RAMBO: Run-time packer Analysis with Multiple Branch Observation

Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, Pablo G. Bringas
University of Deusto, Cisco Talos, Eurecom
13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment

Outline

- Why multi-path exploration for packers?
- Approach
  - Domain-specific optimizations
  - Heuristics
- Evaluation
- Discuss the results

Run-time packers…

- Widely used by malware authors to obfuscate/ protect their code
- 2 main goals
  - Hide the original code from static analysis
  - Implement anti-analysis methods
    - Anti-debug
    - Anti-dump
    - VM / Sandbox / Tool detection
- Making both **automated** and **manual** analysis more difficult

Shifting-decode-frames

- Also known as "partial code revelation"
- Takes advantage of the limitation of dynamic analysis
  - Single path!
- Decrypt code/data on-demand
- Prevent "run and dump"
- Used by certain "advanced" protectors (i.e. Armadillo)
- Presented in academic literature (Bilge et. al.)
  - Compile time function based protection

001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101

Talos

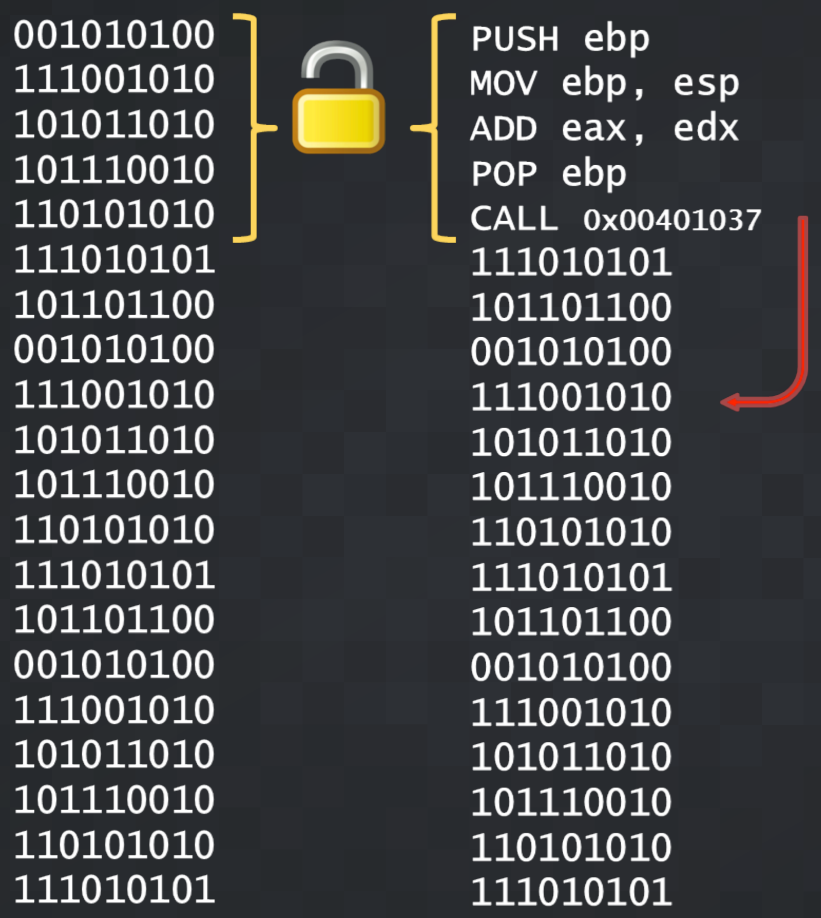
001010100
111001010
101011010
101110010
110101010

111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101

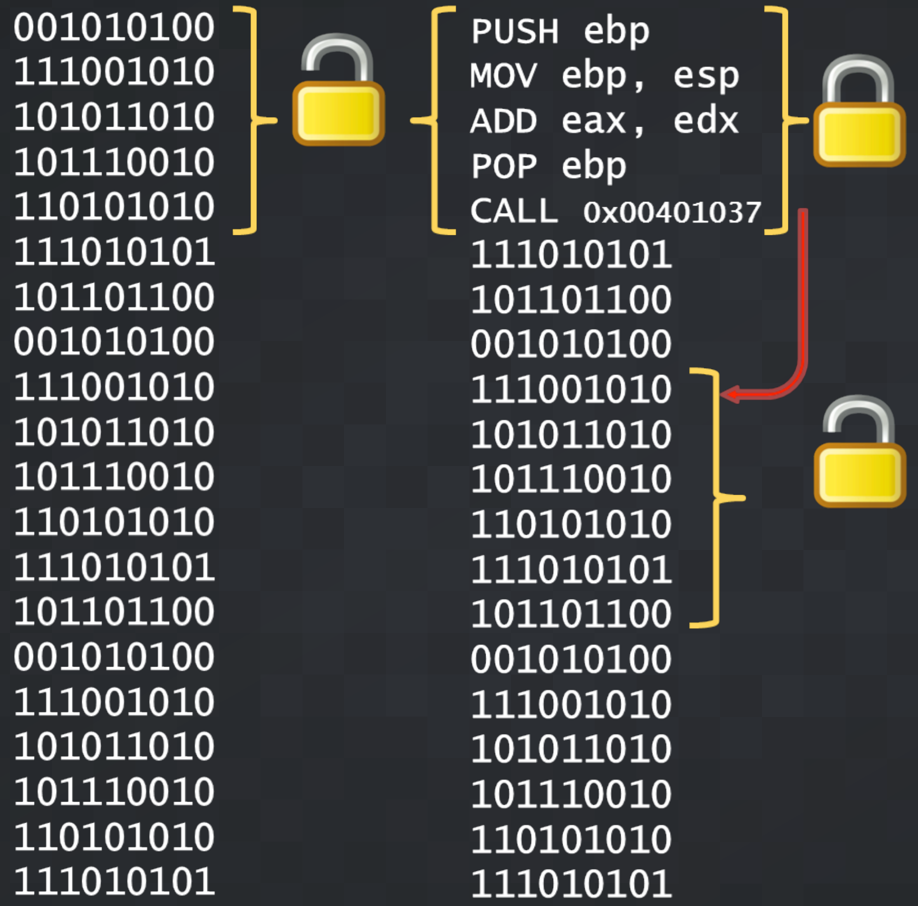
TALOS

```
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
```

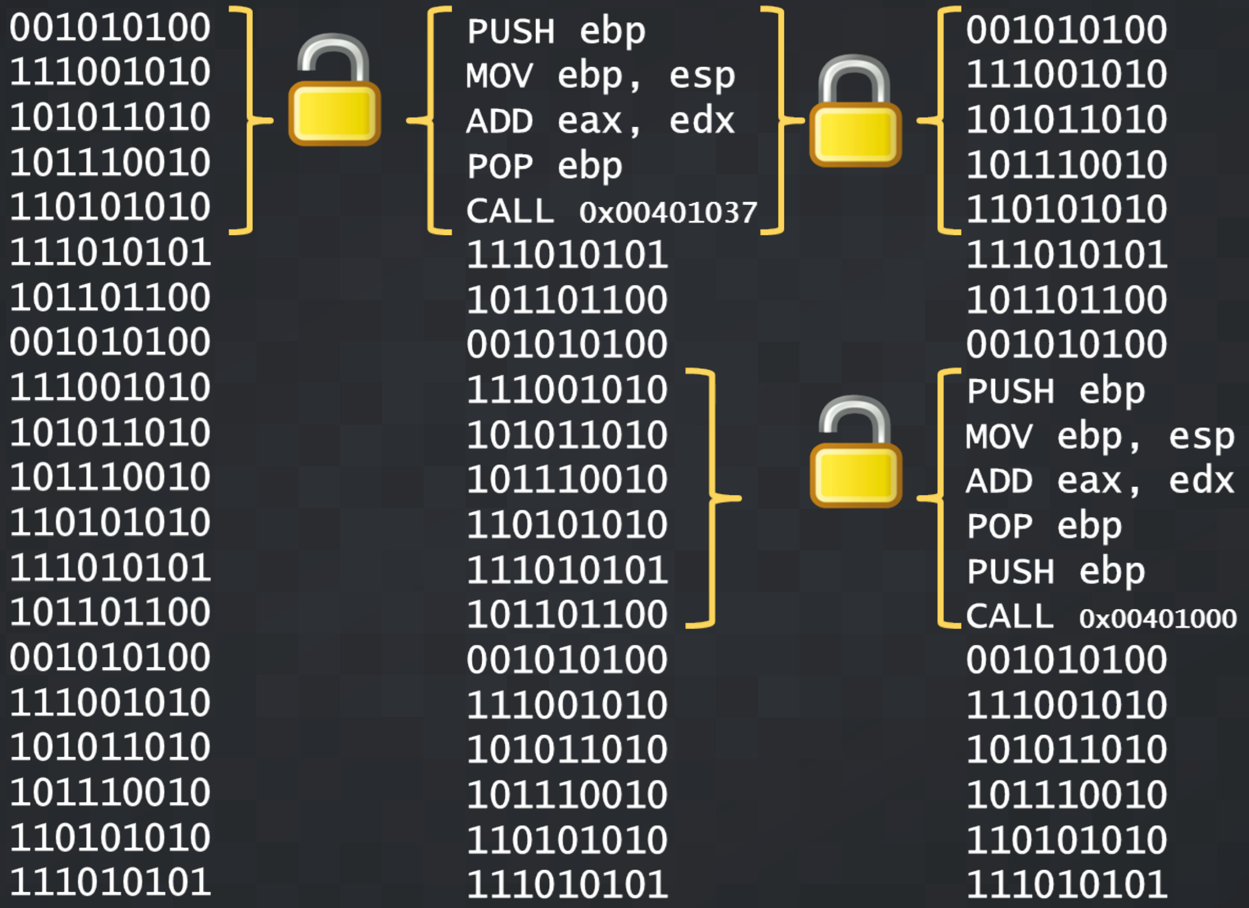
```
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
CALL 0x00401037
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
```


001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
CALL 0x00401037
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101

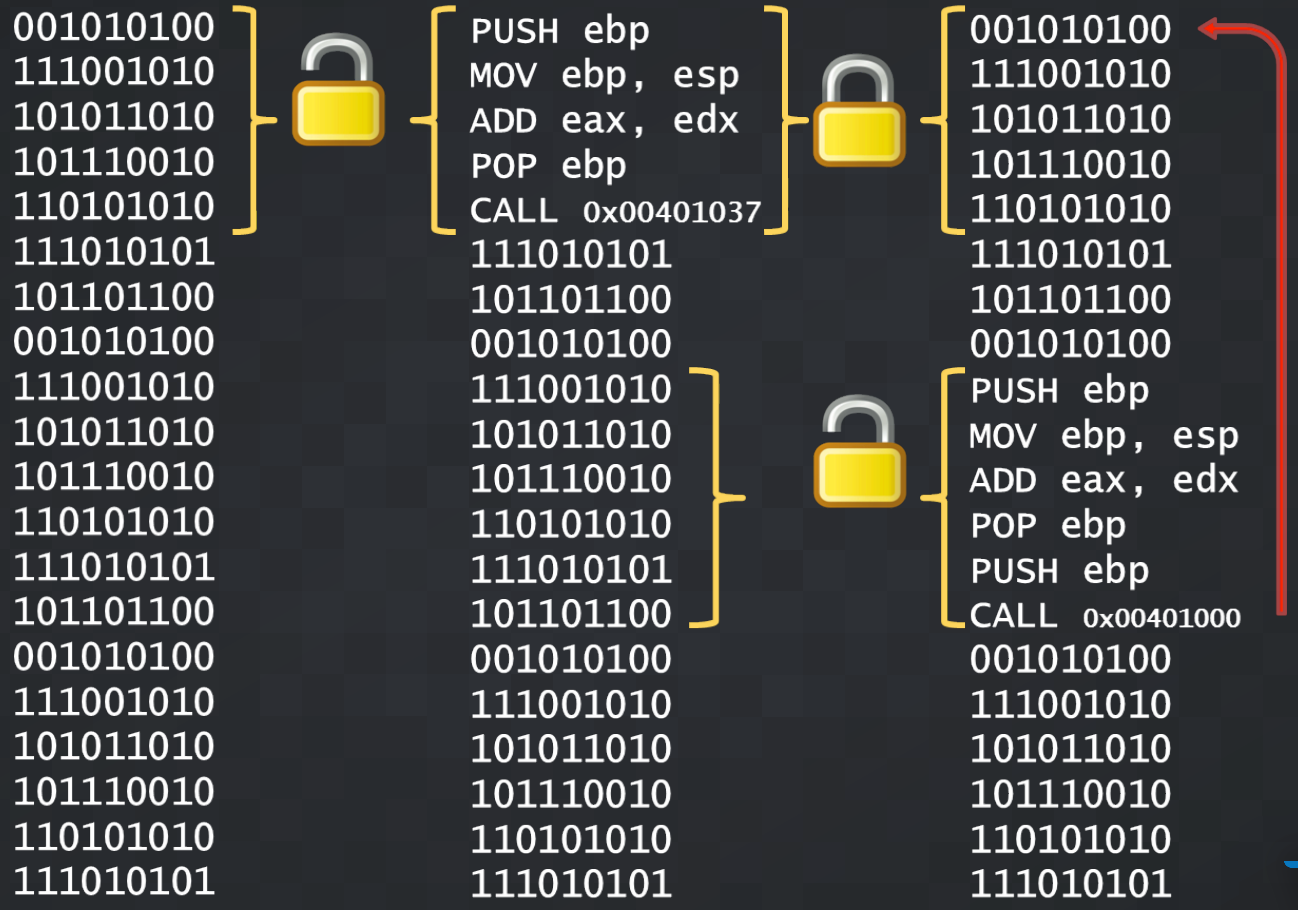
Talos

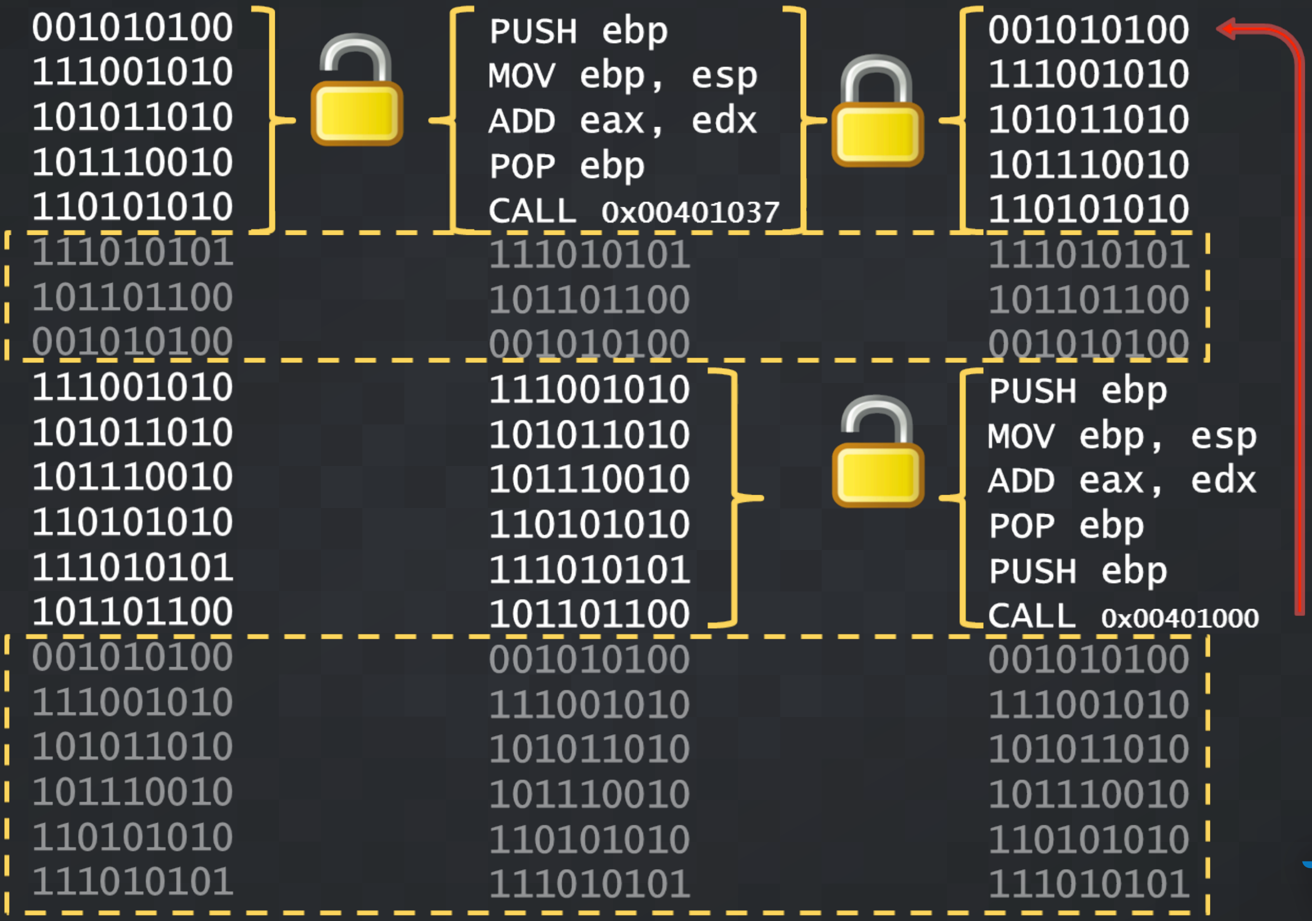


001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
CALL 0x00401037
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101


001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
CALL 0x00401037
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
PUSH ebp
CALL 0x00401000
001010100
111001010
101011010
101110010
110101010
111010101

TALOS


001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
CALL 0x00401037
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
PUSH ebp
CALL 0x00401000
001010100
111001010
101011010
101110010
110101010
111010101


001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
CALL 0x00401037
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
111001010
101011010
101110010
110101010
111010101
001010100
111001010
101011010
101110010
110101010
111010101
101101100
001010100
PUSH ebp
MOV ebp, esp
ADD eax, edx
POP ebp
PUSH ebp
CALL 0x00401000
001010100
111001010
101011010
101110010
110101010
111010101

Multi-path exploration…

- Computationally complex
  - Specially with obfuscated (even self-modifying) code
- Does not scale to real-world, large, complex malware

Multi-path exploration...

- Computationally complex
  - Specially with obfuscated (even self-modifying) code
- Does not scale to real-world, large, complex malware

Can we apply optimizations to multi-path exploration
for this specific use case?

Multi-path exploration

- Computationally complex
  - Specially with obfuscated (even self-modifying) code
- Does not scale to real-world, large, complex malware

Can we apply heuristics to multi-path exploration
for unpacking this type of packers?

Some intuitions…

- We do NOT need to explore *every single path* in the binary, just enough paths to uncover all the interesting regions.
- We do NOT need to understand which are the *conditions* to reach each path (unlike other use-cases, such as vulnerability analysis.
- We do NOT need to maintain the environment / system perfectly *consistent*. We just need to make sure that the execution is stable enough to uncover the protected regions.

Multi-path exploration

- Baseline implementation
  - Based on the concepts presented by Moser et al.
- Bitblaze platform
  - Dynamic taint analysis (Temu)
    - Taint result of function calls:
      - Network/file/argument/time related
  - Symbolic analysis (Vine)
    - Based on Weakest precondition & queries to STP
    - Concrete address for indirect memory accesses
  - System-level snapshots
    - Heavier, but we avoid dealing with system level inconsistencies: handles, open files, sockets…

Optimizations

#1 Partial symbolic execution

- Only execute certain ***regions of interest***

#2 Inconsistent multi-path exploration

- Ignore path constraints if solver cannot provide a solution
- Give ***priority*** to paths that can be solved consistently

#3 Sacrifice global consistency

- Maintain consistency only for the ***regions of interest***

Optimizations

#4 Discard long traces

#5 Bypass blocking API calls

#6 String comparisons

- Our model avoids exploring string comparison API calls
- We taint the output whenever input arguments are tainted
- This relaxes the constraints, allowing certain inconsistencies

The general goal is to simplify symbolic processing

General workflow

Approach:

1. Extract unpacked memory regions (frames)
   - Generically detect the frames & dump at the appropriate point
     - Prev. work: Deep Packer Inspection
2. Process extracted code (disassemble, compute CFG)
3. Find interesting points in the code (specific instructions)
4. Compute which paths lead to these points
5. Prioritize these paths during multi-path exploration

Talos

General workflow

Approach:

1. Extract unpacked memory regions (frames)
   - Generically detect the frames & dump at the appropriate point
     - Prev. work: Deep Packer Inspection
2. Process extracted code (disassemble, compute CFG)
3. Find interesting points in the code (specific instructions)
4. Compute which paths lead to these points
5. Prioritize these paths during multi-path exploration

Talos

Heuristic

Decide which paths should be expanded first

- Several paths can trigger the execution of a region
- We can skip paths that can only lead to regions already unpacked

TALOS

Heuristic

Steer the execution to the interesting points:

- **JMP** & **CALL** instructions
  - **that we have not executed** in any run, but:
  - If they lead to a region that has not been unpacked yet
- **CJMP** instructions leading to protected regions
  - That have not been executed (but were unpacked)
  - If we have only explored one of their paths
- Direct memory access (address not unpacked yet)
- Indirect calls (explore all the paths to these points)
- Immediate values that fall in the range of a protected memory region (may represent a memory access)

Talos

Heuristic

Also need to consider inter-procedural CFG:

- Explore all the paths that lead to a function, if it contains *"points of interest"*.

Path selection during MPE:

- Breadth First Search
  - Incrementally expand all the paths in the tree
  - Prioritize **other paths** over loops
- Prioritize branches with the lowest number of expansions
- Prioritize paths that can be forced consistently over inconsistent ones

Heuristic

Last resort: path bruteforcing

- Set **maximum number of expansions** for each branch.
- When this limit is reached for all the tainted branches:
  - Force the alternative path of non-tainted branches (INCONSISTENT!)
- Introduces inconsistencies, but can be useful to:
  - Bypass loops or control structures with very complex internal logic depending on input
    - E.g.: Parsers
  - In some cases, we just need to jump to some point in the code to trigger its unpacking.

Evaluation

Case study #1: Backpack + Kaiten IRC Bot

- Compile-time packer proposed by Bilge et al.
- Function based granularity
- Kaiten: IRC bot that connects a channel and receives commands

	Iteration 0	Iteration 1	Iteration 2	No Heur.
Functions unpacked	5/31	11/31	27/31	8/31
Interesting points	-	52	96	-
Cjmps	-	36	110	-
Snapshots	-	167	544	6015
Tainted-consistent cjmps	-	161	525	5888
Tainted-inconsistent cjmps	-	6	19	127
Untainted cjmps	-	0	40	-
Long traces discarded	-	6	0	-
Time	5m	24m	1.2h	8h

Evaluation

Case study #2: Armadillo

- Page based granularity (based on memory protection)
- Protected 2 bots: SDBot, SpyBot.

Talos

SDBOT	It. 0	It. 1	It. 2	It. 3	No Heur.
Functions unpacked	2/7	4/7	6/7	7/7	4/7
Interesting points	-	3	2	7	-
Cjmps	-	65	162	264	-
Snapshots	-	14	366	367	3974
Tainted-consistent cjmps	-	13	295	296	3660
Tainted-inconsistent cjmps	-	1	71	71	314
Untainted cjmps	-	0	1	1	-
Long traces discarded	-	1	14	14	-
Time	30m	2.2h	2.8h	3.2h	8h

SPYBOT	Iteration 0	Iteration 1	Iteration 2	No Heur.
Functions unpacked	3/9	8/9	9/9	6/9
Interesting points	-	26	1	-
Cjmps	-	163	214	-
Snapshots	-	113	153	4466
Tainted-consistent cjmps	-	17	31	4096
Tainted-inconsistent cjmps	-	96	122	370
Untainted cjmps	-	17	34	-
Long traces discarded	-	9	34	-
Time	30m	3h	2.75h	8h

Conclusions

- Plain vanilla multi-path exploration was not able to recover the code in a reasonable time (even with partial/ inconsistent exploration)
- With heuristic:
  - Almost 100% recovery of code / data
  - Significant reduction of time / resources when applying heuristics

Discussion

- Strong limitations for sample selection
  - For backpack, we needed linux-based source code.
  - We needed sufficiently complex samples:
    - For Armadillo, several pages of code.
    - Complex parsing routines or logic.
  - We needed non-packed samples.
    - Otherwise, the packer would reveal all the original code at once.
  - Simple malware families execute most the code in a single run (we needed bots).

TALOS

Discussion

- Technical complexity of *protectors* may affect multi-path exploration
  - Calling convention violation
  - Alternative methods to redirect control flow (push + ret, indirect calls, SEH/VEH based...)
  - Resource exhaustion (intentionally introduce complexity to exhaust time-consuming analysis engines such as emulators)
  - Nanomites (substitute branches by interrupts, compute the branch in a separate region of code or process)

Talos

Questions!

TALOS

talosintelligence.com
blog.talosintel.com
@talossecurity