Efficient Program Exploration by Input Fuzzing

towards a new approach in malcious code detection

Guillaume Bonfante   Jean-Yves Marion   Ta Thanh Dinh

Université de Lorraine

CNRS - INRIA Nancy

First Botnet Fighting Conference, 2013

Table of Contents

Context

Host-based botnet detection

- The bot need communicate with the bot-master:
  - receives special commands: does malicious things,
  - otherwise: stays inactive.
- In general: trigger-based malwares.

Real-life infamous examples

- Stuxnet: "... checks the value NTVDM TRACE... If this value is equal to... infection will not occur..." Falliere et al. 2011
- Gauss: "... decrypt... the payload using several strings from the system and, upon success, executes it... " GReAT 2013

Researches on the code coverage

Code coverage is considered

- extensively on the source code of programs (Godefroid et al. 2005 and numerous subsequent works).
- but much fewer if one considers
  - binary codes,
  - malicious obfuscated programs

(Moser et al. 2007 and Brumley et al. 2008).

Detecting trigger-based malwares

- The direct dynamic-analysis fails (limited behaviors).
- The static-analysis faces some difficulties:
  - few work on the binary codes,
  - very sensitive to the obfuscation (Moser et al. 2007).
- We propose a hybrid approach.

Table of Contents

Table of Contents

Trace covering: hidden behaviors detection

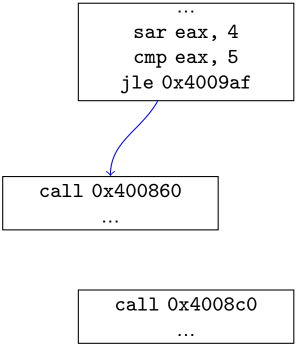
Hybrid approach: execute a program P which receives an input message m, we get a trace t.

as a sequence of instructions

```
...
0x40096b: mov cl, al
0x40096d: mov byte ptr [rbp-17], cl
0x400970: movsx eax, byte ptr [rbp-17]
0x400974: sar eax, 4
0x400977: cmp eax, 5
0x40097c: jle 0x4009af
0x40099b: call 0x400860
0x400860: ....


0x4009af: call 0x4008c0
0x4008c0: ...
```

as a path on the CFG



Trace covering: hidden behaviors detection

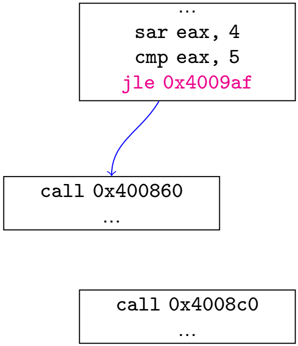
Hybrid approach: execute a program P which receives an input message m, we get a trace t. For each conditional branch $br \in t$,

as a sequence of instructions

```
...
0x40096b: mov cl, al
0x40096d: mov byte ptr [rbp-17], cl
0x400970: movsx eax, byte ptr [rbp-17]
0x400974: sar eax, 4
0x400977: cmp eax, 5
0x40097c: jle 0x4009af
0x40099b: call 0x400860
0x400860: ....


0x4009af: call 0x4008c0
0x4008c0: ...
```

as a path on the CFG



Trace covering: hidden behaviors detection

Hybrid approach: execute a program P which receives an input message m, we get a trace t. For each conditional branch $br \in t$, find m' so that the execution of P leads to a new trace t'

as a sequence of instructions

```
...
0x40096b: mov cl, al
0x40096d: mov byte ptr [rbp-17], cl
0x400970: movsx eax, byte ptr [rbp-17]
0x400974: sar eax, 4
0x400977: cmp eax, 5
0x40097c: jle 0x4009af
0x4009af: call 0x4008c0
0x4008c0: ...
```

as a path on the CFG

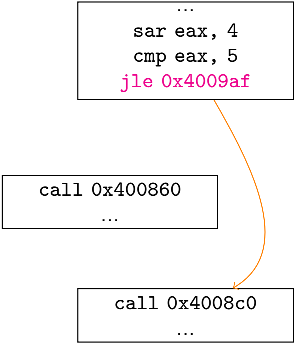


Table of Contents

Backtracking on the CFG of the program

Main ideas: for each checkpoint $br \in t$:

- fuzz testing minimization: find the minimal parts $I_{br} \subseteq m$ of the input message affecting br's decision,
- re-execution trace optimization: find the nearest execution checkpoints $C_{br} \in t$ affecting br's decision.

Backtracking on the CFG of the program

Main ideas: for each checkpoint $br \in t$:

- fuzz testing minimization: find the minimal parts $I_{br} \subseteq m$ of the input message affecting br's decision,
- re-execution trace optimization: find the nearest execution checkpoints $C_{br} \in t$ affecting br's decision.

...let's consider an example

Example (program)

```
0x400966: call 0x4006e0  ;get_msg
0x40096b: mov cl, al
0x40096d: mov byte ptr [rbp-17], cl
0x400970: movsx eax, byte ptr [rbp-17]
0x400974: sar eax, 4
0x400977: cmp eax, 5     ;x[0]>5
0x40097c: jle 0x4009af
0x400982: call 0x400830  ;do_A
0x400987: movsx eax, byte ptr [rbp-17]
0x40098b: and eax, 15
0x400990: cmp eax, 7     ;x[1]<=7
0x400995: jnle 0x4009a5
0x40099b: call 0x400860  ;do_A1
0x4009a0: jmp 0x4009aa   ;...
0x4009a5: call 0x400890  ;do_A2
0x4009af: call 0x4008c0  ;do_B
0x4009b4: movsx eax, byte ptr [rbp-17]
0x4009b8: and eax, 15
0x4009bd: cmp eax, 8     ;x[1]>8
0x4009c2: jle 0x4009d2
0x4009c8: call 0x4008f0  ;do_B1
0x4009d2: call 0x400920  ;do_B2
```

```
m = get_msg();
if (m[0] > 5)
  do_A();
  if (m[1] <= 7) do_A1();
  else do_A2();
else
  do_B();
  if (m[1] > 8) do_B1();
  else do_B2();
...
```

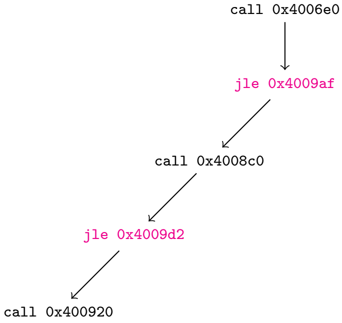
input: m = `byte ptr [rbp-17]`
branches:
$\{0x40097c, 0x400995, 0x4009c2\}$
checkpoints:
$C_{0x40097c} = 0x400970$,
$C_{0x400995} = 0x400987$,
$C_{0x4009c2} = 0x4009b4$

Example (control flow graph)

Input messages

- $m = 47h$

Control flow graph

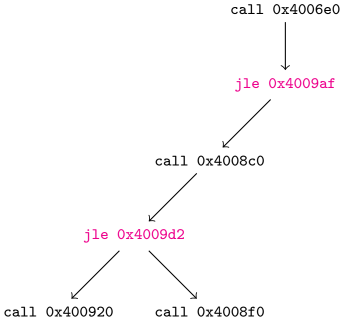


Example (control flow graph)

Input messages

- $m = 47h$
- $m = 49h$

Control flow graph

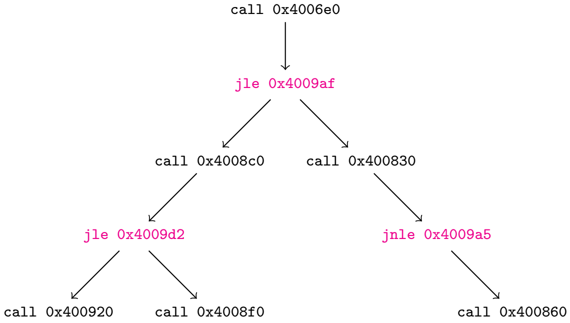


Example (control flow graph)

Input messages

- $m = 47h$
- $m = 49h$
- $m = 67h$

Control flow graph

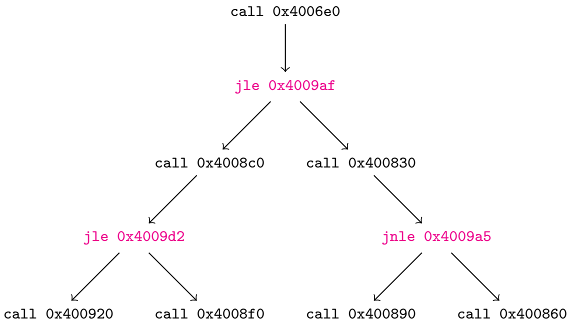


Example (control flow graph)

Input messages

- $m = 47h$
- $m = 49h$
- $m = 67h$
- $m = 66h$

Control flow graph

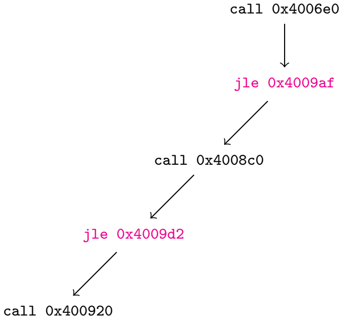


Example (backtracking)

Backtracking

- $m = 47h$

Control flow graph

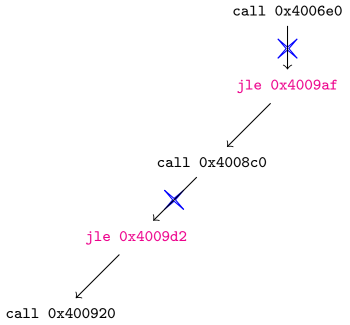


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints

Control flow graph

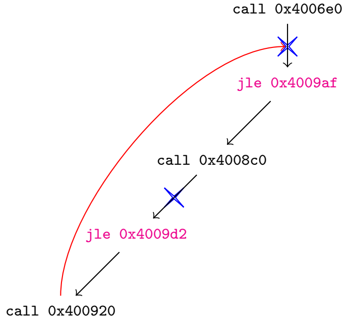


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback

Control flow graph

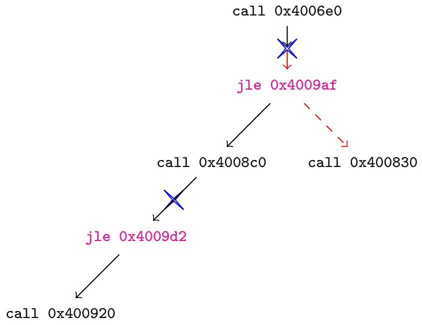


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$

Control flow graph

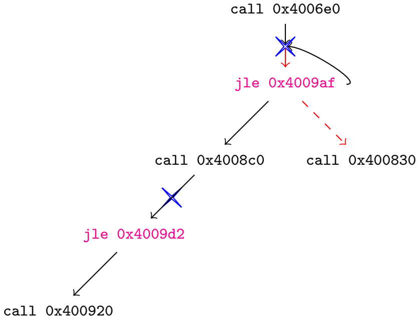


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback

Control flow graph

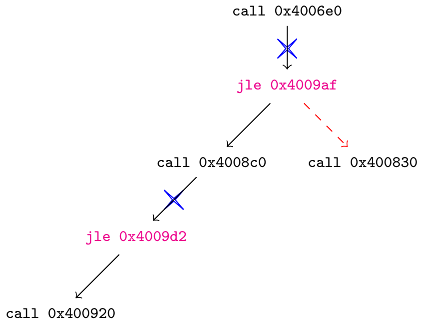


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m

Control flow graph

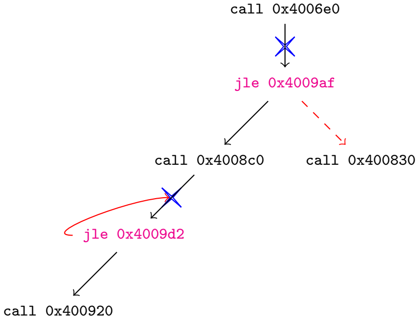


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m
  - rollback

Control flow graph

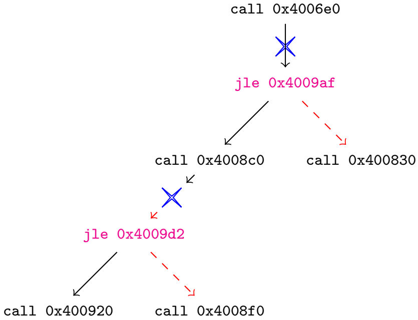


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m
  - rollback
  - try $m = 49h$

Control flow graph

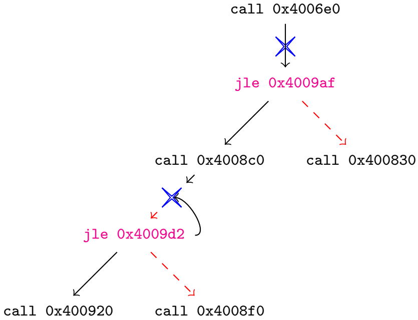


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m
  - rollback
  - try $m = 49h$
  - rollback

Control flow graph

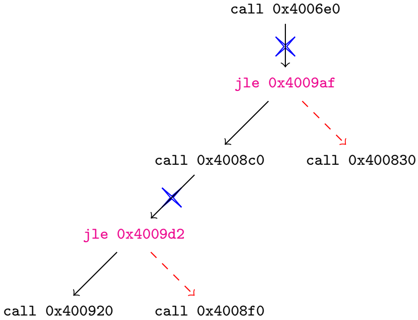


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m
  - rollback
  - try $m = 49h$
  - rollback
  - restore m

Control flow graph

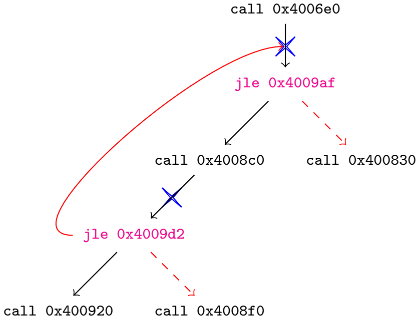


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m
  - rollback
  - try $m = 49h$
  - rollback
  - restore m
  - rollback

Control flow graph

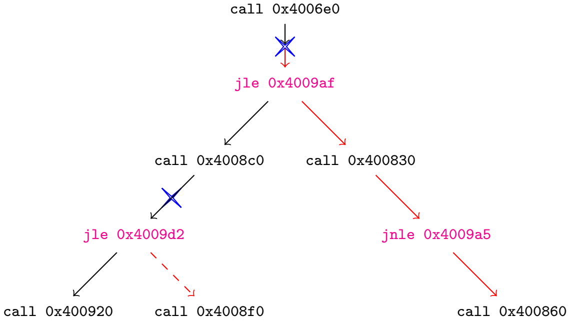


Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m
  - rollback
  - try $m = 49h$
  - rollback
  - restore m
  - rollback
- $m = 67h$
  - ...

Control flow graph



Example (backtracking)

Backtracking

- $m = 47h$
  - get checkpoints
  - rollback
  - try $m = 67h$
  - rollback
  - restore m
  - rollback
  - try $m = 49h$
  - rollback
  - restore m
  - rollback
- $m = 67h$
  - ...

Control flow graph

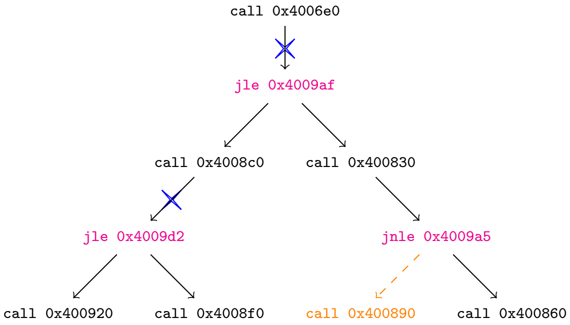


Table of Contents

Fuzz testing optimization

- naive approach (infeasible):
  - e.g. a (compressed) DNS response message of size 79 bytes, has 2^{79x8} possible values!!!
  - re-executing the whole program for each test is expensive.
- our approach: reverse execution and
  - reduce the number of tested inputs,
  - reduce the length of re-execution traces: checkpoints

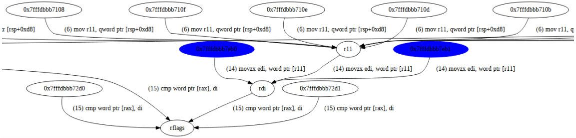
by the dynamic tainting analysis.

Dynamic tainting analysis by the liveness dataflow graph

From the executed trace t, construct a graph with edges are instructions, and for each edge

- source nodes: read operands,
- target nodes: written operands.

```
...
movzx edi, word ptr [11]
cmp word ptr [rax], di
jz 0x35c360b7d8
...
```



Taint propagation from the input message

PathExplorer: a code coveraging tool

- using Pin dynamic binary instrumentation framework [2],
- source codes available at https://github.com/tathanhdinh/PathExplorer.

README.md

PathExplorer: a Pintool for Binary Code Covering

The current version is UNSTABLE, but the following are less unstable:

- Dynamic tainting: construct the dataflow graph based on the liveness analysis (using the outer interface of live variables).
- Checkpoint detection: for each conditional branch there are several execution points which may affect to its decision.
- Reverse execution: an application-layer reverse execution mechanism.

In development:

- Smarter treatment for multiple rollbacks in case of direction fields in the input.
- DFA approximation for CFG.
- A new algorithm for checkpoint detection to shorten the re-execution trace.

Known bugs:

- Does not work for multiple threads programs yet.
- Re-execution is lost for large CFGs after too much rollbacks (lost detected in testing for a CFG with depth of 1000 instructions after nearly 150.000.000 rollbacks).

Experiment

Backtracking traversal on the CFG of `wget`

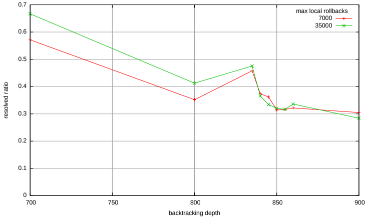
```
pin -t path_explorer.pin -r mlr -l depth -- wget url
```

Options:

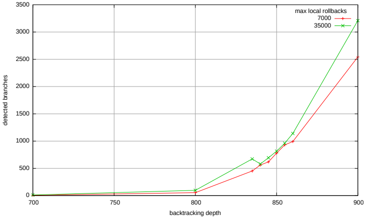
- mlr: the number of rollbacks for each checkpoint,
- depth: the depth of backtracking traversal.

depth	mlr	resolv/detected branches	total rollbacks
700	7000	4/7	21017
700	35000	8/12	169923
800	7000	19/54	301976
800	35000	40/97	2659205
835	7000	207/452	2515703
835	35000	320/673	17061162
840	7000	210/562	3908525
840	35000	212/580	19384515
845	7000	224/619	4913159
845	35000	232/695	26048334
850	7000	245/780	6221047
850	35000	261/815	32299635
855	7000	294/930	8104504
855	35000	304/961	39327555
860	7000	319/992	9273380
860	35000	383/1140	43569664
900	7000	775/2543	22998356
900	35000	911/3210	144671156

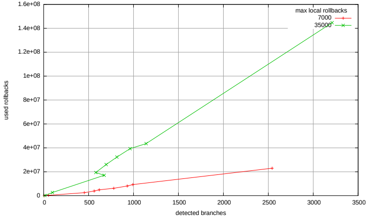
Experiment



(a) Resolved ratio



(b) Detected branches



(c) Detected ratio

Table of Contents

Table of Contents

Malware characterization by message analysis

In malware detection, we look for similarities between a known malcious program M and a suspicious P.

- Traditional approach: trace similarity

Malware characterization by message analysis

Thesis

Two equivalent programs will interpret the input messages equivalently.

In malware detection, we look for similarities between a known malcious program M and a suspicious P.

- Traditional approach: trace similarity
- Our approach: message interpretation similarity
  - compare message relations instead of traces.

Program similarity

Input messages partition

Let $\approx_T$ be an equivalence between traces (e.g. partial similarity, control flow graph similarity, etc), the derived equivalence $\approx_I$ between input messages is defined by:

$$i_1 \approx_I i_2 \Longleftrightarrow P(i_1) \approx_T P(i_2)$$

Program similarity by input messages partition

$$P \sim Q \Longleftrightarrow \text{having the same derived equivalence } \approx_I$$

Table of Contents

Input space partition by FA approximation

Finite State Automata approximation

The input message space are partitioned by Finite State Automata

- The input strings are the inputs of the program,
- The transition traces abstract the execution traces.

That extends the current approach in the Protocol Message Extraction (Caballero et al. 2009).

Corollary (systematic input format extraction)

The precisely obtained FA reveals the format of inputs.

Early results in FA approximation

- $\{...\}$: the parts of the input affecting to the branch's decision
- $0, 1$: the decisions of a branch,
- $\perp$: the execution halts before reaching the limit depth,
- $\perp\!\!\!\perp$: the execution continues after reaching the limit depth.

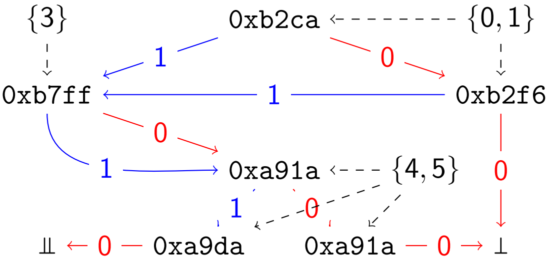


Figure: `wget` and `ping` have the same approximation at the depth 600

Table of Contents

Conclusions

- Smart input fuzzing: hybrid approach for the program covering
  - Dynamic-analysis: runs the program with a concrete input to get an execution trace,
  - Static-analysis: construct the dataflow graph on the trace to detect checkpoints.
  - Improvement in progress: symbolic execution with SMT solver.
- Message analysis: new approach for the program similarity
  - Similarity: relations between traces (instead of traces) are compared,
  - Protocol message extraction: the precisely obtained FA reveals the format of inputs.

Conclusions (a brief comparison)

	current approaches	our approach
analysis method	hybrid	hybrid
covering purpose	functionality	trace
branch resolving technique	symbolic execution	fuzzing+re-execution trace minimization
rollbacking technique	whole-system emulation	application-wide reverse execution
source code	unknown	available

Thanks for your attention
and any question?

Bibliography

- P. Godefroid et al. "DART: Directed Automated Random Testing". In: PLDI. 2005.
- C.-K. Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: PLDI. 2005.
- A. Moser et al. "Exploring Multiple Execution Paths for Malware Analysis". In: SSP. 2007.
- A. Moser et al. "Limits of Static Analysis for Malware Detection". In: ACSAC. 2007.
- D. Brumley et al. "Automatically Identifying Trigger-based Behavior in Malware". In: *Botnet Analysis and Defense*. 2008, pp. 65–88.
- J. Caballero et al. "Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering". In: CCS. 2009.
- N. Falliere et al. *W32.Stuxnet Dossier*. Tech. rep. Symantec Security Response, Feb. 2011.
- GReAT. *The Mystery of the Encrypted Gauss Payload*. Aug. 2013.