TECHNISCHE UNIVERSITÄT DRESDEN

INACCURATE, BUT FAST ESTIMATION OF PROGRAM VULNERABILITY?

SOBRES Lightning Talk

Björn Döbel (TU Dresden)

Braunschweig, 19.09.2012

Program Vulnerability Analysis?

- AVF computation requires deep understanding (read: a model) of underlying hardware
- Software approach: Fault Injection Experiments
  - These require a lot of time...
  - ... even with optimizations[1][2]

[1] Barbosa et al.: *Assembly-Level Pre-Injection Analysis for Improving Fault Injection Efficiency* EDCC 2005

[2] Hari et al.: *Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults*, ASPLOS 2012

Program Vulnerability Analysis?

- AVF computation requires deep understanding (read: a model) of underlying hardware
- Software approach: Fault Injection Experiments
  - These require a lot of time...
  - ... even with optimizations[1][2]
  - Money quote from[2]:

    With our existing simulation speeds, this set of faults [12 samples from SPEC, SPLASH, PARSEC] can be simulated in approximately 11 days on a cluster of 200 cores.
- But what if I want to quickly evaluate the impact of a design decision?

---

[1] Barbosa et al.: *Assembly-Level Pre-Injection Analysis for Improving Fault Injection Efficiency* EDCC 2005

[2] Hari et al.: *Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults*, ASPLOS 2012

Program Vulnerability Analysis!

- Replace AVF hardware dependency with *abstract* set of HW resources[3]
  - Registers
  - Memory
  - Instruction classes (e.g., ALU operations)
- Generate an instruction trace (e.g. from a single run of the application)
- Count ACE vs. un-ACE bits in resources

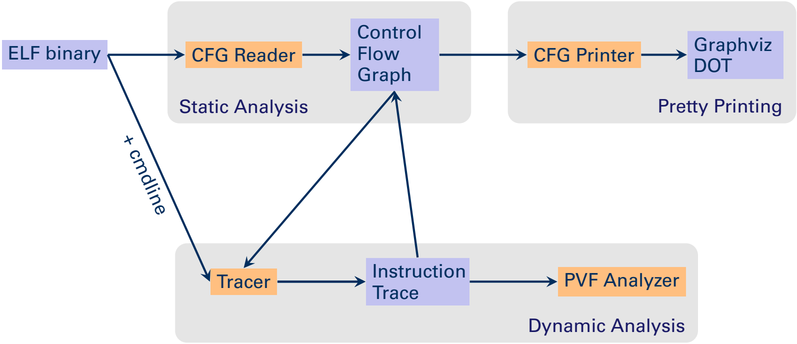
$$PVF(b_1, \ldots, b_n) = \frac{\sum_{i=0}^{n} ACE(b_i)}{n \times total\ cycles}$$

- Conversion between PVF and AVF is possible.
- Fault types: register file, integer ALU

[3] Sridharan, Kaeli: *Eliminating Microarchitectural Dependency from Architectural Vulnerability*, HPCA 2009

PVF Analysis for x86 Binaries

So I built myself some tools:



Register Vulnerability Analysis

- Fault type: register (GPR) bitflip
- Assumption: only content of live registers is vulnerable
- $\rightarrow$Register Liveness Analysis
- Example: Recursive vs. iterative Quicksort

Register Vulnerability Analysis

- Fault type: register (GPR) bitflip
- Assumption: only content of live registers is vulnerable
- →Register Liveness Analysis
- Example: Recursive vs. iterative Quicksort

	EAX	EBX	ECX	EDX	ESP	ESI	EDI	EBP	**AVG**
QSort, rec, GCC, O0	0.78	0.00	0.06	0.33	0.99	0.00	0.00	0.99	0.40
QSort, rec, GCC, O1	0.53	0.76	0.15	0.25	0.97	0.82	0.97	0.96	0.67
QSort, rec, GCC, O2	0.80	0.57	0.57	0.62	0.99	0.28	0.14	0.97	0.62
QSort, rec, GCC, O3	0.42	0.73	0.57	0.56	0.99	0.68	0.30	0.99	0.66
QSort, iter, GCC, O0	0.69	0.00	0.002	0.25	0.0001	0.00	0.00	0.99	0.24
QSort, iter, GCC, O1	0.91	0.14	0.53	0.90	0.99	0.92	0.005	0.98	0.67
QSort, iter, GCC, O2	0.74	0.43	0.52	0.82	0.99	0.03	0.94	0.54	0.63
QSort, iter, GCC, O3	0.75	0.43	0.53	0.82	0.99	0.03	0.94	0.54	0.63
QSort, iter, clang, O0	0.68	0.00	0.74	0.11	0.99	0.00	0.0001	0.99	0.44
QSort, iter, clang, O1	0.61	0.80	0.48	0.02	0.99	0.98	0.0001	0.78	0.58
QSort, iter, clang, O2	0.60	0.80	0.44	0.08	0.99	0.98	0.004	0.78	0.59
QSort, iter, clang, O3	0.60	0.80	0.44	0.08	0.99	0.98	0.004	0.78	0.59

Apples and Oranges!

- PVF is not comparable across different versions of a program!
- Need to factor in that better optimized programs run faster.

Apples and Oranges!

- PVF is not comparable across different versions of a program!
- Need to factor in that better optimized programs run faster.
- PVF-Delay product?
  - Shortcut: simply count machine code instructions executed

Apples and Oranges!

- PVF is not comparable across different versions of a program!
- Need to factor in that better optimized programs run faster.
- PVF-Delay product?
  - Shortcut: simply count machine code instructions executed
- But we might also be interested in other metrics
- Energy-PVF-...-Delay product

Discussion

- Is this better than AVF analysis / fault injection?
  - Higher inaccuracy
  - But: much faster, so it might be a helpful estimate
- For a useful estimate we also need to classify the potential outcome of errors.
  - Heuristics: memory access $\rightarrow$ crash, plain computation $\rightarrow$ data corruption
  - Crazy idea: train a neural network to do so. What would we input then?
- Can we analyse the efficiency of fault tolerance mechanisms?
  - Replication: $PVF := pvf_1 \times pvf_2 \times pvf_3$
  - Encoded processing: per-bit PVF instead of per-register one
- Can we factor in microarchitectural knowledge (our goal being x86 after all)?
  - Might be useful for more accurate results.
- How do we validate the numbers?

Nothing to see here

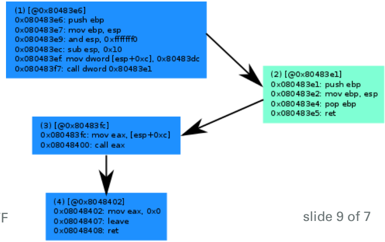
This slide intentionally left blank.

Except for above text.

Static Analysis

- Inputs:
  - ELF binary
  - Start address (if != _start)
- Run through statically detectable code and build CFG
- Dynamic jumps left unresolved
- Dynamic libs ignored for now
  - Ignore jumps to PLT/GOT
  - Explicitly specified terminator addresses to halt CFG building at calls to `exit@PLT`.

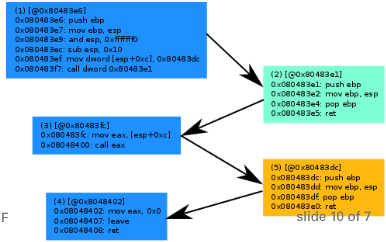
```
void foo() { }
void bar() { }
int main()
{
    void (*func)(void) = foo;
    bar();
    func();
}
```



Dynamic Analysis

- Goals:
  - Resolve dynamic jumps
  - Generate exact instruction trace
- Place & inspect appropriate breakpoints using `ptrace`
  - Unresolved `jmp/call` instructions
  - Last instruction of any BB with more than one successor
- “Room for optimization”

```
void foo() { }
void bar() { }
int main()
{
    void (*func)(void) = foo;
    bar();
    func();
}
```



TECHNISCHE UNIVERSITÄT DRESDEN

Dependence on Input Size

QSort, iterative, GCC, -O3