# Identifying Dormant Functionality in Malware Programs

| | |
|---|---|
| Paolo Milani Comparetti | Vienna University of Technology |
| Guido Salvaneschi | Polotecnico di Milano |
| Engin Kirda | Institute Eurecom |
| Clemens Kolbitsch | Vienna University of Technology |
| Christopher Kruegel | UC Santa Barbara |
| Stefano Zanero | Politecnico di Milano |

# Motivation



- Malicious code (malware) at the root of many internet security problems
  - ~50000 new samples each day!
- Automated dynamic analysis
  - run samples in an instrumented sandbox
- Dynamic analysis provides limited coverage
  - different behavior based on commands from C&C channel
- How can we learn more about malware samples?

# Our Approach



- Leverage code reuse between malware samples
- Automatically generate semantic-aware models of malicious behavior
  - based on 1 execution of a behavior
  - model 1 implementation of the behavior
- Use these models to statically detect the malicious functionality in samples that do not perform that behavior during dynamic analysis

# 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 binaries

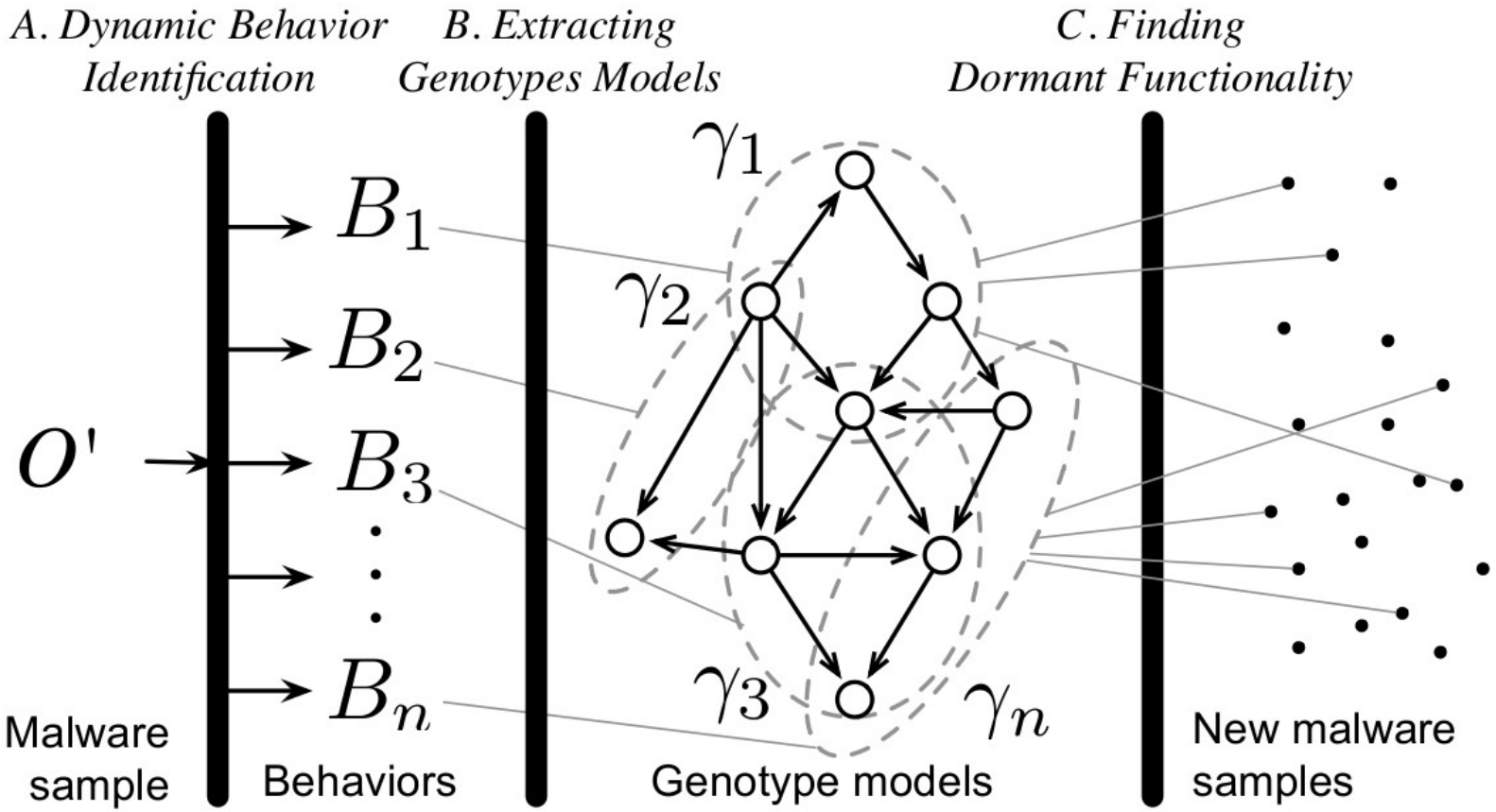
# Outline



- **REANIMATOR**: Identifying dormant functionality
  - Dynamic behavior identification
  - Extracting genotype models
  - Finding dormant functionality
- Evaluation
- Conclusions

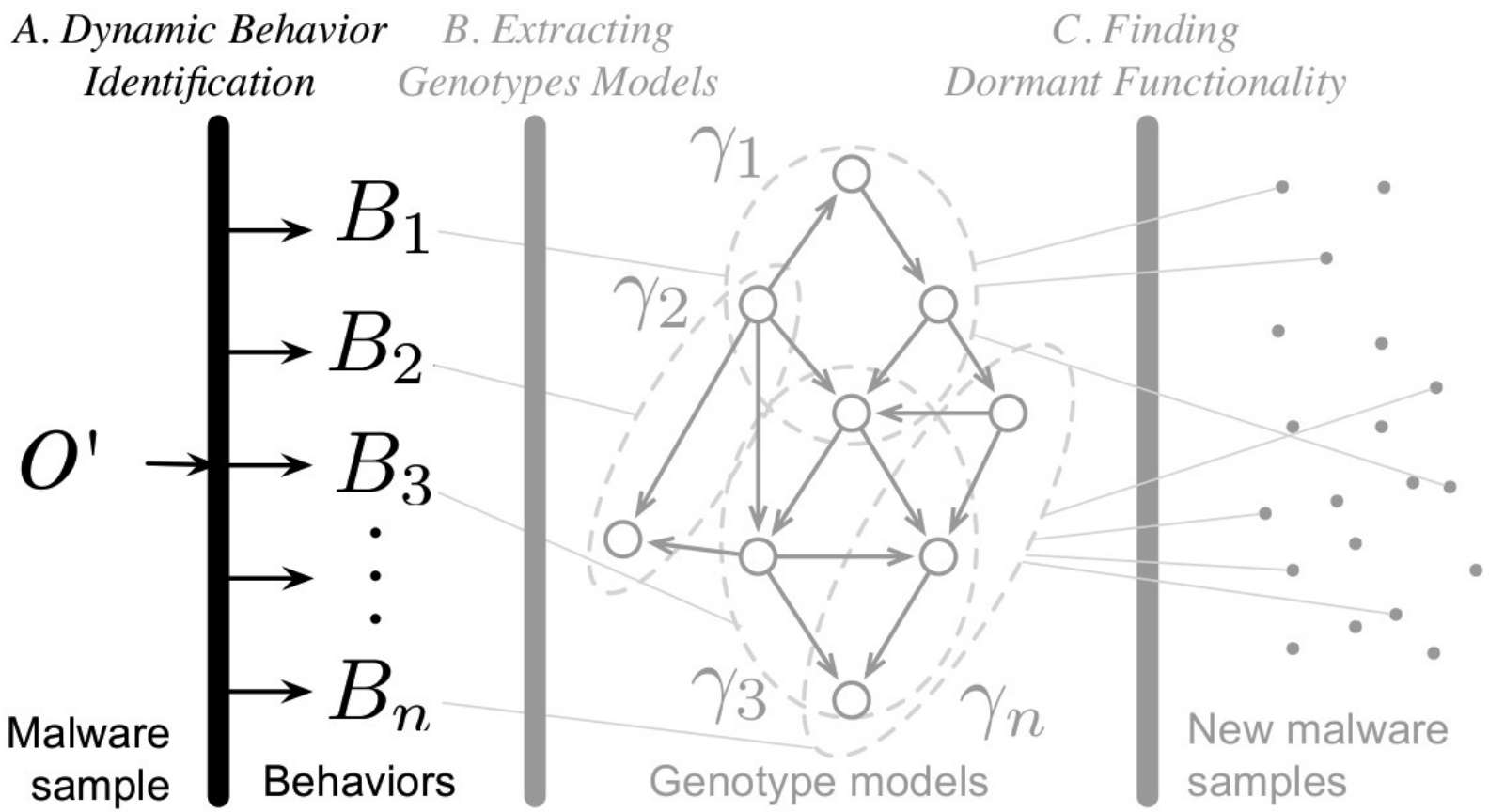
# REANIMATOR




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

- 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

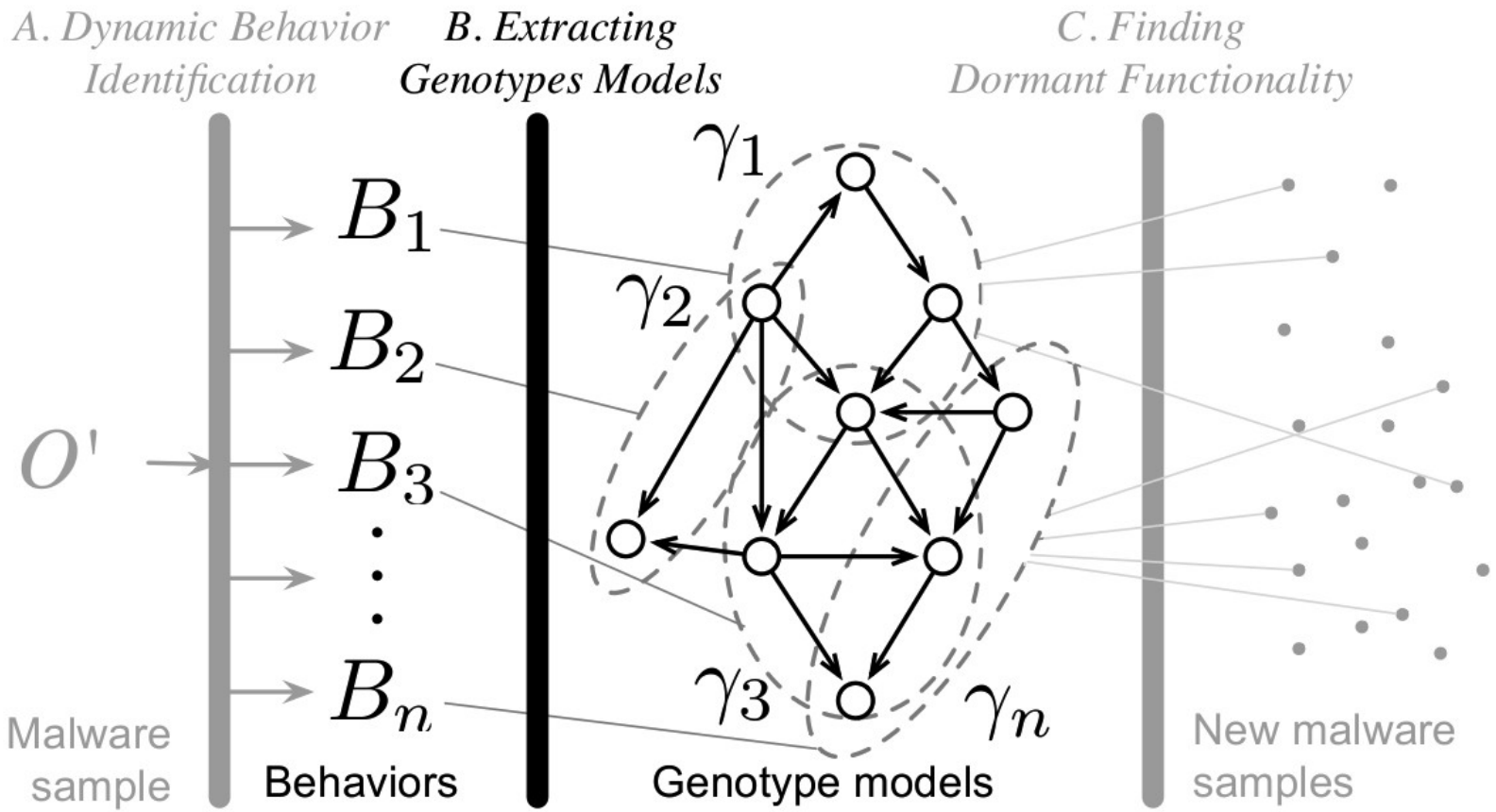
# Behavior Detection Examples



- **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



- Take as input the set $R_B$ of relevant system/API calls
- **Identify** the code responsible for behavior B *(genotype)*
- **Model** the code responsible for behavior B *(genotype model)*
- The genotype model can then be statically, efficiently used for detecting the corresponding genotype and phenotype in other binaries that did not perform B during dynamic analysis

# 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,..)

# Extracting Genotype Models: Steps



- Slicing:
  - obtain an initial set of instructions (genotype) ϕ that are related to $R_B$
- Filtering:
  - increase the precision of the genotype by removing from ϕ instructions that are not specific to B
- Germination:
  - increase the completeness of the model by adding instructions to ϕ

# Step 1: 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)

# Step 2: Filtering



- The slice ϕ is not precise
- General purpose utility functions executed as part of behavior are 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

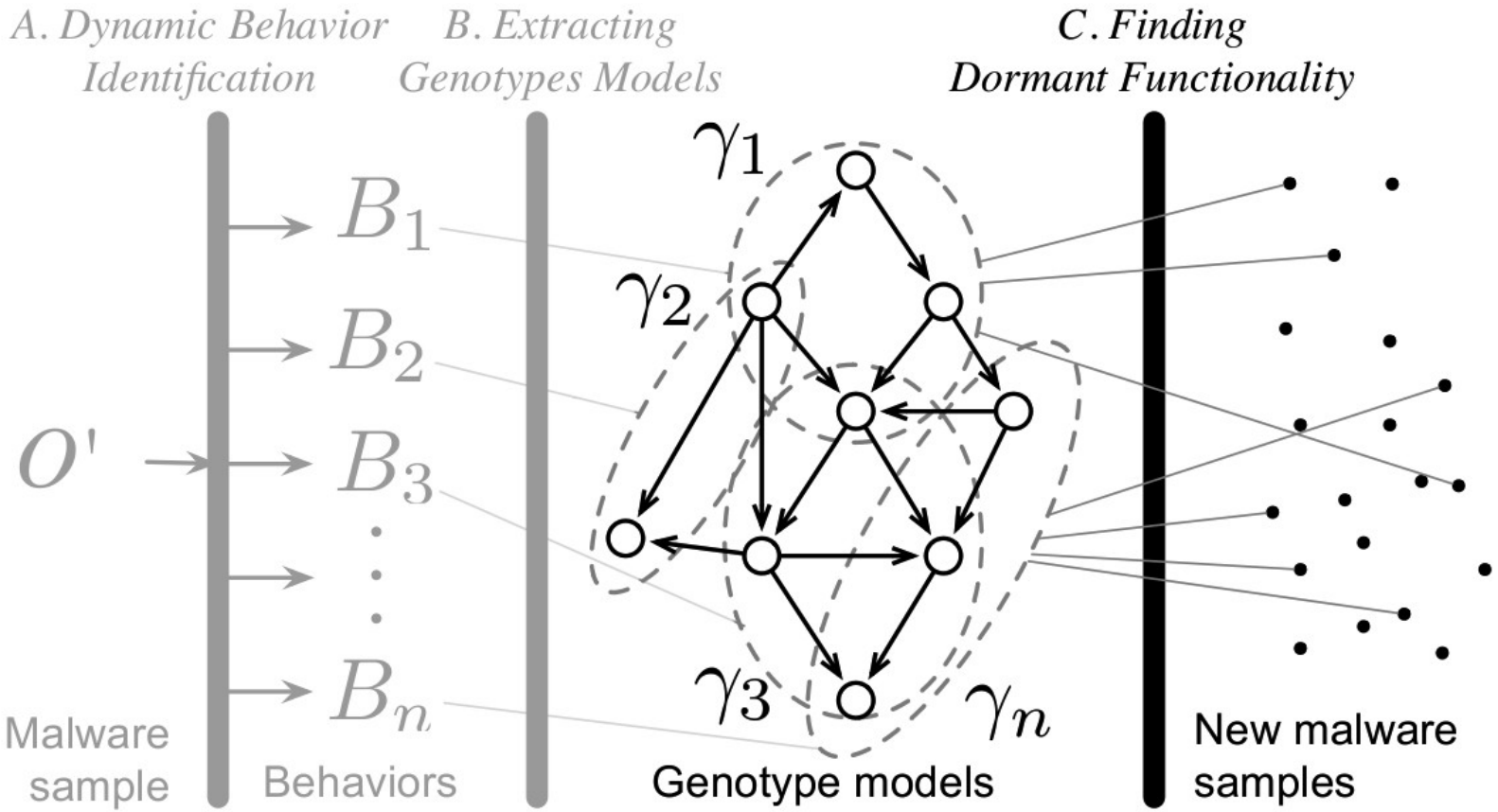
# Step 3: Germination



- 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 a K-Node subgraph (K=10)
- Use techniques from our previous work [1] to efficiently match a binary against a set of genotype models
- We use Anubis as a generic unpacker 

[1] "Polymorphic Worm Detection Using Structural Information of Executables", RAID 2005

# Evaluation

# Evaluation



- Extract genotype models from a sample
- Match these genotypes against other samples
- 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 bots with source code
  - thanks to Jon Oberheide and Michael Bailey from University of Michigan
- Extract 6 genotype models from 1 bot
- Match against remaining 207 bot binaries

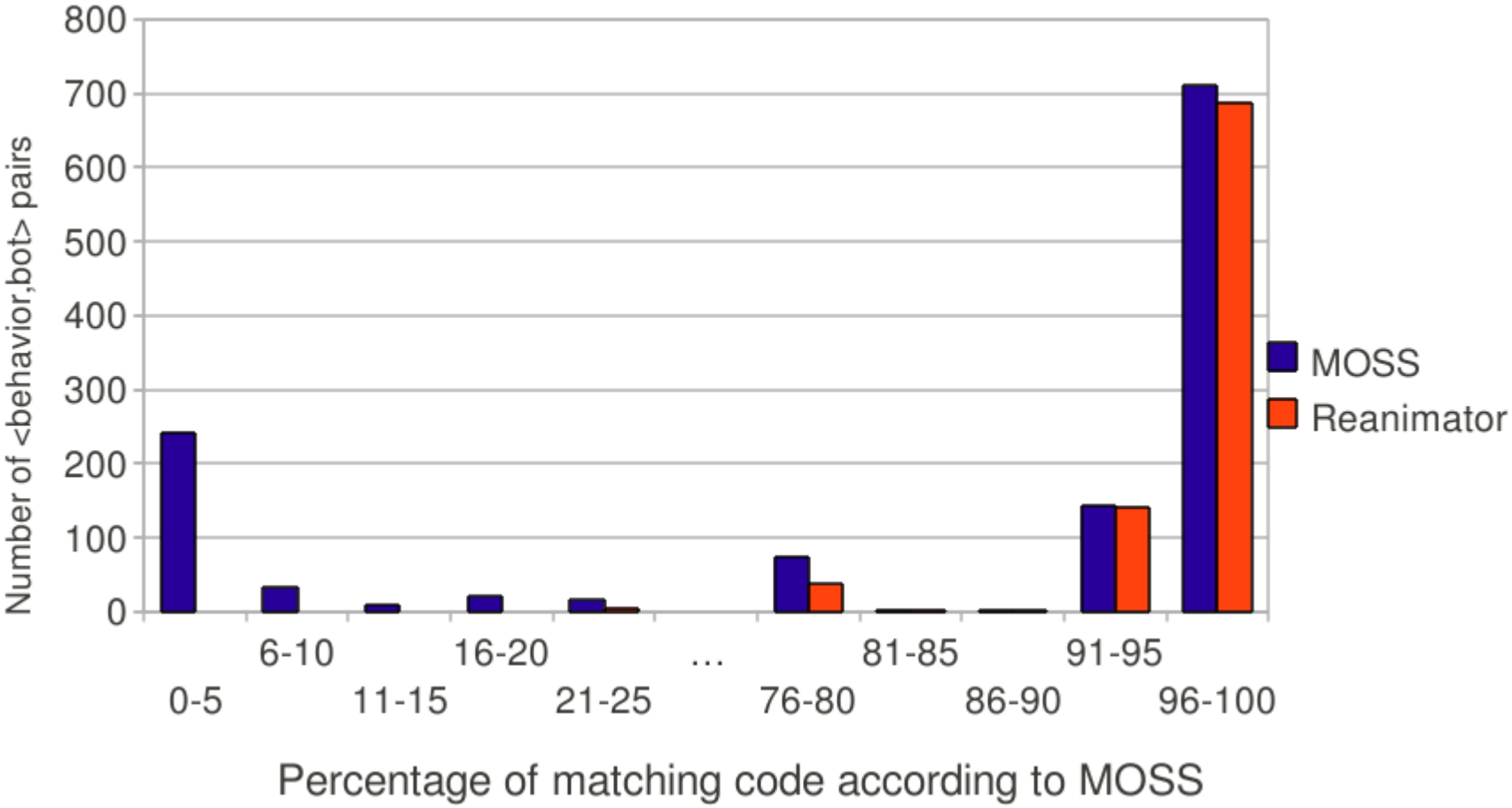
# Accuracy



- 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 bot sources
  - MOSS returns a similarity score in percentage
- We expect **REANIMATOR** to match in cases where MOSS returns high similarity scores

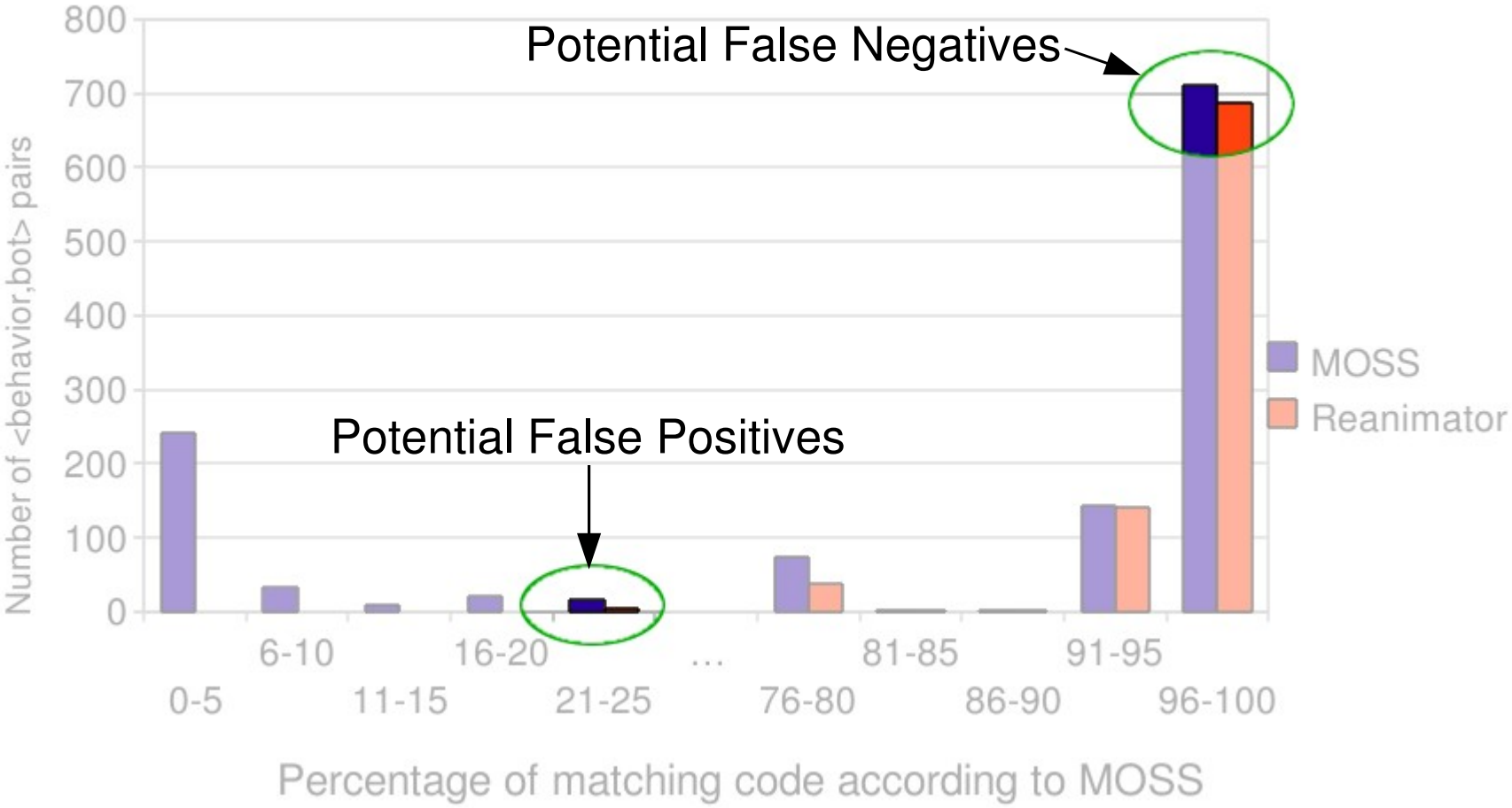
# MOSS Comparison





# 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



- 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)
  - Visual Studio vs. Intel
- Some robustness to malware metamorphism was demonstrated in [1]

# 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

# Conclusions



- Identify security-relevant behavior during dynamic analysis of a malware sample
- Automatically identify and model the code that is responsible for that behavior
- Use these models to statically detect similar code in other samples
- Our experiments demonstrate accuracy and robustness
- Testing against in-the-wild datasets shows improved detection of malicious functionality

# Questions?