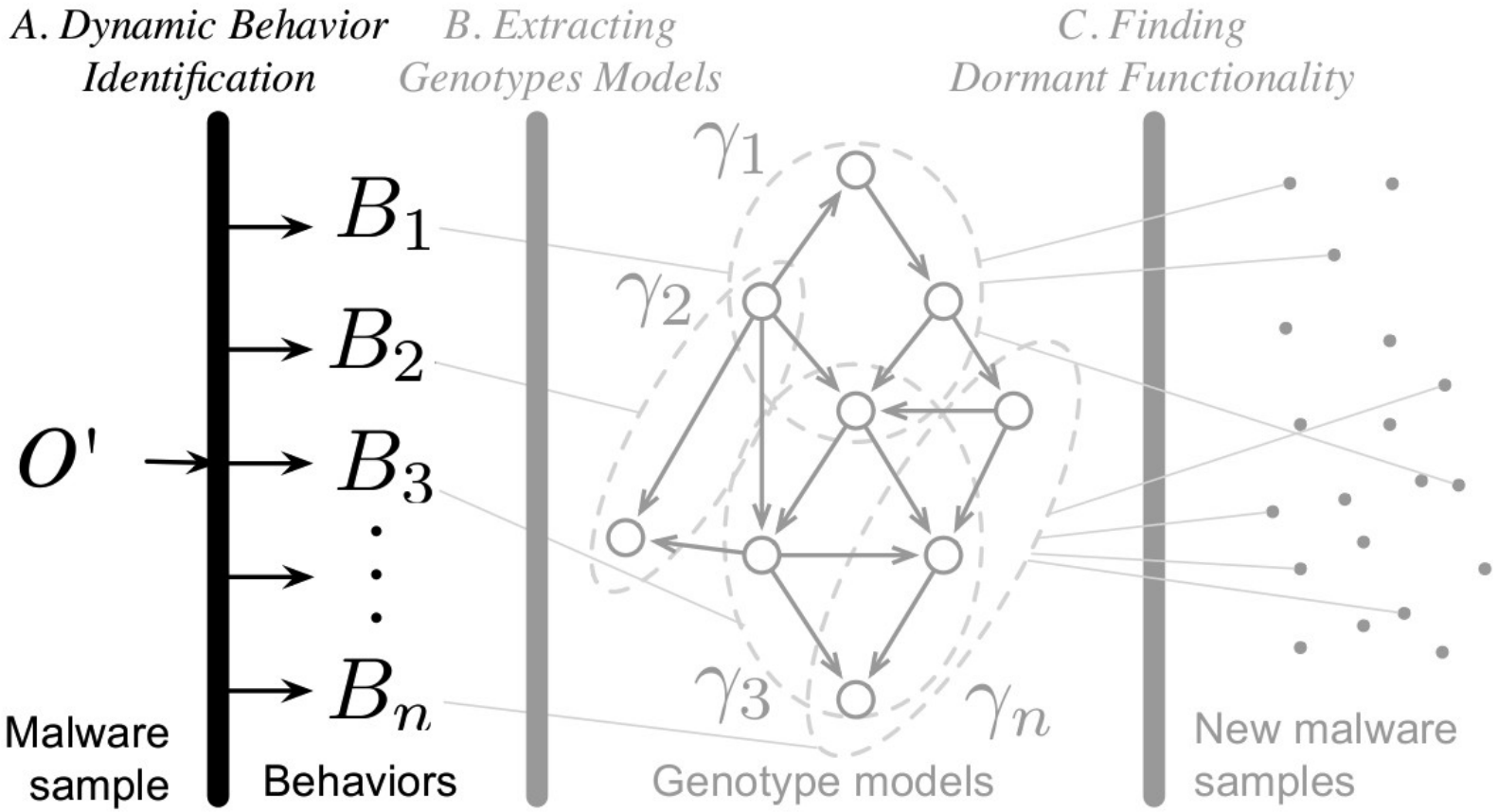
Identify and model the code responsible for the malicious behavior *(genotype model)*

Match genotype model against other unpacked binaries

REANIMATOR


A. Dynamic Behavior Identification
B. Extracting Genotypes Models
C. Finding Dormant Functionality
O'
B1
B2
B3
Bn
γ1
γ2
γ3
γn
Malware sample
Behaviors
Genotype models
New malware samples

Dynamic Behavior Identification



Dynamic Behavior Identification

Run malware in instrumented sandbox

- Anubis (anubis.iseclab.org)

Dynamically detect a behavior B *(phenotype)*

Map B to the set R_B of system/API call instances responsible for it

R_B is the output of the behavior identification phase

Behavior Detection Examples

spam: send SMTP traffic on port 25

- network level detection

sniff: open promiscuous mode socket

- system call level detection

rpcbind: attempt remote exploit against a specific vulnerability

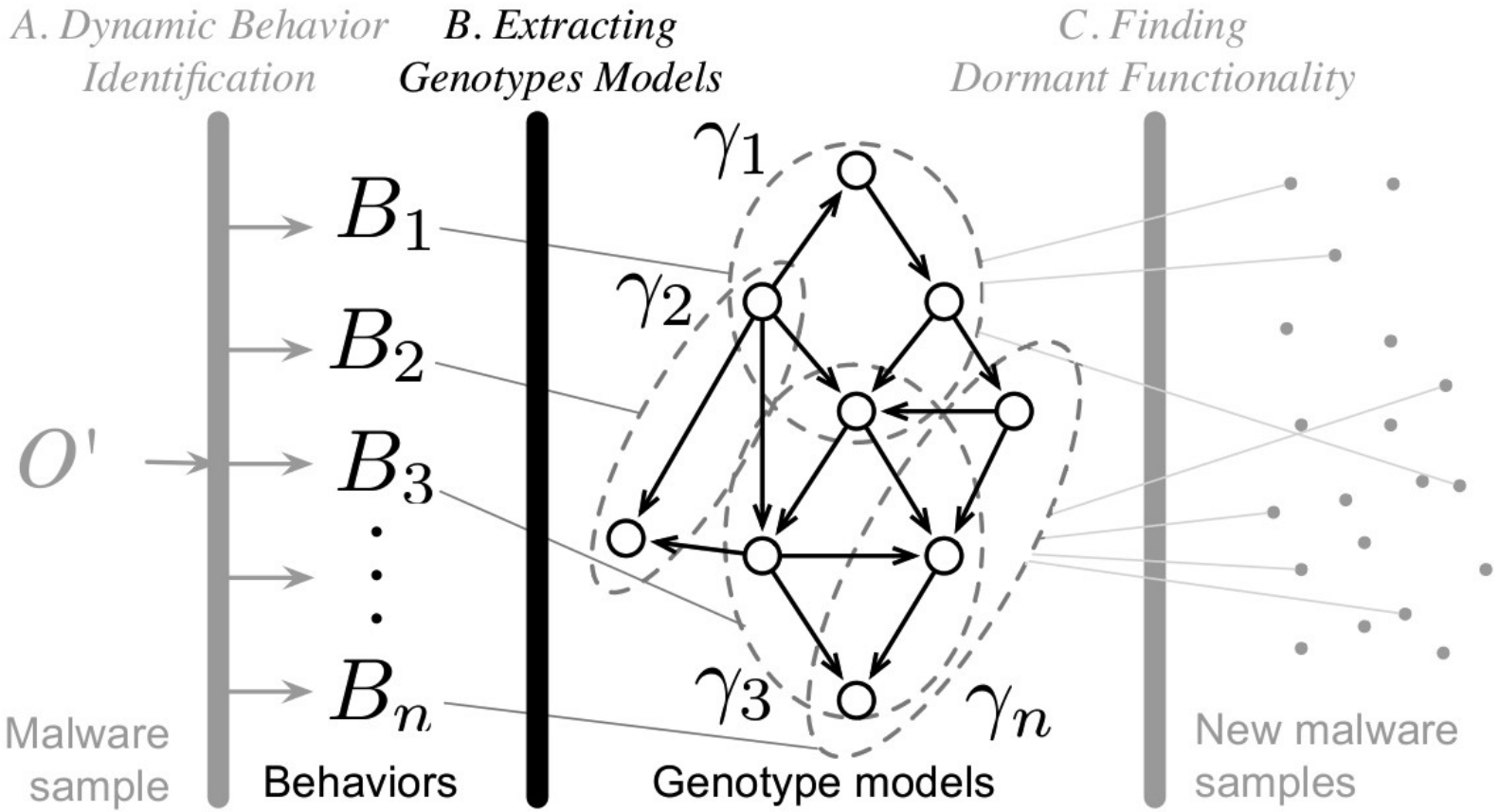
- network level detection, with snort signature

drop: drop and execute a binary

- system call level detection, using data flow information

...

Extracting Genotype Models



Extracting Genotype Models: Goals

Identified genotype should be precise and complete

- Complete: include all of the code implementing B
- Precise: do not include code that is not specific to B (utility functions,..)

We proceed by *slicing* the code, then *filtering* it to remove support code, and *germinating* to complete it

Slicing

Start from relevant calls R_B

Include into slice ϕ instructions involved in:

- preparing input for calls in R_B
  - follow data flow dependencies backwards from call inputs
- processing the outputs of calls in R_B
  - follow data flow forward from call outputs

We do not consider control-flow dependencies

- would lead to including too much code (taint explosion problem)

Filtering

The slice ϕ is not precise

General purpose utility functions are frequently included (i.e: string processing)

- may be from statically linked libraries (i.e: libc)
- genotype model would match against any binary that links to the same library

Backwards slicing goes too far back: initialization and even unpacking routines are often included

- genotype model would match against any malware packed with the same packer

Filtering Techniques

Exclusive instructions:

- set of instructions that manipulate tainted data **every time** they are executed
- utility functions are likely to be also invoked on untainted data

Discard whitelisted code:

- whitelist obtained from other tasks or execution of **the same sample**, that do not perform B
- could also use foreign whitelist
  - i.e: including common libraries and unpacking routines

Germination

The slice ϕ is not complete

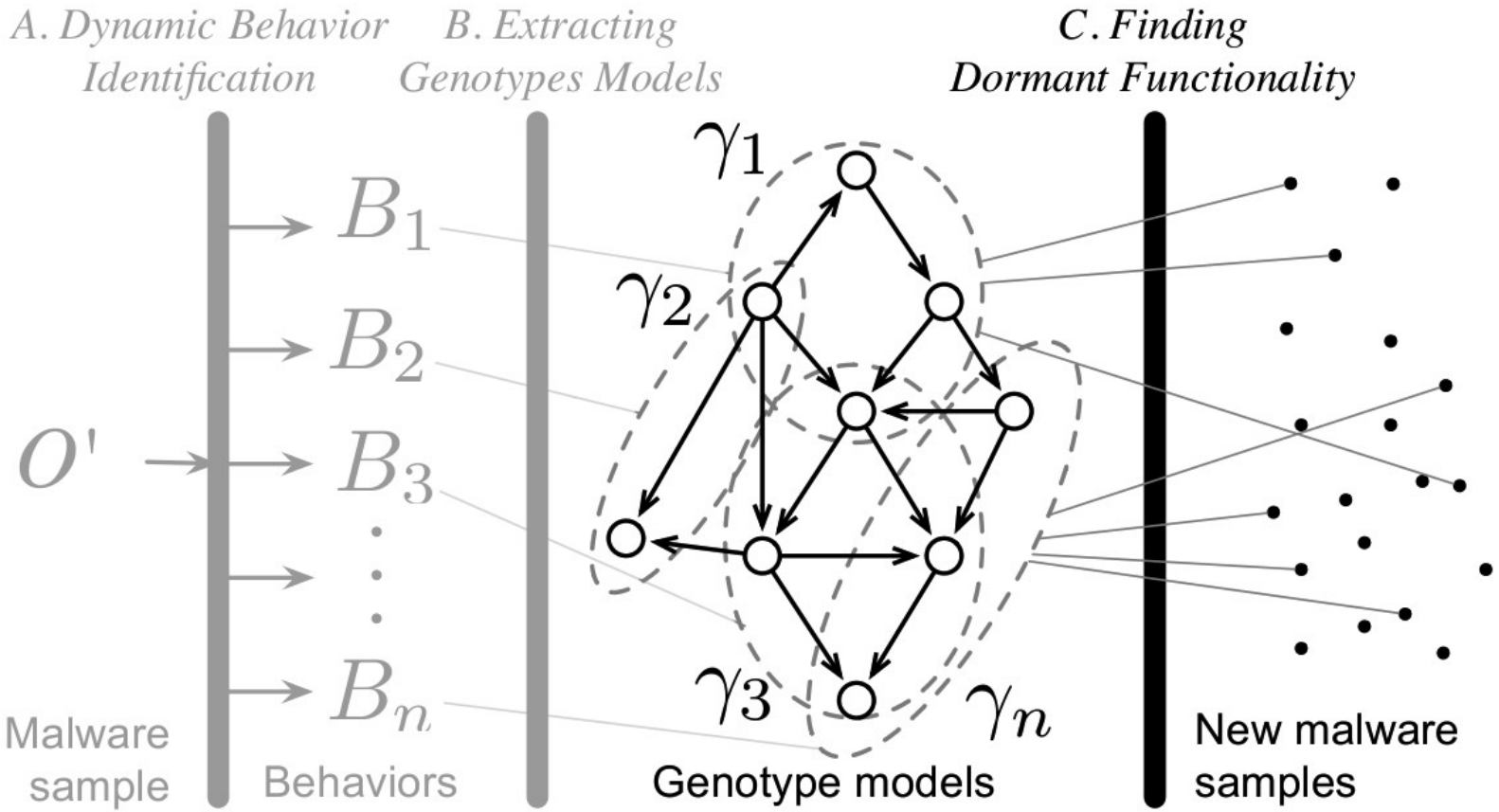
Auxiliary instructions are not included

- loop and stack operations, pointer arithmetic, etc

Add instructions that cannot be executed without executing at least one instruction in ϕ

Based on graph reachability analysis on the intra-procedural Control Flow Graph (CFG)

Finding Dormant Functionality



Finding Dormant Functionality

Genotype is a set of instructions

Genotype model is its colored control flow graph (CFG)

- nodes colored based on instruction classes

2 models match if they share at least one K-Node subgraph (K=10)

Use techniques by Kruegel et al. to efficiently match a binary against a set of genotype models

We use Anubis as a generic unpacker

Evaluation

Are the results accurate?

- when REANIMATOR detects a match, is there really the dormant behavior?
- how reliably does REANIMATOR detect dormant behavior in the face of recompilation or modification of the source code?

Are the results insightful?

- does REANIMATOR reveal behavior we would not see in dynamic analysis?

Accuracy

To test accuracy and robustness of our system we need a ground truth

Dataset of 208 malware samples with source code

- thanks to Jon Oberheide and Michael Bailey from University of Michigan

Extract 6 genotype models from 1 sample

Match against remaining 207 binaries

Accuracy

Even with source, manually verifying code similarity is time-consuming

Use a source code plagiarism detection tool

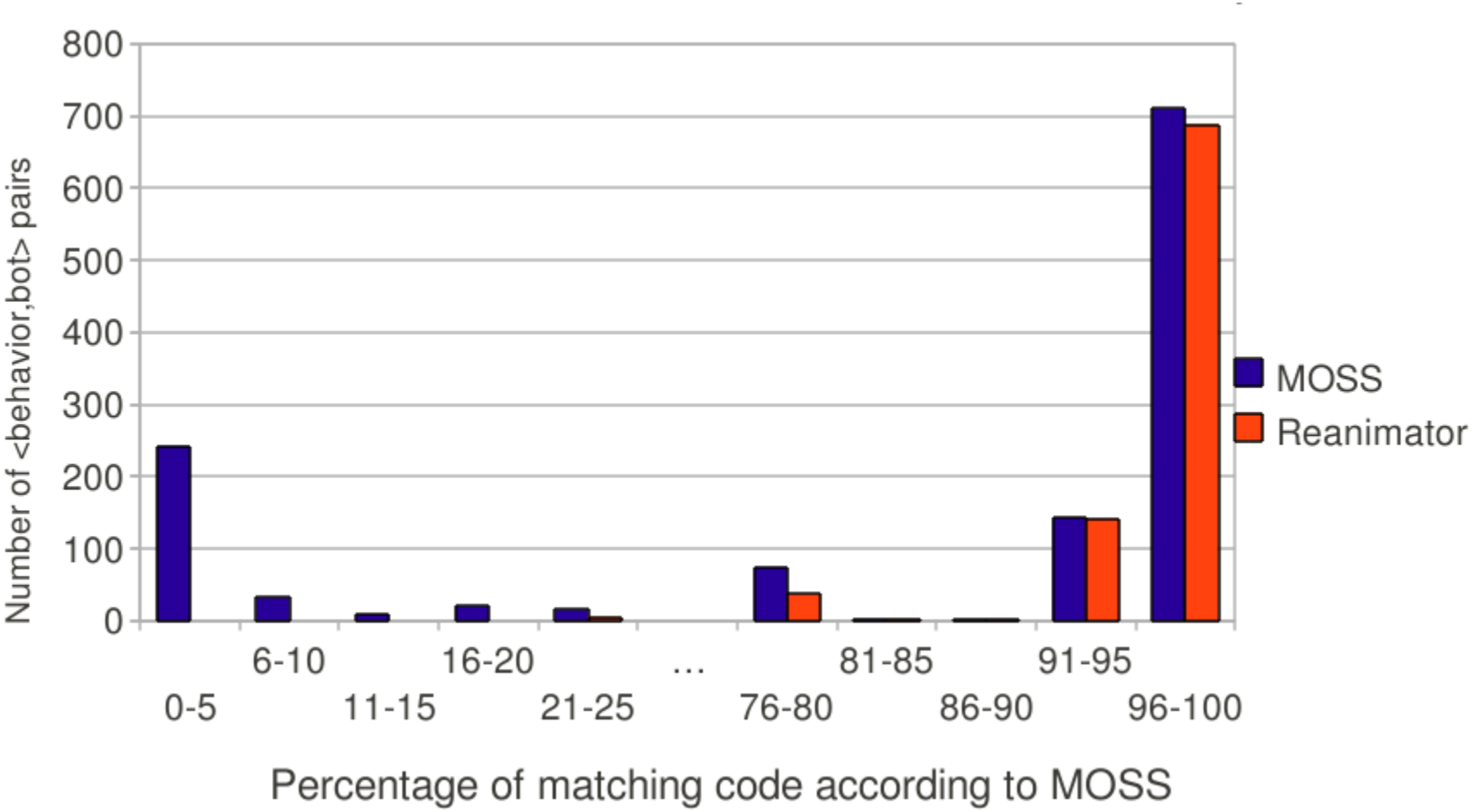
- MOSS

We feed MOSS the source code corresponding to each of the 6 behaviors

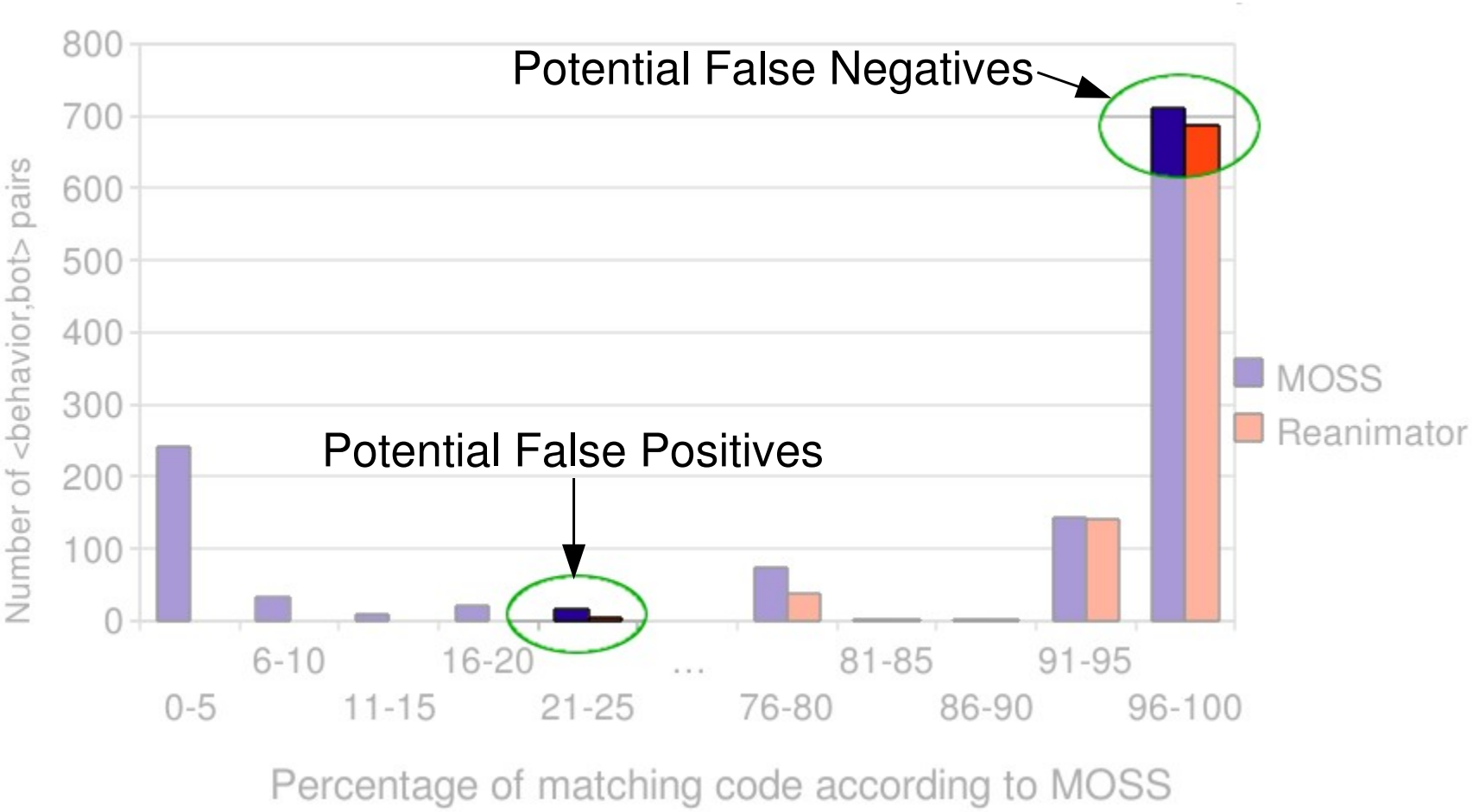
- match it against the other 207 sources
- MOSS returns a similarity score in percentage

We expect REANIMATOR to match in cases where MOSS returns high similarity scores

MOSS Comparison



MOSS Comparison



Accuracy Results

We manually investigated the potential false positives and false negatives

Low false negative rate (~1.5%)

- mostly small genotypes

No false positives

- genotype model match always corresponds to presence of code implementing the behavior

Also no false positives against dataset of ~2000 benign binaries

- binaries in system32 on a windows install

Robustness results when re-compiling same source

- Robust against different compilation options (<7% false negatives)
- Robust against different compiler versions
- Not robust against completely different compiler (>80% false negatives)
- Some robustness to malware metamorphism was demonstrated by Kruegel in a previous work

In-the-Wild Detection

10 genotype models extracted from 4 binaries

4 datasets

- irc_bots: 10238 IRC bots
- packed_bots: 4523 packed IRC bots
- pushdo: 77 pushdo binaries (dropper, typically drops spam engine cutwail)
- allaple: 64 allaple binaries (network worm)

Reanimator reveals a lot of functionality not observed during dynamic analysis

In-the-Wild Detection

Genotype	Phenotype	irc_bots				packed_bots			
		B	S	D	B ∩ S	B	S	D	B ∩ S
httpd	backdoor	2014	636	635	279	840	425	425	264
keylog	keylog	0	293	254	0	0	120	111	0
killproc	killproc	0	400	400	0	4	62	62	0
simplespam	spam	154	409	409	0	53	204	204	0
udpflood	packetflood	0	374	342	0	0	139	122	0
sniff	sniff	43	270	72	0	120	204	45	0

Genotype	pushdo				allaple			
	B	S	D	B ∩ S	B	S	D	B ∩ S
drop	50	54	54	46	0	0	0	0
spam	1	43	42	1	0	0	0	0
scan	23	0	0	0	58	61	61	58
rpcbind	5	9	0	1	62	61	61	58

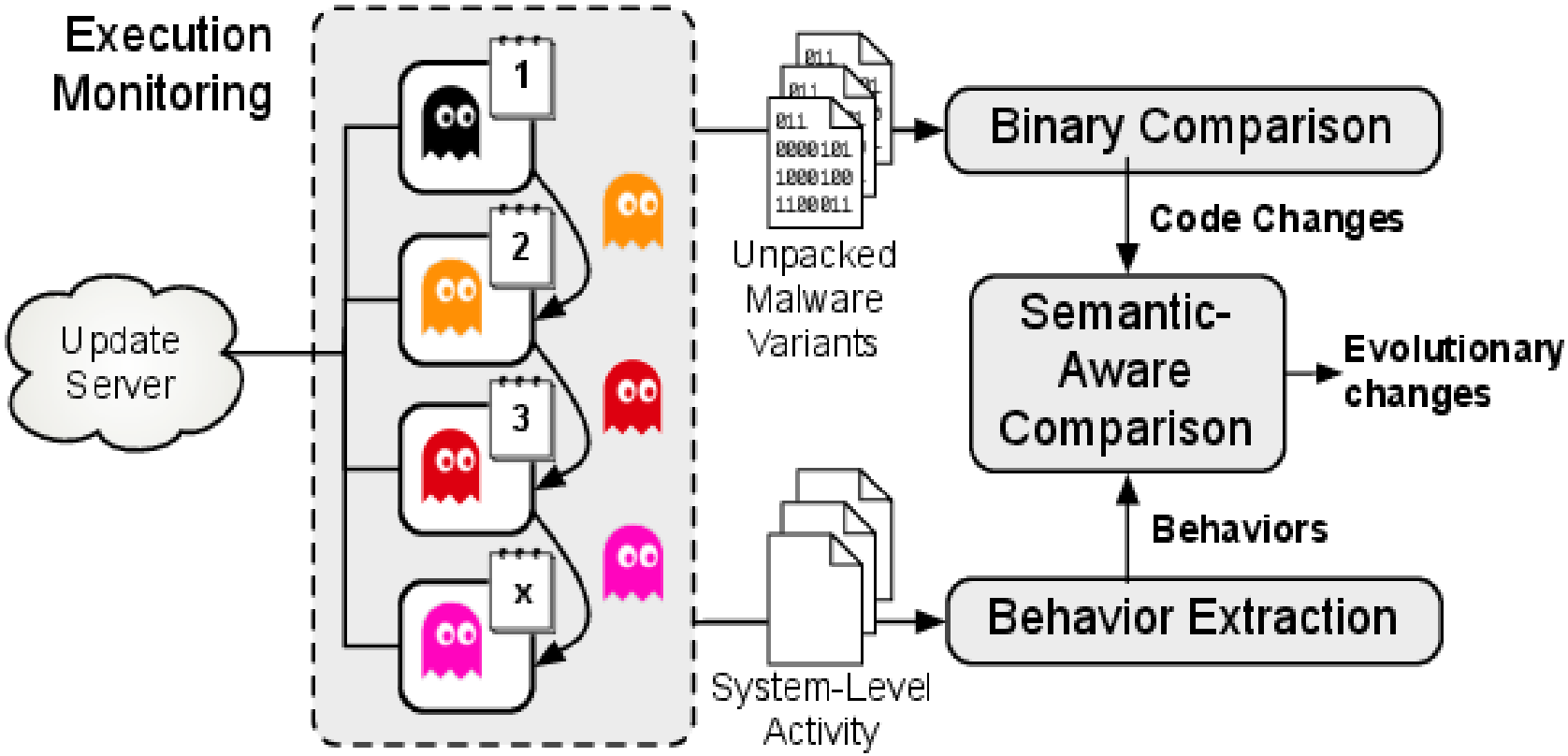
B: Behavior observed in dynamic analysis.

S,D: Functionality detected by Reanimator

Beagle

- Tracking of malware evolution over time
- Let malware update and at each step:
  - Run malware in monitored environment to see behaviors
  - Identify the code changes responsible for malicious behavior changes
- Use the same techniques of REANIMATOR for identifying and labeling behaviors, and evolutions of binary code

Beagle: overview



Beagle: how do we define a behavior

- We make use of an Anubis-like sandbox to automatically analyze system level activity
- We extract automatically graphs of connected actions that we call (unlabeled) behaviors
- We then label (some of) them manually, and can recognize with simple rules them across different samples
- This is similar to the REANIMATOR behavior signatures
- Opposed to REANIMATOR we tag code with behavior at a function-level granularity

Beagle: our dataset

Family Name and Label	Source	1st day	Days	Executions	MD5s	Lifespan
Banload TrojanDownloader:Win32/Banload.ADE	(1)	2012-01-31	87	78	3	2.00/83.00/29.33/37.95
Cycbot Backdoor:Win32/Cycbot.G	(1)	2011-09-15	73	73	69	1.00/73.00/2.04/8.60
Dapato Worm:Win32/Cridex.B	(2)	2012-02-24	65	62	25	1.00/43.00/4.60/8.31
Gamarue Worm:Win32/Gamarue.B	(2)	2012-02-10	78	77	19	1.00/76.00/8.47/16.44
GenericDownloader TrojanDownloader:Win32/Banload.AHC	(1)	2012-01-31	82	79	5	2.00/69.00/16.80/26.16
GenericTrojan Worm:Win32/Vobfus.gen!S	(1)	2012-02-07	76	73	55	1.00/44.00/2.71/6.32
Graftor TrojanDownloader:Win32/Grobim.C	(1)	2012-02-17	37	39	22	1.00/17.00/6.00/5.53
Kelihos TrojanDownloader:Win32/Waledac.C	(2)	2012-03-03	56	38	8	1.00/54.00/21.00/22.88
Llac Worm:Win32/Vobfus.gen!N	(1)	2012-02-07	32	33	82	1.00/10.00/1.49/1.71
OnlineGames Worm:Win32/Taterf.D	(1)	2011-09-02	87	80	47	1.00/38.00/3.94/7.28
ZeuS PWS:Win32/Zbot.gen!AF `1be8884c7210e94fe43edb7edebaf15f`	(3)	2012-02-09	79	78	6	1.00/78.00/26.67/28.70
ZeuS PWS:Win32/Zbot `9926d2c0c44cf0a54b5312638c28dd37`	(3)	2012-02-15	74	73	4	1.00/50.00/18.50/19.63
ZeuS PWS:Win32/Zbot.gen!AF* `c9667edbbcf2c1d23a710bb097cddbcc`	(3)	2012-02-23	66	63	6	1.00/36.00/11.00/13.43
ZeuS PWS:Win32/Zbot.gen!AF* `dbedfd28de176cbd95e1cacdc1287ea8`	(3)	2012-02-09	79	78	4	1.00/78.00/20.25/33.34
ZeuS PWS:Win32/Zbot.gen!AF* `e77797372fbe92aa727cca5df414fc27`	(3)	2012-02-10	79	77	5	1.00/77.00/16.20/30.40
ZeuS PWS:Win32/Zbot.gen!AF* `f579baf33f1c5a09db5b7e3244f3d96f`	(3)	2012-03-03	57	55	11	1.00/30.00/5.64/9.75

Table 1: Dataset. The labels in the first columns are based on Microsoft AV naming convention. The MD5 column is the number of distinct binaries encountered. Lifespan is the duration in days of the interval in which an MD5 was observed (min/max/mean/stdev).

Beagle: some global results

FAMILY NAME	%TAGGED	%LABELED	%RATIO	%ADDED	%REMOVED	%SHARED	NEW	#LABELS
Banload	7.31 ± 1.70	6.68 ± 0.75	91.43	2.48 ± 2.96	2.83 ± 3.10	94.69 ± 3.75	176.2 ± 409.2	5
Cycbot	32.36 ± 2.40	31.23 ± 2.95	96.50	10.59 ± 10.36	10.30 ± 10.42	79.11 ± 12.80	1361.4 ± 3937.2	11
Dapato	2.81 ± 1.22	1.15 ± 0.55	40.90	5.15 ± 5.14	5.57 ± 5.63	89.28 ± 7.48	2402.9 ± 7165.3	4
Gamarue	15.90 ± 14.06	14.06 ± 13.40	88.42	12.08 ± 8.16	12.50 ± 9.32	75.41 ± 11.57	2500.1 ± 7747.2	12
GenericDownloader	9.10 ± 1.93	8.58 ± 1.59	94.30	9.80 ± 9.85	9.58 ± 8.81	80.62 ± 12.48	3330.6 ± 7367.8	6
GenericTrojan	22.94 ± 11.05	20.18 ± 10.69	87.97	16.66 ± 16.15	17.03 ± 15.15	66.31 ± 18.76	4974.1 ± 14339.6	11
Graftor	12.66 ± 6.20	9.58 ± 4.70	75.70	6.47 ± 10.40	6.84 ± 9.96	86.69 ± 13.48	682.0 ± 1662.8	4
Kelihos	24.20 ± 2.24	24.09 ± 2.26	99.53	5.18 ± 8.69	5.60 ± 10.10	89.23 ± 12.64	2145.3 ± 4065.3	12
Llac	19.13 ± 14.25	19.11 ± 14.26	99.91	12.82 ± 12.53	14.45 ± 14.70	72.73 ± 19.05	3323.3 ± 7899.1	10
OnlineGames	2.18 ± 0.30	1.96 ± 0.21	89.97	3.35 ± 3.12	3.37 ± 3.12	93.28 ± 5.44	420.0 ± 718.0	9
Zeus	8.37 ± 2.59	6.15 ± 1.32	73.44	2.10 ± 2.24	3.59 ± 11.27	94.31 ± 11.28	1910.8 ± 6148.0	11
Zeus	8.26 ± 1.56	6.44 ± 1.14	78.00	3.65 ± 3.07	5.25 ± 11.85	91.09 ± 12.41	4086.0 ± 11936.3	12
Zeus	10.45 ± 2.67	7.91 ± 2.49	75.73	2.61 ± 2.20	4.51 ± 12.64	92.88 ± 12.47	2234.5 ± 7117.9	11
Zeus	8.55 ± 2.15	6.53 ± 1.19	76.41	2.55 ± 2.51	3.93 ± 11.26	93.52 ± 11.35	2013.6 ± 6874.5	12
Zeus	8.82 ± 1.79	7.73 ± 1.36	87.65	3.12 ± 2.78	4.57 ± 11.33	92.32 ± 11.46	3245.9 ± 7456.3	12
Zeus	7.44 ± 1.31	6.41 ± 0.88	86.06	2.24 ± 2.51	4.53 ± 13.46	93.23 ± 13.46	2523.9 ± 6834.9	13

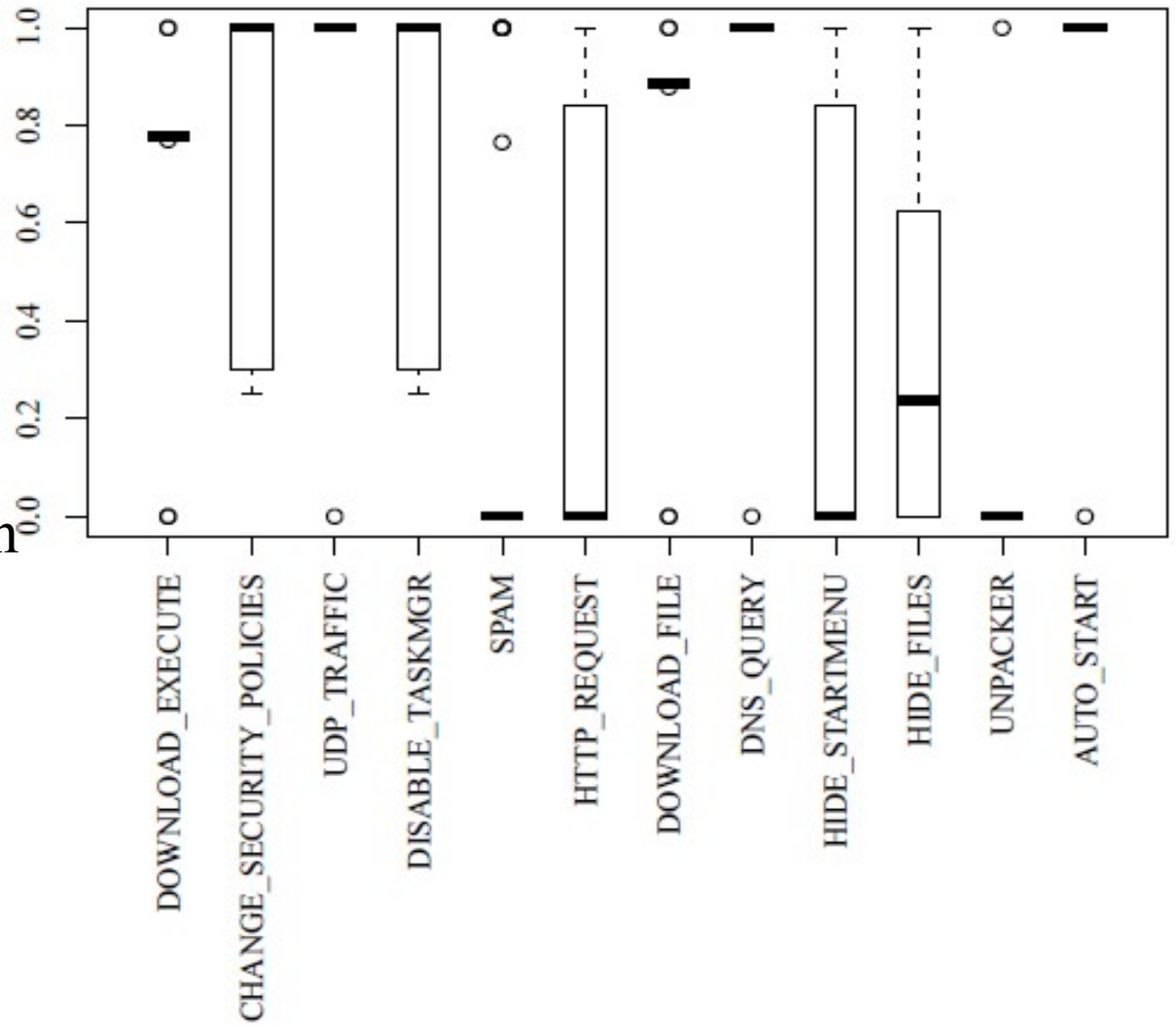
Table 2: Overall tagged and labeled code (in each version), added, removed, shared code (between consecutive versions), and new code (with respect to all previous versions) for each family (mean±variance, measured in basic blocks). #Labels is the number of distinct behavior labels detected throughout the versions.

Beagle: breakdown of changes on behaviors

Gamarue family

Distribution of similarity

Bold line = median
Box = quantiles (0,25,75,100)
Circle = outlier



Beagle: some of the insights

- Changes and evolution
  - Some families are much more actively developed than others
  - Also we can pinpoint changes over individual behaviors, sometimes across the collection
  - In some cases, overall development appears constant/low, but we can disaggregate it to significant changes
- Effort
  - We have blocks in ASM, not LoC in source, but we can do some estimate
  - We estimate that avg added code in Zeus over each variation is 140–280 LoC, with peaks up to 9,000
  - Roughly holds for other families but we are less certain
  - Significant effort of development in malware

Next step: automatic extraction of malicious behaviors from large datasets of malware

- Reanimator and Beagle rely on manually-identified relevant behaviors
  - This is not a dramatic requirement as we saw
  - Still, tedious manual step we may wish to avoid
  - Biased from analyst's previous perspective, not receptive to novelty
- We are working to demonstrate that we can:
  - Automatically extract relevant “sets-of-calls” that might be behaviors (done: grouped call by dataflow dependency)
  - Match this dictionary of unlabeled behaviors across different variants (in fieri)
  - Try to associate as much semantic information as possible to these “emerging” behaviors before presenting them to human analysts (todo)

Next step: Classifying malware by structural and behavioral features

Several works perform either:

- Structural clustering based on code features (e.g. works by H. Flake, Ero Carrera, and others)
- Behavioral clustering based on program execution traces (e.g. works by P. M. Comparetti, C. Kruegel, and others)

Our next research: using the same backward-forward techniques we used in the previous 2 works to map these two clustering approaches to each other. This will improve the quality of the families, help cluster correctly malware which is obfuscated or which has dormant behaviors

Conclusions

- Structural analysis alone is too time and brain consuming
- Dynamic analysis alone has too many blind points
- We can combine both to obtain:
  - Dormant code analysis and tagging
  - Evolution tracking
  - Triage of new samples
  - (hopefully) better means of classifying specimens in families
- Much work needs to be done in this area
  - Automatic identification of behaviors
  - Using these insights to automatically generate a sensible classification of malware into families

Thanks for your attention!

stefano.zanero@polimi.it
@raistolo

Most of the work presented was/is joint work with:

UCSB – Christopher Kruegel

Lastline – Paolo Milani Comparetti

Northeastern University – Engin Kirda

Technical University of Vienna – Martina Lindorfer

Politecnico di Milano - Federico Maggi, Alessandro di Federico, Guido Salvaneschi, Mario Polino, Andrea Scorti

Of course, errors and opinions are mine solely :-)

Research partially funded by the European Commission under FP7 project SysSec