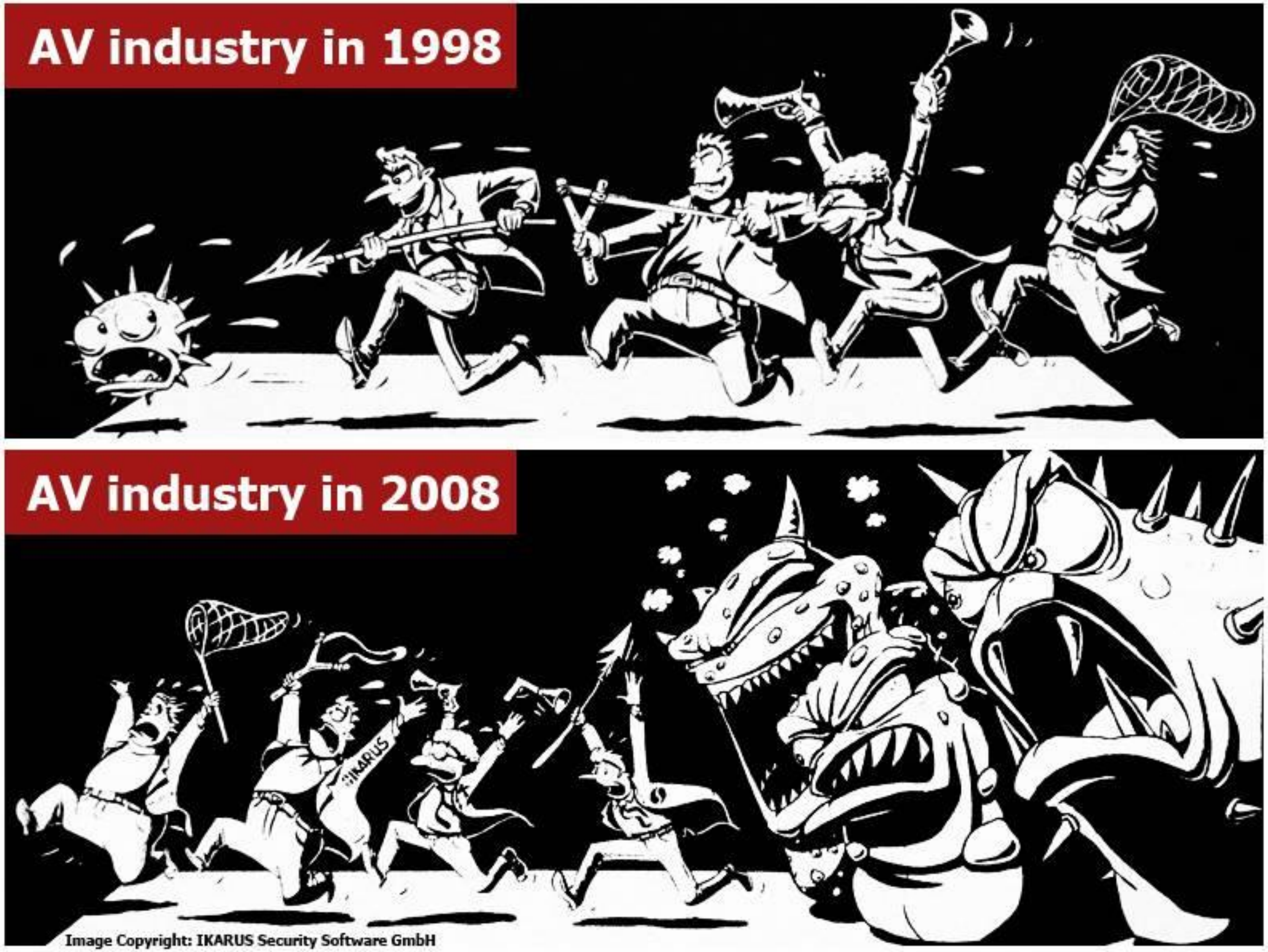
Lines of Malicious Code:
Insights Into the Malicious Software Industry

Martina Lindorfer — Vienna University of Technology
Alessandro Di Federico — Politecnico di Milano
Federico Maggi — Politecnico di Milano
Paolo Milani Comparetti — Vienna University of Technology, Lastline Inc.
Stefano Zanero — Politecnico di Milano


AV industry in 1998
AV industry in 2008
IKARUS
Image Copyright: IKARUS Security Software GmbH

State of Malware

- Underground economy of cybercrime:
  spam, identity theft, DoS, Fake AV scams, …
- Malicious software industry
- Arms race against security researchers
- Overwhelming amount of samples
  - > 70,000/day in 2011 (PandaLabs)
- Need for analysis automation
- Limits of static/dynamic analysis
- Incremental updates of functionality
- Focus manual analysis on novel functionality

Approach (1/2)

- Identify focus of development effort of malware authors
- Take advantage of auto-update functionality in malware
- Collect subsequent updates of malware variants
- Identify code changes between versions
- Identify evolution of functional components
  - e.g. spam, Fake AV
- Estimate development effort
- Highlight significant code changes for further analysis

Approach (2/2)

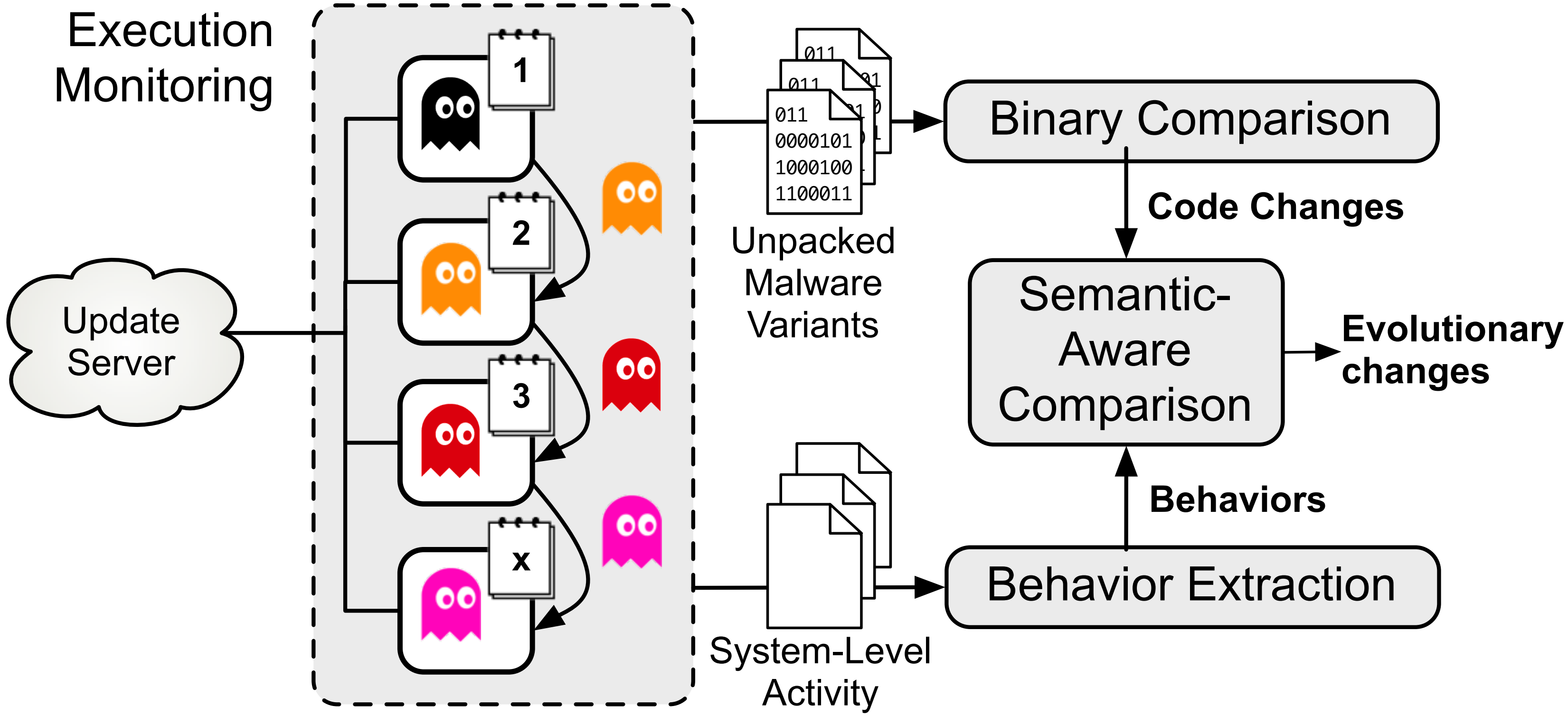
- Combination of static and dynamic analysis
- Builds upon REANIMATOR (Oakland 2010)
  - “Identifying Dormant Functionality in Malware Programs”
- Run samples in sandbox
- Let samples connect to the C&C server to update
- Find differences in binary code
- Map differences in binary code to behavior

- BEAGLE
  - 16 malware samples from 11 families
  - > 1,000 executions, 381 distinct binaries

Outline

- **BEAGLE**
  - Step 1: Execution Monitoring
  - Step 2a: Binary Comparison
  - Step 2b: Behavior Extraction
  - Step 3: Semantic-Aware Comparison
- Experimental Results
- Conclusion

BEAGLE



Execution Monitoring
Update Server
1
2
3
x
011
0000101
1000100
1100011
Unpacked Malware Variants
System-Level Activity
Binary Comparison
Code Changes
Semantic-Aware Comparison
Evolutionary changes
Behaviors
Behavior Extraction

Step 1: Execution Monitoring

- Based on Anubis sandbox
  - Logging of Native + Windows API, dynamic taint tracing
- Stateful analysis:
  - Save analysis state (filesystem and registry changes)
  - Restore analysis state
  - Invoke persistence mechanism
- Logging of call stack for each API call
- Generic unpacker (dump memory)
- Output:
  - Unpacked binaries
  - System calls and taint dependencies

Step 2a: Binary Comparison

- Input:
  - Unpacked malware variants

- Preprocessing: Code whitelisting
  - Generic unpacker dumps all memory
  - Includes code injected into benign processes
  - Includes DLLs loaded into malware's address space
  - Identify all code (EXE and DLL) from the clean image and ignore it

Step 2a: Binary Comparison

- Refined techniques of Kruegel et al. (RAID 2005)
  - "Polymorphic Worm Detection Using Structural Information of Executables"
- Color nodes in CFG based on classes of instructions
- Shared code = finding isomorphic k-node subgraphs
- Fingerprints = hash of normalized subgraphs
- Match fingerprints between malware versions
- Output:
  - Shared/added/removed basic blocks
  - Measure of code change (Jaccard Similarity):
    # of shared BB over the total shared/added/removed BBs

Step 2b: Behavior Extraction

- Input:
  - System calls and taint dependencies from dynamic analysis
- Behavior = connected graph of system-level events
  - Nodes = system calls
  - Edges = data flow dependencies
- Define rules to detect high-level behaviors
  - e.g. Download & Execute = data flow from network to a file that is later executed
  - Unlabeled: no high-level meaning
  - Labeled: behavior matches known patterns
- Output:
  - List of behaviors with responsible code

Step 3: Semantic-Aware Comparison

- Input:
  - Labeled & unlabeled behaviors
  - Shared/added/removed BBs

- Map behavior to code
  - Dynamic analysis at system call level
  - Better scaling than instruction-level tracing
  - Mapping at function-level granularity
  - Locate function boundaries of addresses in call stack

Step 3: Semantic-Aware Comparison

- Expansion of mapping:
  - Statically identify code path between individual system calls
  - Use call stack for each system call as landmark

- Dormant functionality:
  - Locate fingerprints from active components in other executions

- Output:
  - Evolutionary changes in functional components

Outline

- BEAGLE
  - Step 1: Execution Monitoring
  - Step 2a: Binary Comparison
  - Step 2b: Behavior Extraction
  - Step 3: Semantic-Aware Comparison
- **Experimental Results**
- Conclusion

Dataset (1/2)

- 16 samples (11 families, 6 ZeuS)
- Sources:
  - ZeuS Tracker
  - Anubis (download & execute heuristics)
  - Top threats from Microsoft Malware Protection Center

- September 2011 - April 2012
- 15 minutes each, once a day
- 1,023 executions of 381 distinct binaries

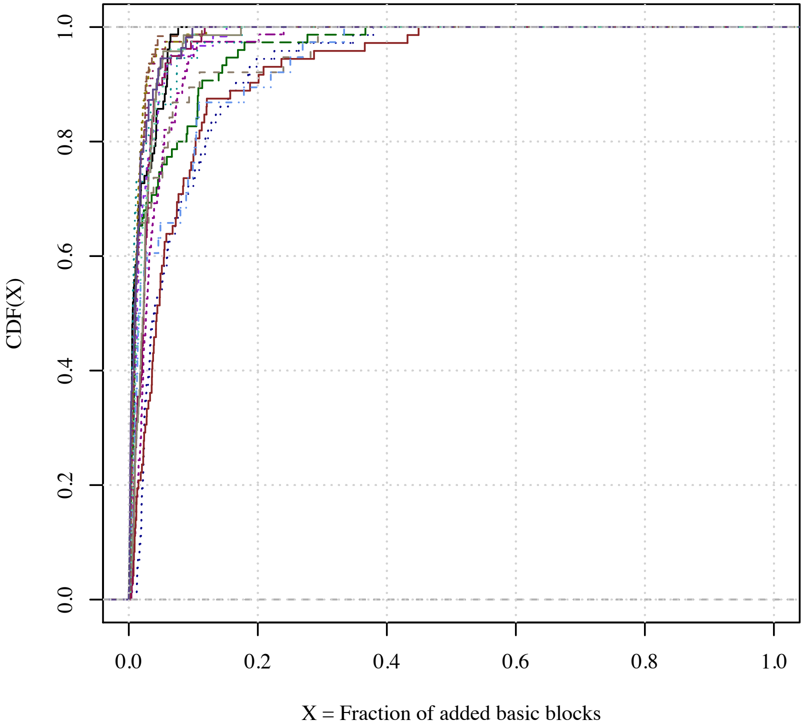
Dataset (2/2)

Family Name and Label	Source	1st Day	Days	Executions	MD5s
Banload TrojanDownloader:Win32/Banload.ADE	(1)	2012-01-31	87	78	3
Cycbot Backdoor:Win32/Cycbot.G	(1)	2011-09-15	73	73	69
Dapato Worm:Win32/Cridex.B	(2)	2012-02-24	65	62	25
Gamarue Worm:Win32/Gamarue.B	(2)	2012-02-10	78	77	19
GenericDownloader TrojanDownloader:Win32/Banload.AHC	(1)	2012-01-31	82	79	5
GenericTrojan Worm:Win32/Vobfus.gen!S	(1)	2012-02-07	76	73	55
Graftor TrojanDownloader:Win32/Grobim.C	(1)	2012-02-17	37	39	22
Kelihos TrojanDownloader:Win32/Waledac.C	(2)	2012-03-03	56	38	8
Llac Worm:Win32/Vobfus.gen!N	(1)	2012-02-07	32	33	82
OnlineGames Worm:Win32/Taterf.D	(1)	2011-09-02	87	80	47
ZeuS PWS:Win32/Zbot.gen!AF `1be8884c7210e94fe43edb7edebaf15f`	(3)	2012-02-09	79	78	6
ZeuS PWS:Win32/Zbot `9926d2c0c44cf0a54b5312638c28dd37`	(3)	2012-02-15	74	73	4
ZeuS PWS:Win32/Zbot.gen!AF* `c9667edbbcf2c1d23a710bb097cddbcc`	(3)	2012-02-23	66	63	6
ZeuS PWS:Win32/Zbot.gen!AF* `dbedfd28de176cbd95e1cacdc1287ea8`	(3)	2012-02-09	79	78	4
ZeuS PWS:Win32/Zbot.gen!AF* `e77797372fbe92aa727cca5df414fc27`	(3)	2012-02-10	79	77	5
ZeuS PWS:Win32/Zbot.gen!AF* `f579baf33f1c5a09db5b7e3244f3d96f`	(3)	2012-03-03	57	55	11

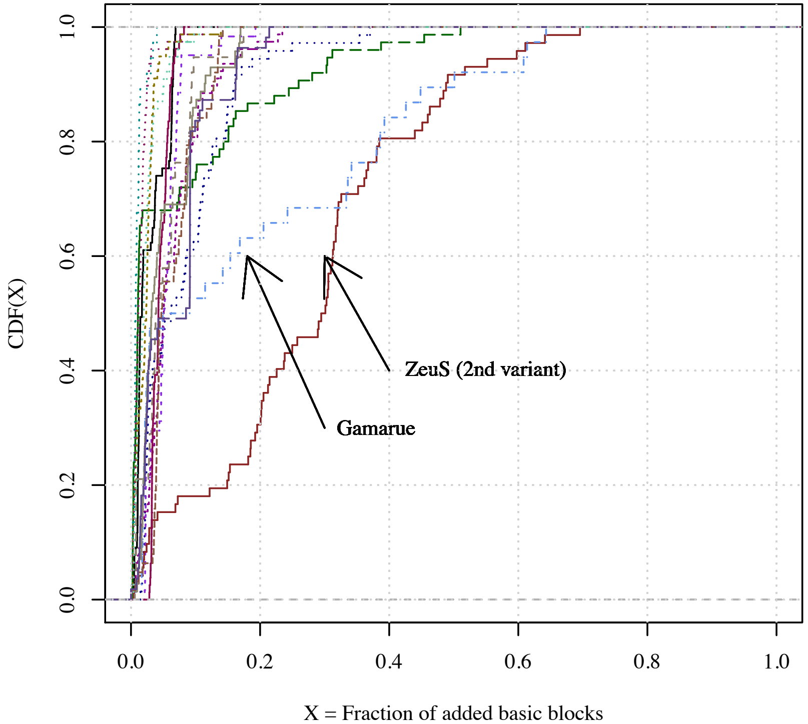
Behaviors in Dataset

HTTP REQUEST
DOWNLOAD FILE
CHANGE SECURITY POLICIES
HIDE FILES DOWNLOAD EXECUTE
DNS QUERY EMAIL HARVESTING
EXECUTE TEMP FILE FTP CREDENTIALS
DISABLE TASKMGR SPAM OPEN PORT
INJECT CODE AUTO RUN HTTPS REQUEST
INTERNET HISTORY BITCOIN WALLET FIREFOX SETTINGS
UDP TRAFFIC IE SECURITY SETTINGS
TCP CONNECTION
LOCAL CONNECTION START SERVICE
RESTORE SSDT DOWNLOAD INJECT
IE PROXY SETTINGS AUTO START
REMOVE FLASHPLAYER FILES HIDE STARTMENU
FIREWALL SETTINGS
ENUMERATE PROCESSES

Overall Code Changes

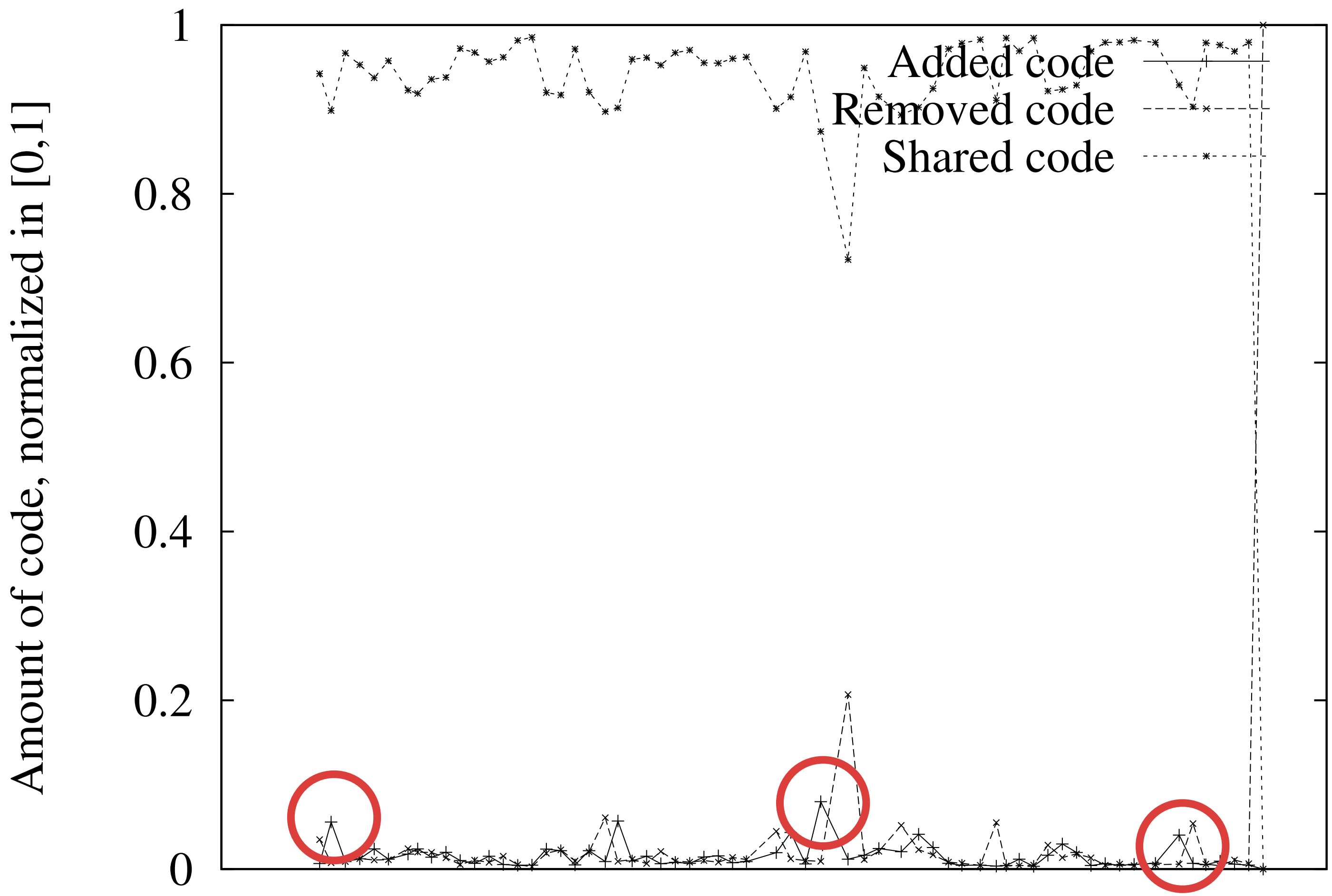


(a) $t - 1$ vs. t

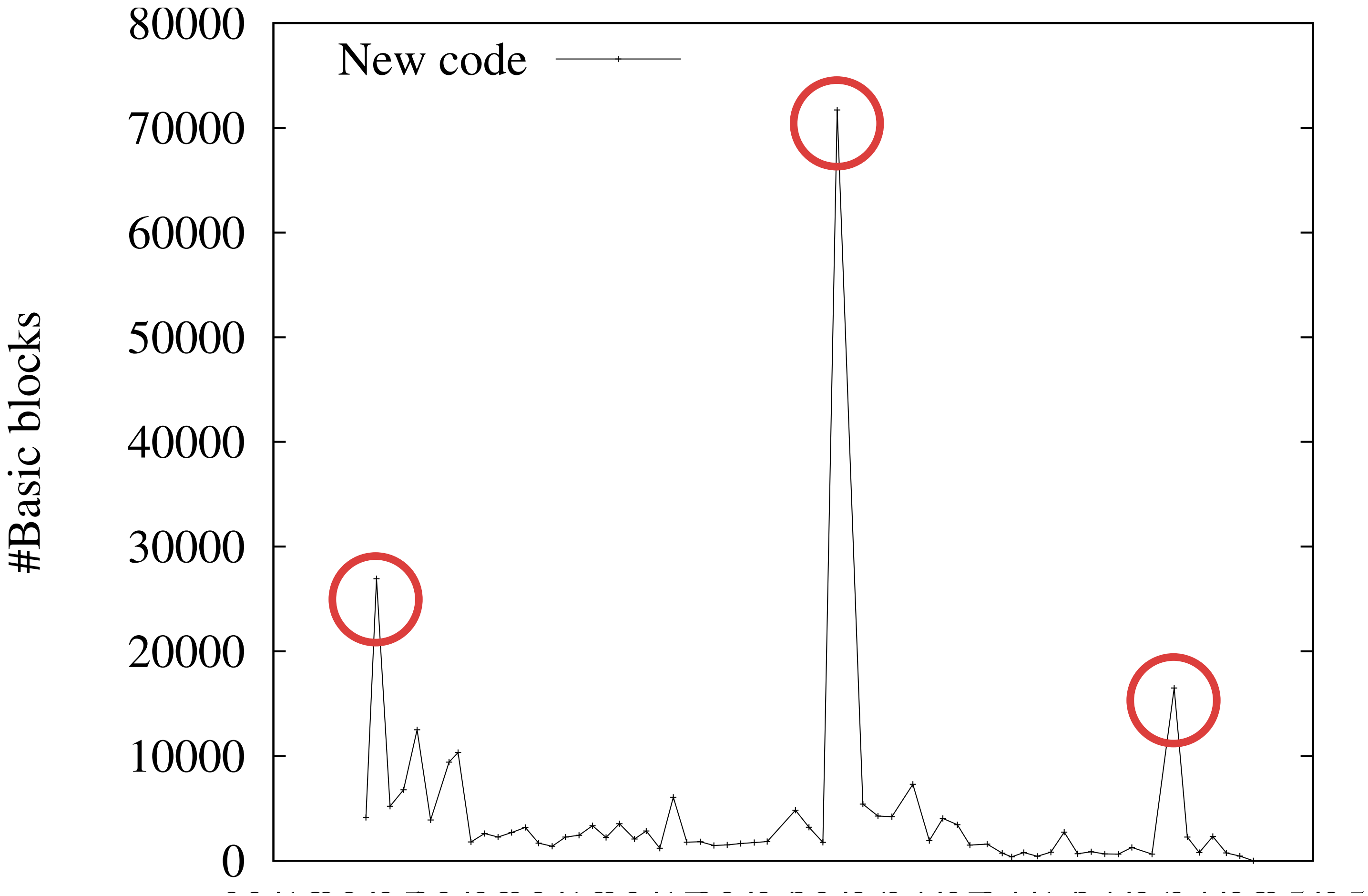


(b) t_0 vs. t

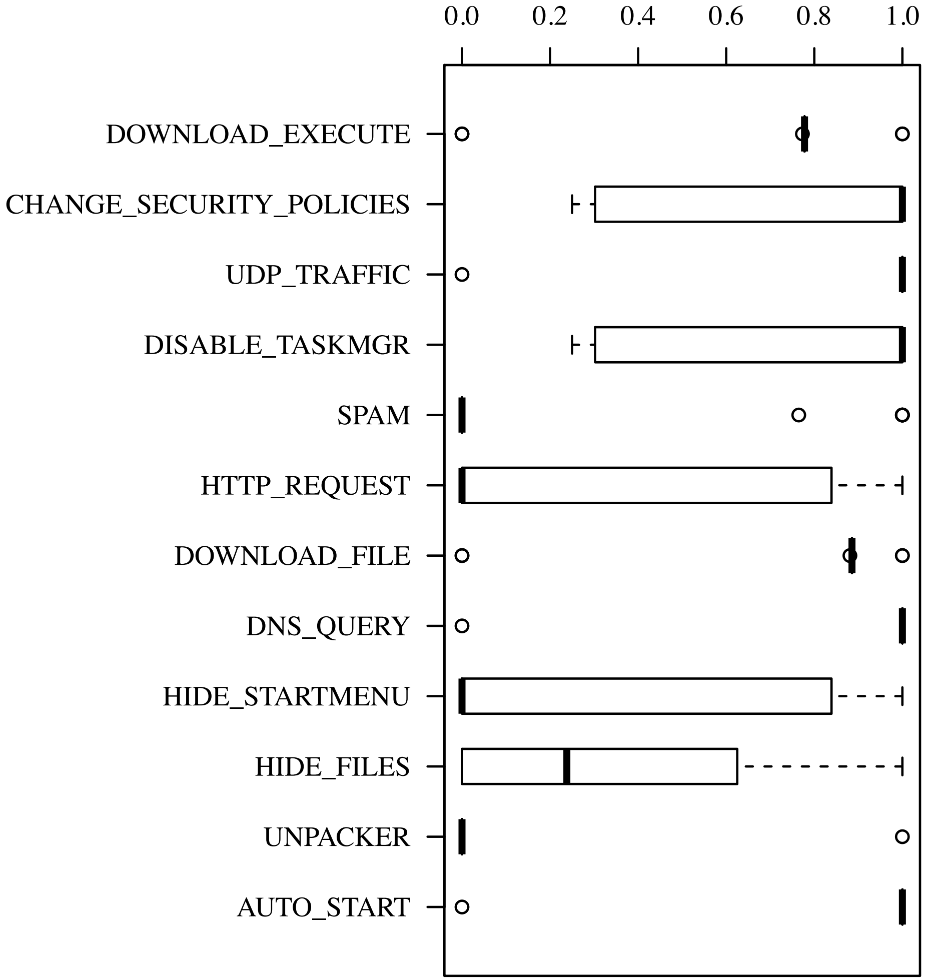
Code Changes: Zeus



Code Changes: Zeus



Behavior Evolution: Gamarue



Evaluation Results

- Core insights
  - Frequency of code changes
  - Most actively developed components
  - Overall amount of development effort

- Some families more actively developed than others
- Incremental updates reuse most of the code
- Peaks of new code added
- Pinpoint changes over individual behaviors
- Pinpoint changes over the whole dataset

Lines of Malicious Code

- Estimation of development effort:
  - Amount of source code for observed changes
- Blocks of ASM, not LoC in source
- ZeuS + 150 bots with source code:
  - 50-100 LoC/basic block
  - 14.64 LoC/basic block for ZeuS
- Significant effort of development in malware
  - Zeus: 140-180 new (peak 9,000) LoC
  - Other: 100-300 new (peak 4,600-9,000) LoC

Outline

- BEAGLE
  - Step 1: Execution Monitoring
  - Step 2a: Binary Comparison
  - Step 2b: Behavior Extraction
  - Step 3: Semantic-Aware Comparison
- Experimental Results
- **Conclusion**

Limitations

- Unpacking (multi-layer or emulation-based packing)
- Dynamic analysis evasion
- Limited code coverage
- Semantics of code changes (human analysis)

- Future work:
  - Patch analysis techniques to understand how the update of a component changes the functionality
  - Automatic classification of high-level behaviors

Conclusion

- Combination of static and dynamic analysis to track evolution of malware
- Measure code changes between malware versions
- Associate observed behavior with implementing components
- Measure evolution of individual components
- Highlight interesting code changes for manual inspection
- Insights on the development efforts in malicious code

Questions?

mlindorfer@iseclab.org

http://www.iseclab.org/people/mlindorfer