Practical Automated Vulnerability Monitoring Using Program State Invariants

Cristiano Giuffrida  Lorenzo Cavallaro  Andrew S. Tanenbaum

Vrije Universiteit Amsterdam

43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Budapest, Hungary
June 24-27, 2013
Source: “The Total Growth of Open Source”
Vulnerability Disclosure Growth by Year
1996 to 2012

Unpatched Vulnerabilities

Overall Vulnerabilities without Remediation
2006 to 2012

Source: "IBM X-Force 2012 Trend and Risk Report"
Automated Vulnerability Monitoring

- Dynamic program monitoring to identify security vulnerabilities.
- Proactive security solution in production systems.
- Feedback to both system administrators and software developers.
- Discover *new* vulnerabilities that escape software testing.
- Prioritize *known* vulnerabilities to go after.
- Focus on memory error vulnerabilities.
Existing Solutions

- Memory error detectors [Valgrind, Bounds checkers, WIT].
  - **Pros**: Accurate detection model.
  - **Cons**: High overhead for production systems.

- Memory error monitors [Cruiser, Kruiser, HeapSentry].
  - **Pros**: Low overhead.
  - **Cons**: Noncomprehensive detection model.

- Likely invariant-based error detectors [Daikon, AccMon, SWAT].
  - **Pros**: Comprehensive detection model.
  - **Cons**: False positives subject to training coverage.
Our approach

- Analyze source code and annotate *Program State Invariants*.

- Monitor target application in realtime and check invariants.

- Detect program state corruption from invariants violations.

- Provide feedback on memory errors and other dangerous behavior.

- **Low-overhead, conservative, and comprehensive** analysis.
enum week\_day today = MONDAY;
int admin = 0;
...
today = (today+1)\%7;
...
admin = 1;

Value-based invariants
void *pointer = NULL;
...
pointer = &var;
blob = malloc(sizeof(blob_t));
...
pointer = blob;

---

**Target-based invariants**

```
void *pointer = NULL;
...
pointer = &var;
blob = malloc(sizeof(blob_t));
...
pointer = blob;
```

- **target:** var, blob, ...
- **pointer:** target
blob_t blob;
ptr_t *pointer = (ptr_t *) &blob;
...
data = (data_t *) pointer;
...
text = (text_t *) pointer;

T*: blob_t data_t text_t

pointer: ptr_t,T*
The RCORE Architecture

**Link time**

**Original Program**
- Data
- Code

**Static Instrumentation**

**Instrumented Program**
- Metadata
- Data
- Instrumented Code
- Metadata Framework

**Runtime**

**Application Threads**
- Application Code
- Library Wrapper
- Dynamic Instrumentation
- Modify Metadata

**Core 1**
- Metadata Framework

**Core 2**
- Run-time Analyzer
- Lookup Metadata

**Monitoring Threads**
Static instrumentation

Before Instrumentation

Original Program
- Data
- Code

After Instrumentation

Statically Instrumented Program
- Data
- Metadata
- Instrumented code
- Metadata Framework

LLVM
Metadata framework

- Data structures and API to manage metadata.
- API to create/destroy/update metadata dynamically.
- API to lookup metadata for invariants analysis.
- Metadata wrappers for memory allocator instrumentation.
- Generic library to support multiple static/runtime analyzers.
Dynamic instrumentation

- Metadata management for uninstrumented shared libraries.
- Necessary to avoid false positives.
- Create metadata for shared libraries at runtime.
- Index new text/data memory regions as untyped.
- Dynamic memory management instrumentation.
Schedule the monitoring activity on a spare core.

Sample execution periodically and check invariants.

The monitoring cycle:

- State introspection.
- Invariants analysis.
- Recording.
- Feedback generation.
Evaluation

- Applications tested: NGINX, LIGHTTPD, EXIM, PROFTPD.

- Performance analysis shows negligible overhead.

- Real-world vulnerability testing confirms effectiveness.

- Free-run effectiveness testing:
  - Dangling pointers.
  - Off-by-one pointers.
  - New vulnerability found in EXIM.
Performance

Normalized execution time

Instrumentation only
Instrumentation and 1 run-time analyzer on 1 core
Instrumentation and 7 run-time analyzers on 4 cores
Instrumentation and 15 run-time analyzers on 8 cores
Detection latency

![Graph showing monitoring cycle time (ms) and PSI violations detected (%).](image)
Summary

- **RCORE**: an infrastructure for automated vulnerability monitoring.
- Combine static + dynamic analysis to check *program state invariants*.
- Perform conservative analysis to avoid false positives.
- Rely on spare cores to keep the analysis efficient.
- Provide comprehensive memory error detection.
Practical Automated Vulnerability Monitoring
Using Program State Invariants

Thank you!
Any questions?

Cristiano Giuffrida, Lorenzo Cavallaro, Andy Tanenbaum
{giuffrida,sullivan,ast}@cs.vu.nl

Vrije Universiteit Amsterdam