Symbolic execution for BIOS security

Mark Tuttle, Lee Rosenbaum,

Oleksandr Bazhaniuk, John Loucaides, Vincent Zimmer

Intel Corporation

August 10, 2015

Overview

Message:

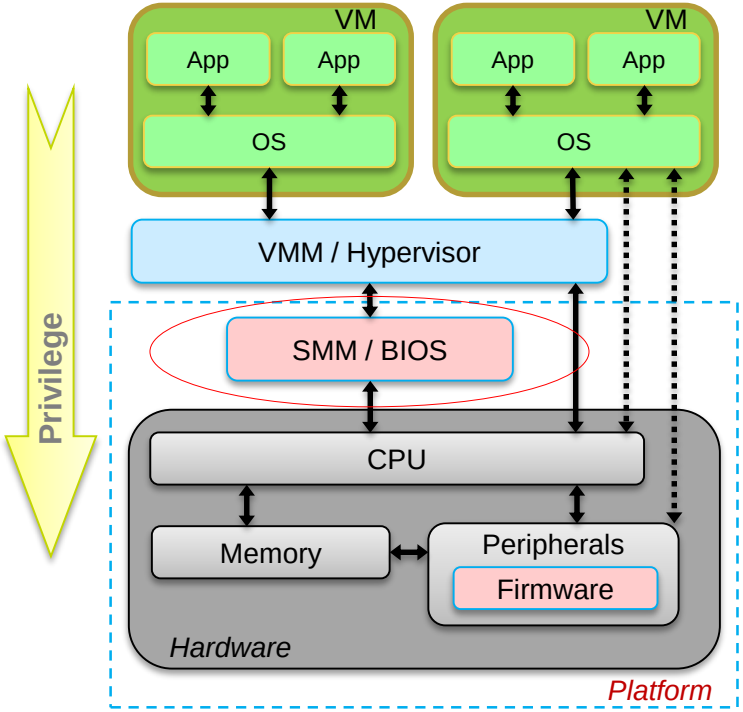
- Symbolic execution is now a believable path to BIOS validation

Outline:

- BIOS/UEFI Background
- The problem: BIOS security
- The approach: Symbolic execution
- Status, risks, and mitigations

BIOS/UEFI Overview

Where is BIOS/UEFI?



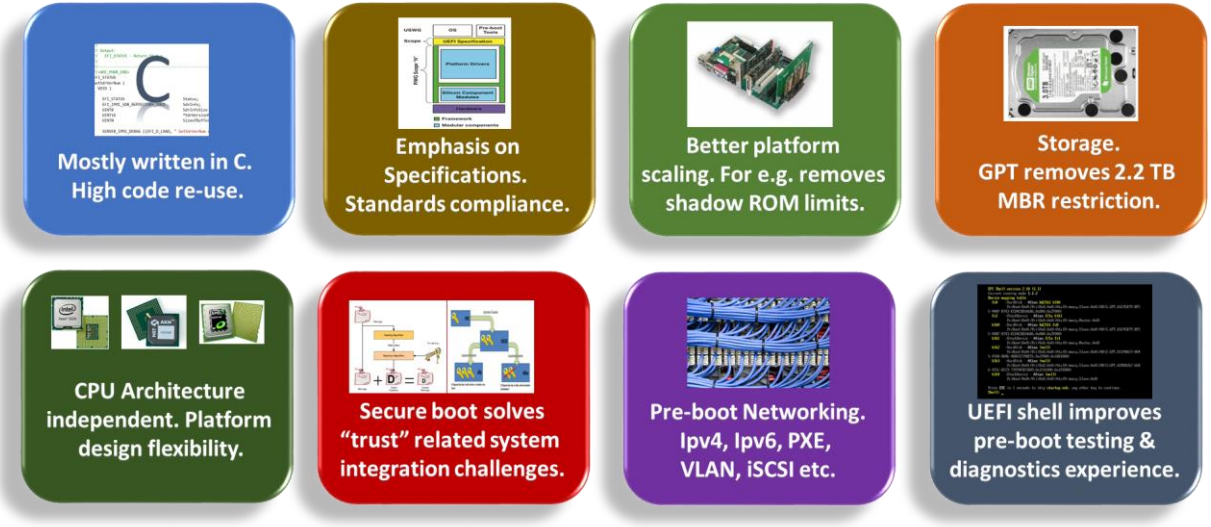
Acronyms

UEFI – Unified Extensible Firmware Interface
UEFI Forum @ www.uefi.org

SMM – System Management Mode

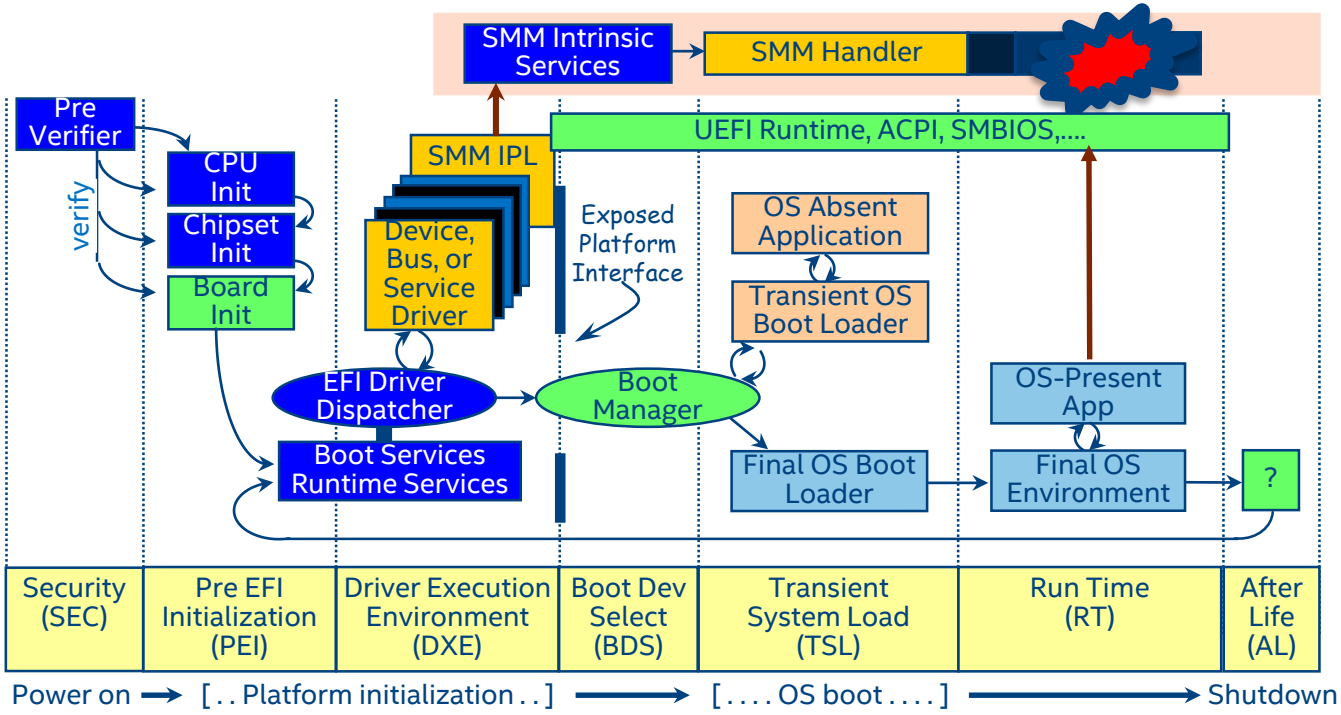
VM – Virtual Machine
VMM – Virtual Machine Monitor

What's in UEFI?



Specification pages:	UEFI 2,000+, PI 2,000+, Also: ACPI, TCG, PECOFF, USB, …
Tianocore.org:	UDK2014: 2 million+ open source LOC
Typical platform:	200,000 open source LOC, 100,000 closed source LOC or binary modules

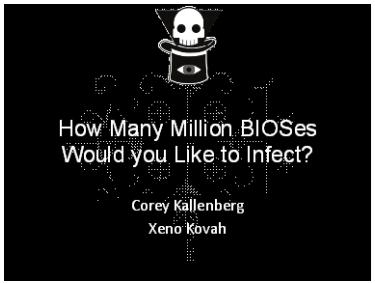
UEFI Boot Timeline

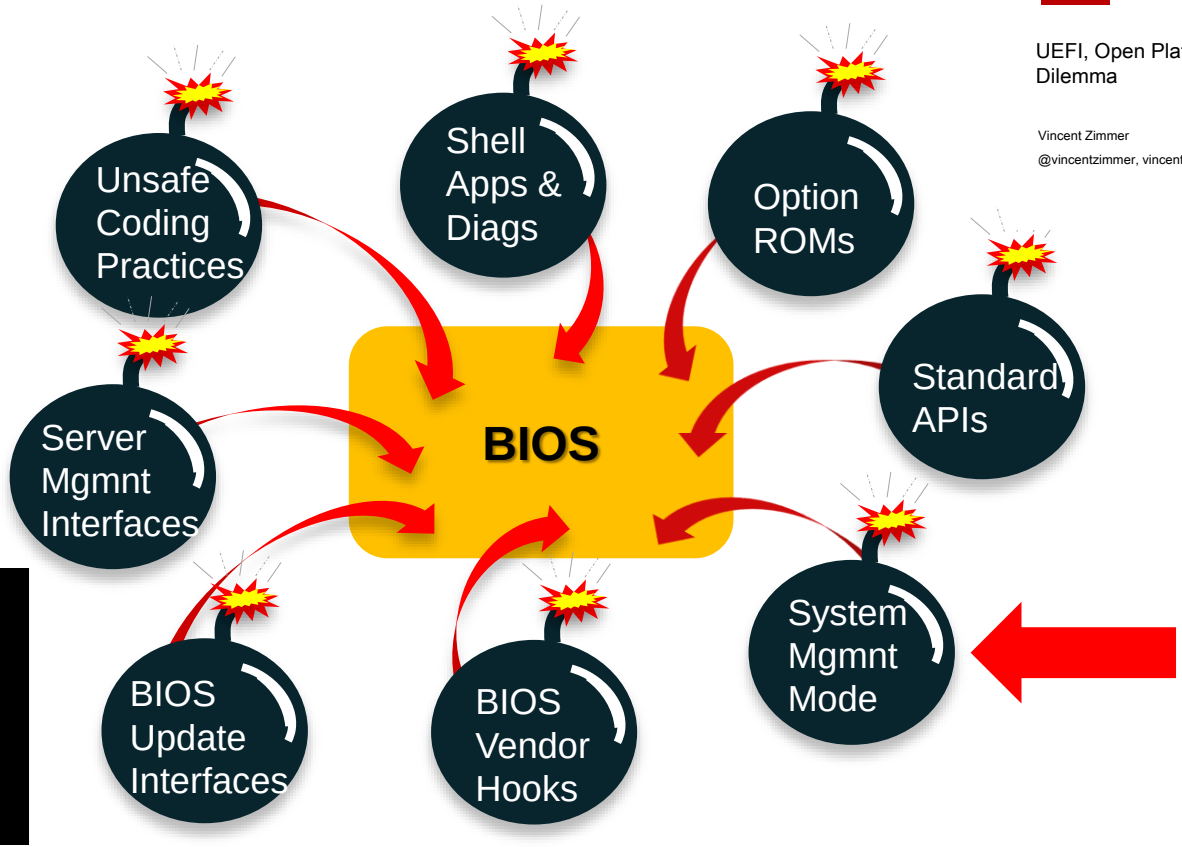


What Could Possibly Go Wrong???

BIOS Attack Surfaces

Attacking Hypervisors Using Firmware and Hardware
Bulygin, Matrosov, Gorobets, & Bazhaniuk





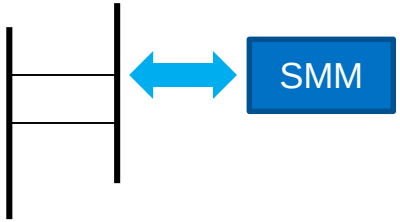
CanSecWest 2015 Vancouver, Canada

UEFI, Open Platforms, and the Defender's Dilemma

Vincent Zimmer
@vincentzimmer, vincent.zimmer @intel.com | @gmail.com

The problem: SMM security

System Management Mode



SMM is valuable because it:

- Is invisible to Operating System, Anti Virus, Virtual Machines …
- Can see all memory and access all host accessible resources
- Is used to protect flash – *which contains UEFI code and variables*

Threats

- Elevation
  - View secrets or own the system by subverting RAM

Mitigations include code reviews to:

- Validate “external” / “untrusted” input
- Remove calls from inside SMM to outside SMM

SMM security with Symbolic Execution

Goal

- Eliminate vulnerabilities during development,
- So they can't be exploited

Approach: Search for vulnerabilities with S^2E

- Integer overflow, division by zero
- Pointers invalid or out of range, buffer overflow
- Insecure memory references

Current target: SMM interrupt handlers + call outs

- **Searching for SMI memory references outside of SMRAM**

The approach: symbolic execution

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

OSDI 2008

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat the coverage of the developers' own hand-written test suites. When we did the same for 75 equivalent tools in the BUSYBOX embedded system suite, results were even better, including 100% coverage on 31 of them.

bolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

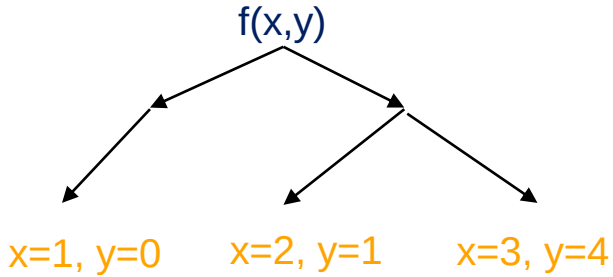
Results are promising. However, while researchers have shown such tools can sometimes get good coverage and find bugs on a small number of programs, it has been an open question whether the approach has any hope of consistently achieving high coverage on real ap-

KLEE

Cristian Cadar, Imperial College

KLEE

Symbolic execution for code coverage and bug hunting



- Coverage: minimal test cases inducing maximal code coverage
- Bugs: test cases inducing common program vulnerabilities

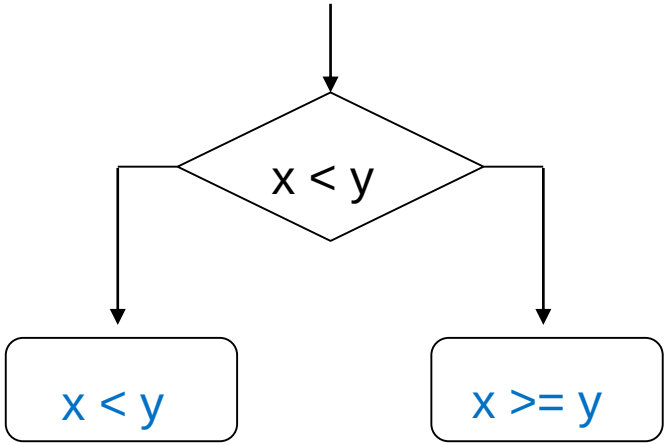
Symbolic execution

Program

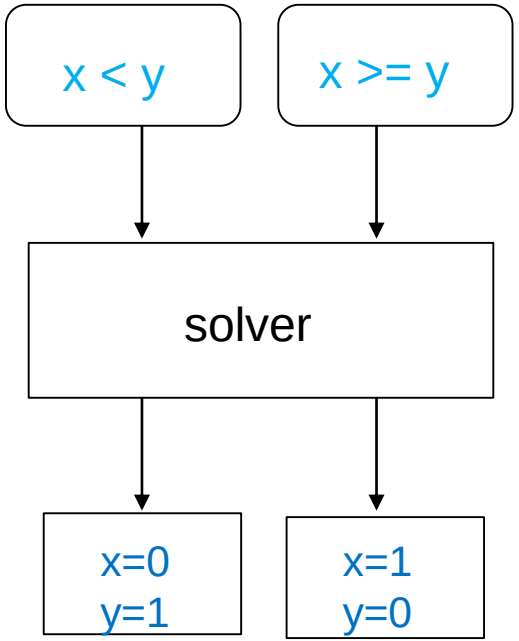
```
if (x < y)
  print(“small”)
else
  print(“large”)
```

Harness

```
make_symbolic(x);
make_symbolic(y);
```

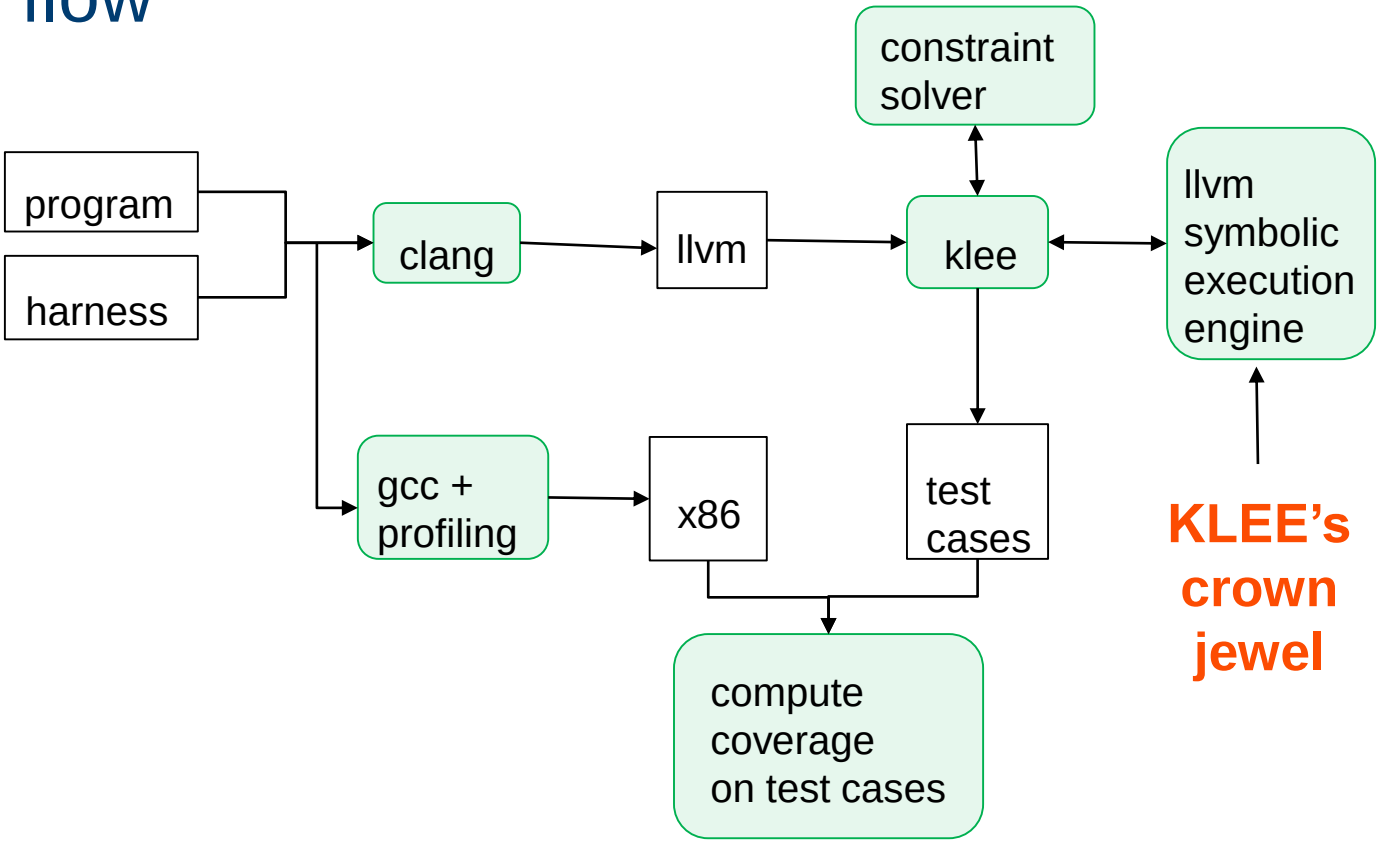


constraints



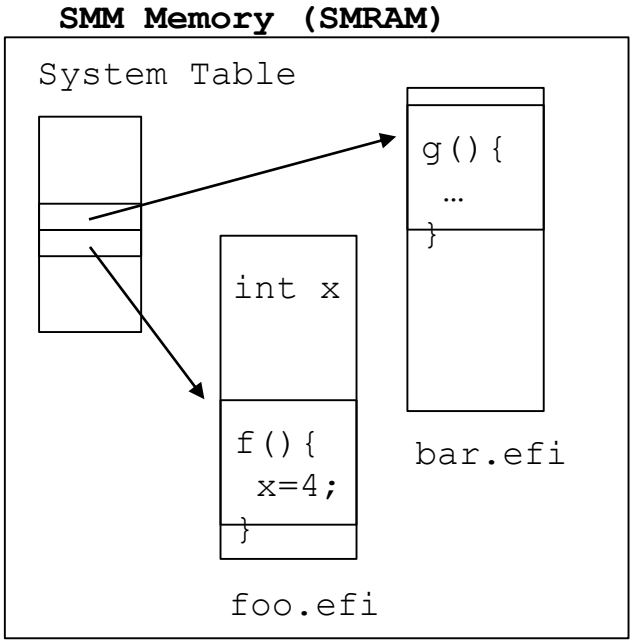
test cases

KLEE flow


constraint solver
program
harness
clang
llvm
klee
llvm symbolic execution engine
gcc + profiling
x86
test cases
KLEE's crown jewel
compute coverage on test cases

But the test harness is a problem

Has to model the environment of the software under test



- SMRAM is the model
  - The code is there
  - The data and data layout there
- S^2E lets us use SMRAM
  - Boot to SMRAM and dump it
  - Load it into S^2E
  - Jump to an entry point
  - And execute symbolically

S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems

Vitaly Chipounov, Volodymyr Kuznetsov, George Candea

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{vitaly.chipounov,vova.kuznetsov,george.candea}@epfl.ch

ASPLOS 2011

Abstract

This paper presents S^{2}E, a platform for analyzing the properties and behavior of software systems. We demonstrate S^{2}E's use in developing practical tools for comprehensive performance profiling, reverse engineering of proprietary software, and bug finding for both kernel-mode and user-mode binaries. Building these tools on top of S^{2}E took less than 770 LOC and 40 person-hours each.

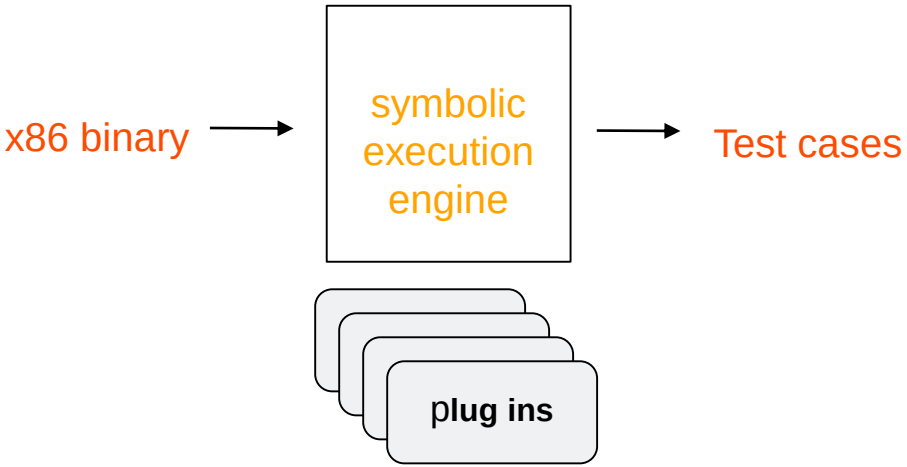
a customer site. Ideally, system designers would also like to be able to do quick *what-if analyses*, such as determining whether aligning a certain data structure on a page boundary will avoid all cache misses and thus increase performance. For small programs, experienced developers can often reason through some of these questions based on code alone. The goal of our work is to make it feasible to answer such questions for large, complex, real systems.

We introduce in this paper a platform that enables easy con-

S^{2}E

Vitaly Chipounov, et al, EPFL

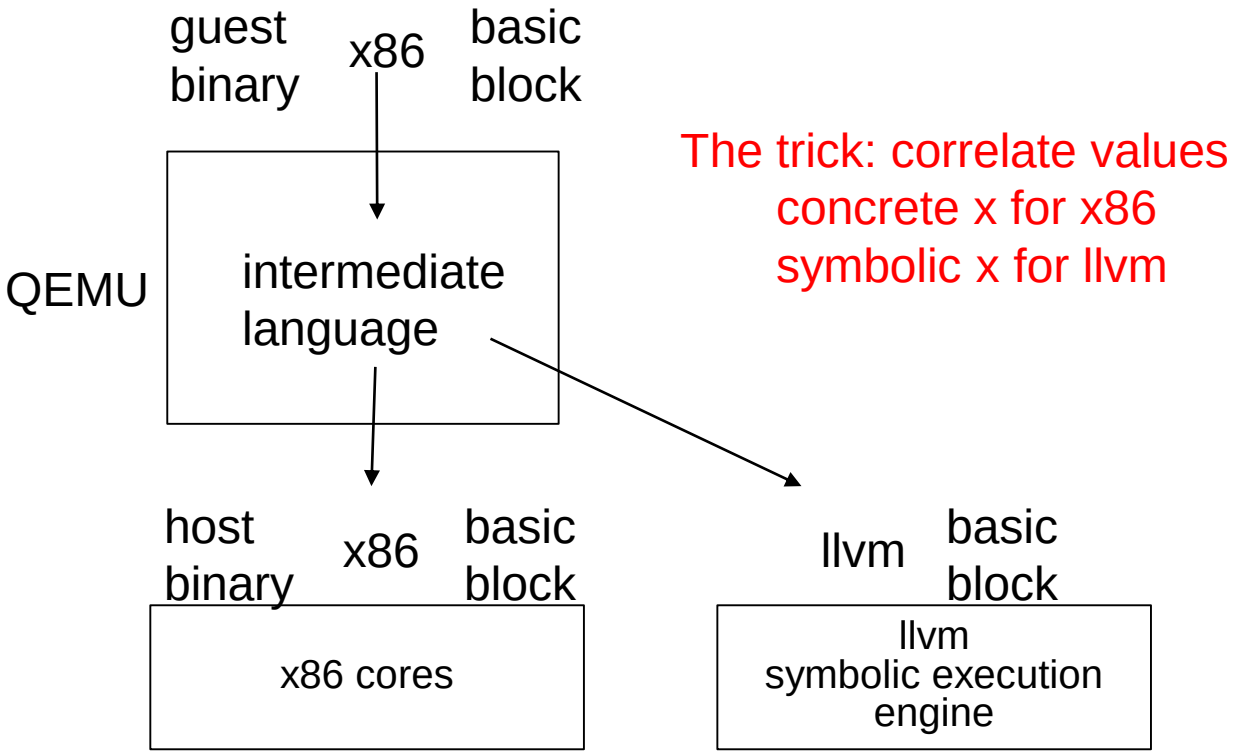
S^2E does symbolic execution on binaries



A powerful plug-in mechanism instruments and extends S^2E

Check memory references
Simulate buggy devices

S²E: dynamic binary translation



The trick: correlate values
concrete x for x86
symbolic x for llvm

Our approach

Use Open Source HW, SW and Tools

HW: Minnow Board MAX Open hardware platform

64-bit Intel® Atom™ SoC E38xx Series

http://firmware.intel.com/projects

SW: Minnow Board MAX UEFI Open Source (EDKII project)

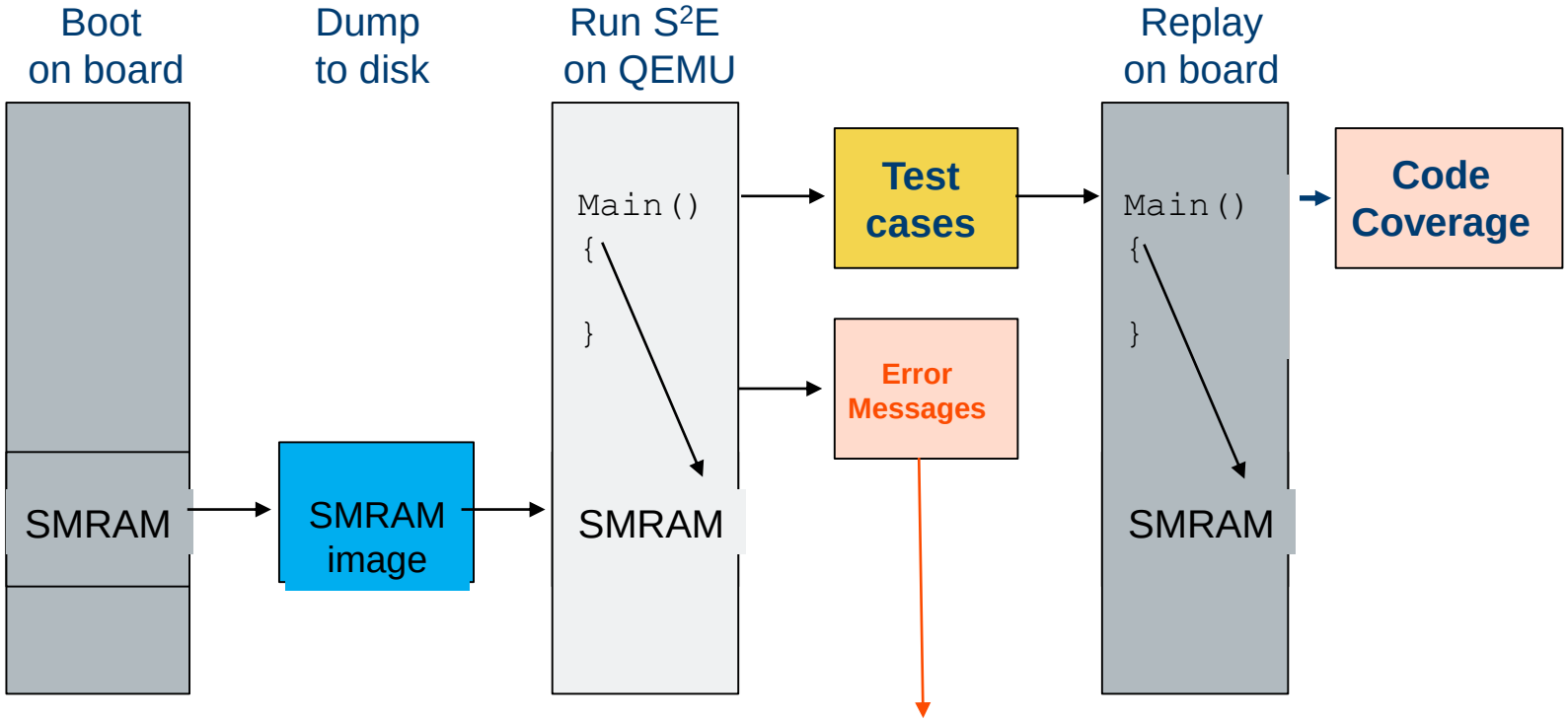
http://tianocore.sourceforge.net/wiki/EDK2

Builds using Microsoft Visual Studios or GNU C Compiler

Tools: S^2E

http://s2e.epfl.ch/

Our Process


Boot
on board
Dump
to disk
Run S2E
on QEMU
Replay
on board
SMRAM
SMRAM
image
Main()
{
}
SMRAM
Test
cases
Error
Messages
Main()
{
}
SMRAM
Code
Coverage
SmmMemoryChecker: address 0xffffffff8172eef4 out of range at pc 0x7b3ec435

Our status

Our status

For a SMM handler, we need:

- SMRAM image, its base & size and the address of the entry point

We have three tools

- ***excite-generate***: generate test cases from Linux shell
  - Generates 4000 tests in 4 hours [1]
- ***excite-replay***: replay test cases from Linux shell
- ***s2eReplay.nsh*** - UEFI shell application:
  - replay test cases on the board in 30 min
  - and measure the code coverage

[1] Intel® Core™ 2 Quad 2.66 GHZ CPU with 2GB ram running Ubuntu 14.04 LTS

For SmmVariableHandler in MdeModulePkg\Universal\Variable\RuntimeDxe\VariableSmm.c

Inducing dangerous memory ops

MemoryCheck plugin from S^2E

- Traps on every memory reference
- Checks address of every memory reference

We are modifying MemoryCheck to induce bad addresses

- Invoke solver: Could the address be outside SMRAM?

excite-checker tool in process …

Conclusion

Conclusion

We have a believable path to detecting SMRAM callouts in SMI handlers

- Test harness identifies symbolic data, but does no additional modeling
- Boot system on a board or simulator (Simics, zsim …) to desired state and dump SMRAM
- Execute entry points symbolically from that state

We believe it is extendable to BIOS in general

- UEFI capsules, binary modules, DXE drivers, UEFI applications …

as well as other embedded firmware

References

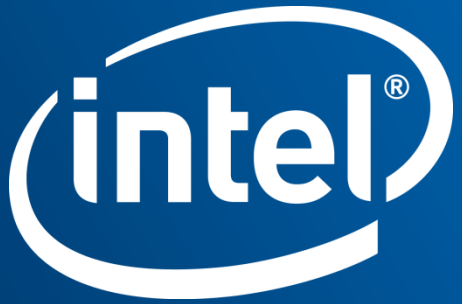
1. UEFI Forum http://www.uefi.org
2. EFI Developer Kit II http://www.tianocore.org
3. White papers, training, projects http://firmware.intel.com
4. UEFI Overview UEFI Intel Technology Journal
5. MinnowMax http://www.minnowboard.org/meet-minnowboard-max/
6. SMM Attacks

   https://cansecwest.com/csw15archive.html - See: Wojtczuk, Kallenberg, Loucaides and Zimmer

   https://cansecwest.com/csw09/csw09-duflot.pdf

   https://www.blackhat.com/us-15/briefings.html#attacking-hypervisors-using-firmware-and-hardware

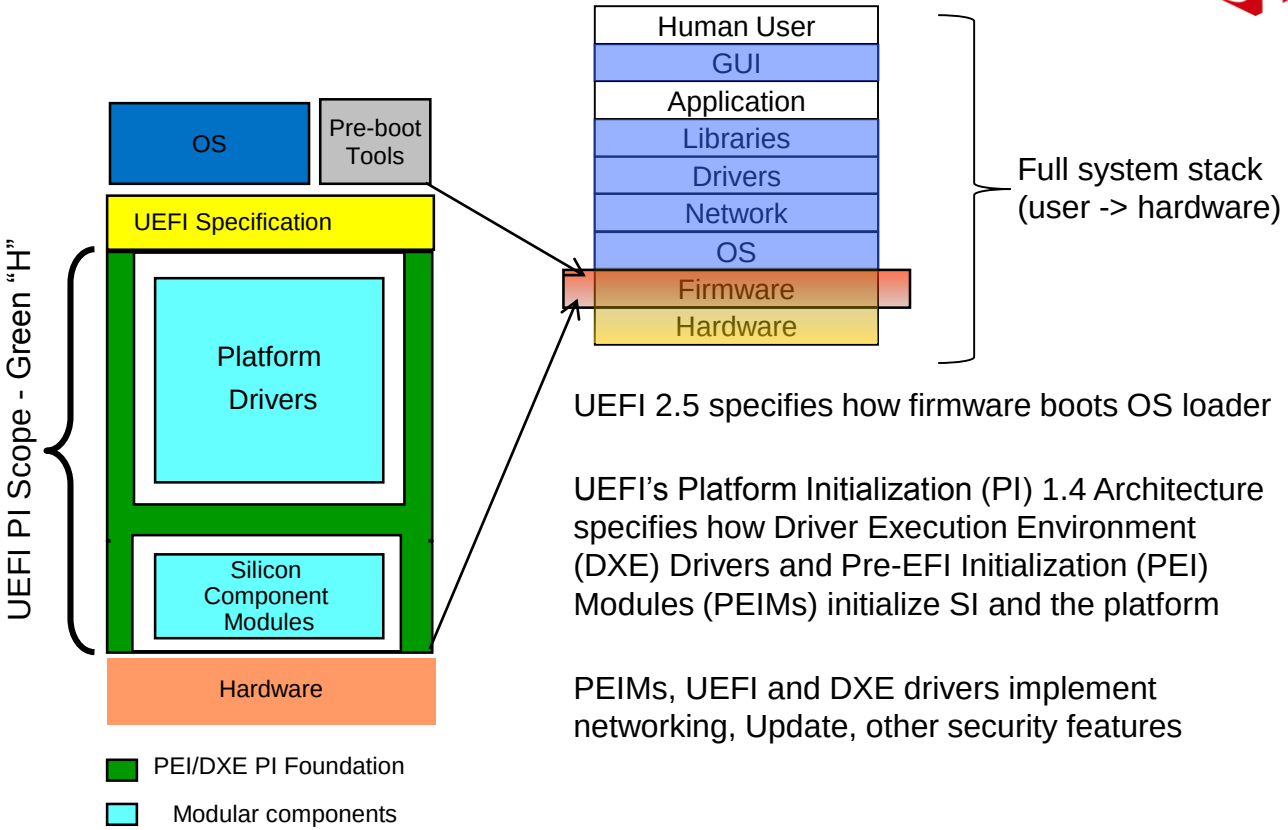
7. S^2E http://s2e.epfl.ch/


intel®

Backup

What is UEFI/PI?

www.uefi.org

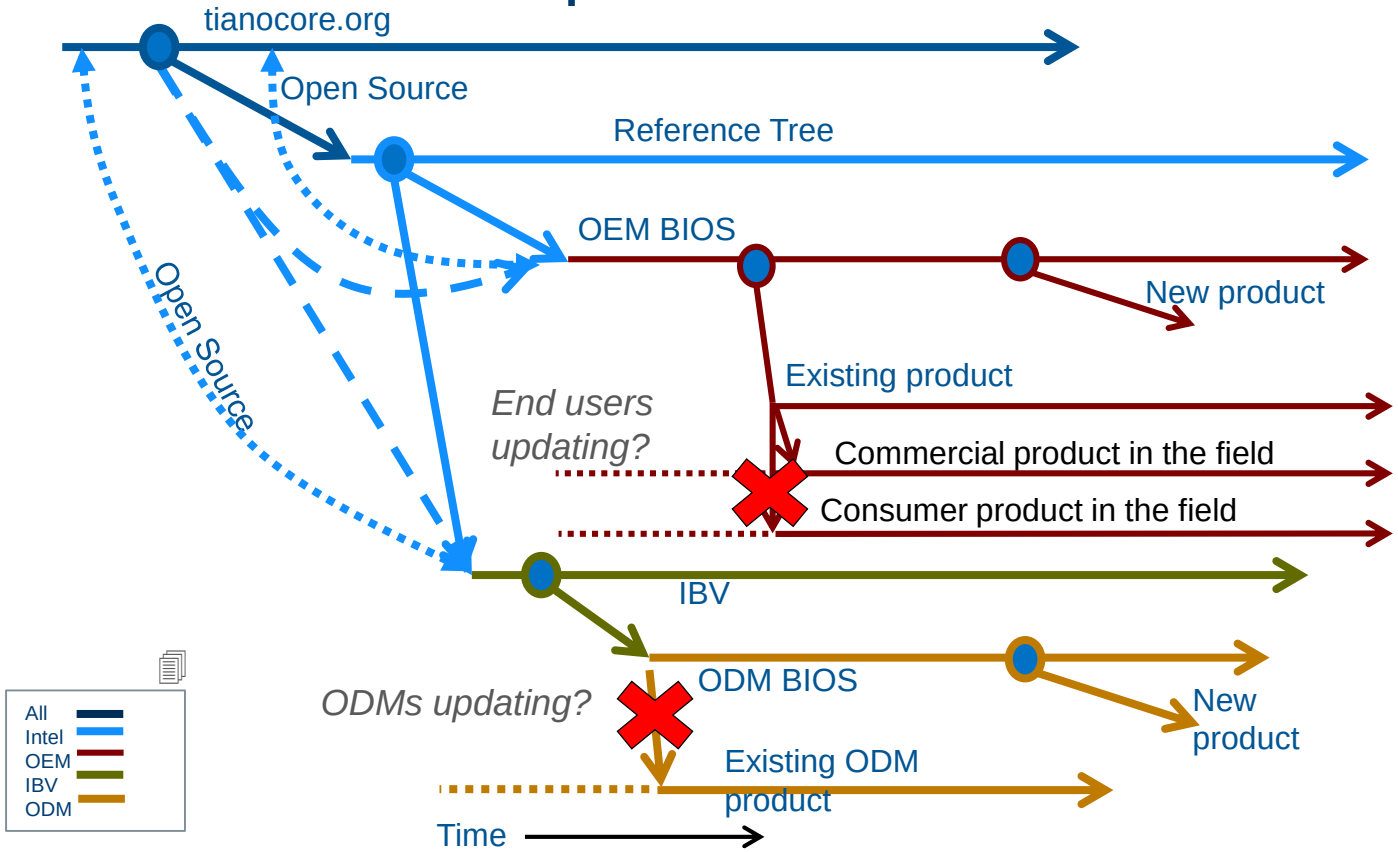


UEFI 2.5 specifies how firmware boots OS loader

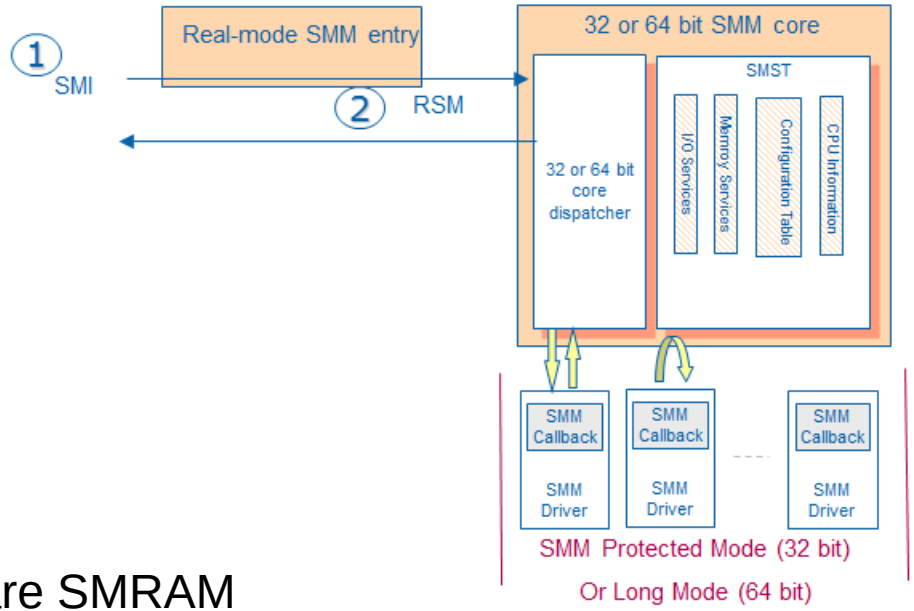
UEFI's Platform Initialization (PI) 1.4 Architecture specifies how Driver Execution Environment (DXE) Drivers and Pre-EFI Initialization (PEI) Modules (PEIMs) initialize SI and the platform

PEIMs, UEFI and DXE drivers implement networking, Update, other security features

The road from core to platform



System Management Mode with UEFI PI



- Orange regions are SMRAM
- Software model defined in PI 1.4 specification, volume 4
- Implementation at edk2\MdeModulePkg\Core\PiSmmCore

SMM Attacks

- SMI Call-outs (aka “Incursions”) – Legbacore

  Kallenberg & Kovah, LegbaCore - “*How many million BIOSes would you like to infect*?”,
  “Wojtczuk & Kallenberg - *Attacks on UEFI Security*”,
- SMI Pointer Inputs – Intel ATR

  Loucaides & Furtak, Intel - “*A new class of vulnerability in SMI Handlers of BIOS/UEFI Firmware*”

- SMM Cache Poisoning – Duflot and Invisible Things Lab
- Compatibility SMRAM Locking – Duflot

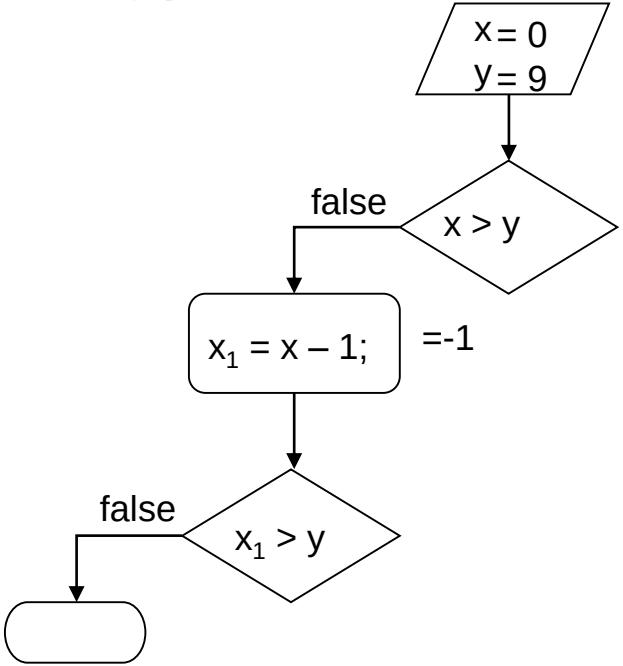
Symbolic Execution Example

Exploring f(int x, int y)

Exploration: 1

- choose x, y

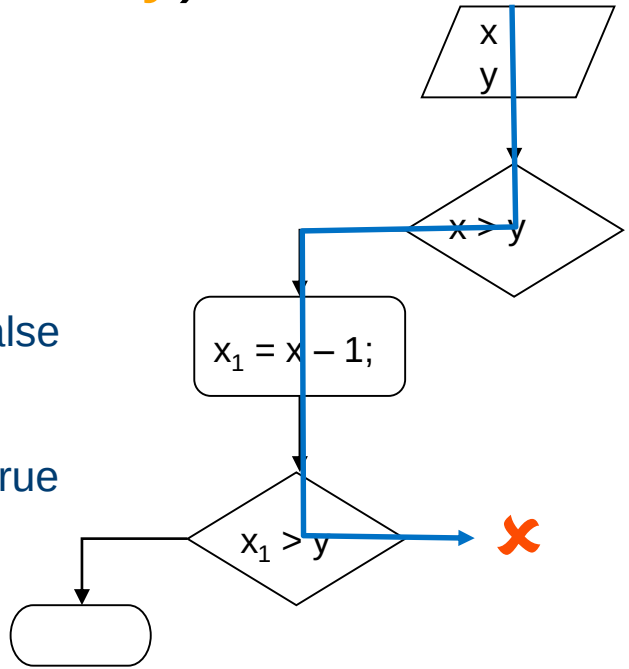
- inputs
  - x = 0
  - y = 9



Exploring f(int x, int y)

Exploration: 2

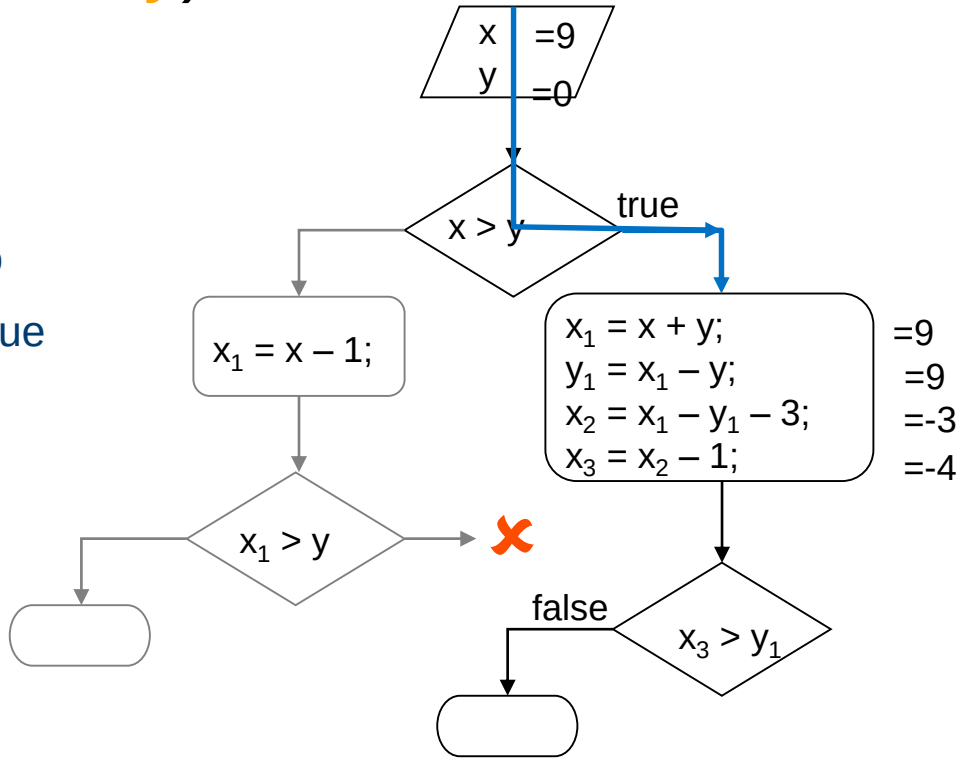
- choose x, y so
  - (x > y) = false
  - $x_1 = x - 1$
  - $(x_1 > y)$ = true
- inputs
  - no such x, y!



Exploring f(int x, int y)

Exploration: 3

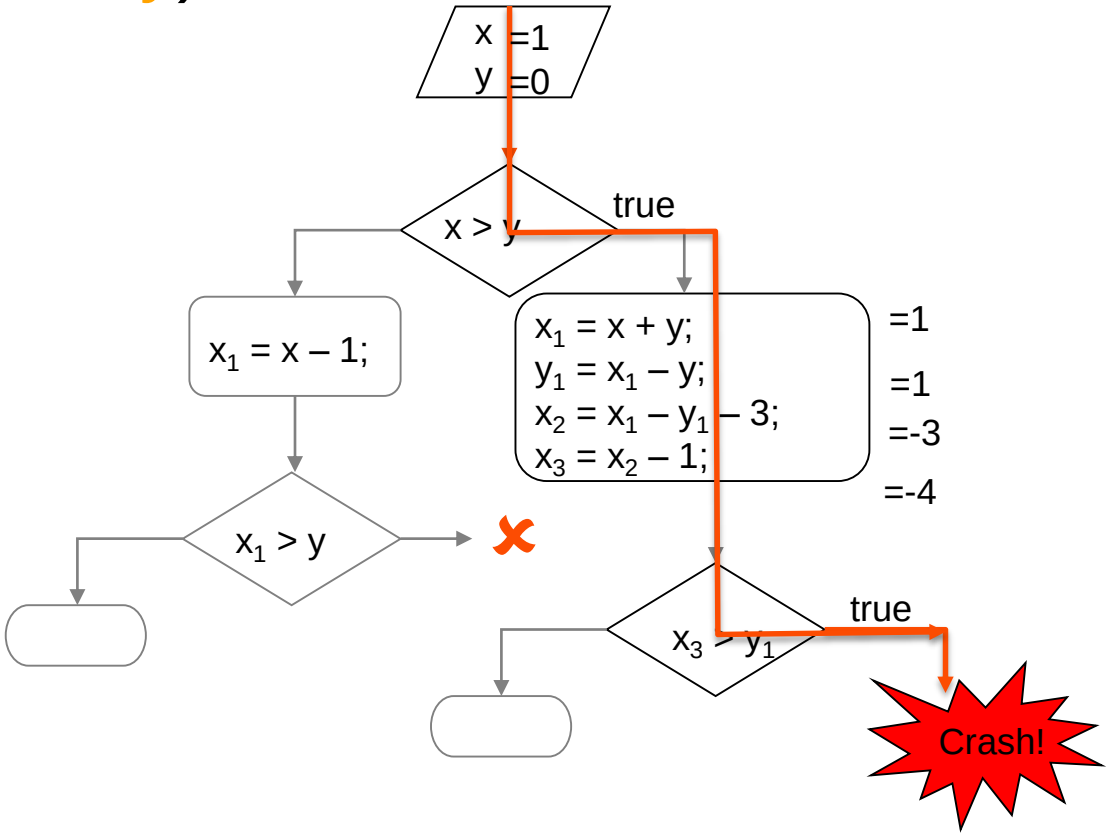
- choose x, y so
  - (x > y) = true
- inputs
  - x = 9
  - y = 0



Exploring f(int x, int y)

Exploration: 4

- choose x, y so
  - (x > y) = true
  - $x_1 = x + y$
  - $y_1 = x_1 - y$
  - $x_2 = x_1 - y_1 + 3$
  - $x_3 = x_2 - 1$
  - $(x_3 > y_1)$ = true
- inputs
  - x = 1
  - y = 0



Program

```
void f (int x, int y) {
        if (x > y) {
            x = x + y;
            y = x – y – 3;
            x = x – y;
        }
        x = x – 1;
        if (x > y) {
            abort ();
        }
}
```

Test Harness

```
int main () {
    int x,y;
    klee_make_symbolic (x);
    klee_make_symbolic (y);

    f (x, y);
    return 0;
}
```

More status

Opens

Bug hunting

- Existing plug ins only detect bugs
- We must extend them to induce bugs

Device behavior

- Model devices
  - SymbolicHardware plug in for PCI devices would be a good start
- Use devices
  - Avatar runs S^2E on devices. How about SIMICS device models?

Integration

- Goal: command-line, turn-key tool checking all handlers
- Seamless integration into product group's development and test processes
- Source annotations to identify symbolic data

Issues

State explosion

- Use path selector plug-in: heuristics for loops fill the literature

Automate handler checking

- Find the handlers to check
- Find the data to make symbolic (annotations required?)

ITS coverage tool

- Not open source (we're aiming for the broader UEFI ecosystem)
- Not easy to enable tool on code base

Coverage Data not based on the paths explored by S^2E

Mapping S^2E's errors from assembly code to corresponding C source line

Details: Test case generation

Run test harness from Bash shell on QEMU

- Memory map SMRAM into address space, jump to entry point
- Status: working

Run test harness from UEFI shell (OVMF) on QEMU

- Allocate pages and write SMRAM into memory
  - But SMRAM pages must be unused by OVMF
- Status: working for quicksort, in progress for SMM

Run test harness on seabios or s2ebios on QEMU

- Only minimal hardware initialized, small loader required
- Status: unimplemented

Details: Test case replay for coverage

Run test harness from Bash shell on Debian (no QEMU)

- Replay works
- No coverage: Considering a gcov for embedded systems

Run test harness on MinnowMax board

- SMRAM unlocked: Use SSG coverage package
  - Status: SSG is fixing the mechanism to boot unlocked
- SMRAM locked: Use SSG coverage package, write to port B2
  - Some test cases cannot be run with SMRAM locked
  - Status: In progress

S²E Configuaration – per Section 8.1 of paper

```
-- File: config.lua

s2e = {
  kleeArgs = {
  "--enable-speculative-forking=false",
  "--state-shared-memory=true",
  "--flush-tbs-on-state-switch=false"
  }
}

…
```