Redundant Computing for Security

David Evans
University of Virginia
Work with Ben Cox, Anh Nguyen-Tuong, Jonathan Rowanhill, John Knight, and Jack Davidson

Yahoo! Tech Talk
16 October 2008

The Basic Idea

Input (Possibly Malicious)

Server Variant 0

Monitor

Server Variant 1

Output

Attacker must find one input that compromises both variants

A Combination Hardware—Software Debugging System

K. C. KNOWLTON

Abstract—A solution is proposed for automatically detecting many programming errors, in particular, those errors which can cause a program to malfunction in different ways, depending upon how the faulty program and the data are sequenced into storage. Error detection is accomplished by simultaneously running two variants of a program which permit to be logically identical, with appropriate hardware checking between them.

IEEE Transactions on Computers, Jan 1968

Nevil Maskelyne
5th English Astronomer Royal, 1765-1811

Maskelyne’s Redundant Computing Data Diversity

Data for computing positions at noon

Data for computing positions at midnight


Image: Michael Daly, Wikimedia Commons
Babbage’s Review
“...I wish to God these calculations had been executed by steam.”
Charles Babbage, 1821

...back to the 21st century (and beyond)
• Moore’s Law: number of transistors$/ increases exponentially
• Einstein’s Law: speed of light isn’t getting faster
• Eastwood/Turing Law: “If you want a guarantee, buy a toaster.”
• Sutton’s Law: “That’s where the money is.”

Using Extra Cores for Security
• Despite lots of effort:
  – Automatically parallelizing programs is still only possible in rare circumstances
  – Human programmers are not capable of thinking asynchronously
• Most server programs do not have fine grain parallelism and are I/O-bound
• Hence: lots of essentially free cycles for security

Security Through Diversity
• Address-Space Randomization
• Instruction Set Randomization
  – [Kc+ 2003, Barrantes+ 2003]
• Data Diversity

Example: Instruction Set Randomization

Limitations of Diversity Techniques
• Weak security assurances
  – Probabilistic guarantees
  – Uncertain what happens when it works
• Need high-entropy variations
  – Address-space may be too small [Shacham+, CCS 04]
• Need to keep secrets
  – Attacker may be able to incrementally probe system [Sovarel+, USENIX Sec 2005]
  – Side channels, weak key generation, etc.
N-Variant System Framework

- **Polygrapher**
  - Replicates input to all variants
- **Variants**
  - \( N \) processes that implement the same service
  - Vary property you hope attack depends on: memory locations, instruction set, system call numbers, calling convention, data representation, ...

No secrets, high assurances, no need for entropy

Variants Requirements

- **Detection Property**
  Any attack that compromises one variant causes the other to “crash” (behave in a way that is noticeably different to the monitor)
- **Normal Equivalence Property**
  Under normal inputs, the variants stay in equivalent states:
  \[ A_0(S_0) \equiv A_1(S_1) \]

Requirement: Actual states are different, but abstract states are equivalent.

Opportunity for Variation

- All Possible Inputs
- Malicious Inputs
- Inputs with Well-Defined Behavior
Can’t change “well-defined” behavior, but can change “undefined” behavior

 Disjoint Variants

- **Variant 0**
  - Inputs with Well-Defined Behavior
- **Variant 1**
  - Inputs with Well-Defined Behavior

Interpreter Model of Execution

- Each interpreter manipulates different data types, protecting inner interpreters.
- Malicious data finds a way through protections in one interpreter to exploit functionality in lower interpreters.
- Our goal: replace interpreters so malicious data is interpreted.
Example: Address-Space Partitioning

- Variation
  - Variant 0: addresses all start with 0
  - Variant 1: addresses all start with 1
- Normal Equivalence
  - Map addresses to same address space
  - Assumes normal behavior does not depend on absolute addresses
- Detection Property
  - Any injected absolute load/store is invalid on one of the variants

Example: Instruction Set Tagging

- Variation: add an extra bit to all opcodes
  - Variation 0: tag bit is a 0
  - Variation 1: tag bit is a 1
  - Run-time: check and remove bit (software dynamic translation)
- Normal Equivalence:
  - Remove the tag bits
  - Assume well-behaved program does not rely on its own instructions
- Detection Property
  - Any (tagged) opcode is invalid on one variant
  - Injected code (identical on both) cannot run on both

Data Diversity

[Amman & Knight, 1987] and [Maskelyne, 1767]

Variations on Interpreters

<table>
<thead>
<tr>
<th>Variation</th>
<th>Data Type</th>
<th>Variant 0</th>
<th>Variant 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Space</td>
<td>Address</td>
<td>( R_0 (a) = a ) | ( 0 )</td>
<td>( R_1 (a) = a + 0x800... )</td>
</tr>
<tr>
<td>Partitioning</td>
<td></td>
<td>( R_0^{-1}(a) = a ) | ( 0 )</td>
<td>( R_1^{-1}(a) = a - 0x800... )</td>
</tr>
<tr>
<td>Instruction Set</td>
<td>Instruction</td>
<td>( R_0 (\text{inst}) = 0 | \text{inst} )</td>
<td>( R_1 (\text{inst}) = 1 | \text{inst} )</td>
</tr>
<tr>
<td>Tagging</td>
<td></td>
<td>( R_0^{-1}(0 | \text{inst}) = \text{inst} )</td>
<td>( R_1^{-1}(1 | \text{inst}) = \text{inst} )</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>( ? )</td>
<td>( ? )</td>
</tr>
</tbody>
</table>

Data Diversity in N-Variant Systems

UID Corruption Attacks

```c
uid_t user;
...
user = authenticate();
...
setuid(user);
```

Examples in [Chen, USENIX Sec 2005]

Goal: thwart attacks by changing data representation
UID Data Diversity

<table>
<thead>
<tr>
<th>User</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>bin</td>
<td>0x7FFFFFFE</td>
</tr>
<tr>
<td>nobody</td>
<td>0x7FFFFF9C</td>
</tr>
</tbody>
</table>

Identity Re-expression

\[ R_0(u) = u \]
\[ R_0^{-1}(u) = u \oplus 0x7FFFFFFF \]

Flip Bits Re-expression

\[ R_1(u) = u \]
\[ R_1^{-1}(u) = u \oplus 0x7FFFFFFF \]

Variant 0 Variant 1

Data Transformation Requirements

- Normal equivalence:
  - \( \forall x: R_i^{-1}(R_i(x)) = x \)
  - All trusted data of type \( T \) is transformed by \( R \)
  - All instructions in \( P \) that operate on data of type \( T \) are transformed to preserve original semantics on re-expressed data

- Detection:
  - \( \forall x: T, R_0^{-1}(x) \neq R_1^{-1}(x) \) (disjointedness)

Ideal Implementation

- Polygrapher
  - Identical inputs to variants at same time
- Monitor
  - Continually examine variants completely
- Variants
  - Fully isolated, behave identically on normal inputs

Infeasible for real systems

Framework Implementation

- Modified Linux 2.6.11 kernel
- Run variants as processes
- Create 2 new system calls
  - n_variant_fork
  - n_variant_execve
- Replication and monitoring by wrapping system calls

Wrapping System Calls

- All calls: check each variant makes the same call
- I/O system calls (process interacts with external state)
  - open, read, write
    - Make call once, send same result to all variants
  - Fork, execve, wait
    - Make call once per variant, adjusted accordingly
- Dangerous
  - Some calls break isolation (mmap) or escape framework (execve)
  - Current solution: disallow unsafe calls

```c
sys_write_wrapper(int fd, char __user * buf, int len) {
    if (IS_VARIANT(current)) {
        perform system call normally
    } else {
        if (currentSystemCall(current->nv_system) == SYS_WRITE) {
            DIVERGENCE – different system calls
        } else if (currentSystemCall(current->nv_system) != SYS_WRITE) {
            DIVERGENCE – different parameters
        } else if (isLastVariant(current->nv_system)) {
            Sleep
            Return Result Value
        } else if (isLastVariant(current->nv_system)) {
            Sleep
            Return Result Value
        } else if (isLastVariant(current->nv_system)) {
            Sleep
            Return Result Value
        } else if (isLastVariant(current->nv_system)) {
            Sleep
            Return Result Value
        } else {
            Perform System Call
            Save Result
            Wake Up All Variants
            Return Result Value
        }
    }
}
```
Implementing Variants

- Address Space Partitioning
  - Specify segments' start addresses and sizes
  - OS detects injected address as SEGV
- Instruction Set Tagging
  - Use Diablo [De Sutter’ 03] to insert tags into binary
  - Use Strata [Scott’ 02] to check and remove tags at runtime

Implementing UID Variation

- Assumptions:
  - We can identify UID data (uid_t, gid_t)
  - Only certain operations are performed on it:
    - Assignments, Comparisons, Parameter passing

  Program shouldn’t depend on actual UID values, only the users they represent.

Code Transformation

- Re-express UID constants in code
  
  \[
  \text{if (getuid())} \Rightarrow \text{if (getuid() == 0)} \Rightarrow \text{if (getuid() == 0x7fffffff)}
  \]
- Preserve semantics
  - Flip comparisons
- Fine-grained monitoring:
  - `uid_t uid_value(uid_t), bool check_cond(bool)`
- External Trusted Data (e.g., `/etc/passwd`)

Thwarting UID Corruption

Injected UID: $\forall x: T, R_y^{-1}(x) \neq R_y^{-1}(x)$ => detected
Open Problems and Opportunities

- Dealing with non-determinism
  - Most sources addressed by wrappers
    - e.g., entropy sources
  - ...but not multi-threading [Bruschi, Cavallero & Lanzi 07]
- Finding useful higher level variations
  - Need specified behavior
  - Opportunities with higher-level languages, web application synthesizers
- Client-side uses (e.g., JavaScript interpreters)
- Giving variants different inputs
  - Character encodings

N-Variant Framework Summary

- Force attacker to simultaneously compromise all variants with same input
- Advantages
  - Enables low-entropy variations
  - High security assurance with no secrets
    - Easier to deploy and maintain than secret diversity
- Disadvantages
  - Expensive for CPU-bound applications
  - Variations limited by need to preserve application semantics

http://www.cs.virginia.edu/nvariant/
Papers: USENIX Sec 2006, DSN 2008
Collaborators: Ben Cox, Anh Nguyen-Tuong, Jonathan Rowanhill, John Knight, Jack Davidson

Related Work

- Design Diversity
  - HACQIT [Just+, 2002], [Gao, Reiter & Song 2005]
- Probabilistic Variations
  - DieHard [Berger & Zorn, 2006]
- Other projects exploring similar frameworks
  - [Bruschi, Cavallaro & Lanzi 2007], [Salamat, Gal & Franz 2008]