N-Variant Systems
A Secretless Framework for Security through Diversity
Cox et al.

Presented by: Stephen McLaughlin
The problem

Software homogeneity makes the process of leveraging a known exploit easy.
Some solutions

- Address space randomization
- Instruction set randomization
- But how random are they...
“Typically, these properties are determined by a secret key used to control the randomization.”

Discussion: Is this really good enough?
Solution to problem with solutions

- Authors posit that we can eliminate the need for secrets!
- All we need to do is run variants of a program with mutually exclusive exploits, hopefully with the property that:
- **Not all variants can be exploited at the same time.**
Contributions

- The N-variant concept
- A model for reasoning about properties of N-variant systems
- Two examples of variants:
Security Model - Framework overview

Presented by: Stephen McLaughlin

N-Variant Systems
A Secretless Framework for Security through Diversity

Cox et al.
Only mentioned in passing
Let’s try to reason about it:

▸ Variant programs:
Variant programs: Obviously not in TCB - receive malicious inputs, and are not verified

Monitor:
Variant programs: Obviously not in TCB - receive malicious inputs, and are not verified
Monitor: Must be in the TCB if it is to provide reference monitor guarantees
Polygrapher:
Security Model - TCB

- Variant programs: Obviously not in TCB - receive malicious inputs, and are not verified
- Monitor: Must be in the TCB if it is to provide reference monitor guarantees
- Polygrapher: No idea for now
Not explicitly mentioned in the paper
Let’s try to reason about it:

▶ Variant programs: Obviously not trusted
From a reference monitor perspective

- What does it mediate?
- Is there a policy?
- Will come back to the next two
- Tamperproof?
- Verifiable?
Related Work

- Automated program diversity - Random instruction sets, system calls, address space layouts, etc.
- Redundant execution - fault tolerance, reliability
- Misc. - Non-executable pages, memory tainting, canaries
N-variant model

Three parts

- The model - executions for original program and its variants
- Two properties about the model - Normal equivalence and Detection
Each program variant creates an execution: \([S_0, S_1, \ldots]\), a possibly infinite sequence of program states.

The set of states for all variants is represented as a tuple of states for each variant at each step of execution:
\([<S_{0,0}, S_{0,1}, \ldots S_{0,N-1}>, <S_{1,0}, S_{1,1}, \ldots S_{1,N-1}>, \ldots]\).

A canonicalization function \(C\), is needed to transform the state of each variant to the form of the states of the original program. So, \(C(S_{0,0}) = C(S_{1,0})\).
Normal Equivalence Property

- **normal state**: the variant is executing as intended.
- **Normal Equivalence**: If all variants are in a normal state, then they must have the same canonical state.
- More formally: \( \forall s_1, s_2 \in S_i, s_1, s_2 \in \text{Normal} \rightarrow C(s_1) = C(s_2) \).
- Proved by induction over the number of normal state transitions.
Detection Property

- **compromised state**: the variant has been successfully compromised by an attack
- **alarm state**: a variant’s anomalous behavior is detected by the monitor.
- **Detection Property**: If the Normal Equivalence Property is satisfied then if a variant is in a compromised state, another variant is in an alarm state.
So what?

» What does this model actually tell us?
» Can we build a system that follows this model?
» What would this require?
Threat: Exploits based on absolute addresses
example: format string `printf(str);`
Mitigation: Variants have mutually exclusive address spaces making a malicious address unreachable by more than one variant
Variants - Instruction Set Tagging

- **Threat**: Exploits that inject executable code to the stack
- **example**: Buffer overflow to overwrite return address
- **Mitigation**: Place a different tag on instructions from each variant
Goals:
- Reduce nondeterminism - Processes should be synchronized - Why is this required?
- Increase granularity of monitoring - Why is this required?
Kernel Implementation - Reducing Nondeterminism

- System calls are wrapped with synchronization primitives that force variants to execute calls in direct succession.

```c
ssize_t sys_read(int fd, const void *buf, size_t count) {
    if (!hasSibling (current)) { make system call normally } // not a variant process
    else {
        record that this variant process entered call
        if (!inSystemCall (current->sibling)) { // this variant is first
            save parameters
            sleep // sibling will wake us up
            get result and copy *buf data back into address space
            return result;
        } else if (currentSystemCall (current->sibling) == SYS_READ) { // this variant is second, sibling waiting
            if (parameters match) { // what it means to “match” depends on variation and system call
                perform system call
                save result and data in kernel buffer
                wake up sibling
                return result;
            } else { DIVERGENCE ERROR! } // sibling used different parameters
        } else { DIVERGENCE ERROR! } } } // sibling is in a different system call
```
Meh, we did it. It works. It’s not too slow.

- What did they evaluate?
- Did we learn anything about N-variants that we didn’t know before?
- Were there any lessons learned about the implementation?

<table>
<thead>
<tr>
<th>Configuration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Unmodified</td>
<td>Unmodified</td>
<td>2-variant</td>
<td>Apache</td>
<td>Apache</td>
<td>2-variant</td>
</tr>
<tr>
<td></td>
<td>Apache, unmodified kernel</td>
<td>Apache, N-variant kernel</td>
<td>system, address partitioning</td>
<td>running under Strata</td>
<td>with instruction tags</td>
<td>system, instruction tags</td>
</tr>
<tr>
<td>Unsaturated</td>
<td>Throughput (MB/s)</td>
<td>2.36</td>
<td>2.32</td>
<td>2.04</td>
<td>2.27</td>
<td>2.25</td>
</tr>
<tr>
<td></td>
<td>Latency (ms)</td>
<td>2.35</td>
<td>2.40</td>
<td>2.77</td>
<td>2.42</td>
<td>2.46</td>
</tr>
<tr>
<td>Saturated</td>
<td>Throughput (MB/s)</td>
<td>9.70</td>
<td>9.59</td>
<td>5.06</td>
<td>8.54</td>
<td>8.30</td>
</tr>
<tr>
<td></td>
<td>Latency (ms)</td>
<td>17.65</td>
<td>17.80</td>
<td>34.20</td>
<td>20.30</td>
<td>20.58</td>
</tr>
</tbody>
</table>
complete mediation:
Back to reference monitor guarantees

- **complete mediation**: No - monitoring is done at intervals
- **tamperproof**: 
Back to reference monitor guarantees

- **complete mediation**: No - monitoring is done at intervals
- **tamperproof**: No - by definition depends on input from potentially low integrity or malicious programs
- **verifiable**: ...
Limitations

- Does not address recover from exploit, only detection
- May lead to denial of service
- Model is not subject to race conditions but actual implementation is
- Variant properties could be spoofed by a well crafted exploit
- Actual implementation only monitors the output of the variants
- $N=2$