Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors

Sven Grünewald

Christian-Albrechts-Universität Kiel
Department of Computer Science and Applied Mathematics

Zuverlässigkeit von Software in sicherheitskritischen Systemen
Outline

1. Introduction
   - ERRORS
   - P4 vs. G4

2. Error Injection Campaign
   - Setup
   - Categorization

3. Experimental Results
   - Overview
   - Stack Injection
   - System Register Injection
   - Code Injection
   - Data Injection

4. Conclusion

5. Appendix
Outline

1. Introduction
   - ERRORS
   - P4 vs. G4

2. Error Injection Campaign
   - Setup
   - Categorization

3. Experimental Results
   - Overview
   - Stack Injection
   - System Register Injection
   - Code Injection
   - Data Injection

4. Conclusion

5. Appendix
Outline

1. Introduction
   - ERRORS
   - P4 vs. G4

2. Error Injection Campaign
   - Setup
   - Categorization

3. Experimental Results
   - Overview
   - Stack Injection
   - System Register Injection
   - Code Injection
   - Data Injection

4. Conclusion

5. Appendix
Outline

1. Introduction
   - ERRORS
   - P4 vs. G4

2. Error Injection Campaign
   - Setup
   - Categorization

3. Experimental Results
   - Overview
   - Stack Injection
   - System Register Injection
   - Code Injection
   - Data Injection

4. Conclusion

5. Appendix
Outline

1. Introduction
   - ERRORS
   - P4 vs. G4

2. Error Injection Campaign
   - Setup
   - Categorization

3. Experimental Results
   - Overview
   - Stack Injection
   - System Register Injection
   - Code Injection
   - Data Injection

4. Conclusion

5. Appendix
Errors Occur

- It is hard to produce correct software

It is hard to produce correct software. Even correct software can be corrupted by hardware faults or the environment, e.g., noise, radiation, etc. Therefore, it is important to build fault-resistant hardware as well.
Errors Occur

- It is hard to produce correct software
- Even correct software can be corrupted by hardware faults or the environment e.g. noise, radiation etc.
Errors Occur

- It is hard to produce correct software
- Even correct software can be corrupted by hardware faults or the environment e.g. noise, radiation etc.
- So it is important to build fault-resistant hardware as well
Errors Occur

- It is hard to produce correct software
- Even correct software can be corrupted by hardware faults or the environment e.g. noise, radiation etc.
- So it is important to build fault-resistant hardware as well
- Therefore we compare two different systems’ fault tolerance:
Errors Occur

- It is hard to produce correct software
- Even correct software can be corrupted by hardware faults or the environment e.g. noise, radiation etc.
- So it is important to build fault-resistant hardware as well
- Therefore we compare two different systems’ fault tolerance:
  - The Pentium 4 and the PowerPC G4
Errors Occur

- It is hard to produce correct software
- Even correct software can be corrupted by hardware faults or the environment e.g. noise, radiation etc.
- So it is important to build fault-resistant hardware as well
- Therefore we compare two different systems’ fault tolerance:
  - The Pentium 4 and the PowerPC G4
  - Both running with Linux Kernel 2.4.22
There have been many other studies analysing the failure behavior of operating systems with different focus:
Related Work

There have been many other studies analysing the failure behavior of operating systems with different focus:

- OS’s behavior while executing API/system calls with erroneous arguments
There have been many other studies analysing the failure behavior of operating systems with different focus:

- OS’s behavior while executing API/system calls with erroneous arguments
- Error injection into kernel and user space
Related Work

There have been many other studies analysing the failure behavior of operating systems with different focus:

- OS’s behavior while executing API/system calls with erroneous arguments
- Error injection into kernel and user space
- Static, compiler analyses of sourcecodes
Related Work

There have been many other studies analysing the failure behavior of operating systems with different focus:

- OS’s behavior while executing API/system calls with erroneous arguments
- Error injection into kernel and user space
- Static, compiler analyses of sourcecodes
- and many other more
Related Work

There have been many other studies analysing the failure behavior of operating systems with different focus:

- OS’s behavior while executing API/system calls with erroneous arguments
- Error injection into kernel and user space
- Static, compiler analyses of sourcecodes
- and many other more

This experiment is focused on direct comparison of Pentium 4 and PowerPC G4
System configurations

<table>
<thead>
<tr>
<th>Processor</th>
<th>CPU Clock</th>
<th>Memory</th>
<th>Distribution</th>
<th>Kernel vers.</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>1.5</td>
<td>256MB</td>
<td>RedHat 9.0</td>
<td>2.4.22</td>
<td>GCC 3.2.2</td>
</tr>
<tr>
<td>MPC 7455</td>
<td>1.0</td>
<td>256MB</td>
<td>YellowDog 3.0</td>
<td>2.4.22</td>
<td>GCC 3.2.2</td>
</tr>
</tbody>
</table>

Both systems are running with the same Version of Linux kernel and compiler and have the same amount of memory. So they are running under (nearly) the same conditions.
## RISC vs. CISC

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many instructions (some with long runtime)</td>
<td>Few and always fast instructions</td>
</tr>
<tr>
<td>Few registers</td>
<td>Many registers</td>
</tr>
<tr>
<td>Variable instruction length</td>
<td>Fixed instruction length</td>
</tr>
<tr>
<td>Stack-intensive procedure linkage</td>
<td>Register-intensive procedure linkage</td>
</tr>
<tr>
<td>Arithmetic/logic operations on memory and registers</td>
<td>Operations only on registers (load/store)</td>
</tr>
</tbody>
</table>
This error injection campaign uses the software framework NFTAPE (which is built for fault/error injection experiments)
The Setup

- This error injection campaign uses the software framework NFTAPE (which is built for fault/error injection experiments)
- UnixBench is running as workload
This error injection campaign uses the software framework NFTAPE (which is built for fault/error injection experiments)

- UnixBench is running as workload
- Single bit errors are injected using a driver-based kernel module
The Setup

- This error injection campaign uses the software framework NFTAPE (which is built for fault/error injection experiments)
- UnixBench is running as workload
- Single bit errors are injected using a driver-based kernel module
- They are automatically injected into pregenerated targets
The Setup

- This error injection campaign uses the software framework NFTAPE (which is built for fault/error injection experiments)
- UnixBench is running as workload
- Single bit errors are injected using a driver-based kernel module
- They are automatically injected into pregenerated targets
- Error activation, propagation and crash latency are monitored
The Setup

- This error injection campaign uses the software framework NFTAPE (which is built for fault/error injection experiments)
- UnixBench is running as workload
- Single bit errors are injected using a driver-based kernel module
- They are automatically injected into pregenerated targets
- Error activation, propagation and crash latency are monitored
- Data are sent over Ethernet to a remote system
The Setup

- This error injection campaign uses the software framework NFTAPE (which is built for fault/error injection experiments)
- UnixBench is running as workload
- Single bit errors are injected using a driver-based kernel module
- They are automatically injected into pregenerated targets
- Error activation, propagation and crash latency are monitored
- Data are sent over Ethernet to a remote system
- A watchdog card is used to detect crashes and to reboot the system
Target Generation

The generated targets are for the following:

- Code injection: Selected, often used kernel functions
Target Generation

The generated targets are for the following:

- Code injection: Selected, often used kernel functions
- Stack injection: Randomly chosen kernel process
Target Generation

The generated targets are for the following:

- Code injection: Selected, often used kernel functions
- Stack injection: Randomly chosen kernel process
- System register injection
Target Generation

The generated targets are for the following:

- Code injection: Selected, often used kernel functions
- Stack injection: Randomly chosen kernel process
- System register injection
- Data injection: Randomly chosen kernel data
For error injection breakpoints are set using the CPU’s debugging registers.
Error Injection

For error injection breakpoints are set using the CPU’s debugging registers

- In the case of code injection the error is injected *when* the pregenerated instruction breakpoint is reached
For error injection breakpoints are set using the CPU’s debugging registers

- In the case of code injection the error is injected \textit{when} the pregenerated instruction breakpoint is reached
- For data and stack injection the error must be set \textit{before} a breakpoint is reached
Error Injection

For error injection breakpoints are set using the CPU’s debugging registers

- In the case of code injection the error is injected *when* the pregenerated instruction breakpoint is reached
- For data and stack injection the error must be set *before* a breakpoint is reached
- If a breakpoint is reached, the error is called *activated* and performance registers are used to determine the crash latency
For error injection breakpoints are set using the CPU’s debugging registers

- In the case of code injection the error is injected \textit{when} the pregenerated instruction breakpoint is reached
- For data and stack injection the error must be set \textit{before} a breakpoint is reached
- If a breakpoint is reached, the error is called \textit{activated} and performance registers are used to determine the crash latency
- It is impossible to use breakpoints for registers, so activation can not be watched for system register injection
Overview

**Figure 2: Automated Process of Injecting Errors**

- **Setup**
- **Categorization**

**Categorization**

- **Setup**
- **Overview**

**Overview**

- **Introduction**
- **Error Injection Campaign**
- **Experimental Results**
- **Conclusion**
- **Appendix**

**SS 2005**

**Error Sensitivity of the Linux Kernel**

Slide 10
Errors are categorized into the following categories (whether and how they manifest):

<table>
<thead>
<tr>
<th>Outcome Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activated</td>
<td>The corrupted data/instruction is used/executed</td>
</tr>
<tr>
<td>Not Manifested</td>
<td>The corrupted instruction/data is executed/used but it does not cause a visible abnormal impact on the system</td>
</tr>
<tr>
<td>Fail Silence Violation</td>
<td>Incorrect data/response is allowed to propagate out</td>
</tr>
<tr>
<td>Crash</td>
<td>OS stops working, enhanced crash handler dump failure data for off-line analysis</td>
</tr>
<tr>
<td>Hang</td>
<td>System resources are exhausted, system becomes non operational</td>
</tr>
</tbody>
</table>
When the P4 crashes the category is refined into one of the following:

<table>
<thead>
<tr>
<th>Crash Category (P4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL Pointer</td>
<td>Unable to handle kernel NULL pointer de-reference.</td>
</tr>
<tr>
<td>Bad Paging</td>
<td>A page fault: the kernel tries to access some other bad page except NULL pointer.</td>
</tr>
<tr>
<td>Invalid Instruction</td>
<td>An illegal instruction that is not defined in the instruction set is executed.</td>
</tr>
<tr>
<td>General Protection Fault</td>
<td>Exceeding segment limit, writing to a read-only code or data segment, loading a selector with a system descriptor, reading an execution-only code segment.</td>
</tr>
<tr>
<td>Kernel Panic</td>
<td>Operating system detects an error.</td>
</tr>
<tr>
<td>Invalid TSS (Task State Segment)</td>
<td>The selector, code segment, or stack segment is outside the limit, or stack is not writeable.</td>
</tr>
<tr>
<td>Divide Error</td>
<td>Math error.</td>
</tr>
<tr>
<td>Bounds Trap</td>
<td>Bounds checking error.</td>
</tr>
</tbody>
</table>
Errors of the G4 are also refined:

<table>
<thead>
<tr>
<th>Crash Category (G4)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad Area</td>
<td>A page fault: the kernel tries to access a bad page including NULL pointer, or bad memory access.</td>
</tr>
<tr>
<td>Illegal Instruction</td>
<td>An invalid instruction that is not defined in the instruction set is executed.</td>
</tr>
<tr>
<td>Stack Overflow</td>
<td>Stack pointer of a kernel process is out of range.</td>
</tr>
<tr>
<td>Machine Check</td>
<td>An error is detected on the processor-local bus including instruction machine-check errors and data machine-check errors.</td>
</tr>
<tr>
<td>Alignment</td>
<td>Load/Store or other specific instructions' operands are not word-aligned.</td>
</tr>
<tr>
<td>Panic</td>
<td>Operating system detects an error.</td>
</tr>
<tr>
<td>Bus Error</td>
<td>Protection fault.</td>
</tr>
<tr>
<td>Bad Trap</td>
<td>Unknown exception.</td>
</tr>
</tbody>
</table>
Result Overview [1]

<table>
<thead>
<tr>
<th>Overview</th>
<th>Stack Injection</th>
<th>System Register Injection</th>
<th>Code Injection</th>
<th>Data Injection</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution</td>
<td>Stack</td>
<td>System Registers</td>
<td>Data</td>
<td>Code</td>
<td>Intel Pentium 4 Campaign</td>
</tr>
<tr>
<td>Error activation</td>
<td>2973(29.3%)</td>
<td>3459(89.5%)</td>
<td>226(0.5%)</td>
<td>982(54.9%)</td>
<td>2973(29.3%)</td>
</tr>
<tr>
<td>Failure distribution on P4</td>
<td>1305(43.9%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>13(1.3%)</td>
<td>1305(43.9%)</td>
</tr>
<tr>
<td>Processor</td>
<td>1136(38.2%)</td>
<td>305(7.9%)</td>
<td>96(42.5%)</td>
<td>455(46.3%)</td>
<td>1136(38.2%)</td>
</tr>
<tr>
<td>Hang/Unknown Crash on P4</td>
<td>532(17.9%)</td>
<td>102(2.6%)</td>
<td>53(23.4%)</td>
<td>206(21.0%)</td>
<td>532(17.9%)</td>
</tr>
</tbody>
</table>

Error activation and failure distribution on P4 processor

<table>
<thead>
<tr>
<th>Overview</th>
<th>Stack</th>
<th>System Registers</th>
<th>Code</th>
<th>Data</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution</td>
<td>3017</td>
<td>3967</td>
<td>2188</td>
<td>46000</td>
<td>6179</td>
</tr>
<tr>
<td>Error activation</td>
<td>1203(39.9%)</td>
<td>117(0.3%)</td>
<td>1415(46.7%)</td>
<td>704(1.5%)</td>
<td>1203(39.9%)</td>
</tr>
<tr>
<td>Failure distribution on G4</td>
<td>949(78.9%)</td>
<td>3774(95.1%)</td>
<td>580(41.0%)</td>
<td>551(78.3%)</td>
<td>949(78.9%)</td>
</tr>
<tr>
<td>Processor</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>33(2.3%)</td>
<td>7(1.0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Hang/Unknown Crash on G4</td>
<td>172(14.3%)</td>
<td>69(1.7%)</td>
<td>576(40.7%)</td>
<td>55(7.8%)</td>
<td>172(14.3%)</td>
</tr>
</tbody>
</table>

Error activation and failure distribution on G4 processor
## Result Overview [1]

### Error activation and failure distribution on P4 processor

<table>
<thead>
<tr>
<th>Intel Pentium 4 Campaign</th>
<th>Injected</th>
<th>Error Activated</th>
<th>Activated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Not Manifested</td>
</tr>
<tr>
<td>Stack</td>
<td>10143</td>
<td>2973(29.3%)</td>
<td>1305(43.9%)</td>
</tr>
<tr>
<td>System Registers</td>
<td>3866</td>
<td>N/A</td>
<td>3459(89.5%)</td>
</tr>
<tr>
<td>Data</td>
<td>46000</td>
<td>226(0.5%)</td>
<td>77(34.1%)</td>
</tr>
<tr>
<td>Code</td>
<td>1790</td>
<td>982(54.9%)</td>
<td>308(31.4%)</td>
</tr>
<tr>
<td>Total</td>
<td>61799</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Error activation and failure distribution on G4 processor

<table>
<thead>
<tr>
<th>Motorola PPC G4 Campaign</th>
<th>Injected</th>
<th>Error Activated</th>
<th>Activated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>3017</td>
<td>1203(39.9%)</td>
<td>949(78.9%)</td>
</tr>
<tr>
<td>System Register</td>
<td>3967</td>
<td>N/A</td>
<td>3774(95.1%)</td>
</tr>
<tr>
<td>Data</td>
<td>46000</td>
<td>704(1.5%)</td>
<td>551(78.3%)</td>
</tr>
<tr>
<td>Code</td>
<td>2188</td>
<td>1415(64.7%)</td>
<td>580(41.0%)</td>
</tr>
<tr>
<td>Total</td>
<td>55172</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The activation rates on both systems don’t differ much
Result Overview [1]

<table>
<thead>
<tr>
<th>Intel Pentium 4 Campaign</th>
<th>Injected</th>
<th>Error Activated</th>
<th>Activated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Not Manifested</td>
</tr>
<tr>
<td>Stack</td>
<td>10143</td>
<td>2973(29.3%)</td>
<td>1305(43.9%)</td>
</tr>
<tr>
<td>System Registers</td>
<td>3866</td>
<td>N/A</td>
<td>3459(89.5%)</td>
</tr>
<tr>
<td>Data</td>
<td>46000</td>
<td>226(0.5%)</td>
<td>77(34.1%)</td>
</tr>
<tr>
<td>Code</td>
<td>1790</td>
<td>982(54.9%)</td>
<td>308(31.4%)</td>
</tr>
<tr>
<td>Total</td>
<td>61799</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Error activation and failure distribution on P4 processor

<table>
<thead>
<tr>
<th>Motorola PPC G4 Campaign</th>
<th>Injected</th>
<th>Error Activated</th>
<th>Activated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Not Manifested</td>
</tr>
<tr>
<td>Stack</td>
<td>3017</td>
<td>1203(39.9%)</td>
<td>949(78.9%)</td>
</tr>
<tr>
<td>System Register</td>
<td>3967</td>
<td>N/A</td>
<td>3774(95.1%)</td>
</tr>
<tr>
<td>Data</td>
<td>46000</td>
<td>704(1.5%)</td>
<td>551(78.3%)</td>
</tr>
<tr>
<td>Code</td>
<td>2188</td>
<td>1415(64.7%)</td>
<td>580(41.0%)</td>
</tr>
<tr>
<td>Total</td>
<td>55172</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Error activation and failure distribution on G4 processor

- The activation rates on both system don’t differ much
- But crash rates differ considerably
The G4 has significant lesser manifestation rates for stack and data errors (round 20% for the G4 each against 56% respectively 66% for the Pentium 4)
The G4 has significant lesser manifestation rates for stack and data errors (round 20% for the G4 each against 56% respectively 66% for the Pentium 4)

The reason is that the PowerPC allways operates on 32-bit wide data items (has 32-bit alignment)
The G4 has significant lesser manifestation rates for stack and data errors (round 20% for the G4 each against 56% respectively 66% for the Pentium 4)

The reason is that the PowerPC allways operates on 32-bit wide data items (has 32-bit alignment)

Many bits that are flipped in the error injection campaign are unused and so the errors does not manifest
Stack injection

Crash Cause for Kernel Stack Injection

Stack injection

Crash Cause of Pentium Stack
(Total 1136)

- General Protection Fault: 5.5%
- Invalid Instruction: 15.9%
- NULL Pointer: 31.5%
- Kernel Panic: 0.4%
- Invalid TSS: 1.0%
- Divide Error: 0.2%
- Bad Paging: 45.4%

Crash Cause of PPC Stack
(Total 172)

- Bad Area: 53.5%
- Stack Overflow: 41.9%
- Illegal Instruction: 2.9%
- Alignment: 1.2%
- Machine Check: 0.8%

Error Injection Campaign
System Register Injection
Code Injection
Data Injection

Overview
Introduction
Experimental Results
Conclusion
Appendix

SS 2005 Error Sensitivity of the Linux Kernel Slide 16
Stack injection

Crash Cause for Kernel Stack Injection

- Stackoverflows seem not to appear on the P4 but indeed they do. They are not listed because they are not handled by any exception. Stackoverflows are propagated and are listed as other errors e.g. bad paging
An error leads to a wrong Stackpointer

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Corrupted Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm/page_alloc: <code>&lt;free_pages_ok&gt;</code>: ...</td>
<td>mm/page_alloc: <code>&lt;free_pages_ok&gt;</code>: ...</td>
</tr>
<tr>
<td>c013ec55: 8d 65 f4 lea 0xffffffff(%ebp),%esp</td>
<td>c013ec65: 8d 64 f4 5b lea 0x5b(%esp,%esi,8)),%esp</td>
</tr>
<tr>
<td>c013ec68: 5b pop %ebx</td>
<td>c013ec69: 5e pop %esi</td>
</tr>
<tr>
<td>c013ec69: 5e pop %esi</td>
<td>c013ec6a: 5f pop %edi</td>
</tr>
<tr>
<td>c013ec6a: 5f pop %edi</td>
<td>c013ec6b: 5d pop %ebp</td>
</tr>
<tr>
<td>c013ec6b: 5d pop %ebp</td>
<td>c013ec6c: c3 ret</td>
</tr>
</tbody>
</table>

**Return Address (in the stack) pattern:**

c0119eb2 c0107784 c010799a c0108067 c0110eb2 c0107784 c010799a c0108067 c0119eb2 c0107784 c010799a c0108067 c0110eb2 c0107784 c010799a c0108067 c0119eb2 c0107784 c010799a c0108067 c0110eb2 c0107784 c010799a c0108067 c0119eb2 ...  

**Unable to handle kernel paging request at virtual address 170fc2a5**

```
%eax=0x170fc2a5  
Crash Latency=13116444
```
An error leads to a wrong Stackpointer

Stackpointer exceeds kernel-stack bounds (*stackoverflow*)

Example

An error leads to a wrong Stackpointer

Stackpointer exceeds kernel-stack bounds (*stackoverflow*)
Example

- An error leads to a wrong Stackpointer
- Stackpointer exceeds kernel-stack bounds (*stackoverflow*)
- But an *bad paging* exception is raised
An error leads to a wrong Stackpointer

Stackpointer exceeds kernel-stack bounds (stackoverflow)

But an bad paging exception is raised

The system crashes after 13M cycles (G4 crashes by stackoverflow within 3k cycles)
Crash Cause for System Register Injection

Both systems have a low crash rate in this scenario (P4: 10.5%, G4: 4.9%). 99 (G4) respectively 20 (P4) system registers are corrupted but only 15 respectively 7 lead to crashes.
System Register Injection [2]

- General protection errors are e.g. caused by manipulating the control register cr0 so that protected mode operation are disabled.
System Register Injection [2]

- General protection errors are e.g. caused by manipulating the control register cr0 so that protected mode operation are disabled
- Invalid instructions are caused by altering the instruction pointer (P4) or e.g special purpose register 274 (G4) (used by stack switch during exception)
System Register Injection [2]

- General protection errors are e.g. caused by manipulating the control register cr0 so that protected mode operation are disabled.
- Invalid instructions are caused by altering the instruction pointer (P4) or e.g special purpose register 274 (G4) (used by stack switch during exception).
- Invalid task state segment errors are raised after altering the nested task bit stored in the EFLAG register. This leads to return to an invalid task after an interrupt.
System Register Injection [2]

- General protection errors are e.g. caused by manipulating the control register cr0 so that protected mode operation are disabled.
- Invalid instructions are caused by altering the instruction pointer (P4) or e.g. special purpose register 274 (G4) (used by stack switch during exception).
- Invalid task state segment errors are raised after altering the nested task bit stored in the EFLAG register. This leads to return to an invalid task after an interrupt.
- Machine Check exceptions are raised after enabling/disabling address translation in the machine state register.
Most errors result in bad memory access e.g. by altering a parameter of a memory access instruction. In the case of the P4 an instruction can be transformed into a sequence of other valid but incorrect instructions, so more memory-access and less invalid instruction errors are observed.
The variable instruction length of the P4 makes it possible to alter a sequence of instruction into an completely other sequence by a single flipped bit:

![Diagram showing variable instruction length example](image)

**Figure 14: Bit Error Causing Change in Instruction Group (P4)**
Data Injection

- Crash Cause for Kernel Data Injection
  - Activation rate is very low on both systems (0.5% / 1.5%)
Data Injection

Crash Cause for Kernel Data Injection

- Activation rate is very low on both systems (0.5% / 1.5%)
- The reason is the sparse usage of this data
Crash Cause for Kernel Data Injection

- Activation rate is very low on both systems (0.5% / 1.5%)
  - The reason is the sparse usage of this data
  - But not activated errors may persist and could be activated later
Data Injection

Crash Cause for Kernel Data Injection

- Activation rate is very low on both systems (0.5% / 1.5%)
  - The reason is the sparse usage of this data
  - But not activated errors may persist and could be activated later
- Illegal instruction exception are raised also for other exception by the kernel
We have seen:

- More wasteful data and memory management (fixed 32 bit alignment) of the G4 leads to much more fault tolerance.
- Variable instruction length of the P4 leads to a poor diagnosability.
- The raised exception can be misleading.
- Even a single wrong bit can have disastrous consequences.
Summary

We have seen:

- More wasteful data and memory management (fixed 32 bit alignment) of the G4 leads to much more fault tolerance
We have seen:

- More wasteful data and memory management (fixed 32 bit alignment) of the G4 leads to much more fault tolerance.
- Variable instruction length of the P4 leads to a poor diagnosability.
Summary

We have seen:

- More wasteful data and memory management (fixed 32 bit alignment) of the G4 leads to much more fault tolerance.
- Variable instruction length of the P4 leads to a poor diagnosability.
- The raised exception can be misleading.
We have seen:

- More wasteful data and memory management (fixed 32 bit alignment) of the G4 leads to much more fault tolerance
- Variable instruction length of the P4 leads to a poor diagnosability
- The raised exception can be misleading
- Even a single wrong bit can have disastrous consequences
Assessment

- Many errors (over 100k) injected, but:
  
  - Three times more stack errors on the P4 than on the G4
  - There are many more registers on the G4 but only little more errors are injected
  - Different implementations of the Linux kernel and compiler makes it hard/impossible to compare/assess pure hardware differences
  - Data errors that are overwritten (and not read) are marked activated
  
  But nevertheless G4's better fault-tolerance comes out clear
Many errors (over 100k) injected, but:

- Three times more stack errors on the P4 than on the G4 are injected.
Assessment

- Many errors (over 100k) injected, but:
- Three times more stack errors on the P4 than on the G4 are injected
- There are many more registers on the G4 but only little more errors are injected
Many errors (over 100k) injected, but:

- Three times more stack errors on the P4 than on the G4 are injected
- There are many more registers on the G4 but only little more errors are injected
- Different implementations of the Linux kernel and compiler makes it hard/impossible to compare/asses pure hardware differences
Assessment

- Many errors (over 100k) injected, but:
  - Three times more stack errors on the P4 than on the G4 are injected
  - There are many more registers on the G4 but only little more errors are injected
  - Different implementations of the Linux kernel and compiler makes it hard/impossible to compare/asses pure hardware differences
  - Data errors that are overwritten (and not read) are marked activated
Assessment

- Many errors (over 100k) injected, but:
- Three times more stack errors on the P4 than on the G4 are injected.
- There are many more registers on the G4 but only little more errors are injected.
- Different implementations of the Linux kernel and compiler makes it hard/impossible to compare/asses pure hardware differences.
- Data errors that are overwritten (and not read) are marked activated.
- But nether the less G4’s better fault-tolerance comes out clear.
(1) Error at 0x0375bc5 changes 4E to 0E

C code to use the data:

```c
static inline void spin_unlock(spinlock_t *lock) {
    #if SPINLOCK_DEBUG
        if ((lock->magic != SPINLOCK_MAGIC) 
            BUG());
        if (!spin_is_locked(lock)) 
            BUG();
    #endif
    asm __volatile __
        " spin_unlock_string
    
}
```

Pentium 4 kernel Data Section:

```c
c0375bc0 <kernel_flag_cacheline>:
c0375bc0:  01 00
1

c0375bc2:  00 00
c0375bc4:  ad
2

c0375bc5:  4e
3

c0375bc6:  ad
c0375bc7:  de 00
```

Example Data

- Cmpl generates wrong flags because of bad data

- Illegal Instruction

- Test %eax,%eax
- Jns c0158538 <sys_ioctl+0x2d8>
- Push $0x42
- Call c015efb0 __out_of_line_bug
- Dec %eax
- Test %eax,%eax
- Mov %eax,0xic(%edx)
- Jns c015856e <sys_ioctl+0x30e>
- Cmpl
- Je c0158554 <sys_ioctl+0x2f4>
- Ude2a

SS 2005 Error Sensitivity of the Linux Kernel Slide 25
Example [Data]

C code to use the data:
```c
static inline void spin_unlock(spinlock_t *lock) {
    #if SPINLOCK_DEBUG
        if ((lock)->magic != SPINLOCK_MAGIC) {
            BUG();
        }
    #endif
    asm volatile ("spin_unlock_string")
}
```

Pentium 4 kernel Data Section:
```c
0375bc0 <kernel_flag_cache_line>: 01 00
0375bc2: 00 00
0375bc4: ad
0375bc5: 0e
0375bc6: ad
0375bc7: 00 00
```

- (1) Error at 0x0375bc5 changes 4E to 0E
- (2) Cmpl detects an error
- cmpl generates wrong flags because of bad data

```
<sys_ioctl>:
  Corresponding Assembly:
  c015852d: 85 c0
  c015852f: 79 07
  c0158531: 6a 42
  c0158533: 68 78 6a ff
  c0158538: 49
  c0158539: 85 c0
  c015853b: 89 42 1c
  c015853c: 79 2e
  c0158540: 81 3d c4 5b 37 c0 ad
  c0158547: 4e ad ds
  c015854a: 74 08
  c015854c: 0f 0b
```

- test %eax, %eax
- jns c0158538 <sys_ioctl+0x2d8>
- push $0x42
- call c011efb0 __out_of_line_bug
- dec %eax
- test %eax, %eax
- mov %eax, 0x1c(%edx)
- jns c015856e <sys_ioctl+0x30e>
- cmpl
- je 0xdead4ead, 0xc0375bc4
- ud2a
- Illegal Instruction
Example [Data]

C code to use the data:

```c
static inline void spin_unlock(spinlock_t *lock) {
    #if SPINLOCK_DEBUG
        if ((lock)->magic != SPINLOCK_MAGIC) BUG();
    #endif
        asm __volatile__ ({
            spin_unlock_string
        });
}
```

Pentium 4 kernel Data Section:

```c
0375bc0: <kernel_flag_cacheline>
0375bc0: 01 00
0375bc2: 00 00
0375bc4: ad 4e
0375bc6: ad 00
0375bc8: de 00
```

- (1) Error at 0x0375bc5 changes 4E to 0E
- (2) Cmpl detects an error
- (3) Illegal instruction exception is raised
Example [Stack P4]

- (1) tsk→state is at location 0xffffffff0(%ebp)

```c
#define TASK_STOPPED 8
int kupdate(void *startup)
{
    ... for (; ;) {
        /* update interval */
        if (interval) {
            tsk->state = TASK_INTERRUPTIBLE;
            schedule_timeout(interval);
        } else {
            tsk->state = TASK_STOPPED;
            schedule(); /* wait for SIGCONT */
        }
        /* check for sigstop */
        if (signal_pending(tsk)) {
            int stopped = 0; ...
        }
        sync_oldbuffers();
        run_task queue(&tq_disk);
    }
}
Crash latency is 12864 cycles
```

Machine Code and Assembly

1. Get address of data structure tsk from stack at location of 0xffffffff0(%ebp). But one bit flip in stack alters the address saved in EAX.
2. Using wrong EAX to set “tsk→state.”
3. Crash because EDX coming from stack is 0.
Example[Stack P4]

(1) tsk→state is at location 0xffffffff0(%ebp)

(2) Error is injected and tsk→state is no more TASK-STOPPED

---

```
#define TASK_STOPPED 8
int kupdate(void *startup) {
    ...
    for (;;) {
        /* update interval */
        tsk->state = TASK_INTERRUPTIBLE;
        schedule_timeout(interval);
    } else {
        tsk->state = TASK_STOPPED;
        schedule(); /* wait for SIGCONT */
    }
    /* check for sigstop */
    if (signal_pending(tsk)) {
        int stopped = 0;
    }
    sync_old_buffers();
    run_task_queue(&q_disk);
}
```

---

```
1
get address of data structure tsk from stack at location of Oxffffffe0(%ebp).
But one bit flip in stack alters the address saved in EAX.

2
Using wrong EAX to set “tsk->state.”

3
Crash because EDX coming from stack now is 0.
```

---

Machine Code  Assembly
8b 6e 00 mov 0xffffffff0(%ebp),%eax
8b 06 00 movl 0xffffffff0(%eax),%eax
8b 55 00 mov 0x0(%ebp),%edx
8b 4a 08 mov 0x8(%edx),%ecx
85 c9 test %ecx,%ecx
ff e0 ca 00 je c014eac6<update+0xff6>

---

Crash latency is 12864 cycles
Example[Stack P4]

1. `tsk→state` is at location `0xffffffff(%ebp)`
2. Error is injected and `tsk→state` is no more `TASK-STOPPED`
3. Incorrect parameters are passed to the scheduler

```
#define TASK_STOPPED 0
int kupdate(void **startup)
{
    ...
    /* update interval */
    for (;;) {
        // on-fly initialization
        if (interval) {
            tsk->state = TASK_INTERRUPTIBLE;
            schedule_timeout(interval);
        } else {
            tsk->state = TASK_STOPPED;
            schedule(); /* wait for SIGCONT */
        }
        /* check for sigstop */
        if (signal_pending(tsk)) {
            int stopped = 0; ... 
        }
        sync_old_buffers();
        run_task_queue(&tq_disk);
    }
}
```

```
Crash latency is 12864 cycles
```

```
Example
```
```
Machine Code Assembly
```
Example[Stack P4]

(1) tsk→state is at location 0xffffffff0(%ebp)

(2) Error is injected and tsk→state is no more TASK-STOPPED

Incorrect parameters are passed to the scheduler

When schedule() returns tsk→state is 0 (stored in edx)
Example [Stack P4]

(1) tsks\rightarrow state is at location 0xffffffff(\%ebp)

(2) Error is injected and tsks\rightarrow state is no more TASK-STOPPED

Incorrect parameters are passed to the scheduler

When schedule() returns tsks\rightarrow state is 0 (stored in edx)

(3) mov crashes and raises a null pointer exception
### Example [Stack G4]

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>c008d794:</td>
<td>4b f8 b8 d9</td>
<td>bl c001906c &lt;interruptible_sleep_on&gt;</td>
</tr>
<tr>
<td>c008d798:</td>
<td>81 7f 00 28</td>
<td>lwz r11, 40(r31)</td>
</tr>
</tbody>
</table>

C code: `transaction = journal->j_running_transaction`

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>c008d79c:</td>
<td>2c 0b 00 00</td>
<td>cmpwi r11, 0</td>
</tr>
<tr>
<td>c008d7a0:</td>
<td>41 82 ff c4</td>
<td>beq+ c008d764 &lt;kjournald+0x178&gt;</td>
</tr>
<tr>
<td>c008d7a4:</td>
<td>80 1b 83 68</td>
<td>lwz r0, -31896(r27)</td>
</tr>
<tr>
<td>c008d7a8:</td>
<td>81 2b 00 4c</td>
<td>lwz r9, 76(r11)</td>
</tr>
</tbody>
</table>

C code: `transaction->t_expires`

- r31 is used as temporary stack pointer

Invalid address 1 erroneously is loaded to r11.

Wrong value in stack pointed by r31 is loaded to r11.

Crash of "kernel access of bad area"
Example [Stack G4]

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>c008d794:</td>
<td>4b f8 b8 d9</td>
<td>bl c001906c &lt;interruptible_sleep_on&gt;</td>
</tr>
<tr>
<td>c008d798:</td>
<td>81 7f 00 28</td>
<td>lwz r11, 40(r31)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C code: transaction = journal-&gt;j_running_transaction</td>
</tr>
<tr>
<td>c008d79c:</td>
<td>2c 0b 00 00</td>
<td>cmpwi r11, 0</td>
</tr>
<tr>
<td>c008d7a0:</td>
<td>41 82 ff c4</td>
<td>beq+ c008d764 &lt;kjournald+0x178&gt;</td>
</tr>
<tr>
<td>c008d7a4:</td>
<td>80 1b 83 68</td>
<td>lwz r0, -31896(r27)</td>
</tr>
<tr>
<td>c008d7a8:</td>
<td>81 2b 00 4c</td>
<td>lwz r9, 76(r11)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C code: transaction-&gt;t_expires</td>
</tr>
</tbody>
</table>

- r31 is used as temporary stack pointer
- Invalid address 1 erroneously is loaded to r11

Wrong value in stack pointed by r31 is loaded to r11

Crash of “kernel access of bad area”
Example [Stack G4]

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>c008d794:</td>
<td>4b f8 b8 d9</td>
<td>bl c001906c &lt;interruptible_sleep_on&gt;</td>
</tr>
<tr>
<td>c008d798:</td>
<td>81 7f 00 28</td>
<td>lwz r11, 40(r31)</td>
</tr>
</tbody>
</table>

C code: `transaction = journal->j_running_transaction`

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>c008d79c:</td>
<td>2c 0b 00 00</td>
<td>cmpwi r11,0</td>
</tr>
<tr>
<td>c008d7a0:</td>
<td>41 82 ff c4</td>
<td>beq+ c008d764 &lt;kjournald+0x178&gt;</td>
</tr>
<tr>
<td>c008d7a4:</td>
<td>80 1b 83 68</td>
<td>lwz r0,-31896(r27)</td>
</tr>
<tr>
<td>c008d7a8:</td>
<td>81 2b 00 4c</td>
<td>lwz r9,76(r11)</td>
</tr>
</tbody>
</table>

C code: `transaction->t_expires`

- r31 is used as temporary stack pointer
- Invalid address 1 erroneously is loaded to r11
- The kernel crashes while trying to access to 0x4d (=76(1))

Randel E.Bryant, David O’Hallaron ”Computer Systems - a programmer’s perspective”

www.wikipedia.org