Static Analysis Based Invariant Detection for Commodity Operating Systems

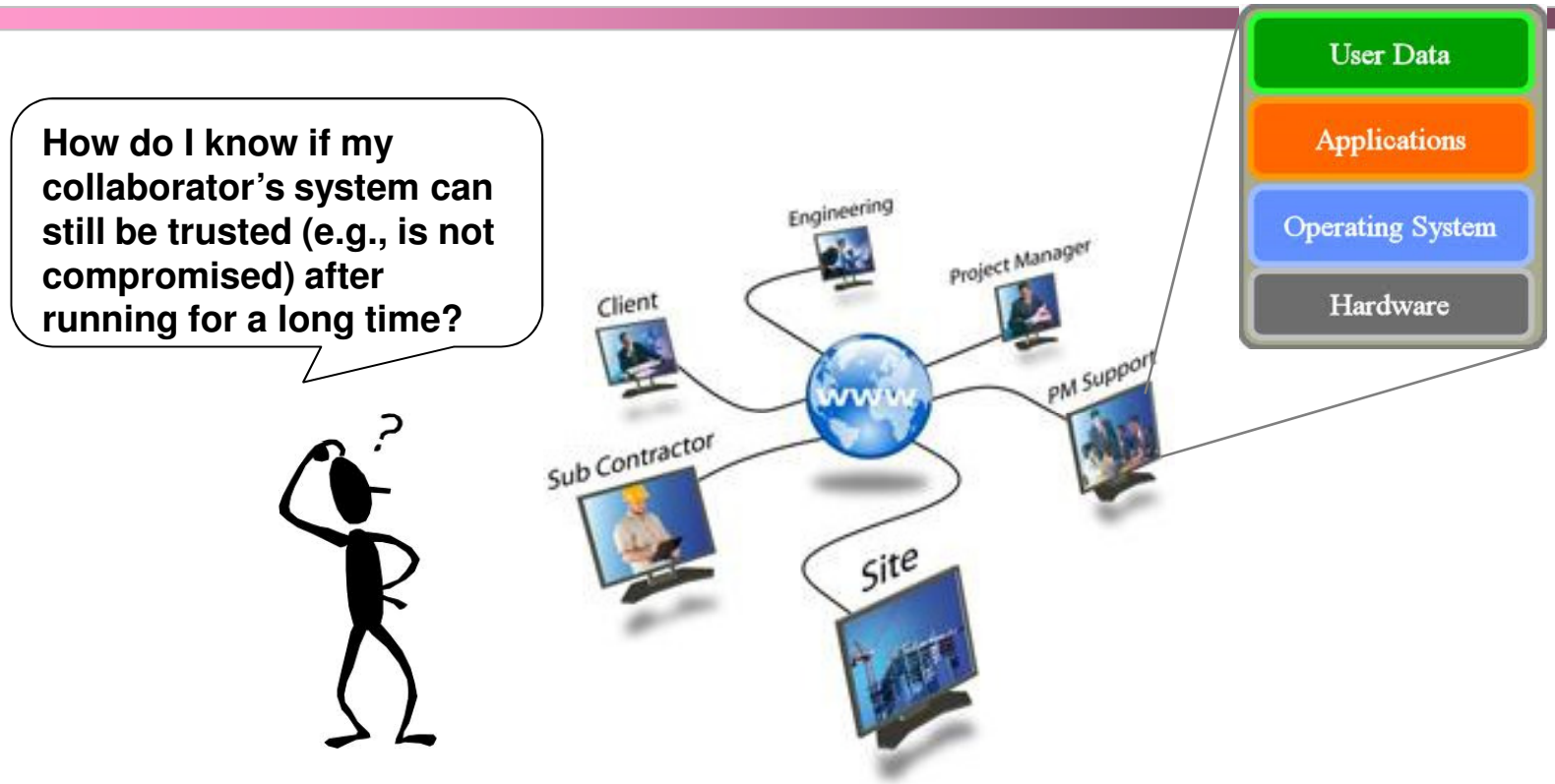
Jinpeng Wei, Feng Zhu
Florida International University
Miami, FL, USA

Yasushi Shinjo
University of Tsukuba
Tsukuba, Ibaraki, Japan

7th International Conference on Collaborative Computing:
Networking, Applications and Worksharing

Orlando, Florida, USA
October 15 - 18, 2011

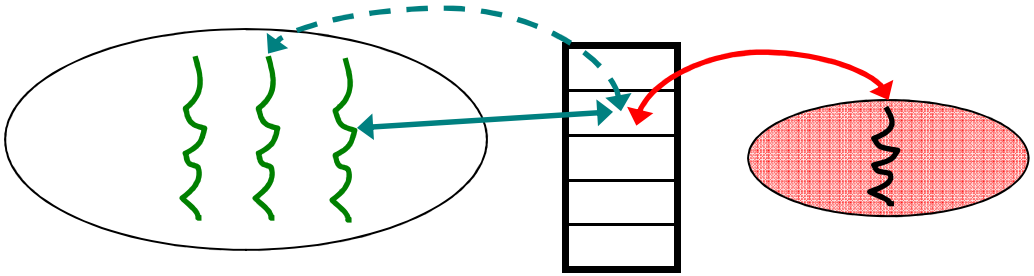
Trust Issues in Collaborative Environments



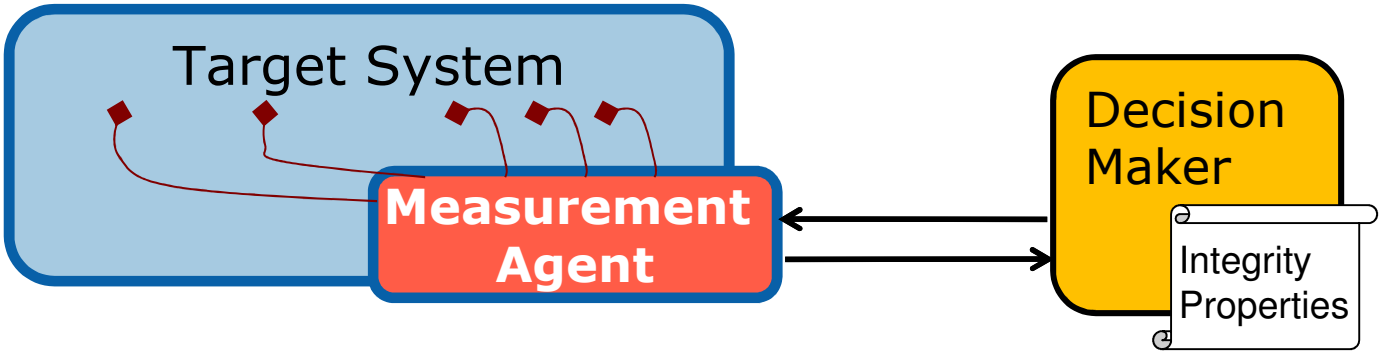
- Constant attacks exploiting vulnerabilities (e.g. buffer overflow, SQL injection)
- Configuration errors
- Malware (e.g., rootkits) are increasingly stealthy

One Solution: Integrity Checks through Remote Attestation

- Example malware: persistent control flow attacks



One Solution: Integrity Checks through Remote Attestation



- Integrity guarantee is only as strong as the completeness of the integrity model
- E.g., if integrity properties only cover system call table, a new rootkit can manipulate other function pointers (such as those found in device driver jump tables) to achieve its goal and remain undetected
- Question: Given the target software, how can we systematically and accurately identify its integrity properties?

Challenge of Runtime Attestation: Precise Integrity Models

- Two classic attestation errors when the model is not precise
  - False positives: the model is overly stringent
  - False negatives: the model is too loose
  - Both kinds of errors are undesirable

Integrity Model Derivation Approaches

- Manual analysis
  - Hard to scale, hard to counter novel attacks that move their targets to less-known places
- Dynamic analysis
  - Inability to explore all possible program execution paths
  - Gibraltar generates about 4,673 false positives
  - ReDAS has to create 70 training scenarios and 13,000 training sessions
- Static analysis …

Example Program

```
int v = 100;

while (1) {
    …
    if (tcp connect request from port 8088){
        v = v + 2;
    }
    …
}
```

- Dynamic analysis may report *v* as an invariant
- Static analysis will not

Our Contributions

- A program analysis tool that can automatically derive *global invariants* from source code, using static analysis
- A thorough study of global invariants detection for the Linux kernel
- An invariant monitor based on the result of the static analysis with low false positive and false negative rates

Outline

- Background
- Design and Implementation
- Evaluation
- Conclusion

Background

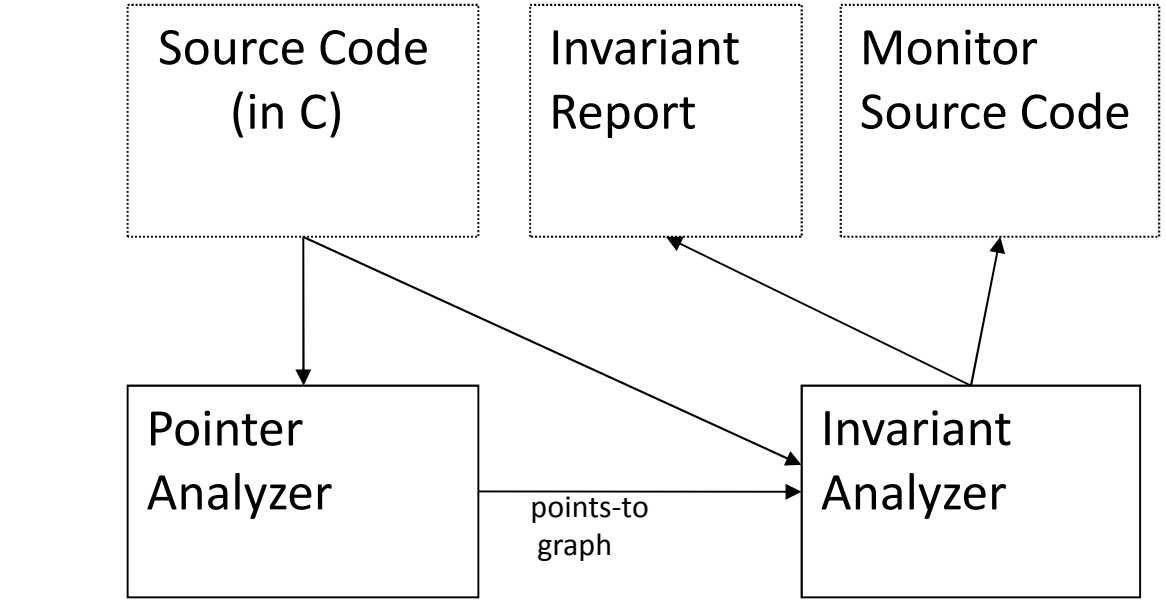
- Definition of Global Invariant
  - Global variable that has a *known-good* value during the runtime of the system
- Relevance in Integrity Protection
  - Represent the immutability of critical internal control data of the target system, e.g, Interrupt Descriptor Table (IDT),system call table, System Service Descriptor Table
  - Popular targets of attacks by rootkits (e.g., SucKIT, Hacker defender, NTIllusion, HE4Hook)
  - Basis for rootkit detectors (e.g., ReDAS, Copilot, and several commercial tools)

Outline

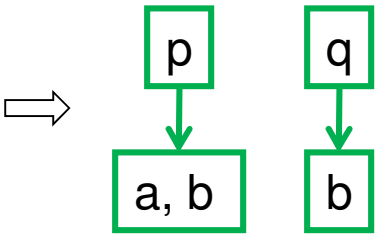
- Background
- Design and Implementation
- Evaluation
- Conclusion

Design and Implementation (1)

- Architecture




```
int *p, *q, a, b;
p = &a; q = &b;
if (...) p = q;
*p = 2;
```



Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis

Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis
    - Example: field-insensitive analysis would consider the entire structure **v** as non-invariant, including **v.a**, and **v.c** → need to be field-sensitive

```
struct {char* a; int b; long c} v;
…
v.b = v.b + 2;
```

Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis
    - Example: array-insensitive analysis would consider the entire array **v** as non-invariant, including **v[1]**, **v[2]**, …, **v[9]** → need to be array-sensitive

```
int v[10], m;
…
v[0] = v[0] + m;
```

Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis
    - Example: conservative pointer analysis would consider **c** non-invariant as well → need precise pointer analysis

```
char* a; int b; int c;
int *p;
…
p = &b; …
*p = *p + 1;
```

Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis
    - Solution: field- and array-sensitivity, precise pointer analysis
  - False positives
    - Causes: implicit assignments and incomplete points-to analysis

Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis
    - Solution: field- and array-sensitivity, precise pointer analysis
  - False positives
    - Causes: implicit assignments and incomplete points-to analysis
    - Example: a structure-level assignment implicitly assigns to all fields. Here **foo = bar** modifies both **foo.a** and **foo.b**

```
struct {int a; int b;} foo, bar;
…
foo = bar;
```

Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis
    - Solution: field- and array-sensitivity, precise pointer analysis
  - False positives
    - Causes: implicit assignments and incomplete points-to analysis
    - Example: If the points-to set of **q** does not contain **a**, then **a** is mistakenly considered an invariant → need precise pointer analysis

```
int a; int b; int *p, int *q;
p = &a; …;
if (some condition) {q = &b; …}
else {q = p; …}; …
*q = *q + 2;
```

Design and Implementation (2)

- Goal: high precision by minimizing false negative and false positive rates
  - False negatives
    - Cause: a lack of fine-granularity and imprecise pointer analysis
    - Solution: field- and array-sensitivity, precise pointer analysis
  - False positives
    - Causes: implicit assignments and incomplete points-to analysis
    - Solution: heuristics and precise pointer analysis

Design and Implementation (3)

- Assignment Recognition
  - Field Sensitivity: v.a and v.b are different variables
  - Array Sensitivity: v[0] and v[1] different variables
  - Pointer Analysis: use a precise algorithm called generalized one level flow (GOLF)
  - Union Support: treat each field of a union as an alias of other fields in the same union. E.g., **union uarg{int a; int b} c**, if **c.a** non-invariant, **c.b** non-invariant, either
  - Heuristics-base Assignment Recognition

Design and Implementation (3)

- Heuristics-base assignment recognition
  - Function prototype-based heuristic: capture implicit assignment by assembly code.
    - Example functions: memcpy, copy_from_user, spin_lock
  - Structure-level assignments
    - E.g., given **struct {int a; int b;} foo, bar** , **foo = bar** is translated into **foo.a = bar.a; foo.b = bar.b**

Design and Implementation (4)

- Invariant Recognition
  - Associate a flag (invariant or not) and a legal value list with each global variable
  - Scan global variable declarations and initialization functions and fill global variable's legal value list
  - Scan the remaining kernel functions. If a global variable, which is marked as an invariant, is assigned a non-constant value, or a constant value but the value is not in its legal value list, the analyzer marks it as a non-invariant
  - Generates a report about the invariant status of all global variables based on their flags

Outline

- Background
- Design and Implementation
- Evaluation
- Conclusion

Metrics and Methodology

- Target software: Linux kernel 2.4.32
- Metrics
  - False positives
  - False negatives
- Methodology
  - Comparing with a dynamic invariant detector
  - Checking invariants against real software (benign or malicious)

Test Cases

- Benign test cases

Test program	*Description*
ltp-2005	Linux Test Project: more than 700 test cases for the Linux kernel and more than 60 test cases for the network stack
Iperf	A network testing tool that measures the throughput of a network, thus exercising the network subsystem of the kernel
Andrew benchmark	A file system benchmark
Miscellaneous	Kernel compilation, ssh, scp, common commands

- Malicious test cases: real-world rootkits such as SuckIT

Comparing with a Dynamic Invariant Detector

Category	*Total #*	*# Error static*	*# Error dyna.*
S.NI, D.NI	154	0	0
S.NI, D.I	17,200	18(FN)	17,182(FP)
S.I, D.NI	0	0	0
S.I, D.I	136,778	1(FP)	1(FP)

S: Static, D: Dynamic; NI: Non-invariant; I: Invariant

Verification of the 17,200 Possible Non-invariants

- If a variable is directly modified: the assignment statement logged in the analysis report is straightforward evidence that the variable is a non-invariant

- If a variable is only indirectly modified through a pointer, our analyzer outputs the relevant statements from the source code that support the points-to relationship

Points-to Analysis Report Example

- Why ctrl_map[2] can be indirectly modified through the pointer key_map

<Name>ctrl_map[2]</Name>
<Invariant>No</Invariant>
<Reason1>
*(key_map + 0) = (unsigned short)(((2 << 8) | 126) ^ 61440);
vt.c:224, Indirectly modified through key_map.

Path from ctrl_map[2] to key_map:
<Label>ctrl_map[2]</Label>
<STMT>ctrl_map=&ctrl_map[2] defkeymap.c:65</STMT>
<Label>l_473154</Label>
<STMT>key_maps[4]=ctrl_map defkeymap.c:141</STMT>
<Label>l_479876</Label>
<STMT>key_map = key_maps[tmp.kb_table];vt.c:174
</STMT>

Comparing with a Dynamic Invariant Detector

Category	*Total #*	*# Error static*	*# Error dyna.*
S.NI, D.NI	154	0	0
S.NI, D.I	17,200	18(FN)	17,182(FP)
S.I, D.NI	0	0	0
S.I, D.I	136,778	1(FP)	1(FP)

S: Static, D: Dynamic; NI: Non-invariant; I: Invariant

Example Non-Invariants

Category	*Example variables*
List heads	acpi_bus_drivers.next arp_tbl.gc_timer.list.next
Locks	dev_base_lock.lock exec_domains_lock.lock
Auditing information	kernel_module.kallsyms_start kernel_module.kallsyms_end
Accounting information	console_sem.count.counter con_buf_sem.count.counter
Resource mgmt data	contig_page_data.node_zonelists[0].zones[0] contig_page_data.node_zones[0].free_area[0].map
Configuration data	FDC2,FLOPPY_DMA,FLOPPY_IRQ, can_use_virtual_dma,fifo_depth
Driver-specific data	eth0_dev.allmulti, eth0_dev.dev_addr[0]

Comparing with a Dynamic Invariant Detector

Category	*Total #*	*# Error static*	*# Error dyna.*
S.NI, D.NI	154	0	0
S.NI, D.I	17,200	18(FN)	17,182(FP)
S.I, D.NI	0	0	0
S.I, D.I	136,778	1(FP)	1(FP)

S: Static, D: Dynamic; NI: Non-invariant; I: Invariant

Experimental Evaluation of Accuracy

- We implement and run an Invariant Monitor based on the static analysis result
- False positive: only ONE false invariant out of 141,280
- False negative: successfully detect the SucKIT 2 rootkit, which modifies sys_call_table[59]

```
p=(struct timedia_struct*)0xc0272420;

if (((struct timedia_struct*)p)[3].num!=8)
 {printk(KERN_WARNING "Bad invariant
timedia_data[3].num \n");};
```

Outline

- Background
- Design and Implementation
- Evaluation
- Conclusion

Conclusion

- Core techniques
  - Static analysis – design and implement automated tools that can derive global invariants out of the target kernel without running it
  - Evaluate our methodology
    - Compare with dynamic analyer
  - Static analysis is a viable option for automated integrity property derivation and can have very low false positive and false negative rates