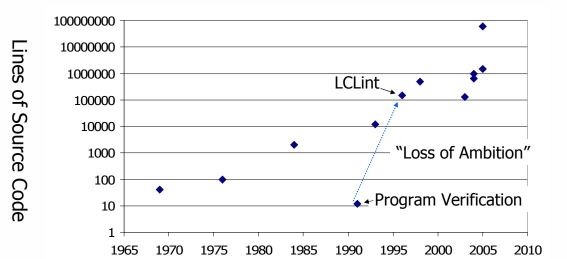
David Evans
University of Virginia

http://www.cs.virginia.edu/evans/talks/sdwest06

Menu

- Retrospective:
  - 10 years of program analysis
  - Limits of static analysis
  - Recent state-of-the-art
- Two current projects:
  - Perracotta
  - N-Variant Systems

Static Program Analysis



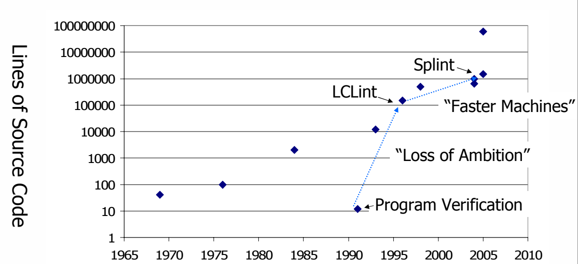
"Loss of Ambition":
Instead of verifying programs, look for simple mistakes
Accept unsoundness: false positives and negatives

Static Analysis in 1996

- Could check 100,000+ line programs
- Unsound and incomplete
- Warnings for "likely" memory leaks, null dereferences, type errors, ignoring possible error results, etc.
- Required source code annotations for inter-procedural checking

David Evans. *Static Detection of Dynamic Memory Errors*. PLDI, May 1996.

LCLint - Splint



Faster machines
Better "marketing"
Target security (buffer overflows, format string, etc.)

2006

coverity FORTIFY SOFTWARE PolySpace Secure Software Reflective OUNCE LABS cigital

- Lots of companies selling security scanning
  - Some of them are even profitable!
- Microsoft PREfix/fast, SLAM→SDV
  - Windows, Office developers required to annotate code to pass checking before commit

What Hasn't Changed

- Programs still ship with buffer overflows**!**
  - **Prediction:** this won't be true 5 years from now
- Perfect checking is still impossible
  - **Prediction:** this will still be true 5 years from now

Why Perfect Checking is Impossible: Theory

- It is impossible to precisely decide any important program property for all programs
- Is this program vulnerable?

```
int main (int argc, char *argv) {
   P ();
   gets();
}
```


Halting Problem [Turing 1936]

- Can we write a program that takes any program as input and returns true iff running that program would terminate?

```
// post: returns true iff p will halt
bool doesItFinish (Program p) {
   ???
}
```


Informal Proof

- Proof by contradiction:

```
bool contradict () {
   if (doesItFinish (contradict)) {
       while (true) ; // loop forever
   } else {
       return false; } }
```

What is doesItFinish (contradiction)?

Can't be **true**: contradict would loop forever
Can't be **false**: contradict would finish in else
Therefore, doesItFinish can't exist!

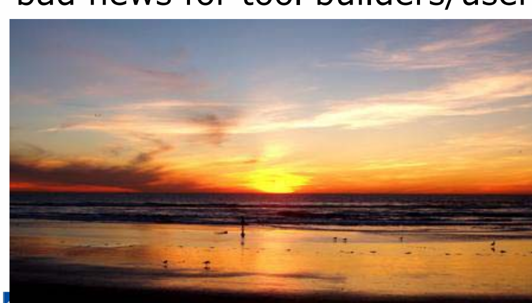
Hopelessness of Analysis

- But this means, we can't write a program that decides any other interesting property either:

```
bool dereferencesNull (Program p)
  // EFFECTS: Returns true if p ever dereferences null,
  //     false otherwise.

bool alwaysHalts (Program p) {
  return (derefencesNull (new Program ("p (); *NULL;")));
}
```


Good news for theoreticians, bad news for tool builders/users



Implication

- Static analysis tools must *always* make compromises
  - Simplifying assumptions
  - Alias analysis: limited depth, precision
  - Paths: group infinitely many possible paths into a finite number
  - Values: sets of possible values

Compromises ⇒ Imperfection

- Unsound: will produce warnings for correct code ("false positives")
- Incomplete: will miss warnings for flawed code ("false negatives")
- Easy to have one:
  - Sound checker: every line is okay
  - Complete checker: every line is flawed
- Impossible to have both
  - Most tools sacrifice some of both

Gödel's lifetime employment guarantee for software security experts!

The Future Still Needs Us

- Imperfect tools mean human expertise is needed to understand output
  - Identify the real bugs and fix them
  - Coerce the tool to do the right thing

Recent State-of-the-Art: Model Checking Security Properties

Hao Chen (UC Davis), David Wagner (Berkeley)

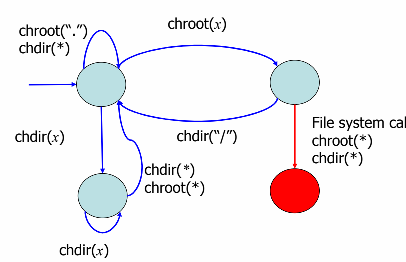
Hao Chen and David Wagner. *MOPS: an infrastructure for examining security properties of software.* ACM CCS 2002.

Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, Wei Tu. *Model Checking An Entire Linux Distribution for Security Violations.* ACSAC 2005

Model Checking

- Simulate execution paths
- Check if a path satisfies some model (finite state machine-like)
- Control state explosion:
  - Merge alike states
  - "Compaction": only consider things that matter for checked model

MOPS Example: Detect chroot() vulnerabilities



Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, Wei Tu. *Model Checking An Entire Linux Distribution for Security Violations.* ACSAC 2005

Checking RedHat

- 60 Million lines, 839 packages
  - Analyzed 87%
- Processing time: ~8 hours per rule
- Human time: up to 100 hours per rule
- Found 108 exploitable bugs
  - 41 TOC-TOU, 34 temporary files, 22 standard file descriptors, etc.

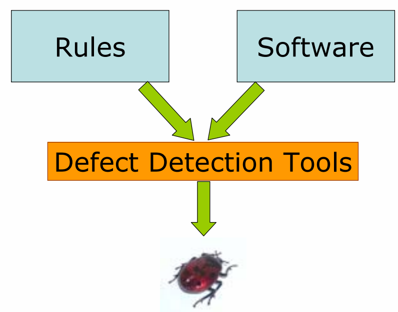
Checkpoint

- Retrospective: 10 years of program analysis
  - 10 years of program analysis
  - Limits of static analysis
  - Recent state-of-the-art
- **Two current projects:**
  - **Perracotta**
  - **N-Variant Systems**

Perracotta:
Automatic Inference and Effective Application of Temporal Specifications

Jinlin Yang
University of Virginia
www.cs.virginia.edu/perracotta

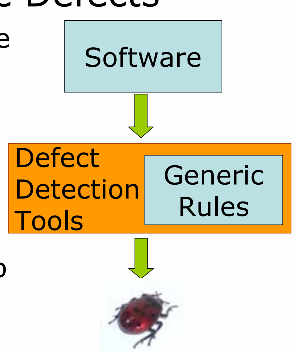
Defect Detection



Generic Defects

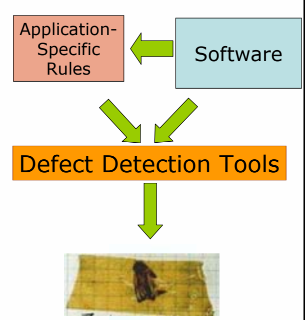
Generic defects are universal:
- Null pointer dereference
- Buffer overflow
- TOCTOU
- etc.

- Expert can develop rules for everyone



Application-Specific Defects

- Application-specific specifications depend on the implementations
- Powerful tools available
- **Rules rarely available**
- Limited adoption of such tools
- Many bugs escape into products



Deadlock Bug in Windows Vista

```
void PushNewInfo (struct1 s1,
     struct2 s2) {
  …
  QXWinLockacquire (s1.lock);
  …
  if ( s2.flag )
     GetData ( s1, FALSE );
  …
}
void GetData (struct1 s1,
     boolean locked ) {
  …
  if ( !locked ) {
     QXWinLockacquire (s1.lock);
  …
}
```

(Note: actual names have been changed to keep MSFT's lawyers happy and fit on slide)

Rule:
QXWinLock*acquire*
must alternate with
QXWinLock*release*

Regular expression:
(*acquire release*)*

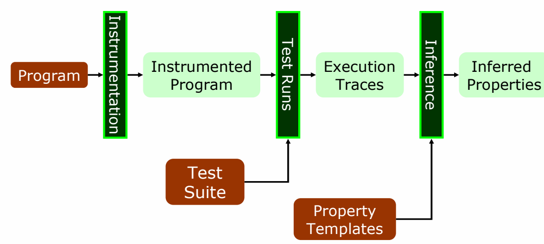
Available tools (e.g., ESP) can check this property, but only if they know the rule!

Getting Specifications from Developers

Problems

- Difficult to get approval (in most states)
- Manual specifications are still incomplete (and often wrong)
- Hard to keep specifications up-to-date when code changes
- **Solution:** guess specs from executions

Getting Specifications Automatically



Jinlin Yang and David Evans. *Dynamically Inferring Temporal Properties.* PASTE 2004.

Inference Example

Alternating Property Template: (*PS*)*

Collected Program Trace	Instantiated Alternating Properties (P, S)	Satisfied by Trace
acquire	acquire, release	acquire, release
release	acquire, open	X
open	acquire, close	X
acquire	release, acquire	X
release	release, open	X
acquire	release, close	X
close	open, acquire	X
release	open, release	X
	open, close	open, close
	close, acquire	X
	close, release	X
	close, open	X

Inference in Real World

- Must scale to large real systems
  - Traces have millions of events
- Infer properties from buggy traces
  - Hard to get perfect traces
- Separate wheat from chaff
  - Most properties are redundant or useless
  - Impossible to analyze all of them
  - Present properties in useful way

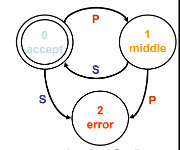
Naïve Inference Algorithm

- Match execution traces against regular expression templates
- n^2 possible substitutions for two-letter regular expressions
- Matches each substitution against the trace
- Time: $O(n^2L)$, doesn't scale to large traces!

Fast ($O(nL)$) Inference

- n distinct events, L events
- $n \times n$ array storing the states
- For event X, update X^{th} column *and* row

Alternating: *(PS)**



Initial

	A	B	C	D
A	2	0	0	0
B	0	2	0	0
C	0	0	2	0
D	0	0	0	2

ABCACDBC

	A	B	C	D
A	2	1	1	1
B	2	2	0	0
C	2	0	2	0
D	2	0	0	2

A**B**CACDBC

	A	B	C	D
A	2	0	1	1
B	2	2	1	1
C	2	2	2	0
D	2	2	0	2

…

Final

	A	B	C	D
A	2	0	2	2
B	2	2	2	2
C	2	2	2	2
D	2	2	2	2

Perracotta [Jinlin Yang]

http://www.cs.virginia.edu/perracotta/

- Inference engine implementation
- Windows kernel traces, 5.85 million events, 500 distinct events
- Only 14 minutes (naïve algorithm would take 5 days)

Dear Professor Evans,
My name is Tim McIntyre, and I work as General Counsel for Terracotta, Inc., a Java software start-up based in San Francisco. I came across your and your colleagues' webpage the other day (http://www.cs.virginia.edu/terracotta), and while I applaud your impeccable

Fast Inference is the Easy Part…

- Must scale to large real systems
  - Traces have millions of events
- Infer properties from buggy traces
  - Hard to get perfect traces
- Separate wheat from chaff
  - Most properties are redundant or useless
  - Impossible to analyze all of them
  - Present properties in useful way

Buggy Test Programs

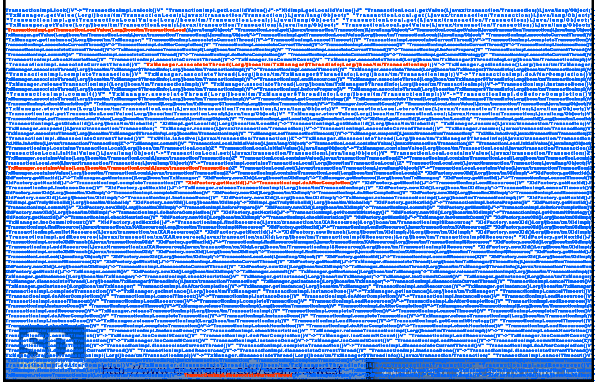
- Causes of imperfect traces:
  - Buggy programs
  - Trace collected by sampling
  - Missing information
- Detect dominant behaviors
  - *PS PS PS PS PS PS PS PS PS* ***P***
  - Matching (*PS*)* precisely misses the property
- Partition the original trace into subtraces
- Decide if each subtrace satisfies a template *T*
- Compute the percentage of satisfaction, P_T
- Rank pairs of events based on P_T

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. *Perracotta: Mining Temporal API Rules from Imperfect Traces.* ICSE 2006.

Selected Results - Windows

P_{AL}	Properties
0.993	ObpCreateHandle→ObpCloseHandle
0.988	GreLockDisplay→GreUnlockDisplay
0.985	RtlActivateActivationContextUnsafeFast→RtlDeactivateActivationContextUnsafeFast
0.982	KeAcquireInStackQueuedSpinLock→KeReleaseInStackQueuedSpinLock

JBoss – Inferred Properties



Selection Heuristic: Call Graph

```
void A(){            void x(){
  ...                  C();
  B();                 ...
  ...                  D();
}                    }
```

Case 1 Case 2

- C→D is often more interesting
- Keep A→B if B is not reachable from A

```
void KeSetTimer(){      void x(){
  KeSetTimerEx();         ExAcquireFastMutexUnsafe(&m);
}                         ...
                          ExReleaseFastMutexUnsafe(&m);
                        }
```


Selection Heuristic: Name Similarity

- The more similar two events are, the more likely that the properties are interesting
- Relative similarity between A and B
  - A has w_A words, B has w_B, w common words:
    $similarity_{AB} = 2w / (w_A + w_B)$
- For example (similarity = 85.7%):
  Ke Acquire In Stack Queued Spin Lock →
  Ke Release In Stack Queued Spin Lock

Windows Experiment Results

- 7611 properties (P_{AL} threshold = 0.90)
- Manual examination: <1% appear to be interesting
- Selection heuristics: 142 properties (1/53)
  - Use the call-graph of ntoskrnl.exe, edit dist > 0.5
- Small enough for manual inspection
  - 56 of 142 are "interesting" (40%)
  - Locking discipline
  - Resource allocation and deletion

Roadmap

- Inference:
  - Scales (Millions of events)
  - Infer properties from buggy traces
    - Partition and use satisfaction threshold
  - Separate wheat from chaff
    - Selection heuristics: ~40% left interesting
- Applications of inferred properties
  - **Program understanding**
  - Program evolution
  - Program verification

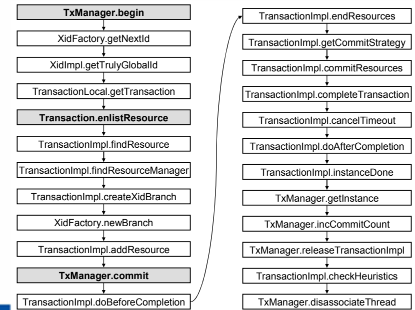
Program Understanding

- Help developers understand how to use a library
- 56 interesting rules of Windows kernel APIs
- Compared with Microsoft Research researchers' efforts in this area (SLAM)
  - Inferred four already documented rules
  - Inferred two other undocumented rules

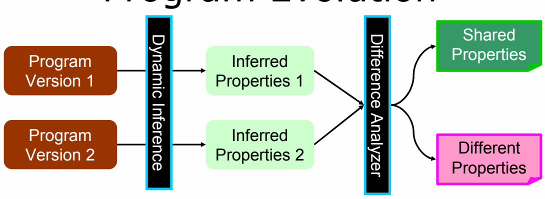
Chaining Method

- JBoss application server
  - Inferred 490 properties for the transaction manager
  - Edit distance not very useful
  - Too many properties to inspect
- Chaining method
  - Explore the relationships among *Alternating* properties: A->B, B->C, and A->C gives A->B->C chain
  - Potentially reduce n^2 properties to a chain of length n
- JBoss: 41 properties after chaining and call-graph reduction
- Longest chain consistent with the J2EE specification

JBoss: Chaining Properties



Program Evolution

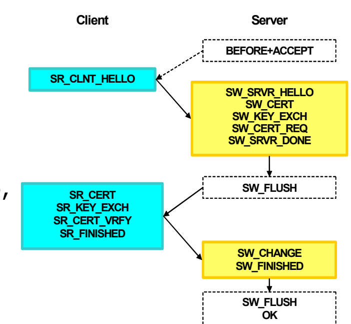


- Use inferred properties it identify differences
- Test beds: course assignments, OpenSSL

Jinlin Yang and David Evans. *Automatically Inferring Temporal Properties for Program Evolution.* ISSRE 2004.

Example: OpenSSL

- Widely used implementation of the Secure Socket Layer protocol
- 6 versions [0.9.6, 0.9.7, 0.9.7a-d]
- Handshake protocol



OpenSSL Evolution Results

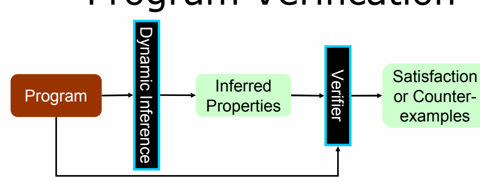
✓ indicates Perracotta inferred the Alternating property

	0.9.6	0.9.7	0.9.7a	0.9.7b	0.9.7c	0.9.7d
SR_KEY_EXCH→ SR_CERT_VRFY	✓	✓	✓	✓		
SW_CERT→ SW_KEY_EXCH		✓	✓	✓	✓	✓

Fixed bug

Documented improvement

Program Verification



- ESP: path-sensitive property checker
  - Found previously unknown bugs in Windows
  - Development team confirmed and fixed bugs

Summary of Applications

- Program understanding
  - Discover API usage rules, 56 rules for Windows kernel APIs
  - Revealed the mechanism of legacy system, a 24-state FSM of JBoss transaction module
- Program verification
  - Found many previously unknown bugs in Windows
- Program evolution
  - Identified differences among different versions
  - Exposed bugs and intended improvements
  - Demonstrated important properties have been preserved

Checkpoint

- Retrospective: 10 years of program analysis
  - 10 years of program analysis
  - Limits of static analysis
  - Recent state-of-the-art
- Two current projects:
  - Perracotta
  - **N-Variant Systems**

Inevitability of Failure

- Despite all the best efforts to build secure software, we will still fail (or have to run programs that failed)
- Run programs in ways that make it harder to exploit vulnerabilities

Security Through Diversity

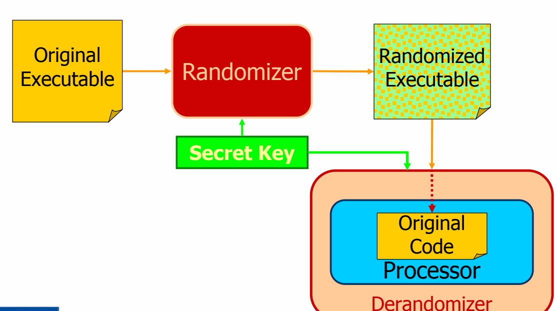
- Today's Computing Monoculture
  - Exploit can compromise billions of machines since they are all running the same software
- Biological Diversity
  - All successful species use very expensive mechanism (sex) to maintain diversity
- Computer security research: [Cohen 92], [Forrest+ 97], [Cowan+ 2003], [Barrantes+ 2003], [Kc+ 2003], [Bhatkar+2003], [Just+ 2004], [Bhatkar, Sekar, DuVarney 2005]

Instruction Set Randomization

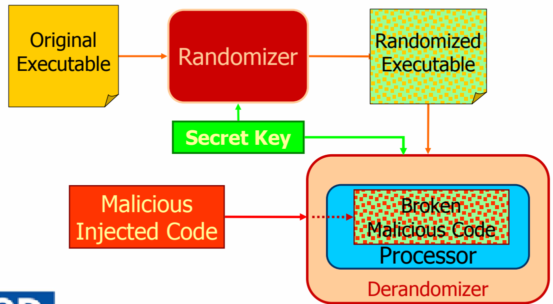
[Barrantes+, CCS 03] [Kc+, CCS 03]

- Code injection attacks depend on knowing the victim machine's instruction set
- Defuse them all by making instruction sets different and secret
  - Its expensive to design new ISAs and build new microprocessors

Automating ISR



ISR Defuses Attacks



ISR Designs

	Columbia [Kc 03]	RISE [Barrantes 03]
Randomization Function	XOR or 32-bit transposition	XOR
Key Size	32 bits (same key used for all locations)	program length (each location XORed with different byte)
Transformation Time	Compile Time	Load Time
Derandomization	Hardware	Software (Valgrind)

How secure is ISR?

Slows down an attack about 6 minutes!

Can use probe injections to incrementally guess the key byte-by-byte (under the right conditions)

Ana Nora Sovarel, David Evans and Nathanael Paul. *Where's the FEEB?* Effectiveness of Instruction Set Randomization. *USENIX Security Symposium*, August 2005.

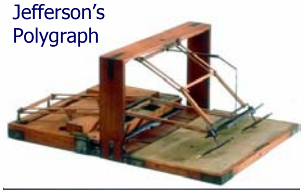
Better Solution

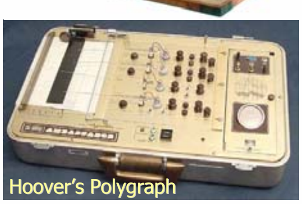
- Avoid secrets!
  - Keeping them is hard
  - They can be broken or stolen
- Prove security properties without relying on assumptions about secrets or probabilistic arguments

Polygraphing Processes: N-Variant Systems for Secretless Security

work with Ben Cox, Jack Davidson, Adrian Filipi, Jason Hiser, Wei Hu, John Knight, Anh Nguyen-Tuong, Jonathan Rowanhill

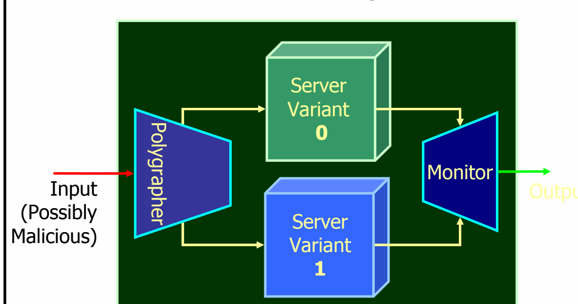
Jefferson's Polygraph





Hoover's Polygraph

2-Variant System



N-Version Programming

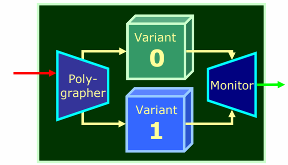
[Avizienis & Chen, 1977]

- Multiple teams of programmers implement same spec
- Voter compares results and selects most common
- No guarantees: teams may make same mistake

N-Variant Systems

- Transformer automatically produces diverse variants
- Monitor compares results and detects attack
- Guarantees: variants behave differently on particular input classes

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, scheduler, calling convention, …
- Monitor
  - Observes variants
  - Delays external effects until all variants agree
  - Initiates recovery if variants diverge

Variants Requirements

- *Detection* Property

  Any attack that compromises Variant 0 causes Variant 1 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:

  $\mathcal{A}_0(\mathrm{S}_0) \equiv \mathcal{A}_1(\mathrm{S}_1)$

  Actual states are different, but abstract states are equivalent

Memory Partitioning

- Variation
  - Variant 0: addresses all start with **0**
  - Variant 1: addresses all start with **1**
- Normal Equivalence
  - Map addresses to same address space
- Detection Property
  - Any *absolute* load/store is invalid on one of the variants

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**
  - At run-time check bit and remove it
    - Low-overhead software dynamic translation using Strata [Scott, et al., CGO 2003]
- Normal Equivalence: Remove the tag bits
- Detection Property
  - Any (tagged) opcode is invalid on one variant
  - Injected code (identical on both) cannot run on both

Implementing N-Variant Systems

- Competing goals:
  - Isolation: of monitor, polygrapher, variants
  - Synchronization: variants must maintain normal equivalence (nondeterminism)
  - Performance: latency (wait for all variants to finish) and throughput (increased load)
- Two implementations:
  - Divert Sockets (prioritizes isolation over others)
    - Maintaining normal equivalence is too difficult
  - Kernel modification (sacrifices isolation for others)

Kernel Implementation [Ben Cox]

- Modify process table to record variants
- Create new fork routine to launch variants
- Intercept system calls:
  - 289 calls in Linux
  - Check parameters are the same for all variants
  - Make call once
- Low overhead, lack of isolation

Wrapping System Calls

- I/O system calls (process interacts with external state) (e.g., open, read, write)
  - Make call once, send same result to all variants
- Process system calls (e.g, fork, execve, wait)
  - Make call once per variant, adjusted accordingly
- Special:
  - mmap: each variant maps segment into own address space, only allow MAP_ANONYMOUS (shared segment not mapped to a file) and MAP_PRIVATE (writes do not go back to file)

System Call Wrapper Example

```
ssize_t sys_read(int fd, const void *buf, size_t count) {
  if (hasSibling (current)) {
    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) {
       // I'm second variant, sibling is waiting
       if (parameters match) { // match depends on variation
          perform system call
          save result and data in kernel buffer
          wake up sibling
          return result;
       } else {
         DIVERGENCE ERROR! }
    } else { // sibling is in a different system call!
       DIVERGENCE ERROR! } }
  ...
}
```


Overhead

Results for Apache running WebBench 5.0 benchmark

Description	Unmodified Apache, unmodified kernel	2-variant system, address space partitioning	2-variant system, instruction tagging
Throughput (MB/s)	6.46	5.57	4.01
Latency (ms)	9.06	10.52	14.84

14% decrease in throughput

68% increase in latency

Diversity depends on your adversary

Slide from my USENIX Security 2004 Talk, *What Biology Can (and Can't) Teach us about Security*

N-Variant Summary

- Producing artificial diversity is easy
  - Defeats undetermined adversaries
- Keeping secrets is hard
- N-variant systems framework offers provable defense without secrets
  - Effectiveness depends on whether variations vary things that matter to attack

Questions?

N-Variant Systems:
http://www.cs.virginia.edu/nvariant
Ben Cox, Jack Davidson, Adrian Filipi, Jason Hiser, Wei Hu, John Knight, Anh Nguyen-Tuong, Jonathan Rowanhill

Perracotta:
http://www.cs.virginia.edu/perracotta
Jinlin Yang
Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das (Microsoft Research)

Funding: National Science Foundation