Lines of Malicious Code: Insights Into the Malicious Software Industry

Martina Lindorfer  
Alessandro Di Federico  
Federico Maggi  
Paolo Milani Comparetti  
Stefano Zanero  

Vienna University of Technology  
Politecnico di Milano  
Vienna University of Technology, Lastline Inc.  
Politecnico di Milano
AV industry in 1998

AV industry in 2008

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

Binary Comparison

Unpacked Malware Variants

Semantic-Aware Comparison

Code Changes

Evolutionary changes

Behaviors

Behavior Extraction

System-Level Activity
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
We tag all functions in the identified path as part of the behavior.

We use the addresses in the call stack as landmarks this path should

way, we use the CFG fingerprints from §4.2 to identify a compo-

we select the largest implementation by number of basic blocks and

we apply the techniques discussed in §4.2 to successive versions of a

component, instead of considering entire unpacked binaries. Among

new code, as well as quickly identifying significant updates to the

functionality over time. This allows us to get an idea of the development effort

of a malware, we can observe the evolution of that functionality

the components that implement a behavior in successive versions

is a functional component of a malware instance. By comparing

Behavior Evolution.

Previous work that performed a similar mapping of behavior to code

in §4.1, our modified sandbox logs events at the system API level.

Like any dynamic code analysis approach,

incomplete code coverage. In a typical

Like any dynamic code analysis approach,

random evolution. In the limit, if a behavior is observed only once, we do not

out our experiments. This limits our visibility in the component's

the versions of a component observed throughout our experiments,

is a functional component of a malware instance. By comparing

Behavior Evolution.

Previous work that performed a similar mapping of behavior to code

in §4.1, our modified sandbox logs events at the system API level.

Like any dynamic code analysis approach,

random evolution. In the limit, if a behavior is observed only once, we do not

out our experiments. This limits our visibility in the component's

the versions of a component observed throughout our experiments,

is a functional component of a malware instance. By comparing

Behavior Evolution.
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
We tag all functions in the identified path as part of the behavior. We use the addresses in the call stack as landmarks this path should traverse and in case the dynamic path cannot be resolved statically. We select the largest implementation by number of basic blocks and apply the techniques discussed in §4.2 to successive versions of a component even in executions where it is dormant and the corresponding behavior cannot be observed. For this, we identify the dormant component from an off some precision in delimiting functional components, to achieve a limitation of reference behavior.

Table 1: Dataset. The labels in the first columns are based on Microsoft AV naming convention. The MD5 column is the number of variants from the top threats according to the Microsoft Malware Protection Center [2] (3) ZeuS samples from ZeuS Tracker [3].

We run on a desktop-class, dual-core machine with 4GB RAM, and execute each sample for 15 minutes approximately. Working at function granularity is a design decision that trades off some precision in delimiting functional components, to achieve a limitation of reference behavior.

Previous work that performed a similar mapping of behavior to code and the corresponding components by locating the functions in a sample that match fingerprints from an active component even in executions where it is functional component only in some of a sample's executions, even though the code implementing the functionality is present throughout. By comparing the versions of a component observed throughout our experiments, this allows us to get an idea of the development effort changes (a) are concentrated around low values for all the families, showing different development efforts.

We selected samples from three different sources: (1) Recent submissions to Anubis for which the data flow detection of Jackstraws [17] indicated download & execute behavior. (2) Malware variants from the top threats according to the Microsoft Malware Protection Center [2] (3) ZeuS samples from ZeuS Tracker [3].

As summarized in Tab. 4.3, we analyzed the evolution of 16 samples from the top threats. (1) 2011-09-02 87 80 47 1.00/38.00/3.94/7.28. (2) 2012-02-07 32 33 82 1.00/10.00/1.49/1.71. (3) 2012-02-10 79 77 5 1.00/77.00/16.20/30.40. (4) 2012-02-09 79 78 6 1.00/78.00/26.67/28.70. (5) 2011-09-15 73 73 69 1.00/73.00/2.04/8.60. (6) 2012-01-31 82 79 5 2.00/69.00/16.80/26.16. (7) 2012-03-03 57 55 11 1.00/30.00/5.64/9.75. (8) 2012-02-03 56 38 8 1.00/54.00/21.00/22.88. (9) 2011-09-23 1 1 1 1.00/1.00/1.00/1.00. (10) 2012-03-02 42 42 1 1.00/1.00/1.00/1.00. (11) 2012-01-30 1 1 1 1.00/1.00/1.00/1.00. (12) 2012-02-08 42 42 1 1.00/1.00/1.00/1.00. (13) 2012-03-01 42 42 1 1.00/1.00/1.00/1.00. (14) 2012-02-06 42 42 1 1.00/1.00/1.00/1.00. (15) 2012-02-04 42 42 1 1.00/1.00/1.00/1.00. (16) 2012-02-05 42 42 1 1.00/1.00/1.00/1.00.
Code Changes: Zeus

Amount of code, normalized in [0,1]
Behavior Evolution: Gamarue

![Behavior Evolution: Gamarue](image_url)

---

**Download File**: [Download File](DOWNLOADFILE)

**UDP Traffic**: [UDP_TRAFFIC]

**Download Execute**: [DOWNLOAD_EXECUTE]

**Change Security Policies**: [CHANGE_SECURITY_POLICIES]

**Disable TaskMGR**: [DISABLE_TASKMGR]

**Spam**: [SPAM]

**HTTP Request**: [HTTP_REQUEST]

**Download File**: [DOWNLOAD_FILE]

**DNS Query**: [DNS_QUERY]

**Hide StartMenu**: [HIDE_STARTMENU]

**Hide Files**: [HIDE_FILES]

**Unpacker**: [UNPACKER]

**Auto Start**: [AUTO_START]

---

Annual Computer Security Applications Conference, December 2012
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