Systems vs. Programs

... and The Battle Against Bugs

George Candea

School of Computer & Communication Sciences

ECOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Outline

• Systems Research
• The S2E Platform
• Three Use Cases
What Is A Systems Researcher?

- *Alto personal computer* (1972)
- *Ethernet* (1973)
- *Mesa programming language* (1975)
- *Bravo text editor* (1973)
- *Fast RPC* (1987)
- *Autonet* (1987)
Systems Challenges

• #1 Challenge: Complexity
  • e.g., preconditions & postconditions

• #1 Problem: poorly understood connections
  • unexplained interference -> loss of predictability
  • need predictability in order to achieve scalable performance, reliable operation, etc.
Some Systems Solutions

• Caching
  • *improve performance by adding another layer*

• Transactions
  • *building blocks for robust systems*

• Virtual Machines
  • *isolation taken to the max (processes, addr spaces, ...)*
Systems Approach

• Get idea
• Build prototype
• Measure & observe
• Adjust prototype ... repeat previous step

• Principles of system construction
  • modularity, hierarchy, layering, abstraction, end to end
The Battle Against Bugs

• Bugs = huge source of unpredictability
  • ... your favorite scary bug story here ...
  • small discontinuities have significant effects

• We have no choice but resort to faith
RTL8029.SYS
The Rigorous Approach

- Specify, model, and verify
  - what you run is not what you verify

- SLAM @ Microsoft
  - models are hard to get right and to maintain

Bug-free programs ⇔ Bug-free systems
A Systems Approach

• If cannot take rigorous constructive approach
  • at least understand in-depth the artifacts we build

• Build tools to understand systems
  • performance profilers, debuggers, tracers, ...

• How to build sophisticated analyzers cheaply?
  • that work for real systems
Outline

• Systems Research

• The S2E Platform
  • Background
  • S2E: The Theory
  • S2E: The System

• Three Use Cases
$S^2E = \text{platform for building analysis tools that are multi-path and in-vivo}$
Example of Analysis: Testing

```c
int main(argc, argv)
{
    if (argc == 2)
        printf("%c", *argv[2]);

    printf("OK");
}
```

$ ./prog
OK

$ ./prog ABC
Segmentation fault

$ valgrind ./prog ABC
Invalid read of size 1
main (prog.c:4)

Analysis = check properties of execution paths
int main(argc, argv)
{
    if (argc == 2)
        printf("%c", *argv[2]);
    printf("OK");
}

Simultaneously analyze multiple paths

Multi-Path Analysis
In-Vivo Analysis

In Vitro

In Vivo

Analyze entire software stack
Single-Path In-Vitro

Multi-Path In-Vivo

User-mode program

OS Kernel
$S^2E$ platform = path exploration + path analysis
Challenge #1: Path Explosion

\[ \text{# of paths} \approx 2^{\text{system size}} \]

• Cannot analyze all paths \( \Rightarrow \) select only some
  • which paths you choose can make a big difference
Challenge #2: No Source Code

• A lot of interesting software is closed-source

• Systems are built from many pieces
  • even if you had source code, it would be a pain

• Solution: analyze machine code
  • the problem becomes even harder
  • more stuff to analyze, loss of type information, etc.
Outline

• Systems Research
• The S2E Platform
  • Background
  • S2E: The Theory
  • S2E: The System
• Three Use Cases
System = Unit + Environment

```c
int main(argc, argv) {
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
}
```
Thorough Automated Testing

```c
int main(argc, argv) {
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```
Thorough Automated Testing

```c
int main(argc, argv) {
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```

System Input

Environment
Thorough Automated Testing

```c
int main(argc, argv) {
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```

System Input

Unit

Environment
Thorough Automated Testing

int main(argc, argv) {
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
Illusion of Full-System Analysis

```c
int main(argc, argv)
{
    if (argc == 0) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
}
```
Relaxed Execution Consistency

- Allow developer to tell us which paths matter
  - *scale-up by ignoring further paths we don’t care about*
  - *appropriate relaxation is analysis-specific*
  - *the illusion is automatic, relaxation is human controlled*

- Execution Consistency Model = set of paths
  - *abstract grammar/specification of which paths matter*
  - *but defined through code, not logic*
SC-UE (SC Unit-level Execution)

Much fewer paths False negatives?

```c
int main(argc, argv) {
    if (argc == 1) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
```

*Environment*

*Unit*
Relaxed Consistency (RC)

No false negatives

int main(argc, argv) {
    if (argc == 1) {
        ...
    }
    p = malloc(...);
    if (p == NULL) {
        ...
    }
    ...
}
Relaxed Consistency (RC)

No false negatives
False positives ?

Unit Input

```c
int main(argc, argv) {
  if (argc == 1) {
    ...
  }
  p = malloc(...);
  if (p == NULL) {
    ...
  }
  ...
}
```

Multi-valued return

\[ p' \in \{ p, \text{NULL} \} \]
Local Consistency (LC)

```c
int main(argc, argv) {
    if (argc == 1) {
        ...
    }
    p = malloc(...);
    ...
}
```

Interface annotation

`malloc() → {p, NULL}`
globally feasible

locally feasible

statically feasible
relax constraints at unit/environment boundary consistently with environment interface specification

arbitrarily relax constraints at unit/environment boundary

arbitrarily relax constraints anywhere

more inputs based on knowledge of constraints from within the unit

more inputs based on knowledge of constraints from environment

SC-CE
Strictly consistent concrete execution

SC-UE
Strictly consistent unit-level execution

SC-SE
Strictly consistent system-level execution

Zero False Positives

RC-CC
CFG consistency

RC-OC
Overapproximate consistency

LC
Local consistency

Systems vs. Programs and the Battle Against Bugs
relax constraints at unit/environment boundary consistently with environment interface specification
arbitrarily relax constraints at unit/environment boundary
arbitrarily relax constraints anywhere
more inputs based on knowledge of constraints from within the unit
more inputs based on knowledge of constraints from environment

**SC-CE**
Strictly consistent concrete execution

**SC-UE**
Strictly consistent unit-level execution

**SC-SE**
Strictly consistent system-level execution

**RC-CC**
CFG consistency

**RC-OC**
Overapproximate consistency

**LC**
Local consistency

**False Positives**

**Zero FPs**
Mix & Match

- ECM = specification of paths to be explored
  - *S2E underneath the covers explores the requested paths*
- Can make principled trade-offs
  - *FPs vs. FNs vs. exploration+analysis performance*
- Minimize the number of explored paths
  - *all the paths in the ECM set, but none extra*

Can imperatively implement in S2E any ECM
Outline

• Systems Research

• The S2E Platform
  • Background
  • S2E: The Theory
  • S2E: The System

• Three Use Cases
Symbolic Execution

```c
int func(int x, int y)
{
    if (x < 0) {
        ...
    }
    if (y > 0) {
        ...
    }
}
```
Selective Symbolic Execution

\[
\text{int main(\text{argc, argv})} \{
\text{if (argc == 0) \{ \quad \alpha \quad \beta \}} \}
\text{\quad \ldots}
\text{\quad 128}
\text{\quad p = malloc(\alpha);}
\text{\quad if (p == NULL) \{ \quad \ldots \quad \}}
\text{\quad \ldots}
\text{\quad \ldots}
\]
Idea #1: Virtualization

• Enable in-vivo analysis
  • run entire system in the tool, not just the program
• Enable true-behavior analysis
  • system is oblivious $\rightarrow$ behaves “naturally”
• Analyze what actually runs
  • i.e., x86 machine code
Idea #2: Dynamic Translation

• Create symbolic & concrete domains
  • *rely on DBT to alternate symbolic vs. concrete execution*
  • *do this entirely at runtime*

• Create hybrid system-state
  • *virtualized hardware shares state between domains*
  • *maintain hybrid symbolic/concrete representation*
S2E System Architecture

- Runs unmodified x86 binaries (incl. proprietary/obfuscated/encrypted binaries)
- Customized virtual machine
- Selection done at runtime
- Most code runs “natively”
- Shared concrete/symbolic state representation
Lazy + Selective Conversion

buf
Application

Libraries

Kernel

Block device driver
if (buf[0] == 3) ...

buf (buf[0] == 3)

Avoids unnecessary concretization
S2E User’s View

Selection interface

User-defined selectors

S²E stock selectors

Applications
- Virtual CPU
- VM Phys memory
- Virtual devices

Libraries

Operating system kernel
- Dynamic binary translation
- Symbolic execution

Drivers

User-defined analyzers

S²E stock analyzers

Analysis interface

Real CPU

Real phys memory

Real devices

QEMU

LLVM

KLEE
Outline

• Systems Research
• The S2E Platform
• Three Use Cases
  • Testing proprietary software
  • Reverse engineering
  • Total performance profiling
A problem has been detected and Windows has been shut down to prevent damage to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBE7617,0x00000000)

*** SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c
<table>
<thead>
<tr>
<th>Tested Driver</th>
<th>Bug Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL8029</td>
<td>Resource leak</td>
</tr>
<tr>
<td>RTL8029</td>
<td>Memory corruption</td>
</tr>
<tr>
<td>RTL8029</td>
<td>Race condition</td>
</tr>
<tr>
<td>RTL8029</td>
<td>Segmentation fault</td>
</tr>
<tr>
<td>RTL8029</td>
<td>Segmentation fault</td>
</tr>
<tr>
<td>AMD PCNet</td>
<td>Resource leak</td>
</tr>
<tr>
<td>AMD PCNet</td>
<td>Resource leak</td>
</tr>
<tr>
<td>Ensoniq AudioPCI</td>
<td>Segmentation fault</td>
</tr>
<tr>
<td>Ensoniq AudioPCI</td>
<td>Segmentation fault</td>
</tr>
<tr>
<td>Ensoniq AudioPCI</td>
<td>Race condition</td>
</tr>
<tr>
<td>Ensoniq AudioPCI</td>
<td>Race condition</td>
</tr>
<tr>
<td>Intel Pro/1000</td>
<td>Memory leak</td>
</tr>
<tr>
<td>Intel Pro/100 (DDK)</td>
<td>Kernel crash</td>
</tr>
<tr>
<td>Intel 82801AA AC97</td>
<td>Race condition</td>
</tr>
</tbody>
</table>
Analysis Time < 20 Minutes

![Graph showing basic block coverage over running time with analysis time < 20 minutes. The graph compares different devices: RTL8029, Intel Pro/100, and Intel 82801AA AC97.]
Symbolic Hardware

- Symbolic HW inputs, symbolic interrupts, etc.
- Test without having the real hardware
- Test for bad hardware behaviors
Outline

• Systems Research
• The S2E Platform
• Three Use Cases
  • Testing proprietary software
  • Reverse engineering
  • Total performance profiling
**RevNIC+ Reverse Engineering**

- **applications**
- **libraries**
- **operating system kernel**
- **drivers**

<table>
<thead>
<tr>
<th>virtual CPU</th>
<th>VM phys memory</th>
<th>virtual devices</th>
</tr>
</thead>
</table>

- **user-defined analyzers**

- **RevNIC Code Synthesizer**

- **Synthetic Driver**

- **Driver/Hardware interaction traces**

---

George Candea

*Systems vs. Programs and the Battle Against Bugs*
Automated Porting

Windows

Windows
Linux
KitOS
μC/OS II

x86 PC
VMware QEMU
FPGA4U
PROFS: Total Performance Profiling

- Apache URL parser
- Microsoft IIS SSL module
- Lua language interpreter
- Various utilities

1,645 instructions

129,086 instructions
Multi-Path In-Vivo Profiling

User-mode program

OS Kernel

Valgrind
Oprofile
PROFS
Productivity Results

PROF $s$
- 20 person-hours
- 767 lines of code

RevNIC$^+$
- 40 person-hours*
- 580 lines of code

DDT$^+$
- 38 person-hours
- 720 lines of code

S2E Platform
- > 100,000 lines of code
- 47,000 lines of new code

* 47,000 lines of new code
Summary

• Approach
  
  • *powerful analysis* -> *deeper understanding* -> *more rigorous composition* -> *formal verification*?

• Borrow ideas to solve systems problems
  
  • *systems research is interdisciplinary by definition*

http://s2e.epfl.ch

Ready-to-play VM, demos, tutorials, source code, documentation
Outline

• Systems Research

• The S2E Platform

• Three Use Cases
  • Testing proprietary software
  • Reverse engineering
  • Total performance profiling
Tools & Systems

Automated Testing
- DDT: Test proprietary code
- LFI: Test recovery code
- ConfErr: Human error testing

Automated Debugging
- ESD: Execution synthesis
- Portend: Date race classifier

Automated Correcting
- Dimmunix: Deadlock immunity
- RevNIC: Reverse engineering

Scalable Certification
- iPrae: End-user verifiability
- iQA: Rating & certification

Core Techs

Cloud9 - Cluster-based parallel symbolic execution

S²E - Selective symbolic execution of full software stacks