Timing Performance Profiling of Substation Control Code

Iris Larsen

Western Washington University

Presentation Outline

- Contribution
- Rationale for our approach
- Building a performance graph from the control flow graph
- Computing shortest paths for IED malware detection
- Validation and testing

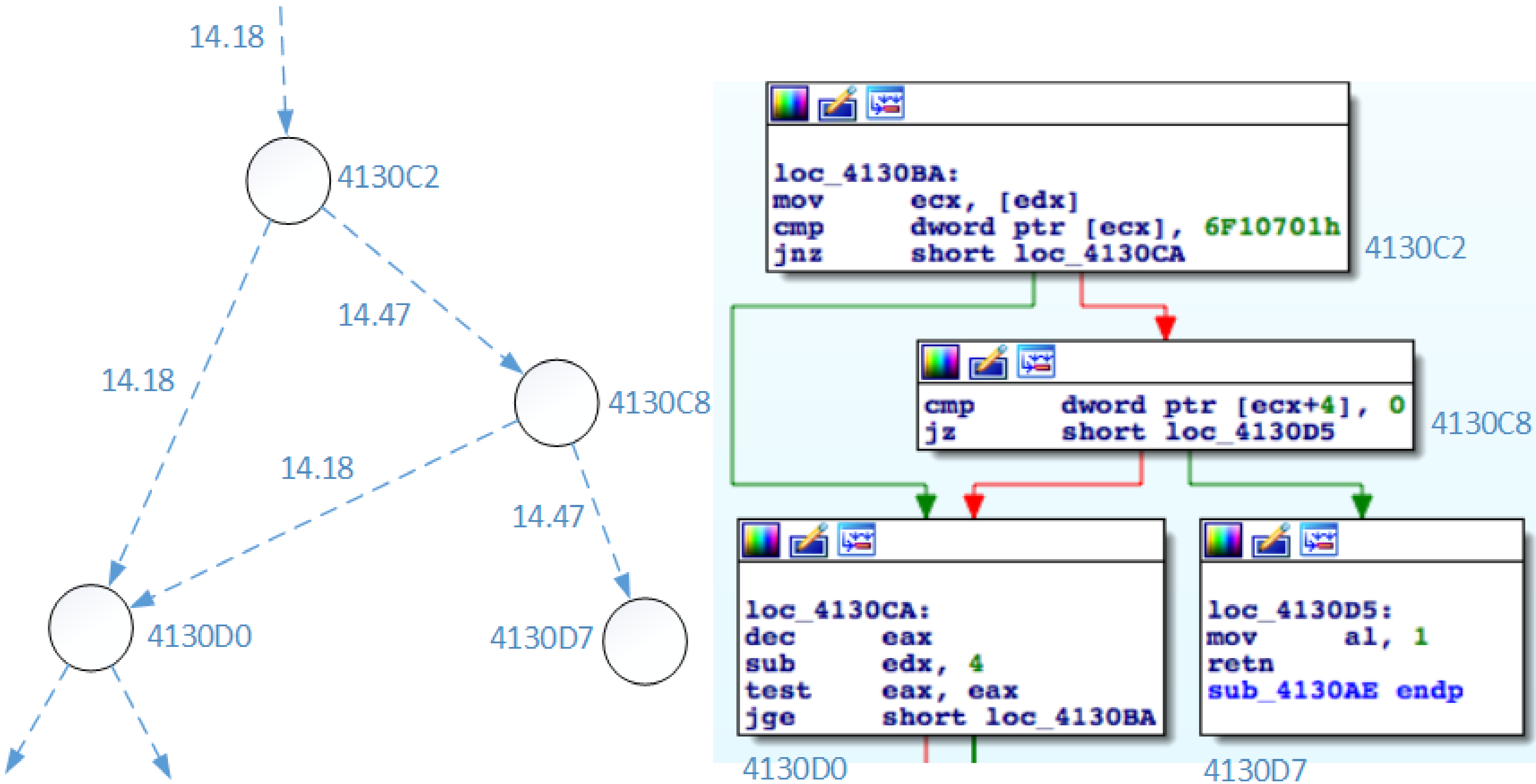
Contribution

- Research goal
  - Detect IED malware on the first encounter
  - No adverse effects on IED operation
- Approach
  - Modeling work to convert malware detection into a classical problem of graph theory
  - Graph-theoretical techniques to tackle the challenge
  - Apply as a binary static analysis technique

Rationale

- IEDs in an electrical substation have real-time requirements
- All faults must be sensed, processed, and addressed within a few milliseconds
- IEC 61850 specifies tolerated delays
- All protection and control code on an IED is tested to comply

Building a Performance Graph



Graph Characteristics

- A DAG from a source to a sink
  - Models execution flows from sensor readings to actuator controls
- The control-flow graph is revised to remove cycles
- Its structure models the cumulative time of execution paths
  - A vertex is labeled by the address of the branch instruction in a block of code
  - The execution time of that block of code is modeled by the weight on the edge that lands onto that vertex

Towards Weight Calculation

- We consider the total amount of clock cycles consumed by the instructions of the respective code block
- The clock cycles per instruction depend on the processor of the IED
- We also consider the IPS metric as measured by Dhrystone
- Note that IPS is cognizant of instruction pipelining

Metric Conversions (I)

- IPS is expressed in terms of million of instructions per second (MIPS)
- Example:
  - Intel Core i7
  - MIPS of 4,800
  - About 4,800 instructions per microsecond
- By counting an average of 1.5 clock cycles per instruction, we get 7,200 clock cycles per microsecond

Metric Conversions (II)

- A time threshold is now converted into clock cycles
- The total amount of clock cycles consumed by the code blocks along each path from a source to a sink should not exceed that threshold
- Time threshold of 2000 ms
  - $2000 * 10^3 * 7200 = 144 * 10^8$ clock cycles
  - Probability of maliciousness = 1.0
- Each clock cycle has a CtM of:
  - $1.0/144 * 10^8 = 0.00000017361111$
- The CtM of each instruction is the product of its number of clock cycles and the CtM of a single clock cycle

Examples of Latencies and their CtMs

Instruction	Clock Cycles	CtM for SCADA	CtM for Sampled Values
mov	1	0.00000000006944	0.00000017361111
cmp	2	0.00000000013888	0.00000034722222
jnz	1	0.00000000006944	0.00000017361111
jz	1	0.00000000006944	0.00000017361111
ret	2	0.00000000013888	0.00000034722222
dec	1	0.00000000006944	0.00000017361111
sub	1	0.00000000006944	0.00000017361111
test	1	0.00000000006944	0.00000017361111
jge	1	0.00000000006944	0.00000017361111

Weight Transformation

- The weight of an edge is set to $-\ln CtM$
- The higher the CtM of an edge is, the lower its negative log probability becomes
- Now it is the shortest path that represents an execution flow which could exceed the time threshold

Edge Relaxation

Algorithm 1: Algorithm to relax the edges of the performance graph of the binary code of an IED executable. Adopted from [13].

```
Function Relax (υ, v, ω);
  Input  : Vertices υ and v, and weight function ω.
  Output : A possible update of the attributes d and π of vertex v.
if v.d > −ln (e^{−υ.d} + e^{−ω(υ,v)}) then
    v.d = −ln (e^{−υ.d} + e^{−ω(υ,v)});
    v.π = υ;
else
    Leave v.d and v.π unchanged;
end
```

Malicious Flows as Shortest Paths

Algorithm 2: Algorithm to calculate the shortest path from a source vertex to a sink vertex, as in [13], in the performance graph of the binary code of an IED executable.

Function ShortestPath (DAG, ω, s);

Input : Performance graph, weight function ω, and source vertex s.

Output : Update the predecessor attributes of the vertices of the performance graph.

Topological sort of the vertices of the performance graph;

Initialize the vertices of the performance graph;

for *each vertex* υ *taken in order* **do**

  for *each vertex* v *such that there is an edge from* υ *to* v **do**

    Relax (υ, v, ω);

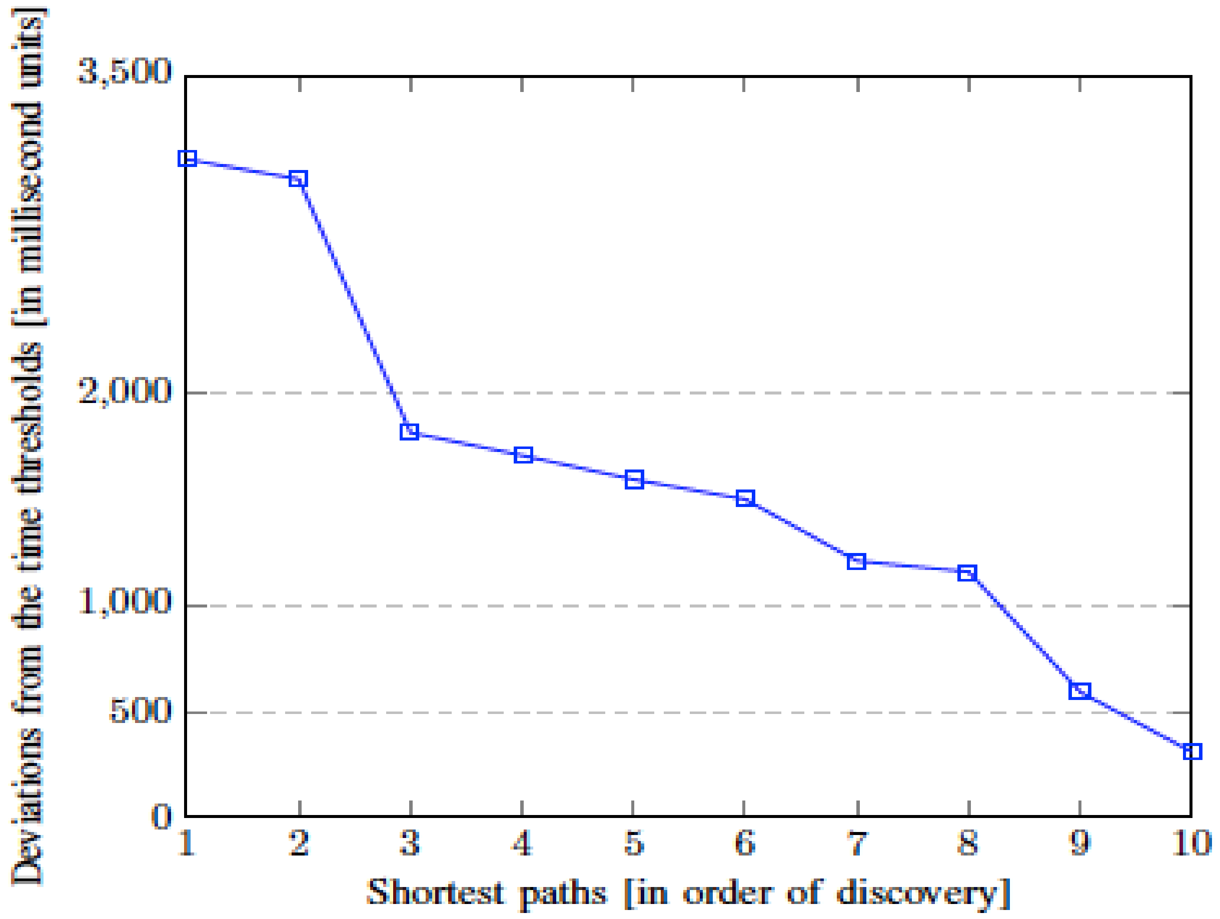
  end

end

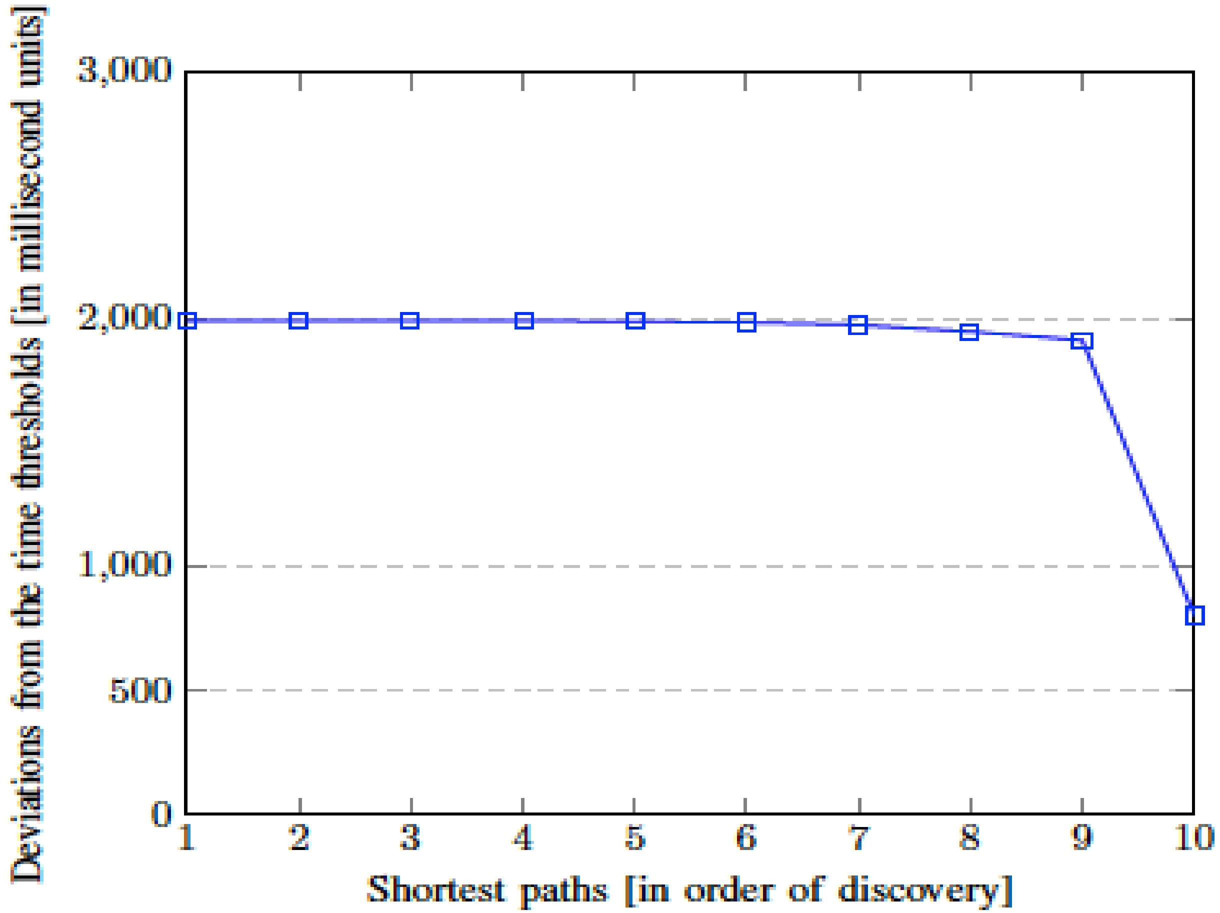
Evaluation

- Testbed
  - Resembled an electrical substation
  - Included Human-Machine Interface, relays, and an OPC server
- Malware
  - Network scanning
  - Exploits, with code injection and heap spraying
  - Installation via a dropper module
  - Public malware samples

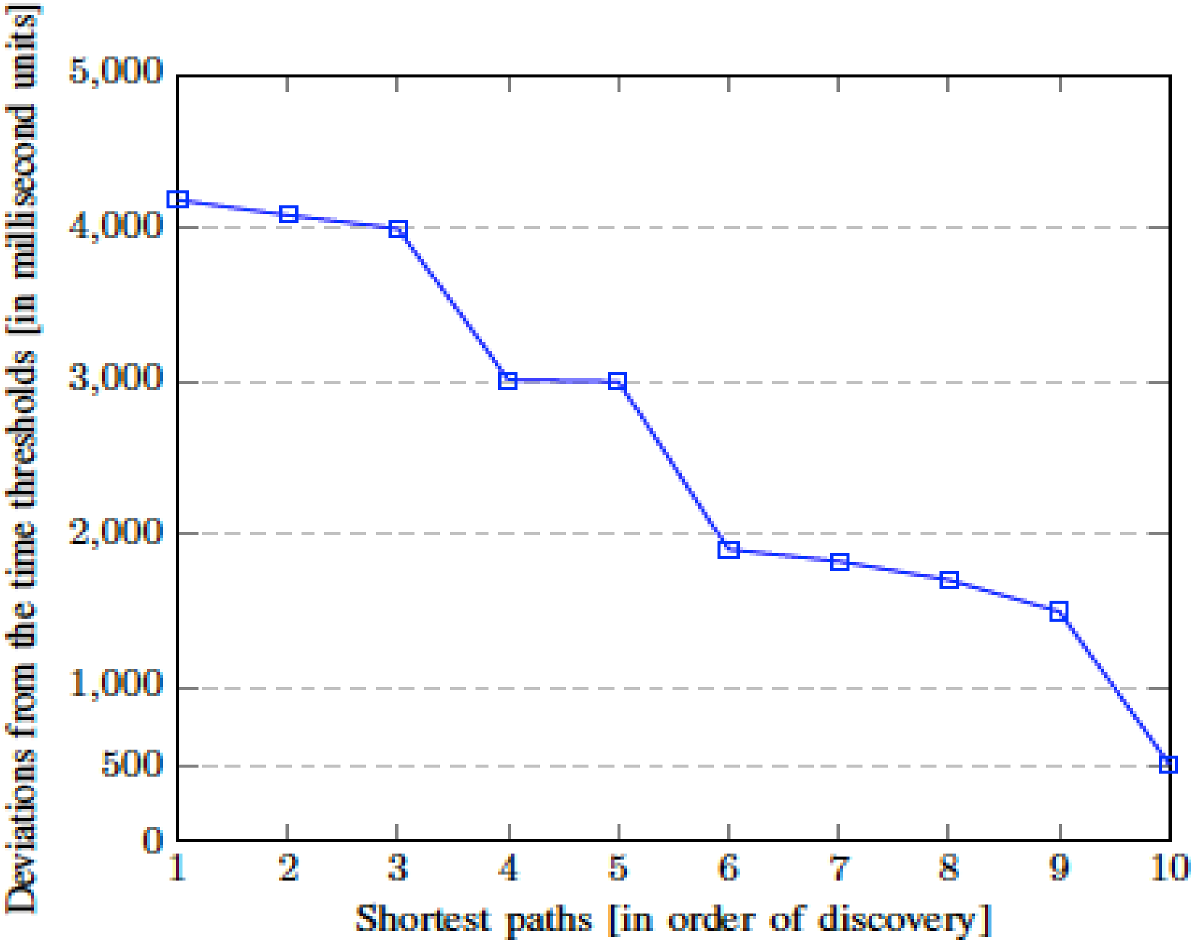
Timing Anomalies in Netscan Code



Timing Anomalies in Exploit Code



Timing Anomalies in Test Malware



Questions?