Automated Software Testing

for the 21st Century

Patrice Godefroid

Microsoft Research

Outline

Two parts:

1. Some recent advances on automated software testing
   - Technical developments
   - Applications
2. Some current trends in the software industry
   - And their impact on software testing

Automatic Code-Driven Test Generation

Problem:

Given a sequential program with a set of input parameters, generate a set of inputs that maximizes code coverage

= "automate test generation using program analysis"

This is not "model-based testing" (= generate tests from an FSM spec)

Example: `Powerpnt.exe <filename>`

- Millions of lines of C/C++, complex input format, dynamic memory allocation, data structures of various shapes and sizes, pointers, loops, procedures, libraries, system calls, etc.

How? (1) Static Test Generation

- Static analysis to partition the program's input space [King76,…]
- Ineffective whenever symbolic reasoning is not possible
  - which is frequent in practice… (pointer manipulations, complex arithmetic, calls to complex OS or library functions, etc.)

Example:

```
int obscure(int x, int y) {
  if (x==hash(y)) error();
  return 0;
}
```

Can't statically generate values for x and y that satisfy "x==hash(y)" !

How? (2) Dynamic Test Generation

- Run the program (starting with some random inputs), gather constraints on inputs at conditional statements, use a constraint solver to generate new test inputs
- Repeat until a specific program statement is reached [Korel90,…]
- Or repeat to try to cover ALL feasible program paths:
  DART = Directed Automated Random Testing
  = systematic dynamic test generation [PLDI'05,…]
  - detect crashes, assertion violations, use runtime checkers (Purify, Valgrind, AppVerifier,…)

DART = Directed Automated Random Testing

Example:

```
int obscure(int x, int y) {
  if (x==hash(y)) error();
  return 0;
}
```

Run 1 : - start with (random) x=33, y=42

- execute concretely and symbolically:
  if (33 != 567) | if $\boxed{\text{(x != hash(y))}}$
  constraint too complex
  → simplify it: x != 567
- solve: x==567 → solution: x=567
- new test input: x=567, y=42

Run 2 : the other branch is executed

All program paths are now covered !

- Observations:
  - Dynamic test generation extends static test generation with additional runtime information: it is more powerful
    - see [DART in PLDI'05], [PLDI'11]
  - The number of program paths can be infinite: may not terminate!
  - Still, DART works well for small programs (1,000s LOC)
  - Significantly improves code coverage vs. random testing

DART Implementations

- Defined by symbolic execution, constraint generation and solving
  - Languages: C, Java, x86, .NET,…
  - Theories: linear arithmetic, bit-vectors, arrays, uninterpreted functions,…
  - Solvers: lp_solve, CVCLite, STP, Disolver, Z3,… → SMT solvers!
- Examples of tools/systems implementing DART:
  - EXE/EGT (Stanford): independent ['05-'06] closely related work (became KLEE)
  - CUTE = same as first DART implementation done at Bell Labs
  - SAGE (CSE/MSR) for x86 binaries and merges it with "fuzz" testing for finding security bugs (more later)
  - PEX (MSR) for .NET binaries in conjunction with "parameterized-unit tests" for unit testing of .NET programs
  - YOGI (MSR) for checking the feasibility of program paths generated statically using a SLAM-like tool
  - Vigilante (MSR) for generating worm filters
  - BitScope (CMU/Berkeley) for malware analysis
  - CatchConv (Berkeley) focus on integer overflows
  - Splat (UCLA) focus on fast detection of buffer overflows
  - Apollo (MIT/IBM) for testing web applications

…and many more!

The Rise of SMT Solvers

- SAT Solvers for propositional logic
  - Ex: Is formula F = *p and (not q)* satisfiable?

    Answer: yes with $p = true$ and $q = false$

- SMT = Satisfiability Modulo Theories
  - Allows more expressive formulas, useful to model sw features
  - Ex: Let F = $\boxed{(b + 2 = c)}$ *and* $(\boxed{f}(\boxed{read}(\boxed{write}(a,\boxed{b},\boxed{3}),\ \boxed{c\text{-}2}) \neq \boxed{f}(\boxed{c\text{-}b\text{+}1}))$

Arithmetic

Array Theory

Uninterpreted Functions

Is formula F satisfiable modulo theory T ?

(A theory T is a set of formulas)

  - SMT solvers have specialized algorithms for each T, and have improved dramatically over the last 10 years

An Application: SAGE @ Microsoft

- #1 application of SMT solvers today (CPU usage)
- Why? Security Testing
- Software security bugs can be very expensive:
  - Cost of each Microsoft Security Bulletin: $Millions
  - Cost due to worms (Slammer, CodeRed, Blaster, etc.): $Billions
- Many security vulnerabilities are in file & packet parsers
  - Ex: MS Windows includes parsers for hundreds of file formats
- Security testing: "hunting for million-dollar bugs"
  - Write A/V (always exploitable), Read A/V (sometimes exploitable), NULL-pointer dereference, division-by-zero (harder to exploit but still DOS attacks), etc.

Hunting for Security Bugs

- Main techniques used by “black hats”:
  - Code inspection (of binaries) and
  - Blackbox fuzz testing
- Blackbox fuzz testing:
  - A form of blackbox random testing [Miller+90]
  - Randomly fuzz (=modify) a well-formed input
  - Grammar-based fuzzing: rules that encode “well-formed”ness + heuristics about how to fuzz (e.g., using probabilistic weights)
- **Heavily** used in security testing
  - Simple yet effective: many bugs found this way…
  - At Microsoft, fuzzing is mandated by the SDL →

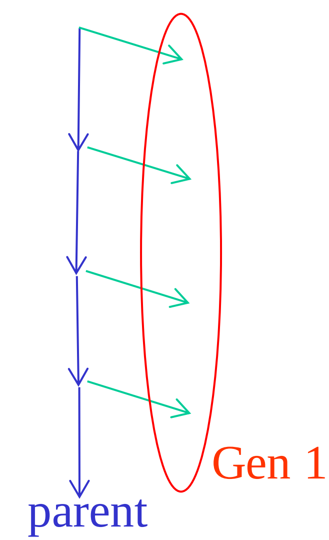
Introducing Whitebox Fuzzing [NDSS'08]

Idea: mix fuzz testing with dynamic test generation

- Dynamic symbolic execution to collect constraints on inputs, negate those, solve new constraints to get new tests, repeat → "systematic dynamic test generation" (= DART)

(Why dynamic ? Because most precise ! [PLDI'05, PLDI'11])

- Apply to large applications (not unit)
- Start with a well-formed input (not random)
- Combine with a generational search (not DFS)
  - Negate 1-by-1 each constraint in a path constraint
  - Generate many children for each parent run
  - Challenge all the layers of the application sooner
  - Leverage expensive symbolic execution



- Implemented in the tool SAGE

Example

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 4) crash();
}
```

input = "good"

Path constraint:

Path constraint			
I_0!='b'	→	I_0='b'	bood
I_1!='a'	→	I_1='a'	gaod
I_2!='d'	→	I_2='d'	godd
I_3!='!'	→	I_3='!'	goo!

SMT solver

good

→ SAT

Gen 1

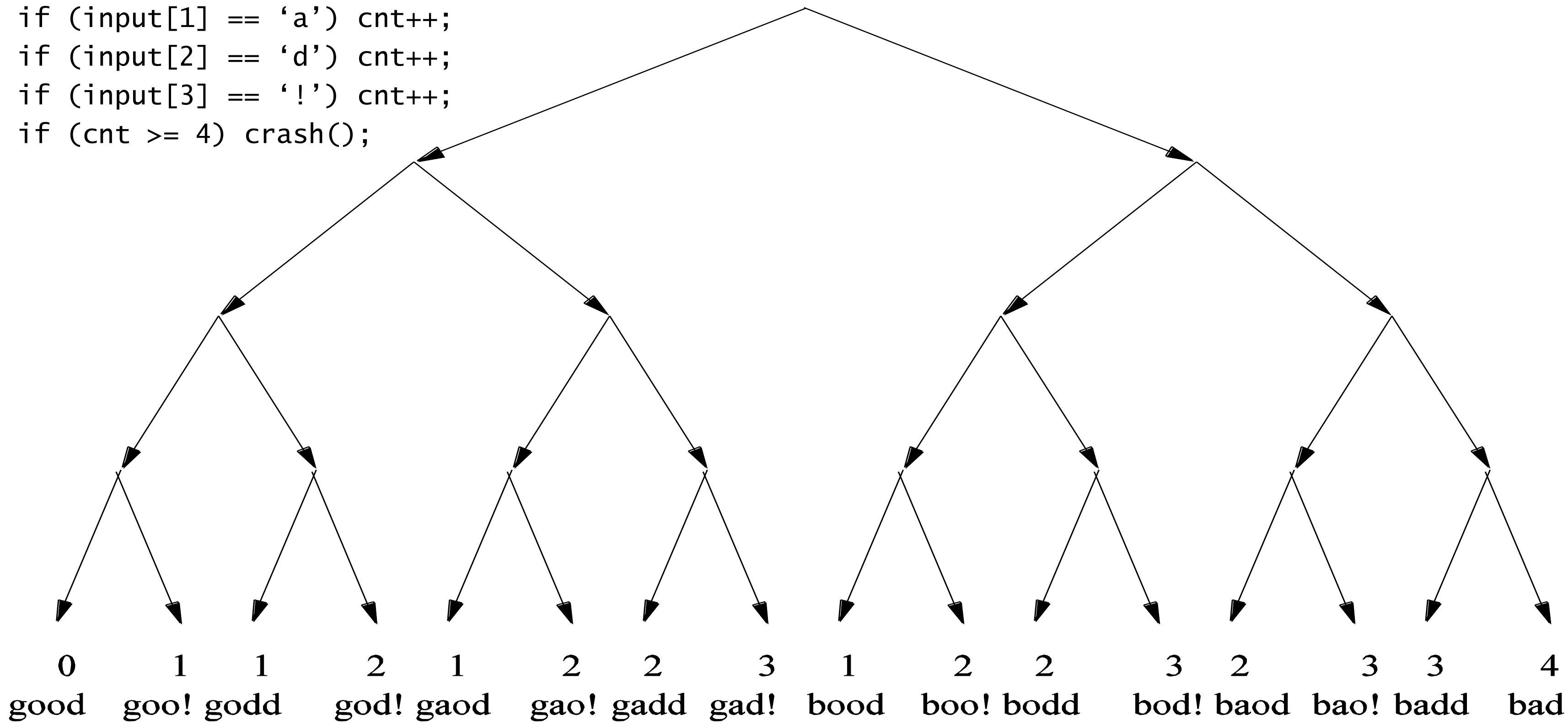
Negate each constraint in path constraint

Solve new constraint → new input

The Search Space

```
void top(char input[4])
{
   int cnt = 0;
   if (input[0] == 'b') cnt++;
   if (input[1] == 'a') cnt++;
   if (input[2] == 'd') cnt++;
   if (input[3] == '!') cnt++;
   if (cnt >= 4) crash();
}
```

If symbolic execution is perfect and search space is small, this is verification !



Some Experiments

Most much (100x) bigger than ever tried before!

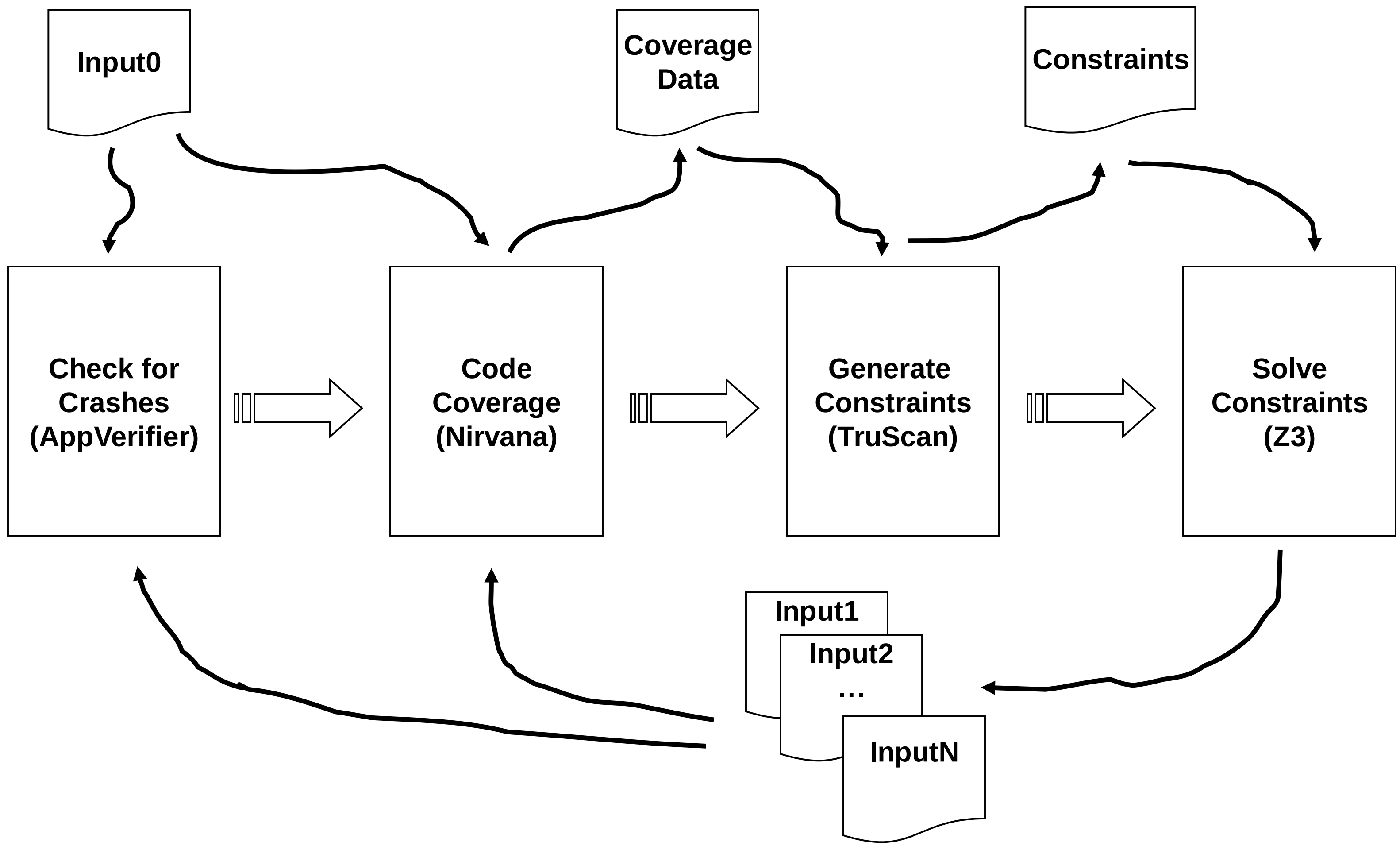
- Seven applications – 10 hours search each

App Tested	#Tests	Mean Depth	Mean #Instr.	Mean Input Size
ANI	11468	178	2,066,087	5,400
Media1	6890	73	3,409,376	65,536
Media2	1045	1100	271,432,489	27,335
Media3	2266	608	54,644,652	30,833
Media4	909	883	133,685,240	22,209
Compressed File Format	1527	65	480,435	634
Excel	3008	6502	923,731,248	45,064

SAGE (Scalable Automated Guided Execution)

- Whitebox fuzzing introduced in SAGE
- Performs symbolic execution of x86 execution traces
  - Builds on Nirvana, iDNA and TruScan for x86 analysis
  - Don't care about language or build process
  - Easy to test new applications, no interference possible
- Can analyse any file-reading Windows applications
- Several optimizations to handle huge execution traces
  - Constraint caching and common subexpression elimination
  - Unrelated constraint optimization
  - Constraint subsumption for constraints from input-bound loops
  - "Flip-count" limit (to prevent endless loop expansions)

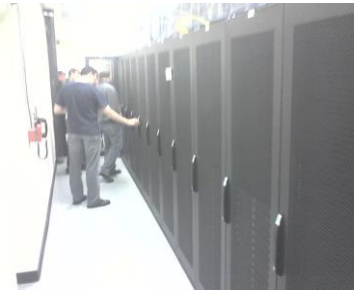
SAGE Architecture

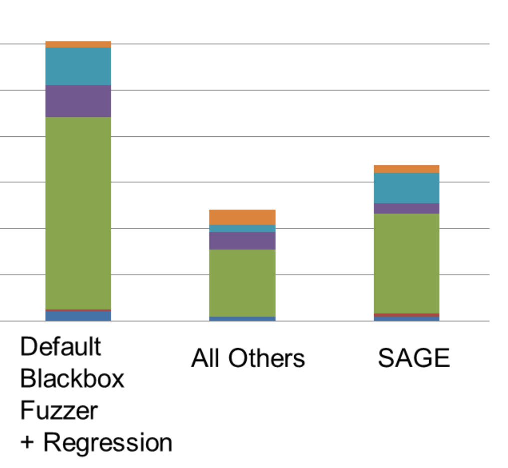


SAGE Results

Since 2007: many new security bugs found
(missed by blackbox fuzzers, static analysis)

- Apps: image decoders, media players, document processors,…
- Bugs: Write A/Vs, Read A/Vs, Crashes,…
- Many triaged as "security critical, severity 1, priority 1" (would trigger Microsoft security bulletin if known outside MS)
- Example: WEX Security team for Win7
  - Dedicated fuzzing lab with 100s machines
  - 100s apps (deployed on 1 billion+ computers)
  - ~1/3 of all fuzzing bugs found by SAGE !





Impact of SAGE (in Numbers)

- 500+ machine-years
  - Runs in the largest dedicated fuzzing lab in the world
  - Largest computational usage ever for any SMT solver
- 100s of apps, 100s of bugs (missed by everything else)
  - Bug fixes shipped quietly (no MSRCs) to 1 Billion+ PCs
  - Millions of dollars saved (for Microsoft and the world)
- "Practical Verification":
  - Eradicate all buffer overflows in all Windows parsers
    - <5 security bulletins in all SAGE-cleaned Win7 parsers, 0 since 2011
    - If nobody can find bugs in P, P is observationally equiv to "verified"!
    - Reduce costs & risks for Microsoft, increase those for Black Hats

2000 2005 2010 2015

Blackbox Fuzzing | Whitebox Fuzzing | "Practical Verification"

What Next?

1. Better Depth: Towards Formal Verification
   - When can we safely stop testing?
   - When we know that there are no more bugs ! = "Verification"
   - Software Model Checking = verification by exhaustive testing (state-space exploration)
   - Active area of research…

2. Better Breadth: More Applications
   - Beyond file fuzzing
   - What other "killer apps"?
   - Active area of research…

More On the Research Behind SAGE

- How to recover from imprecision in symbolic exec.? PLDI'05, PLDI'11
- How to scale symbolic exec. to billions of instructions? NDSS'08
- How to check efficiently many properties together? EMSOFT'08
- How to leverage grammars for complex input formats? PLDI'08
- How to deal with path explosion ? POPL'07, TACAS'08, POPL'10, SAS'11
- How to reason precisely about pointers? ISSTA'09
- How to deal with floating-point instructions? ISSTA'10
- How to deal with input-dependent loops? ISSTA'11
- How to synthesize x86 circuits automatically? PLDI'12
- How to run 24/7 for months at a time? ICSE'13

+ research on constraint solvers

References: see http://research.microsoft.com/users/pg

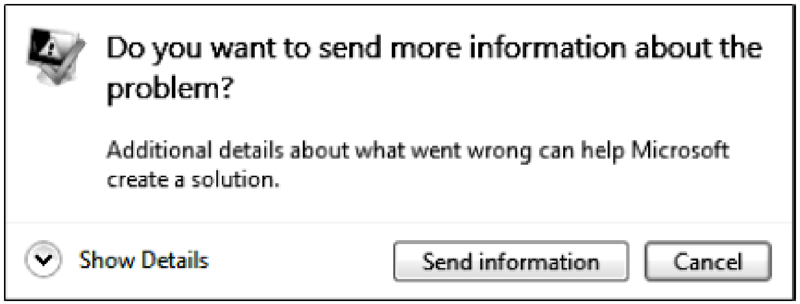
Some Current Trends in the Software Industry

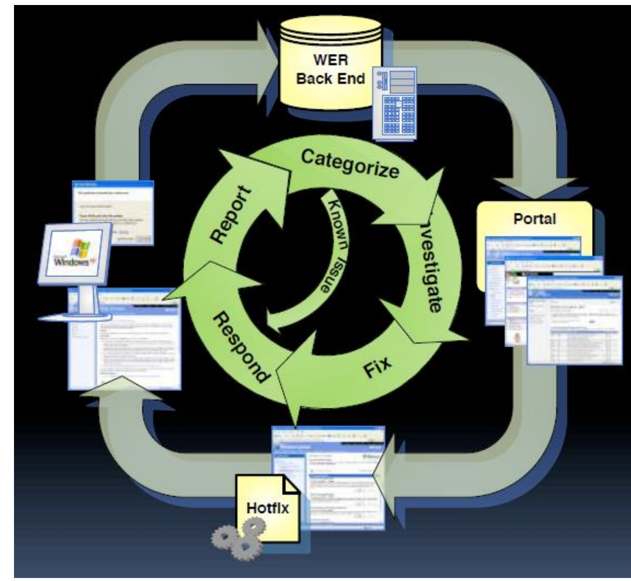
And their Impact on Software Testing

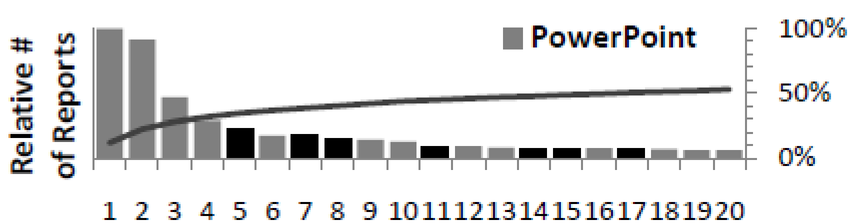
Illustrated with Examples from Microsoft

Telemetry

- Ex: Microsoft's Windows Error Reporting (WER)





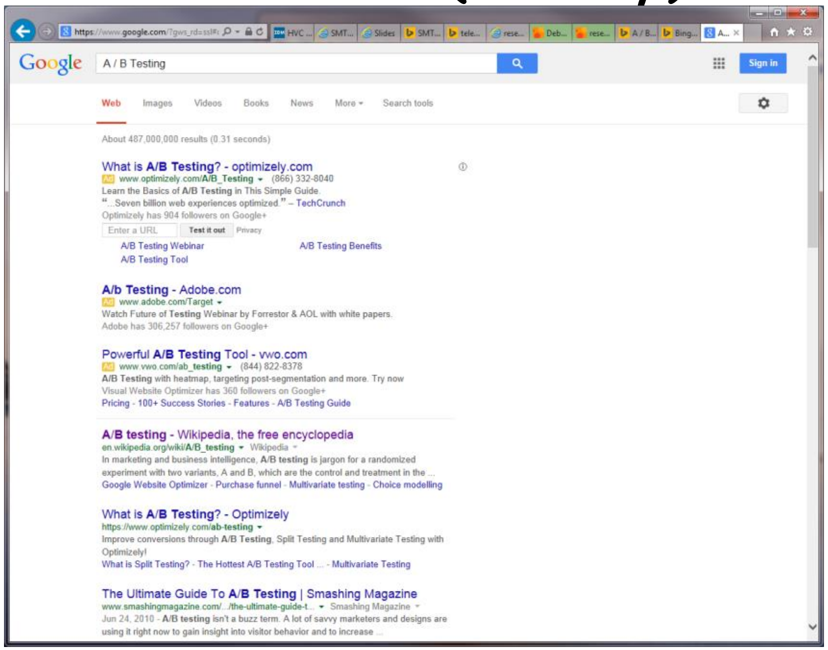


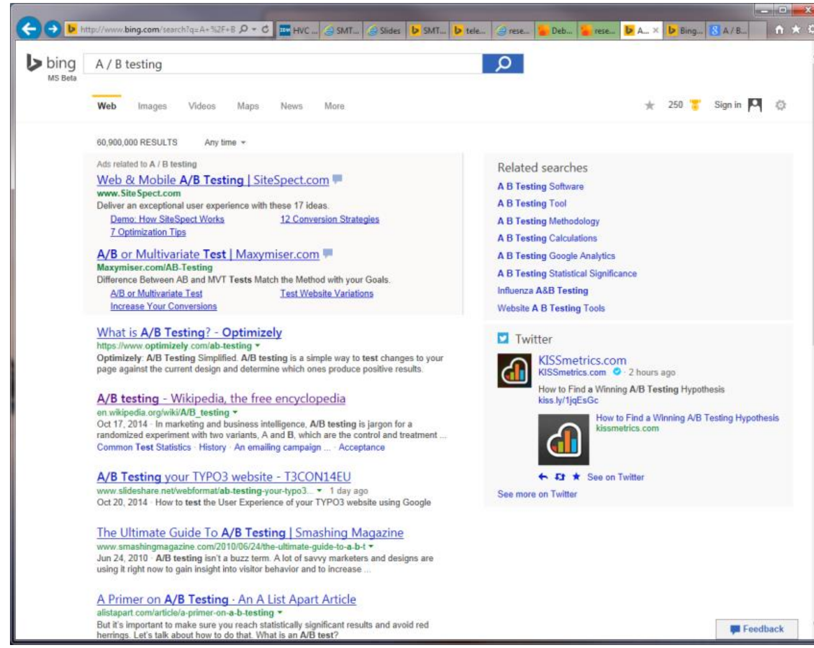
- Valuable automatic feedback
  - Huge help to prioritize, improve customer satisfaction
    - Heavily skewed distributions, maximum benefit from fixed budget
  - Not just Microsoft software ! (>7000 products)

Credit: G. Hunt, J. Larus

"A/B Testing"

- For Services (mostly)





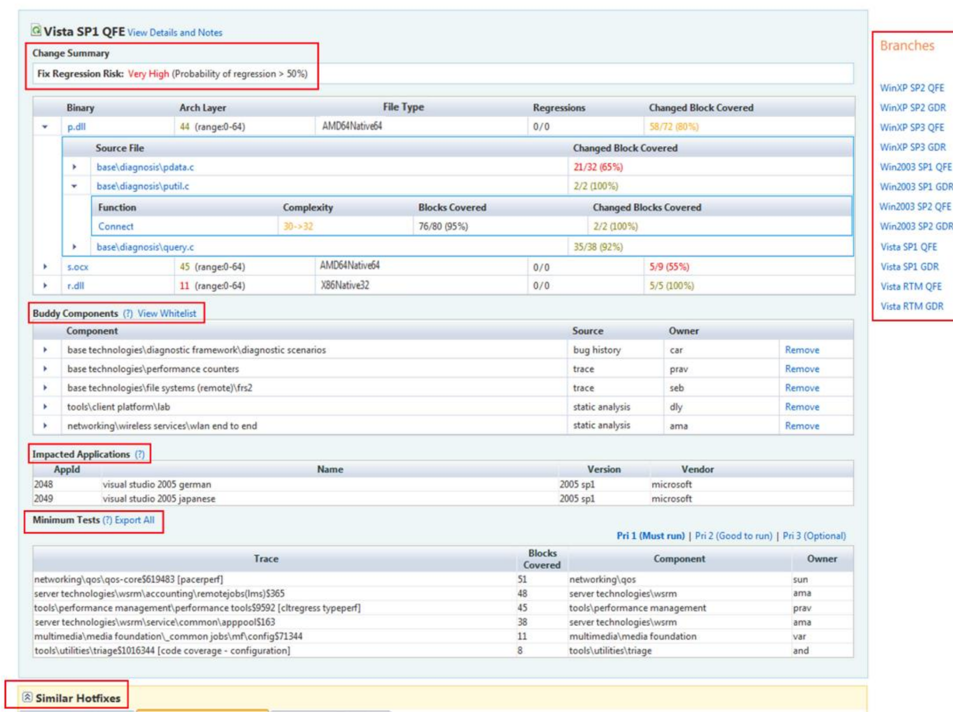
- Deploy first to a small set of users
  - Users are testers, monitoring, log analysis
  - Fix bugs on server side quickly, quietly and cheaply…
  - When stable, deploy further

New World of Smartphones and Clouds

- Lots of new code development !
- How much testing? Varies widely !
  - Many apps are poorly tested
  - Some apps (high-end) are very well tested
    - Small margin for failure (ready-at-launch)
    - Otherwise re-brand/re-launch

Big Data: Program Analysis in the Cloud

- The Cloud is also an opportunity
  - For program analysis, testing, fuzzing, etc.
  - Move software development (and testing) assets to the Cloud
  - Mine data about code, edits (churn), bug DBs, HR-data, etc.
  - Failure-prediction models, change analysis, test prioritization, etc.
    - Ex: Crane @ MSR
  - Continuous monitoring, logging, analysis, etc.
  - Enables new large-scale sophisticated analyses !
    - "Empirical Software Engineering" research

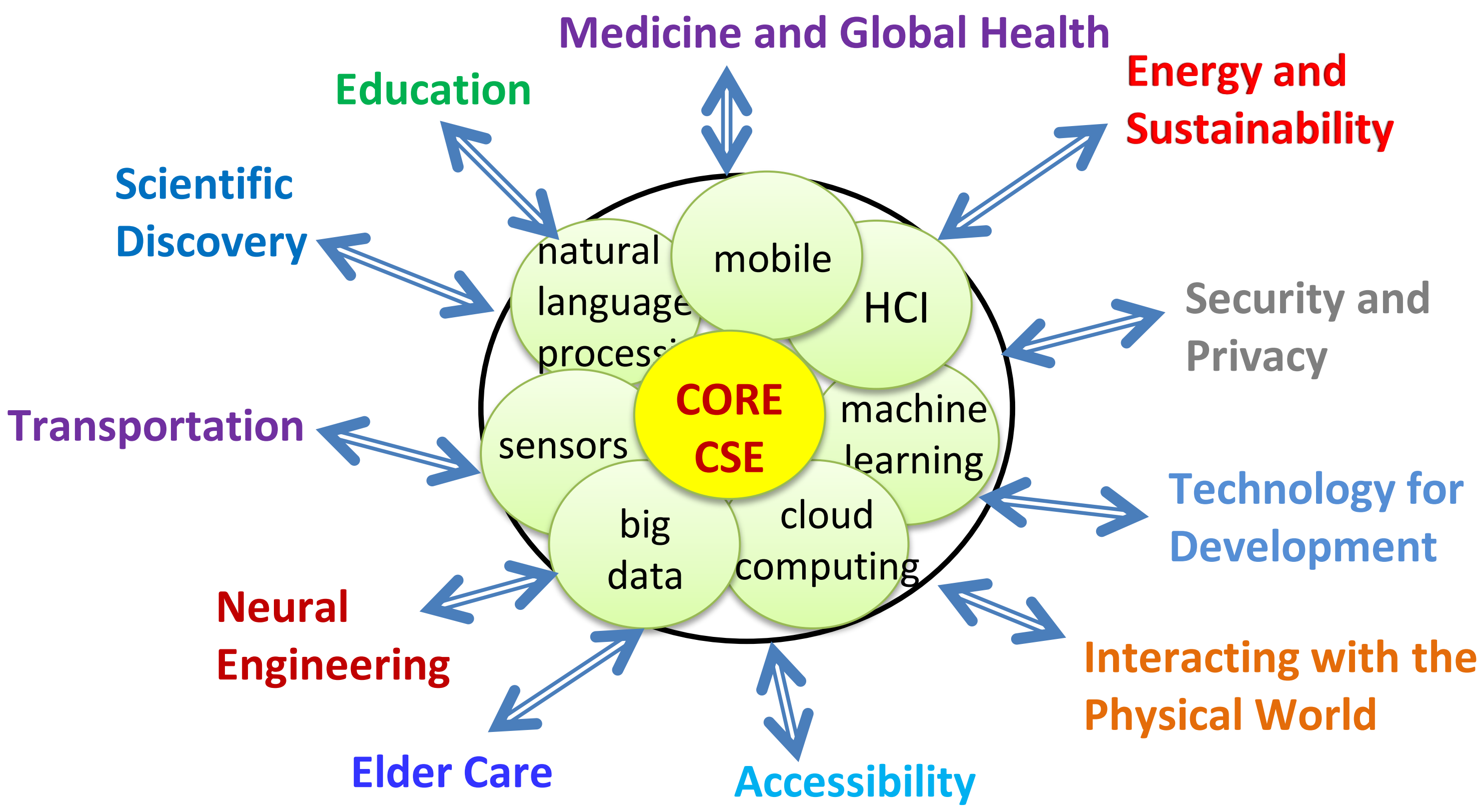


Credit: N. Nagappan

Testing Process

- Separate test organization or "combined engineering" ?
- Current trend: "Agile" software development
  - Speed: Ship frequently, incrementally, independently
    - Especially for services, continuous improvements
  - Modularity: Fine-grained components, libraries, services
  - Test-driven development, devs write ("unit") tests
  - Separate, specialized, end-to-end testing (e.g., for security)
  - Evolution of the Dev:Test ratio
    - Old Microsoft: 1:1
    - New Microsoft: towards 10:1 ? (Like Google, Facebook, etc.)
- Impact on quality? When does this work and not work?

The Rapidly Expanding World of Computing



What testing for such systems?

Credit: E. Lazowska

Conclusions

Automated Software Testing for the 21st Century :

- Some recent advances on automated software testing
  - Dynamic test generation, SMT solvers, whitebox fuzzing
  - Applications to large-scale security testing (500+ machine-years)
  - What next? Towards verification, more applications…
    - Active area of research !
- Some current trends in the software industry
  - Telemetry, A/B Testing, Agile Dev&Test, Cloud & Big Data,…
  - Impact on software testing…

We live in a world of remarkable innovation, diversity, and opportunity

The same is true for testing !