Exploiting Trade-offs* in Symbolic Execution for Identifying Security Bugs

SAS Workshop

Thanassis Avgerinos
thanassis@forallsecure.com
September 8, 2015

*trade-off [def. from Merriam-Webster]
noun
a balance achieved between two desirable but incompatible features; a compromise : a trade-off between objectivity and relevance.
The Security Battle to *Exploit* Bugs
$ iwconfig accesspoint

Exploit

$ iwconfig 01ad 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 fce8 bfff 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 0101 0101
0101 0101 3101 50c0
2f68 732f 6868 622f 6e69
e389 5350 e189 d231

# Superuser
Bug Fixed!

Good

Evil
Fact:
Ubuntu Linux has over 119,000 known bugs
1. \( \text{inp}=`\text{perl} \ -e \ '{print "A"x8000}'`\)
2. for program in /usr/bin/*; do
3.   for opt in {a..z} {A..Z}; do
4.     timeout –s 9 1s
        $program –$opt $inp
5.   done
6. done

1009 Linux programs. 13 minutes. 52 new bugs in 29 programs.
Which bugs are exploitable?
Plaid Parliament of Pwning
CMU Hacking Team
### Team rating

<table>
<thead>
<tr>
<th>Place</th>
<th>Team</th>
<th>Country</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Plaid Parliament of Pwning</td>
<td>🇺🇸</td>
<td>1381.205</td>
</tr>
<tr>
<td>2</td>
<td>More Smoked Leet Chicken</td>
<td>🇷🇺</td>
<td>900.705</td>
</tr>
<tr>
<td>3</td>
<td>Eindbazen</td>
<td>🇳🇱</td>
<td>617.515</td>
</tr>
<tr>
<td>4</td>
<td>sutegoma2</td>
<td>🇰🇷</td>
<td>529.882</td>
</tr>
<tr>
<td>5</td>
<td>disekt</td>
<td>🇺🇸</td>
<td>512.743</td>
</tr>
<tr>
<td>6</td>
<td>FluxFingers</td>
<td>🇹🇼</td>
<td>500.014</td>
</tr>
<tr>
<td>7</td>
<td>Hates Irony</td>
<td>🇺🇸</td>
<td>490.347</td>
</tr>
<tr>
<td>8</td>
<td>int3pids</td>
<td>🇪🇸</td>
<td>480.852</td>
</tr>
<tr>
<td>9</td>
<td>C.o.P</td>
<td>🇺🇸</td>
<td>472.345</td>
</tr>
<tr>
<td>10</td>
<td>European Nopsie</td>
<td>🇺🇸</td>
<td>462.515</td>
</tr>
</tbody>
</table>
DEF CON 2013  

<table>
<thead>
<tr>
<th>Team</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPP</td>
<td>15002</td>
</tr>
<tr>
<td>men in black hats</td>
<td>7924</td>
</tr>
<tr>
<td>raon_ASRT</td>
<td>7107</td>
</tr>
<tr>
<td>more smoked leet chicken</td>
<td>4160</td>
</tr>
<tr>
<td>routards</td>
<td>2503</td>
</tr>
<tr>
<td>sutegoma2</td>
<td>1540</td>
</tr>
<tr>
<td>shellphish</td>
<td>1223</td>
</tr>
<tr>
<td>Alternatives</td>
<td>1095</td>
</tr>
<tr>
<td>The European Nopsled Team 9447</td>
<td>859</td>
</tr>
<tr>
<td>blue lotus</td>
<td>12</td>
</tr>
<tr>
<td>Samurai</td>
<td>0</td>
</tr>
<tr>
<td>APT8</td>
<td>0</td>
</tr>
<tr>
<td>clgt</td>
<td>0</td>
</tr>
<tr>
<td>pwnies</td>
<td>0</td>
</tr>
<tr>
<td>pwningyeti</td>
<td>0</td>
</tr>
<tr>
<td>Robot Mafia</td>
<td>0</td>
</tr>
<tr>
<td>shell corp</td>
<td>0</td>
</tr>
<tr>
<td>[Technopandans]</td>
<td>0</td>
</tr>
<tr>
<td>WOWHacker-BIOS</td>
<td>0</td>
</tr>
</tbody>
</table>

DEF CON 2014  

Final Scores  

<table>
<thead>
<tr>
<th>Team</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plaid Parliament of Pwning</td>
<td>11263</td>
</tr>
<tr>
<td>HITCON</td>
<td>7833</td>
</tr>
<tr>
<td>Dragon Sector</td>
<td>4421</td>
</tr>
<tr>
<td>Reckless Abandon</td>
<td>4020</td>
</tr>
<tr>
<td>blue-lotus</td>
<td>3233</td>
</tr>
<tr>
<td>(Mostly) Men in Black Hats</td>
<td>2594</td>
</tr>
<tr>
<td>raon_ASRT</td>
<td>2281</td>
</tr>
<tr>
<td>StratumAuhuu</td>
<td>1529</td>
</tr>
<tr>
<td>[CBA]9447</td>
<td>1519</td>
</tr>
</tbody>
</table>

DEF CON 2015  

<table>
<thead>
<tr>
<th>Team Name</th>
<th>Final Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFKOR</td>
<td>23949</td>
</tr>
<tr>
<td>Plaid Parliament of Pwning</td>
<td>19896</td>
</tr>
<tr>
<td>Odaysober</td>
<td>17943</td>
</tr>
<tr>
<td>HITCON</td>
<td>13560</td>
</tr>
<tr>
<td>blue-lotus</td>
<td>12442</td>
</tr>
<tr>
<td>Oops</td>
<td>11306</td>
</tr>
<tr>
<td>Dragon Sector</td>
<td>11288</td>
</tr>
<tr>
<td>Samurai</td>
<td>10742</td>
</tr>
<tr>
<td>Shellphish</td>
<td>10591</td>
</tr>
<tr>
<td>LC's BC</td>
<td>9941</td>
</tr>
<tr>
<td>lSPamAndHex</td>
<td>9461</td>
</tr>
<tr>
<td>Gallopedsed</td>
<td>8608</td>
</tr>
<tr>
<td>9447</td>
<td>8410</td>
</tr>
<tr>
<td>CORNDUMP</td>
<td>7508</td>
</tr>
<tr>
<td>Bushwhackers</td>
<td>7447</td>
</tr>
</tbody>
</table>

Unlimited size  

Limited-size teams
A Manual Process
Our Vision: 
*Automatically* Check the World’s Software for *Exploitable* Bugs
Automatic Exploit Generation with Mayhem

March 7, 2012
We owned the machine in seconds
Verification, but with a twist

Program → Verification

- Correct
- Safe paths
- Incorrect
- Exploit

Correctness Property
Un-exploitability Property

33,248 programs
152 new exploitable bugs
Talk Outline

• Basics of Dynamic Symbolic Execution (DSE)
• DSE for exploit generation and 3 tradeoffs:
  1. Preconditioned symbolic execution (Pruning)
  2. Memory modeling (Reduction)
  3. Veritesting (Segmentation)
• Current & Future Work
Automatically and Effectively Finding Exploitable Bugs

Program
\[ e.g., \text{C/x86 assembly} \]

Symbolic Execution

Bugs
\[ e.g., \text{memory corruption} \]
Dynamic Symbolic Execution (DSE)

Path Predicate

\[ \Pi_0 \]

\[ s \neq 42 \]

SMT Solver

Testcase: \( s \rightarrow 0 \)

\[ \Pi_1 \]

\[ s = 42 \]

SMT Solver

Testcase: \( s \rightarrow 42 \)

\[ s = \text{input()} \]

if \( s = 42 \)

\[ \text{safe()} \]

\[ \text{bug()} \]

\[ s \text{ is anything} \]
Every conditional branch potentially doubles the number of states that should be checked.
Finding Exploitable Bugs

- Basics of Exploitation
- Identifying Control Flow Hijacks [1, 2, 3]

Security Policy: Control Flow Hijacks

Control Flow Hijack:
*EIP = Attacker Code

Effective Instruction Pointer points to next instruction to execute
**iwconfig: setuid wireless config**

```c
int get_info(int skfd, char *ifname, ...){
    ... if(iw_get_ext(skfd, ifname)...
    struct ifreq ifr;
    strcpy(ifrifr_name, ifname);
}

print_info(int skfd, char *ifname, ...){
    ... get_info(skfd, ifname, ...);
}

main(int argc, char *argv[]){
    ... print_info(skfd, argv[1], NULL, 0);
}
```

*Inputs triggering bug: strlen(argv[1]) > sizeof(ifr_name)*
1 int get_info(int skfd, char * ifname)
2   ...
3   if(iw_get_ext(skfd, ifname, SIOCGWNAME, &wrq) < 0)
4     {
5       struct ifreq ifr;
6       strcpy(ifr.ifr_name, ifname);
7     }
8 print_info(int skfd, char *ifname,...)
9   ...
10  get_info(skfd, ifname, ...);
11 }
12 main(int argc, char *argv[]){
13   ...
14  print_info(skfd, argv[1], NULL, 0)
15 }
```c
int get_info(int skfd, char *ifname) {
    ...
    if(iw_get_ext(skfd, ifname, SIOCGIWNAME, &wrq) < 0) {
        struct ifreq ifr;
        strcpy(ifr.ifr_name, ifname);
    }
    ...
}

void print_info(int skfd, char *ifname, ...) {
    ...
    get_info(skfd, ifname, ...);
}

int main(int argc, char *argv[]) {
    ...
    print_info(skfd, argv[1], NULL, 0);
}
```
```c
int get_info(int skfd, char *ifname) {
    ...
    if(iw_get_ext(skfd, ifname, SIOCGIWNAME, &wrq) < 0)
    {
        struct ifreq ifr;
        strcpy(ifr.ifr_name, ifname);
    }
    ...
    print_info(int skfd, char *ifname,...)
    ...
    get_info(skfd, ifname,...);
}

main(int argc, char *argv[]){
    ...
    print_info(skfd, argv[1], NULL, 0)
}
```
*EIP = Attacker Code

The next instruction will execute attacker code
Identifying Control Hijack Exploits

- Checking exploitability on every statement

Path predicate $\Pi$ ensures execution can reach the current state:

\[
\text{strlen(input)} > 68 \land \text{mem[EIP]} = \text{<shellcode>}
\]

Exploitability condition checks if $\ast\text{EIP} = \text{Attacker Code}$
Generating Exploits

\[
\text{strlen(input)} > 68 \\
\land \\
\text{mem[EIP]} = \text{<shellcode>}
\]

SMT Solver

Exploit

\[
\begin{array}{cccccccccccccccccccccccccccccccc}
02 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\
01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\
01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\
01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 & 01 \\
01 & 01 & 01 & 01 & 01 & 70 & f3 & ff & bf & 31 & c0 & 50 & 68 & 2f & 2f & 73 & 68 & 68 & 2f & 62 & 69 & 6e & 89 & e3 & 50 & 53 & 89 & e1 & 31 & d2 & b0 & 0b & cd & 80 & 01 & 01 & 01 & 00
\end{array}
\]
Note: Shellcode is parameterizable

\[
\text{mem[EIP]} = \langle \text{shellcode} \rangle
\]

- Allows for immediate \textit{exploit hardening} [1]
  - Return-Oriented Programming (ROP) shellcode can bypass common OS defenses:
    - Data Execution Prevention (DEP)
    - Address Space Layout Randomization (ASLR)
  - Q [1] is a system for automatic ROP shellcode generation with minimal code requirements

First Prototype [2010]

• Built on top of KLEE\(^1\)
  – Required source (C/C++ programs)

• Checked exploitability

\[ \prod \land \text{mem}[EIP] = \langle \text{shellcode} \rangle \]

• Analyzed tens of known buggy applications
  • Found one exploit – iwconfig in \(~5\) minutes

---

[1] Cadar et al., KLEE: Unassisted and automatic generation of high-coverage tests [OSDI’08]
Traditional Symbolic Execution

```c
strcpy(ifr_name, ifname);
for (i = 0 ; ifname[i] ≠ 0 ; i++)
    ifr_name[i] = ifname[i];
ifr_name[i] = 0;
```
Traditional Symbolic Execution

if (ifname[0] ≠ 0)
t
if (ifname[1] ≠ 0)
t
if (ifname[n] ≠ 0)
t
...

20 min exploration

30 min exploration

x min exploration

Exploitable
Bug found

Based on KLEE [Cadar’08]
### Trade-off #1

<table>
<thead>
<tr>
<th>DSE</th>
<th>Preconditioned DSE [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>Checks all paths</td>
</tr>
<tr>
<td>✗</td>
<td>Exploits</td>
</tr>
<tr>
<td>✓</td>
<td>Exploits</td>
</tr>
<tr>
<td>✗</td>
<td>Checks all paths</td>
</tr>
</tbody>
</table>

Pruning: only check part of the state space

Insight: *Precondition Symbolic Execution* to focus on (likely) exploitable paths

- **Bugs**
- **Control Hijack**
- **Precondition**
  
  Only check inputs above a certain length
  
  `strlen(input) > n`
AEG: Preconditioned Symbolic Execution

Precondition Check:
\[ \text{strlen(input)} > n \land \text{ifname[0]} = 0 \]

Exploitable Bug found

Unsatisfiable

Unsatisfiable

Exploitabl e

Unsatisfiabl e

Not explored. Saved 20 min

Not explored. Saved 30 min

Not explored. Saved x min

Unsatisfiable

Exploitabl e
## Generating Exploits

Length precondition + heuristic 10 exploits

<table>
<thead>
<tr>
<th>Name</th>
<th>Advisory ID</th>
<th>Time</th>
<th>Exploit Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iwconfig</td>
<td>CVE-2003-0947</td>
<td>1.5s</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>Htget</td>
<td>CVE-2004-0852</td>
<td>&lt; 1min</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>Htget</td>
<td>-</td>
<td>1.2s</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>Ncompress</td>
<td>CVE-2001-1413</td>
<td>12.3s</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>Aeon</td>
<td>CVE-2005-1019</td>
<td>3.8s</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>Tipxd</td>
<td>OSVDB-ID#12346</td>
<td>1.5s</td>
<td>Format String</td>
</tr>
<tr>
<td>Glftpd</td>
<td>OSVDB-ID#16373</td>
<td>2.3s</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>Socat</td>
<td>CVE-2004-1484</td>
<td>3.2s</td>
<td>Format String</td>
</tr>
<tr>
<td>Expect</td>
<td>OSVDB-ID#60979</td>
<td>&lt; 4min</td>
<td>Buffer Overflow</td>
</tr>
<tr>
<td>Expect</td>
<td>-</td>
<td>19.7s</td>
<td>Buffer Overflow</td>
</tr>
</tbody>
</table>
Finding Exploitable Bugs

Symbolic Execution

Control Hijack

Source:
```c
#include <stdio.h>

int main()
{
    int s = input();
    if (s == 42)
    {
        bug();
    }
    else
    {
        safe();
    }
    return 0;
}
```
Second Prototype: Mayhem [2011]

• Binary-only symbolic executor

• Checks exploitability

• No source code abstractions
  – Types, buffers, datastructures
  – Indirect jumps, partial control flow graph

\[ \Pi \land \text{mem[EIP]} = \langle \text{shellcode} \rangle \]
One Challenge: Symbolic Indices

\[ x := \text{get\_input}(); \]
\[ \ldots \]
\[ y := \text{mem}[x]; \]
\[ \text{assert} \ (y == 42); \]

\( x \) can be anything

Which memory cell contains 42?

\( 2^{32} \) cells to check
Symbolic Indices: Overwritten Pointers

```c
... assert(*ptr==42); return;
ptr = 0x11223344

mem[0x11223344] = 42
mem[input]...
... user input
buf
ret_addr
arg
ptr
...`

Symbolic Indices: Translation Tables

c = get_char();
...
c = tolower(c);

tolower(char c){
    return c >= -128 && c < 256 ? tbl[c] : c;
}

Other causes
• Parsing: sscanf, vfprintf, etc.
• Character test: isspace, isalpha, etc.
• Conversion: toupper, tolower, mbtowc, etc.
• ...

Address is symbolic
Method 1: Concretization

\[ \Pi \land \text{mem}[x] = 42 \]

\[ \Pi \land x = 17 \land \text{mem}[17] = 42 \]

✓ Solvable

✗ Exploits

Misses over 40% of exploits

1 cell to check

Memory

\[ 0 \quad 30 \quad 2^{32} - 1 \]
Method 2: Fully Symbolic

\( \Pi \land \text{mem}[x] = 42 \)

\( \Pi \land \text{mem}[x] = 42 \land \text{mem}[0] = v_0 \land \ldots \land \text{mem}[2^{32} - 1] = v_{2^{32} - 1} \)

✗ Solvable
✓ Exploits
## Trade-off #2

<table>
<thead>
<tr>
<th>Concretization</th>
<th>Partial Memory Modeling [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Solvable</td>
<td>✓ Solvable</td>
</tr>
<tr>
<td>✗ Exploits</td>
<td>✓ Exploits</td>
</tr>
</tbody>
</table>

### Fully symbolic

<table>
<thead>
<tr>
<th>Solvable</th>
<th>Exploits</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Our Observation

Path predicate ($\Pi$) constrains *range* of symbolic memory accesses

$\Pi \Rightarrow 42 < x < 50$

Use symbolic execution state to:

**Step 1:** Bound memory addresses referenced

**Step 2:** Make search tree for memory address values
Step 1 — Find Bounds

\[ \text{mem}[x \& 0xff] \]

Lowerbound = 0, Upperbound = 0xff

1. Value Set Analysis\(^1\) provides initial bounds
   - Over-approximation
2. Query solver to refine bounds

[1] Balakrishnan et al., Analyzing memory accesses in x86 executables, ICCC 2004
Step 2 — Index Search Tree Construction

\[ y = \text{mem}[x] \text{ite}(x < 3, \text{left, right}) \]

\[ \text{ite}(x < 2, \text{left, right}) \]

\[ \begin{align*}
\text{if } x = 1 & \text{ then } y = 10 \\
\text{if } x = 2 & \text{ then } y = 12 \\
\text{if } x = 3 & \text{ then } y = 22 \\
\text{if } x = 4 & \text{ then } y = 20
\end{align*} \]
Index Search Tree Optimization
(reads):

*Piecewise Linear Reduction*

\[ y = 2x + 10 \]
\[ y = -2x + 28 \]
Index Search Tree Optimization
(reads):

Piecewise Linear Reduction

ite(n < 91, ite(n < 64, n, n+32), n)

40% more exploits with partial memory modeling
soritong
muse
gsplayer
galan
dizzy
destiny
coolplayer
xtokkaetama
xgalaga	
tipxd
squirrel mail
socat
sharutils
rsync
psUtils
orzHttpd
nCompress
mbse-bbs
iwconfig
htpasswd
htget
gnugol
glftpdp
ghostscript
freeradius
atphphttpd
aspell
a2ps
State Explosion: An Example in C

1. int counter = 0;
2. for ( i = 0 ; i < 100 ; i ++ ) {
3.   if (input[i] == 0x42) // ‘B’
4.     counter ++;
5. }
6. if (counter == 75) bug ();
7. ...

• 100 consecutive branches
• $2^{100}$ feasible paths
State Explosion: An Example in C

1. int counter = 0;
2. for ( i = 0 ; i < 100 ; i ++ ) {
3.   if (input[i] == 0x42) // ‘B’
4.     counter ++;
}

Can we check all states in a reasonable amount of time?

• Time to check $2^{100}$ states:

DSE executing @ 1 state/ns: $\sim 10^{14}$ years

Age of Universe $< 10^{12}$ years
Yes, but *not* if we check *one* state at a time
Static Symbolic Execution (SSE)$^1$

- **SSE input:**
  - Starting condition for the execution
  - An *acyclic* control flow graph (CFG)

- **SSE output:**
  - One formula per CFG node
    - encompasses *all paths* reaching the node

[1] Variants by Koelbl et al. [IJPP’05], Xie et al. [POPL’05], Babic et al. [ICSE’08]
Static Symbolic Execution (SSE)
Static Symbolic Execution (SSE)

Obtaining Node Coverage

Formulas not exponential

SMT Solver
Static Symbolic Execution (SSE)

• SSE input:
  - Starting condition for the execution
  - An *acyclic* control flow graph (CFG)

• SSE output:
  - One formula per CFG node
    • encompasses *all paths* reaching the node

What about features that cannot be recovered statically?

What about programs with loops?

Are formulas too difficult to solve?
How expensive is formula solving?

• Solve time in DSE (25 million queries)
  – 99.9% solved in less than 1sec
  – 95% solved in less than 100ms
  
  – Mean solve time: 3.67ms
  – Variance: 0.34ms

  – SAGE\(^1\) reports similar results (99% require less than 1sec)

Quick Recap

DSE for Testing

✓ Dynamic execution
✓ Loops unrolled as the code executes
✓ Formula solving time acceptable
✗ Path explosion

SSE for Verification

✗ Missing dynamic features
✗ # Unrolls per loop unknown
✗ Formula solving worse than DSE
✓ No path explosion
Trade-off #3

DSE for Testing

✗ State explosion
✓ Formulas
✓ Dynamic features

SSE for Verification

✓ State explosion
✗ Formulas
✗ Dynamic Features

Veritesting [1]

✗ State explosion
– Formulas
✓ Dynamic features
✓ Bugs & testing

Segment the state space and check sets of states simultaneously


* ACM Distinguished Paper Award (to appear in CACM 2015 Research
**Core Idea:** Alternate DSE + SSE

- **Use DSE to:**
  - Dynamically unroll loops
  - Have access to dynamic features

- **Use SSE to:**
  - Analyze multiple paths simultaneously
DSE vs Veritesting

Step 1: Recover CFG

Step 2: Switch to SSE

Step 3: Switch to DSE

Fork

Merge

Π

Π_

Π₀

Π₁

Π'
DSE vs Veritesting
Experiments on 1,023 Programs

- Source: Debian Squeeze (default install)
- All /bin, /usr/bin, /sbin ELF 32-bit binaries

- Time: 30 minutes each (DSE vs Veritesting)
- Measured:
  1. # of Bugs
  2. Node code coverage (reported by gcov)
  3. Test cases
Code Coverage with Time

- +5% Code Coverage

Test Cases with Time

- +114% More Bugs
- Requires 60% Less Test Cases
Veritesting Profiles: a Trade-off

Programs

Coverage Difference

Veritesting better

DSE better

SMT

73%

67%

DSE

Veritesting

26
43
79

#Bugs

SMT

30%

44%
Statistics* from 7.7 Years CPU-time

- 37,391 programs / 16 billion SMT resolved

207 million test cases
2,606,506 crashes
13,875 unique (stack hash) bugs
152 control hijacks

[*] Statistics and data available at: http://forallsecure.com/debian
Reporting 1.2K Crashes

MergePoint bump
"Thanks for your extensive feedback, it's a pleasure to work with such detailed material (and easy to pin the bug, BTW).

"I have a lot of respect for the Mayhem tool now as a way to find corner cases in simple C parsers. I'm sure the team at CMU's project will find some very real bugs in Debian.

"I am sorry, but it is not a bug if jocamlrun segfaults when you feed it garbage!

"No you *did* not! You might have found a bug in libc but it is not a bug in tart."
Bugs are getting fixed (slowly)

- Outstanding bugs -- Normal bugs; Patch Available (1 bug)
- Outstanding bugs -- Normal bugs; Confirmed (1 bug)
- Outstanding bugs -- Normal bugs; Unclassified (796 bugs)
- Outstanding bugs -- Normal bugs; Will Not Fix (1 bug)
- Outstanding bugs -- Minor bugs; Confirmed (1 bug)
- Outstanding bugs -- Minor bugs; Unclassified (7 bugs)
- Forwarded bugs -- Normal bugs (29 bugs)
- Pending Upload bugs -- Normal bugs (1 bug)
- Resolved bugs -- Normal bugs (28 bugs)

~300 bugs already fixed!
Acknowledgments

David Brumley  
Sang Kil Cha  
Alexandre Rebert Edward J. Schwartz

Maverick Woo

- Brent Lim Tze Hao
- JongHyup Lee
- Ivan Jager

- Jonathan Foote
- David Warren
- Gustavo Grieco
Conclusion

• Automatically finding and demonstrating exploitable bugs is possible

• Exploiting tradeoffs such as state pruning, reduction, and segmentation can improve DSE as a testing/bug-finding tool

The future of binary program analysis should be exciting
Thank You!

Questions?