Fuzzing @ Microsoft

A Research Perspective

Patrice Godefroid

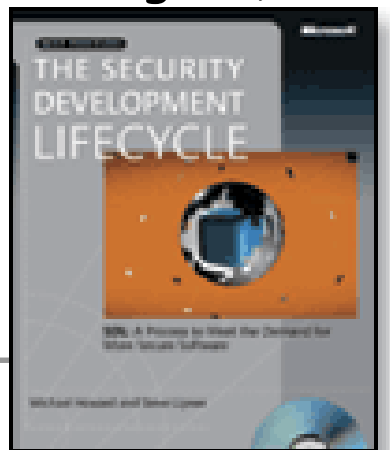
Microsoft Research

Security is Critical to Microsoft

- 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 due to programming errors in file and 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 (2006-)

Idea: mix fuzz testing with dynamic test generation

- Symbolic execution
- Collect constraints on inputs
- Negate those, solve with constraint solver, generate new inputs
- Repeat: "systematic dynamic test generation" (=DART)

By construction, new inputs exercise new program paths !

→ Better code coverage

→ Finds more bugs !

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"

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),
  use dynamic symbolic execution,
  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 blend with model checking !
  - 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 (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,…
- 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 (MSR) for x86 binaries and merges it with "fuzz" testing for finding security bugs (more next)
  - 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!

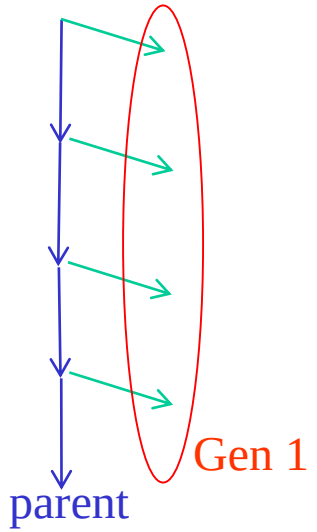
Introducing Whitebox Fuzzing [NDSS'08]

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)

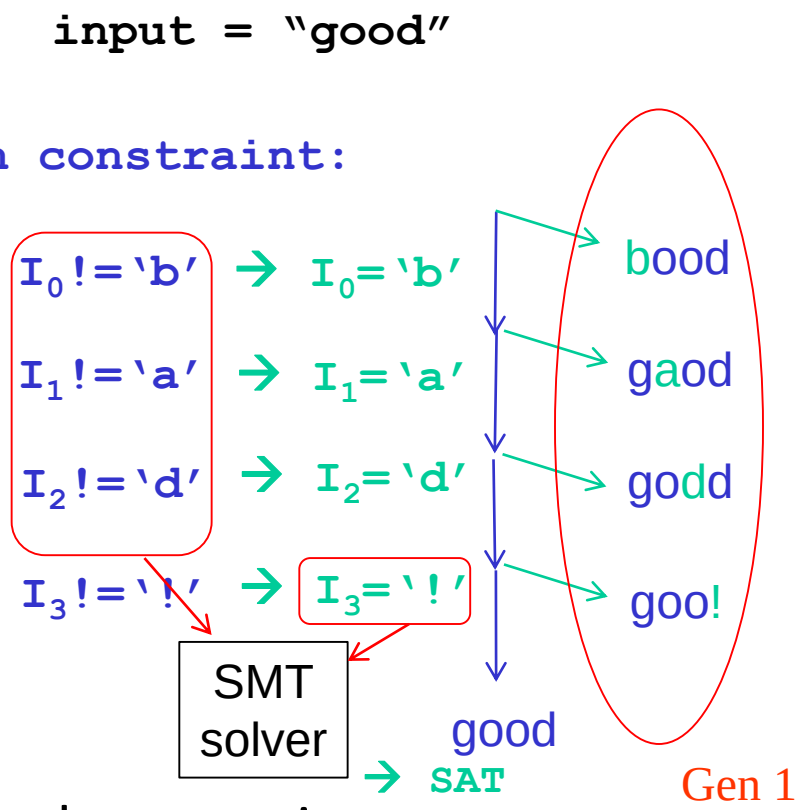
(Why dynamic ? Because most precise ! [PLDI'05, PLDI'11])

- 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



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();
}
```



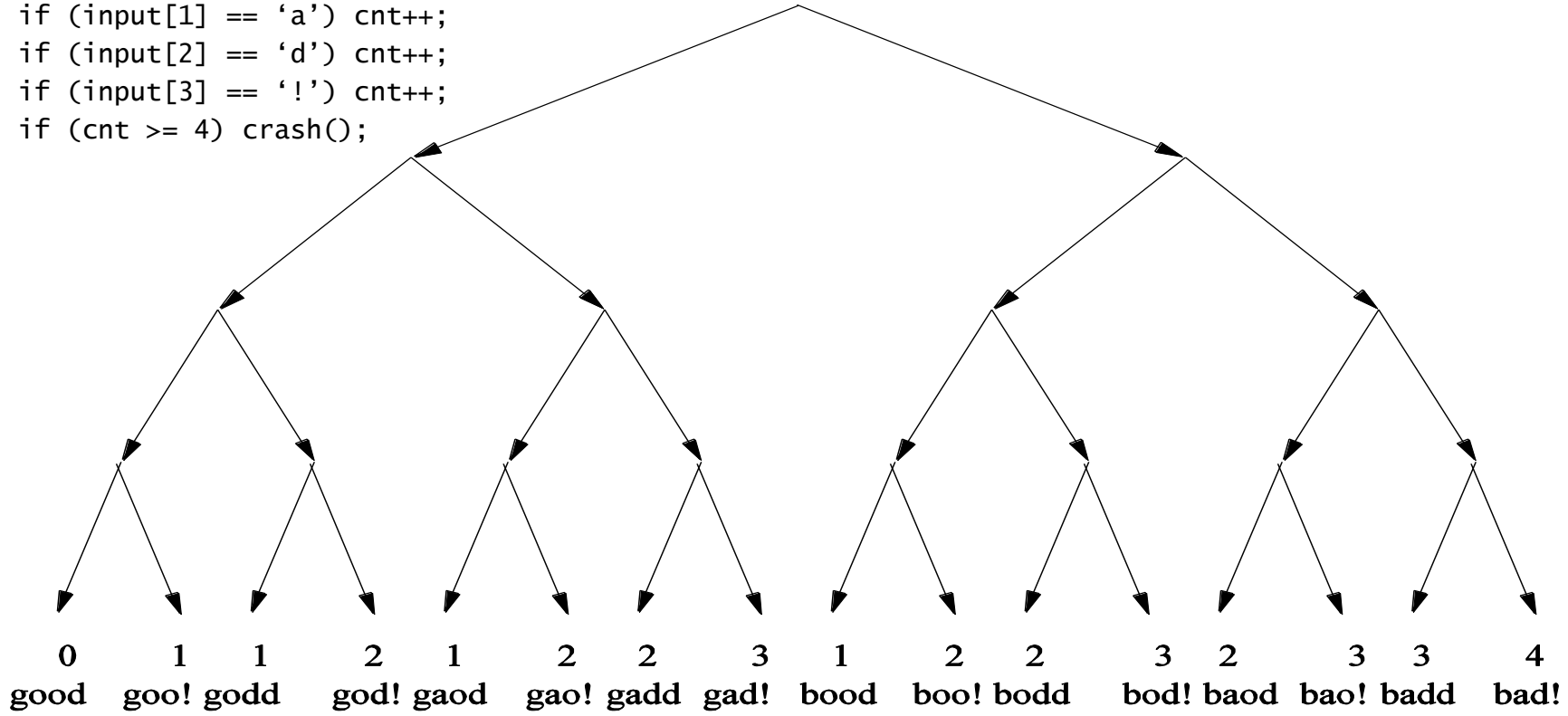
Negate each constraint in path constraint

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!

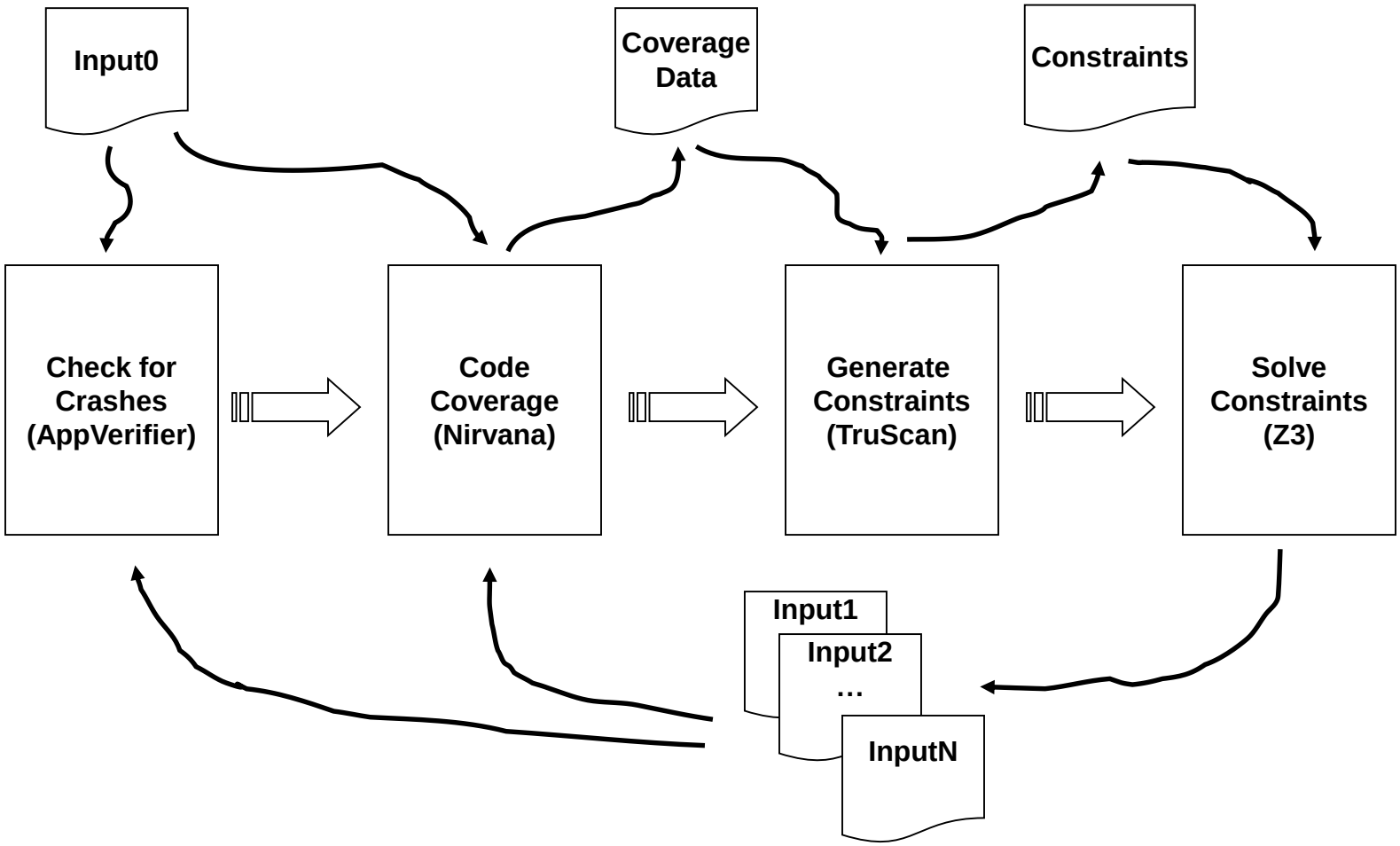
- Seven applications – 10 hours search each

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)

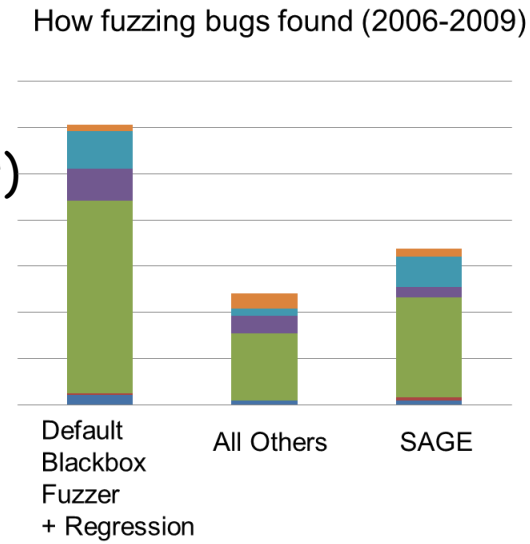
SAGE Architecture



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)

- 1000+ machine-years
  - Ran in the world-largest dedicated fuzzing lab, now in the cloud
  - 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
    - If nobody can find bugs in P, P is observationally equiv to "verified"!
    - Reduce costs & risks for Microsoft, increase those for Black Hats

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

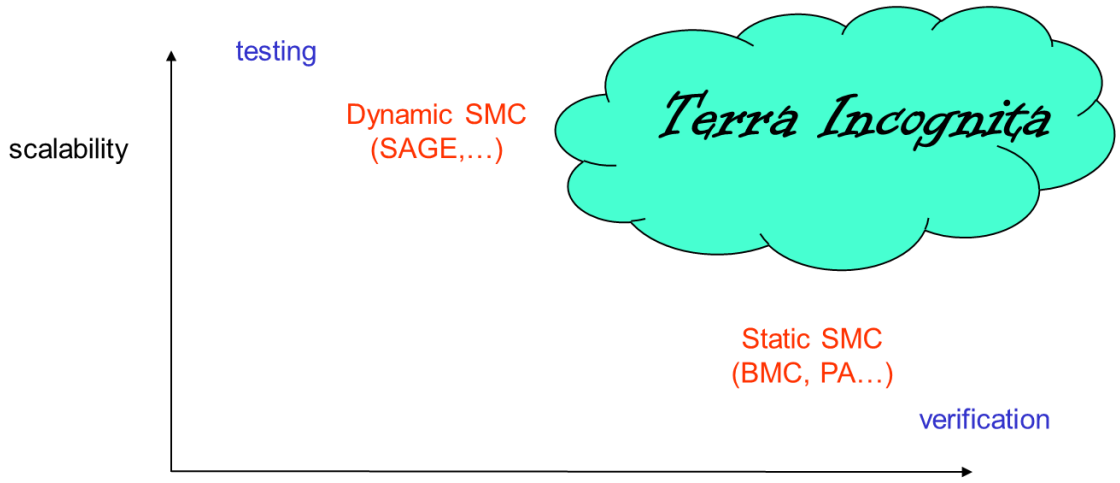
References: see http://research.microsoft.com/users/pg

Post-SAGE Related Work

- Whitebox fuzzing is now popular
  - Ex: now mainstream CS topic in Security, PL and SE conferences
  - Ex: used by all top finalists of DARPA Cyber Grand Challenge
- Many approximations and variants
  - Ex: greybox fuzzing (coverage-guided blackbox fuzzing, AFL)
  - Ex: hybrid fuzzing (mixes greybox + whitebox fuzzing, Driller)
  - Etc.

What Next?

1. Better Depth: Towards Formal Verification
   - When can we safely stop testing?
   - When we know that there are no more bugs ! = "Verification"
   - Two main problems:
     - Imperfect symbolic execution and constraint solving
     - Path explosion
   - Active area of research…



Ex: ANI Windows Image Parser Verification

- The ANI Windows parser

  350+ fcts in 5 DLLs, parsing in ~110 fcts in 2 DLLs, core = 47 fcts in user32.dll →

- Is "attacker memory safe"

  = no attacker-controllable buffer overflow

- How? Compositional exhaustive testing
  - "perfect" symbolic execution in SAGE (max precision, no divergences, no x86 incompleteness, no Z3 timeouts, etc.),
  - *manual* bounding of input-dependent loops (only ~10 input bytes + file size), and
  - 5 *user-guided* simple summaries

- And modulo fixing a few bugs… ☺

- 100% dynamic (=zero static analysis)

- 1st Windows image parser proved attacker memory safe

- See "Proving Memory Safety of the ANI Windows Image Parser using Compositional Exhaustive Testing", [VMCAI'2015]

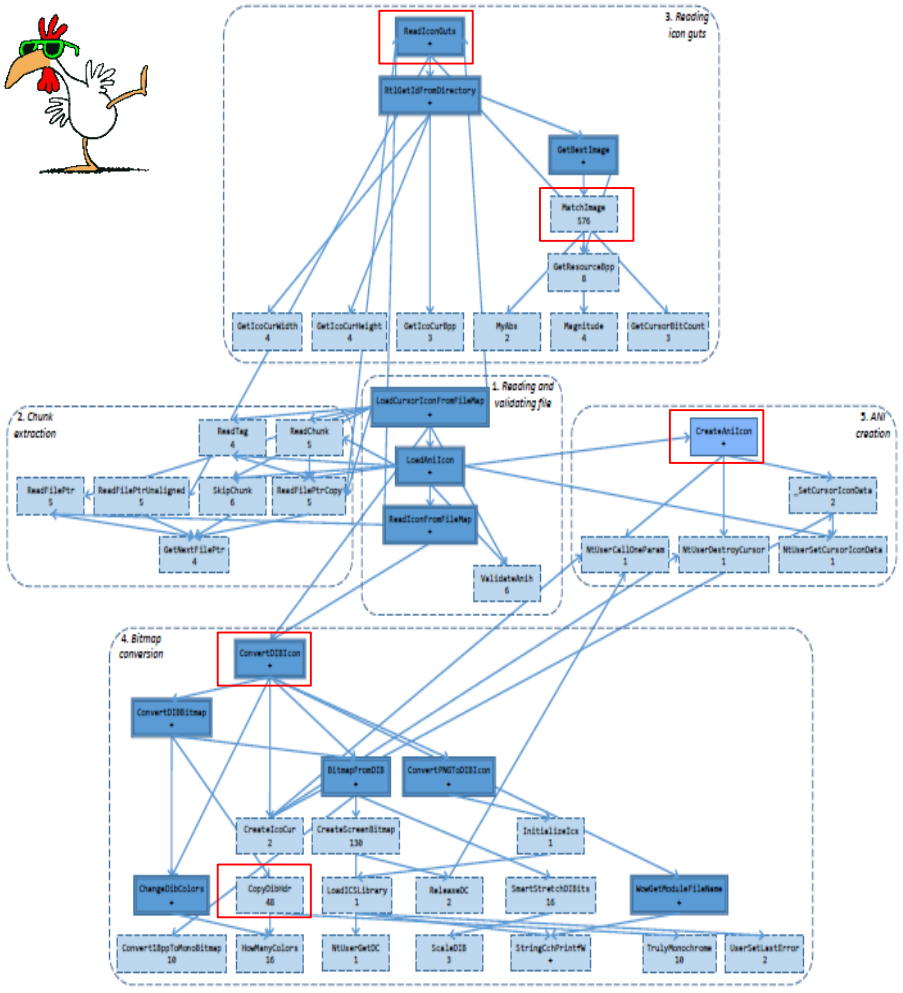


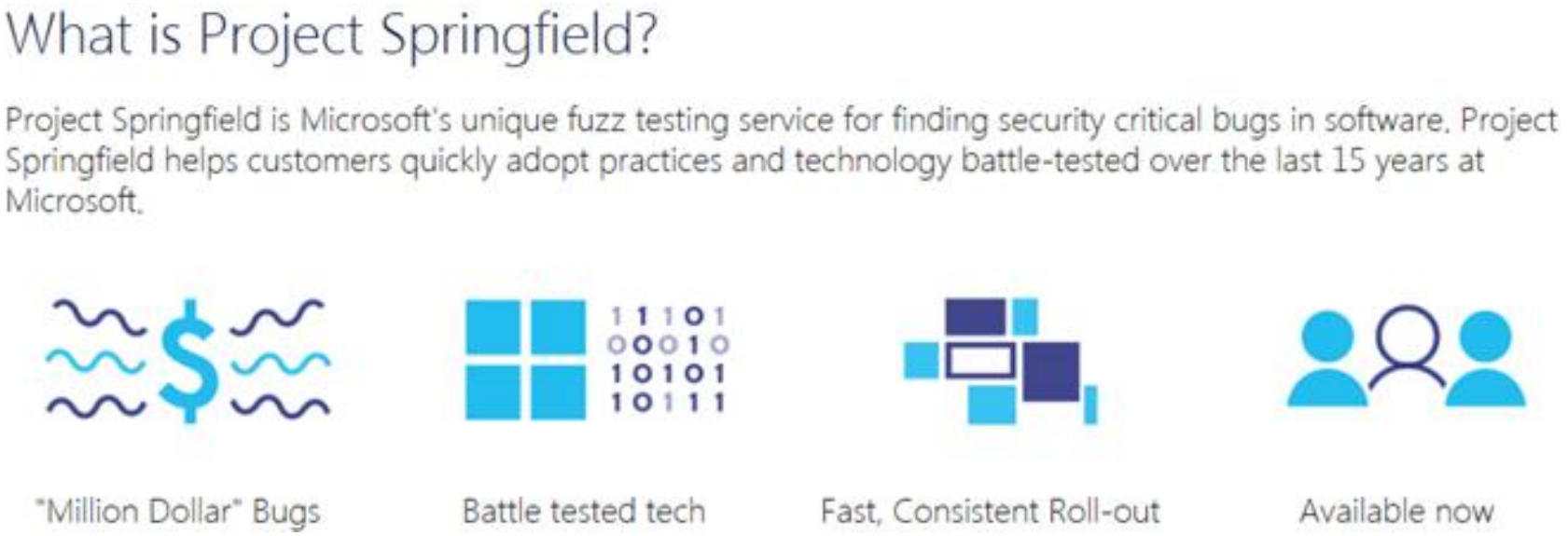
Figure 3. The callgraph of the 47 `user32.dll` functions implementing the ANI parser core. Functions are grouped based on the architectural component of Fig. 2 to which they belong. The different shades and lines of the boxes denote the verification strategy we used to prove memory safety of each function. Functions are annotated with the number of their execution paths. A + indicates that a function contains too many execution paths to be exhaustively enumerated within 12 hours without using additional techniques for controlling path explosion.

What Next?

1. Better Depth: Towards Formal Verification
   - When can we safely stop testing?
   - When we know that there are no more bugs ! = "Verification"
   - Two main problems:
     - Imperfect symbolic execution and constraint solving
     - Path explosion
   - Active area of research…
2. Better Breadth: More Applications
   - Beyond file fuzzing, what other "killer apps"?
   - Active area of research…
   - Ex: Project Springfield

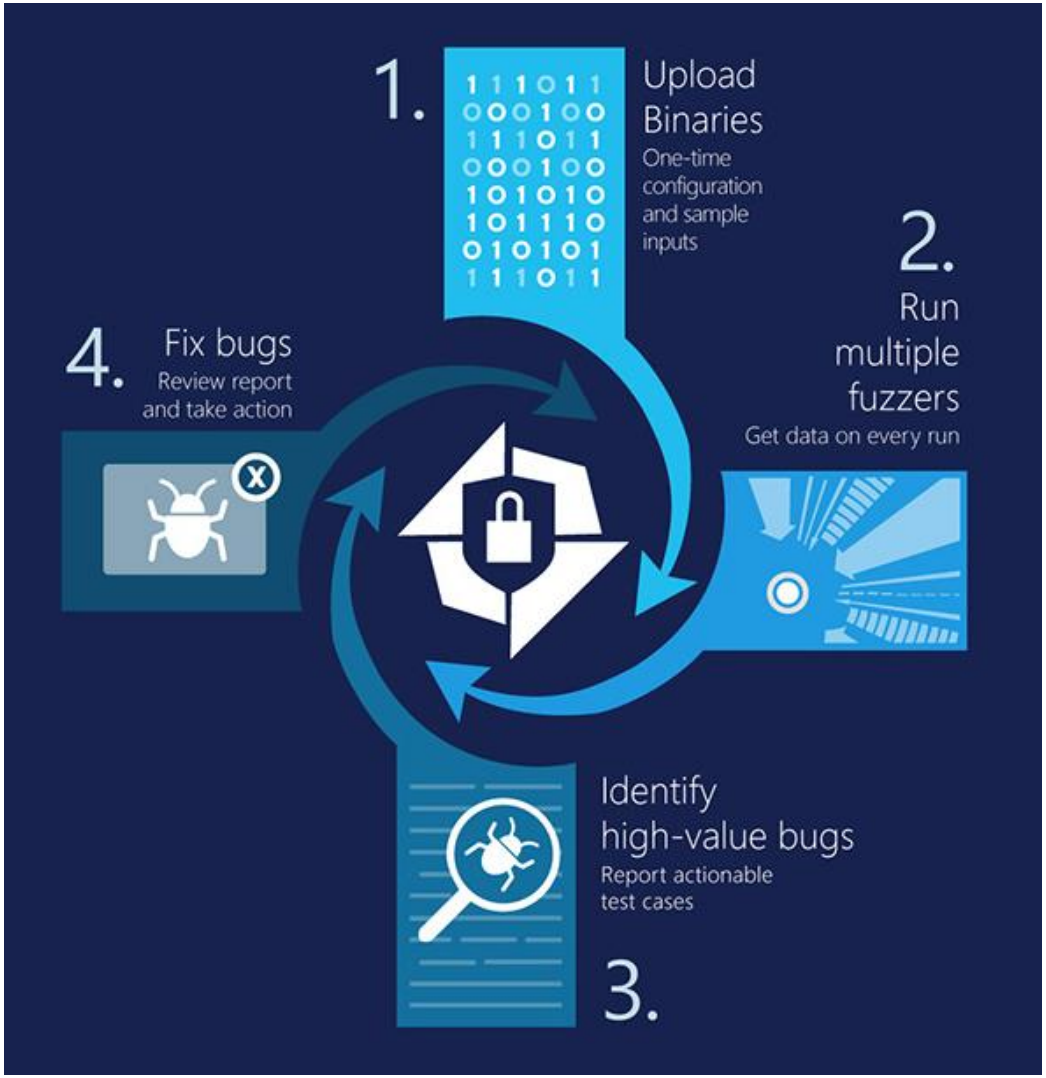
Project Springfield (2015-)

- A simple idea:
  - centralized fuzzing at Microsoft (starting ~2006)
  - Let's package centralized fuzzing as a cloud service !
  - Project Springfield = 1st commercial cloud fuzzing service



Goal = build, sell and operate a cloud fuzzing service (FaaS)

How does Springfield work?



Why Fuzzing in The Cloud

- Faster: easier adoption, time to market
- Cheaper: shared dev costs, no upfront cost, pay as you go
- Better: centralization (better quality, big data optimizations, more scenarios hence more security), elasticity (compute)

Is there a market? Yes - testing the market by shipping

- Private preview in 2015
- Public since Sep 2016
- May 2017: Project Springfield is renamed
  Microsoft Security Risk Detection (MSRD)

Outsourcing fuzzing to MSRD: higher quality at lower cost !

Example: How MSRD helped DocuSign (video link)

Conclusion

- Fuzzing @ Microsoft
- Whitebox Fuzzing with SAGE
- Microsoft Security Risk Detection
  - Fuzzing-as-a-Service, available today !
  - https://www.microsoft.com/Springfield
  - https://www.microsoft.com/en-us/security-risk-detection/

Thank You !

Questions ?