OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
OUTLINE

- Introduction
- Research Overview
- Background and Prior Work
- Hardware and Software Threat Models
- Hardware Trojan Attack and Error Detection in Medical Devices
- Run-time Code Integrity in Embedded and Medical Devices
- Impact on Embedded Systems Security
- Conclusions
- List of Publications
- References
Recent attacks such as WannaCry were major wakeup calls for the medical industry [1]

Researchers at the University of South Alabama hack a pacemaker; kill a man(nequin) [2]

The wireless medical device was implanted in one of the most advanced wireless patient simulators on the market

The simulator had a pacemaker so the researchers could speed the heart rate up and could slow it down

A figure showing the patient simulator (left) along with the attacked software (right)
RECENT THREATS, ATTACKS, AND CONCERNS

• In 2008, Israeli jets bombed a suspected nuclear installation in Syria [3]
  • The success of the strike was partially due to the failure of a Syrian radar
  • Many posts speculated that the microprocessors used by the radar were fabricated with a hidden “backdoor”

• In 2010, the U.S. Navy discovered fake microchips with a “back door” which could have disarmed their missiles [4]

• In 2016, researchers at the University of Michigan created an analog HT which is nearly invisible and is based on capacitor charging to trigger an attack on processors and allow for higher system-level attacks [5]

• In 2018, a Bloomberg report claimed that the supply chain of a major server motherboards manufacturer has been compromised by the insertion of a backdoor chip [6]
MEDICAL DEVICE SECURITY – TAKING ACTION

• In December 2016, the FDA released a report about medical cybersecurity [7]
  • Evaluation of risk of patient harm
  • Use a matrix with combinations of “exploitability” and “severity of patient harm”

• In February 2016, the IEEE Center for Secure Design released a report spotlighting six security red flags for the wearable industry [8]
  • Compromising device integrity
  • Falsifying user’s own health data
  • Falsifying another user’s health data
  • Stealing a user’s health data
  • Abusing health data that are not intentionally shared
  • Denial of service

[Diagram: Evaluating risk of patient’s harm]
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References

© Georgia Institute of Technology, 2018
RESEARCH OVERVIEW

Targeted attacks and errors

Medical Device Hardware

Gamma-rays causing unintentional errors

Malicious code modifications

Medical Device Hardware

Designed and implemented architecture

Chip 1 (trusted)
- Sensors
- MISR
- Analog Signatures

Chip 2 (untrusted)
- ASIC
- Microprocessor
- Memory Signatures
- Checking Logic

© Georgia Institute of Technology, 2018
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
HARDWARE TROJAN (HT) ATTACKS

- Disaggregation of the chip manufacturing process
  - HDL & Design For Test (DFT) $\rightarrow$ Synthesis $\rightarrow$ Placement & routing $\rightarrow$ Pre-fabrication testing $\rightarrow$ Fabrication $\rightarrow$ Post-fabrication testing

- Hardware Trojan (HT) attacks [10]
  - Malicious modifications of hardware during design or fabrication
  - Result in undesired functional behavior or provide backdoors and leak information

<table>
<thead>
<tr>
<th>Physical characteristics</th>
<th>Structure</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution</td>
<td>Layout same</td>
<td>Parametric</td>
</tr>
<tr>
<td>Size</td>
<td>Layout change</td>
<td>Functional</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activation characteristics</th>
<th>Internally activated</th>
<th>Externally activated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Conditional Logic</td>
<td>Antenna</td>
</tr>
<tr>
<td></td>
<td>Sensor</td>
<td>Sensor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action characteristics</th>
<th>Modify function</th>
<th>Transmit information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Change</td>
<td>Modify specification</td>
</tr>
<tr>
<td></td>
<td>Disable</td>
<td></td>
</tr>
</tbody>
</table>

Chip manufacturing process of a digital integrated circuit [9]
HT DETECTION METHODS

• Side-channel analysis [10-13]
  • Analyze information gained from the physical implementation of an architecture and compare to data generated by the normal behavior of a chip
  • Lack the ability to detect HTs that are small in size due to the minor effects that the HT might cause in terms of power and timing variations

• Hardware Trojan triggering [14-16]
  • Extensively testing the design prior to deployment and use (i.e., right after chip fabrication)
  • Cleverly inserted HTs may not be easily triggered
  • Offline full-functionality test for each fabricated chip might turn out to be inefficient and time-consuming

• Functional Verification [10, 15, 17, 18]
  • Checking the functionality of the hardware by monitoring the output and checking for expected behavior
  • Lacks the ability to detect attacks on input values as they initially appear on a Cipher chip
CODE INTEGRITY ATTACKS

- A division of software vulnerabilities exists in the Common Weakness Enumeration (CWE) and Common Vulnerabilities and Exposures (CVE) standards developed by the MITRE Corporation [19,20]

  - Code Injection or Code Execution
    - Malicious code insertion or modification
    - Different classes of attacks including ones that target executing data or user inputs as commands by an application
    - Accounts for 29.6% of all reported vulnerabilities between 1999 and 2017 [21]

  - Buffer Errors and/or Buffer Overflow
    - Significant change in the structure of a process space
    - Exposing code that tries to directly access memory locations outside the bounds of a memory buffer
    - Accounts for 15.6% of all reported vulnerabilities between 1999 and 2017 [21]

  - Resource Management
    - Helps in protecting the memory, CPU and communication interfaces from malevolent code execution
    - Challenges of preventing false positives
Traditionally, code integrity is provided by the Linux Integrity Measurement Unit (IMU) [22].

IMUs typically verify the integrity of executable content in an OS at load-time by inspecting the integrity of executable files before loading them.

Static analysis of code to find possible security vulnerabilities [23-27].

Infeasibility of discovering all vulnerabilities in a given program.

Not able to detect run-time attacks.
CODE INTEGRITY DETECTION METHODS – RUN-TIME CODE INSPECTION

• Dynamic integrity measurements through extensions to the IMU [28-30]
  • Primarily implemented in software and are thus vulnerable
  • Would require code augmentation and recompilation thus incurring significant performance overhead [31-33]

• Intel SGX [34] and ARM TrustZone [35] have been developed to secure and protect code integrity by
  • Isolating user code and allocating private regions of memory
  • Implementing tight access guards between the secure and the insecure compartments (e.g. SGX enclaves)

• Other hardware techniques either depend on control flow execution or rely on instruction-based monitoring [36-41]
  • Introduce coding limitations, frequent false positives and performance overhead
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
THREAT MODEL – HARDWARE TROJAN ATTACKS

- We focus on:
  - Extremely small HT logic inserted in the chip fabrication process [42,43], which when triggered, attempts to corrupt functionality
  - Attack on primary input of a chip
  - HT triggers a payload which modifies the input value
  - Data is affected before any encryption or signature generation

<table>
<thead>
<tr>
<th>Physical characteristics</th>
<th>Structure</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution</td>
<td>Layout same</td>
<td>Parametric</td>
</tr>
<tr>
<td>Small Size</td>
<td>Layout change</td>
<td>Functional</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activation characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Externally activated</td>
</tr>
<tr>
<td>Internally activated</td>
</tr>
<tr>
<td>Antenna</td>
</tr>
<tr>
<td>Sensor</td>
</tr>
<tr>
<td>Always on</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmit information</td>
</tr>
<tr>
<td>Modify specification</td>
</tr>
<tr>
<td>Modify function</td>
</tr>
<tr>
<td>Change</td>
</tr>
</tbody>
</table>
THREAT MODEL – CODE INTEGRITY ATTACKS

- Focus on threats after development and deployment
- Software attack that modifies a user process code at run-time
- Variations of *hollow process injection (process hollowing)* malware [44]
  - Examples: BadNews Android malware and Stuxnet [45, 46]
- Attack is performed on the text segment of a process address space
- Attack targets code down to instruction-level modification
- Attack can be performed by exploiting any software technique that attempts to modify memory contents, e.g., buffer overflow, code injection, etc.
- Expand attack vector to cover critical kernel-level modules
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
RESEARCH OVERVIEW

Targeted attacks and errors

Medical Device Hardware

Gamma-rays causing unintentional errors

Medical Device Hardware

Chip 1 (trusted)
- Sensors
- MISR
- Analog Signatures

Chip 2 (untrusted)
- ASIC
- Microprocessor
- Memory Signatures
- Checking Logic

Designed and implemented architecture
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
  • An Example Scenario
  • Architecture Overview
  • Analog-, Digital- and Physiological-based Signature Generation
  • HT Attacks
  • Experimental Results
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
ECG AND BCG – EXAMPLE SCENARIO

• Electrocardiogram (ECG) and Ballistocardiogram (BCG) sensors are normally used to monitor and capture heart pulse electrical and timing events [48]

• Harvested data is typically further processed and analyzed to extract physiological features that help in the diagnosis of health conditions [49]
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
  • An Example Scenario
  • Architecture Overview
  • Analog-, Digital- and Physiological-based Signature Generation
  • HT Attacks
  • Experimental Results
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
**ARCHITECTURE OVERVIEW**

- Combined analog, digital and physiological based signature generation and testing

**Contributions**
- Address HT attacks on input values as they initially appear on a digital chip
- Further complicate the job of the attacker by incorporating multiple types of signatures
- Help distinguish HT attacks and hardware errors from health problems
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• **Hardware Trojan Attack and Error Detection in Medical Devices**
  • An Example Scenario
  • Architecture Overview
  • **Analog-, Digital- and Physiological-based Signature Generation**
  • HT Attacks
  • Experimental Results
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
CHIP 1 (TRUSTED) – ANALOG AND DIGITAL SIGNATURE GENERATION

\[ \sqrt{ECG^2 + BCG HF^2} \]
MULTIPLE INPUT SIGNATURE REGISTER (MISR)

- MISRs are typically used in digital systems test
- For built-in self tests, Built In Logic Block Observer (BILBO) MISRs are used
- We take advantage of the pre-existing BILBO registers in the design and program them to operate in MISR mode
CHIP 2 (UNTRUSTED) – SIGNATURE REGENERATION AND CHECKING

Digital-based Signature Generation and Testing

Analog-based Signature Generation and Testing

Physiological Features-based Signature Generation and Testing

© Georgia Institute of Technology, 2018
ECG DATA AND BCG HEAD-TO-FOOT DATA FEATURES

• Physiological Features
  • Data dependent

• Heart Rate
  • Calculated by finding the elapsed time between two consecutive ECG peaks (R-peaks) [49]

• R-J Interval
  • Defined as the elapsed time between the previous ECG R-peak and the global maximum (J-peak) in the first 400 ms of the BCG head-to-foot signal [49]
Physiological Features-based Signature Generation and Testing

Physiological Features-based Signature Generation and Testing

© Georgia Institute of Technology, 2018
<table>
<thead>
<tr>
<th>Physiological-based signature</th>
<th>MISR-based Signature</th>
<th>Analog-based Signature</th>
<th>Analog-based Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Anomaly</td>
<td>Match</td>
<td>Mismatch</td>
<td>Match</td>
</tr>
<tr>
<td>Correct operation</td>
<td>Possible attack on the analog-based signature generation architecture</td>
<td>Possible HT attack or hardware error in low order bits</td>
<td>Possible HT attack or hardware error in low order bits</td>
</tr>
<tr>
<td>Potential health problem (person is recommended to seek medical help)</td>
<td>Potential health problem or possible attack on the analog signature generation architecture</td>
<td>Possible HT attack or hardware error along with a potentially minor health problem</td>
<td>Hardware Trojan attack or hardware error</td>
</tr>
<tr>
<td>Gray Zone</td>
<td>Match</td>
<td>Mismatch</td>
<td>Match</td>
</tr>
<tr>
<td>Person is asked to seek immediate medical help</td>
<td>Possibility of a serious health problem or attack on the analog signature generation architecture</td>
<td>HT attack or hardware error along with a possibility of a serious health problem</td>
<td>Hardware Trojan attack or hardware error</td>
</tr>
<tr>
<td>Anomaly</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
  • Hardware Trojan Attack and Error Detection in Medical Devices
    • An Example Scenario
    • Architecture Overview
    • Analog-, Digital- and Physiological-based Signature Generation
    • HT Attacks
      • Experimental Results
  • Run-time Code Integrity in Embedded and Medical Devices
  • Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
HT ATTACK TYPES – COORDINATED ATTACKS

© Georgia Institute of Technology, 2018
COORDINATED ATTACK TYPE 3

- HT simultaneously targets data in the design along with the comparator output.

- Comparator testing logic block is inserted to specifically monitor this type of attack.

© Georgia Institute of Technology, 2018
• Periodically, the comparator testing logic intentionally changes signature 2 by asserting the Test Mode signal
• The comparator’s output is read with the expectation of a mismatch if there is no HT attack
• Data processing is stalled for one cycle during the operation
1. Hardware Trojan attacks ECG data

2. Corrupt Ciphertext

3. Corrupt Plaintext

4. Corrupt Signature

5. Alarm Signal goes off!

6. Corrupt Data

7. Corrupt Analog Signature

8. Alarm Signal goes off!

9. ECG Peaks appear at different rates

10. Counter stores abnormal heart rate values

11. R-J interval values are affected

12. R-J interval variations go beyond the normal limit

13. Alarm Signals go off!

© Georgia Institute of Technology, 2018
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• **Hardware Trojan Attack and Error Detection in Medical Devices**
  • An Example Scenario
  • Architecture Overview
  • Analog-, Digital- and Physiological-based Signature Generation
  • HT Attacks
  • Experimental Results
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
EXPERIMENTAL RESULTS

• Experimental Setup
  • ECG and BCG data of six different subjects were captured at 2 KHz and used in our simulation experiments
  • We implemented our design in VHDL code and carried out our simulations using Mentor Graphics ModelSim SE version 10.2b revision 2013.05 for Linux

• Simulation Results
  • HT attacks and hardware errors are caught on average:
    • within 35ms with a worst case of 64ms for MISR signatures
    • within 1 – 2ms for analog signatures
  • Detecting user health problems varies according to the severity of the condition
    • Minor heart problems are caught on average within 300ms with a worst case of approximately 1 sec (equivalent to the time taken for a heartbeat to occur)
    • Severe heart conditions are caught at an average of 100ms with a worst case of 515ms
SYNTHESIS RESULTS

- Our synthesis results were performed using Synopsys Design Compiler version J-2014.09 for Linux and were mapped to the NCSU 45nm Base Kit Library

<table>
<thead>
<tr>
<th>Module</th>
<th>Area (square microns)</th>
<th>Area (kGE)</th>
<th>Current maximum clock speed: 300 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption Cipher (PRESENT [50])</td>
<td>5517</td>
<td>2.939</td>
<td></td>
</tr>
<tr>
<td>Decryption Cipher (PRESENT [50])</td>
<td>5431</td>
<td>2.893</td>
<td></td>
</tr>
<tr>
<td>MISR-based Signatures Generation Overhead</td>
<td>6172</td>
<td>3.288</td>
<td></td>
</tr>
<tr>
<td>MISR-based Signatures Overhead (MISRs shared with digital systems test)</td>
<td>3575</td>
<td>1.904</td>
<td>14%</td>
</tr>
<tr>
<td>Analog-based Signatures Overhead</td>
<td>1839</td>
<td>0.98</td>
<td>7%</td>
</tr>
<tr>
<td>Physiological Feature Extraction Circuitry</td>
<td>3485</td>
<td>1.856</td>
<td></td>
</tr>
<tr>
<td>Physiological Signatures Testing Overhead</td>
<td>954</td>
<td>0.508</td>
<td>&lt; 4%</td>
</tr>
</tbody>
</table>

© Georgia Institute of Technology, 2018
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
RESEARCH OVERVIEW

Targeted attacks and errors

Medical Device Hardware

- Malicious code modifications
- Gamma-rays causing unintentional errors

Designed and implemented architecture

Medical Device Hardware

Chip 1 (trusted)
- Sensors
- MISR
- Analog Signatures

Chip 2 (untrusted)
- ASIC
- Microprocessor
- Memory Signatures
- Checking Logic

© Georgia Institute of Technology, 2018
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
  • Motivation and Example Scenario
  • Methodology
  • Assumptions and Challenges
  • Kernel-level Code Integrity Assessment
  • Experimental Platform and Results
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
RUN-TIME CODE INTEGRITY – MOTIVATION

- OS attempting to run a user-level application
  - Code is loaded from disk to memory
- Malicious activities can try to modify code at run-time while resident in memory
- G. Holmes classified malware into five variants [51]
  - Three of which use a run-time infection method
  - Modify and/or insert malicious code
  - Modifications were made to the in-memory copy of the executable code
- Design and implement a hardware-assisted technique to detect such malicious activities
PORTABLE ECG – AN EXAMPLE SCENARIO

• System processor does the following tasks [52]
  • Operate the user interface
  • Control electrode amplifiers + A/D
  • Control digital signal processing of inputs
  • Test battery, temperature, etc.
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
  • Motivation and Example Scenario
  • Methodology
  • Assumptions and Challenges
  • Kernel-level Code Integrity Assessment
  • Experimental Platform and Results
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
METHODOLOGY – PROCESS INTEGRITY USING HARDWARE SIGNATURES

- Hardware/software codesign at the memory interface
  - Tight coupling between physical memory and a hardware monitor
  - Perform run-time memory probing with kernel help
- Create page-based hashes of running processes
- Run-time hardware hash generation and run-time testing
Compile Time

- Partition processes into process pages (PPs)
- Precompute signatures of PPs and save them in root-of-trust

Process Starts

Kernel notifies hardware monitor of process’ pages location

Run Time

- Dedicated HW recreates signatures of PPs that are paged in and compares them to stored values

Comparison for any PP fails?
- Yes
- No

Context switch?
- Yes
- No

Page fault?
- Yes
- No

Wait for a pre-defined period

- Trap into kernel and issue an alarm
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
  • Motivation and Example Scenario
  • Methodology
  • Assumptions and Challenges
  • Kernel-level Code Integrity Assessment
  • Experimental Platform and Results
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
ASSUMPTIONS

- Measured or secure boot is implemented
  - Processor boots into a known trusted state
- The ability to duplicate memory maps from the OS and processes
- Access to compiler, linker and loader information, specific cache information of the target system memory, etc.
- Kernel processes may have varying degrees of protection from attack, e.g., a very high level of assurance to a medium level
- Hardware is secure and has been provisioned by system integrator
COMPILE TIME – PARSING ELF FILE AND GENERATING GOLDEN HASHES

• Read Executable and Linkable Format (ELF) Header to extract target platform and structure of file
• Program Header Table in ELF
  • Specifies all different segments
• Section Header Table in ELF
  • Specifies all different sections (smallest grouped unit in ELF)
• Extract code segment which needs to be loaded to memory
• Zero out unmapped regions and feed to hash generator
• Store golden hashes of all executable pages in secure memory

ELF File Structure

© Georgia Institute of Technology, 2018
ADDRESSING DYNAMICALLY LINKED LIBRARIES

• Use specific Linux commands
• Extract private headers from ELF file and scan for dynamic library dependencies
  • objdump/readelf
• Search for shared object (.so) files of the corresponding libraries to be linked
• Extract executable page contents from corresponding library ELF files
• Generate hashes and store with application’s native page hashes
• Dynamically resolved addresses are masked by the indirection provided by Linux
  • Procedure Linkage Tables (PLT) and Global Offset Table (GOT) in the ELF file

File Structure of Concatenated Hashes

N1 hashes in application

256-bit hash

Application’s native pages

DLL 1

DLL 2, 3, …
RUN-TIME – ADDRESSING LINUX MEMORY MANAGEMENT

- Access Linux memory mapping (/proc/pid/maps)
- “test” process memory address space
- Code starts at address 0x400000

<table>
<thead>
<tr>
<th>Address</th>
<th>Kbytes</th>
<th>RSS</th>
<th>Dirty</th>
<th>Mode</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000400000</td>
<td>4</td>
<td>4</td>
<td></td>
<td>r-x-</td>
<td>test</td>
</tr>
<tr>
<td>0000000000600000</td>
<td>4</td>
<td>4</td>
<td></td>
<td>r-w-</td>
<td>test</td>
</tr>
<tr>
<td>0000000381ce0000</td>
<td>128</td>
<td>108</td>
<td></td>
<td>r-x-</td>
<td>ld-2.12.so</td>
</tr>
<tr>
<td>0000000381d02000</td>
<td>4</td>
<td>4</td>
<td></td>
<td>r----</td>
<td>ld-2.12.so</td>
</tr>
<tr>
<td>0000000381d021000</td>
<td>4</td>
<td>4</td>
<td></td>
<td>r-w-</td>
<td>ld-2.12.so</td>
</tr>
<tr>
<td>0000000381d022000</td>
<td>4</td>
<td>4</td>
<td></td>
<td>r-w-</td>
<td>[ anon ]</td>
</tr>
<tr>
<td>0000000381d200000</td>
<td>1576</td>
<td>152</td>
<td></td>
<td>r-x-</td>
<td>libc-2.12.so</td>
</tr>
<tr>
<td>0000000381d38a000</td>
<td>2048</td>
<td>0</td>
<td></td>
<td>-----</td>
<td>libc-2.12.so</td>
</tr>
<tr>
<td>0000000381d58a000</td>
<td>16</td>
<td>8</td>
<td></td>
<td>r----</td>
<td>libc-2.12.so</td>
</tr>
<tr>
<td>0000000381d58e000</td>
<td>8</td>
<td>8</td>
<td></td>
<td>r-w-</td>
<td>libc-2.12.so</td>
</tr>
<tr>
<td>0000000381d590000</td>
<td>16</td>
<td>8</td>
<td></td>
<td>r-w-</td>
<td>[ anon ]</td>
</tr>
<tr>
<td>00000007fed2793e000</td>
<td>12</td>
<td>12</td>
<td></td>
<td>r-w-</td>
<td>[ anon ]</td>
</tr>
<tr>
<td>00000007fed27991000</td>
<td>4</td>
<td>4</td>
<td></td>
<td>r-w-</td>
<td>[ anon ]</td>
</tr>
<tr>
<td>00000007ffd896d8000</td>
<td>84</td>
<td>12</td>
<td></td>
<td>r-w-</td>
<td>[ stack ]</td>
</tr>
<tr>
<td>00000007ffd897df000</td>
<td>4</td>
<td>4</td>
<td></td>
<td>r-x-</td>
<td>[ anon ]</td>
</tr>
<tr>
<td>ffffffff600000000</td>
<td>4</td>
<td>0</td>
<td></td>
<td>r-x-</td>
<td>[ anon ]</td>
</tr>
</tbody>
</table>

---

total kB 3920 336 68
RUN-TIME – ADDRESSING LINUX MEMORY MANAGEMENT

- Page virtual address to physical address translation
  (/proc/pid/pagemap)
- Virtual address: 0x400000
- Page Frame Number (PFN): 0x4BDC26
- Physical address: 0x4BDC26000

```
Vaddr: 0x400000, Page_size: 4096, Entry_size: 8
Reading /proc/61885/pagemap at 0x2000
Result: 0x86000000004bdc26
PFN: 0x4bdc26
```
Page-based memory monitoring of a fetched physical address
- Kernel fetches golden hash of the page pointed to by the PFN
- PFN is passed to the hardware monitor along with the page’s hash (signature)
- Hardware monitor
  - Fetches page from memory over AXI bus
  - Generates hash in hardware
  - Compares generated hash to the one passed from Kernel

**PROCESS CODE INTEGRITY AT RUN-TIME**

Kernel Space

User Space

Process 1

Process n

Physical Memory

Hash = 0xF355…256

PFN = 0xBDC26

Page

Kernel Space

User Space

Process 1

Process n

PFN = 0xBDC26

Hash = 0xA211…24F

Hardware Monitor

HW Crypto Engine

Hash Comparator

Control

Result

Trap

© Georgia Institute of Technology, 2018
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
  • Motivation and Example Scenario
  • Methodology
  • Assumptions and Challenges
  • Kernel-level Code Integrity Assessment
  • Experimental Platform and Results
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
ADDRESSING KERNEL SECURITY

- Dedicated memory attached to the hardware
  - Root of trust embedded in FPGA
- If dedicated memory gets filled then allow kernel to store pages
- Perform signature checking on specific modules of the kernel
- Of specific interest are kernel processes that handle page mapping and allocation
- Use the same method to monitor the integrity of these processes’ pages
• Assess the integrity of specific critical modules in the kernel
  • Mainly interested in modules that perform memory management (MMU)
• Extract the code segment of modules that need to be monitored at compile time
• Create hashes of the pages of these modules
• At run-time, use Linux pseudo files such as “/proc/kallsyms” to locate addresses of monitored modules
• Send physical address of these modules to the hardware monitor
  • If KASLR is not enabled, the addresses of these modules typically remain unchanged at run-time and between different reboots of the system (assuming kernel is loaded in the same place in memory at boot time)
  • If KASLR is enabled, the hardware is informed of the kernel’s loaded address at boot-time
• Hardware monitor assesses the integrity of the pages of these kernel modules along with the pages of the monitored application(s)

Timeline with kernel-level integrity assessment

1. System Boots
2. Secure Boot loads and launches the OS
3. Hardware monitor starts assessing integrity of critical kernel modules
4. Wait on user-level application(s) to start
5. Start assessing code integrity of user-level application(s)
6. Alert kernel in case of any page hash violation
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
  • Motivation and Example Scenario
  • Methodology
  • Assumptions and Challenges
  • Kernel-level Code Integrity Assessment
  • Experimental Platform and Results
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
• Use a board from Digilent in the ZedBoard family with
  • ARM Cortex dual-core A9 processor
  • Zynq 7000-series FPGA from Xilinx
• Use an embedded version of Linux provided by Xilinx called PetaLinux
  • Petalinux (linux-xlnx-xilinx-v2017.2)
  • GNU Compiler (arm-xilinx-linux-gnueabi-gcc version 4.9.2)
ARCHITECTURE TARGETING ZEDBOARD

- Golden hashes are generated and stored in Block RAM
- Kernel level driver sets up communication with HW using GP and HP AXI ports
- Wait for the monitored process to start
- Monitor memory maps and send physical addresses to HW
- AXI DMA uses physical address and stores a copy of an executable page in AXI FIFO
- HW reads FIFO, generates hash and compares to golden hash
- Discrepancy is reported to kernel and process is halted
HEART RATE MONITORING APPLICATION

- User-level application
  - Reads ECG data
  - Calculates heart rate by measuring time difference between two consecutive ECG peaks (R-peaks)
  - Displays heart rate to the user

Current heart rate: 81.41 bpm

© Georgia Institute of Technology, 2018
RUN-TIME MEMORY CORRUPTION MALWARE

- Malware attaches itself to the heartrate monitor using the `ptrace` command
- Changes a single instruction to affect heart rate calculation
- Developed two variations of the malware
  - Change `rsb` to `mov`
  - Change `rsb` to `add`

```c
if (oldDataset > ECGThreshold) {
    peakTime = currentTime-1;
    heartrate = (peakTime - oldPeakTime);
    printf("Heart rate: %.2f bpm\n", 60*2000/(double)heartrate);
    oldPeakTime = peakTime;  //set peak time as old peak time
}
```

```
.text:00008768 e51b2038      ldr r2, [fp, #-56]
.text:0000876c e51b301c      ldr r3, [fp, #-28]
.text:00008770 e0633002      rsb r3, r3, r2
.text:00008774 e50b303c      str r3, [fp, #-60]
.text:00008778 e51b3020      ldr r3, [fp, #-32]
.text:0000877c e2832001      add r2, r3, #1
```

```
.text:00008768 e51b2038      ldr r2, [fp, #-56]
.text:0000876c e51b301c      ldr r3, [fp, #-28]
.text:00008770 e300378f      mov r3, #1935
.text:00008774 e50b303c      str r3, [fp, #-60]
.text:00008778 e51b3020      ldr r3, [fp, #-32]
.text:0000877c e2832001      add r2, r3, #1
```
ATTACKING HEART RATE MONITORING APPLICATION
RUNNING PRELIMINARY TESTS – NO ATTACK
RUNNING PRELIMINARY TESTS – MALWARE ATTACK
CURRENT PERFORMANCE RESULTS

Table 1: Comparing the baseline architecture with the modified process integrity architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Time to run (ms)</th>
<th>Attacked application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heart rate</td>
<td>Malware 1</td>
</tr>
<tr>
<td></td>
<td>application</td>
<td></td>
</tr>
<tr>
<td>Baseline</td>
<td>60362</td>
<td>61855</td>
</tr>
<tr>
<td>Modified</td>
<td>60671 (halted early)</td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td>0.5%</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 2: Time taken to detect the malware after it is triggered

<table>
<thead>
<tr>
<th>Malware variation</th>
<th>Time to detect malware (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>220</td>
</tr>
<tr>
<td>2</td>
<td>235</td>
</tr>
</tbody>
</table>
KERNEL- AND USER-LEVEL MONITORING
### PERFORMANCE RESULTS WITH KERNEL-LEVEL MONITORING

Table 1: Comparing the baseline architecture with the kernel- and process-level monitoring architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Time to run (ms)</th>
<th>Attacked application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Heart rate</td>
<td>Malware 1</td>
</tr>
<tr>
<td></td>
<td>application</td>
<td></td>
</tr>
<tr>
<td>Baseline</td>
<td>60362</td>
<td>61855</td>
</tr>
<tr>
<td>Modified</td>
<td>60928</td>
<td>halted early</td>
</tr>
<tr>
<td>Overhead</td>
<td>0.94%</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 2: Time taken to detect the malware after it is triggered

<table>
<thead>
<tr>
<th>Malware variation</th>
<th>Time to detect malware (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>635</td>
</tr>
<tr>
<td>2</td>
<td>647</td>
</tr>
</tbody>
</table>
RESOURCE UTILIZATION AND POWER ESTIMATES

- Implementation results reported by the Vivado Design Suite showing the hardware overhead imposed by our architecture and the power estimates targeting the Digilent Zedboard

<table>
<thead>
<tr>
<th>Zynq-7000 FPGA Resource Utilization</th>
<th>LUT</th>
<th>LUT RAM</th>
<th>Flip Flop</th>
<th>Block RAM</th>
<th>IO</th>
<th>BUFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilization</td>
<td>4051</td>
<td>941</td>
<td>3348</td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Overall %</td>
<td>7.61</td>
<td>5.41</td>
<td>3.15</td>
<td>1.42</td>
<td>4</td>
<td>3.13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Power (mW)</th>
<th>Clocks</th>
<th>Signals</th>
<th>Logic</th>
<th>Block RAM</th>
<th>IO</th>
<th>PS7</th>
<th>Dynamic</th>
<th>Device Static</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>15</td>
<td>13</td>
<td>1</td>
<td>6</td>
<td>1532</td>
<td>1579</td>
<td>146</td>
<td>1725</td>
</tr>
</tbody>
</table>

© Georgia Institute of Technology, 2018
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
EXAMPLE SCENARIO – GENERIC SENSOR NODES

• A generic wireless sensor node used to manage the capture and transmission of data

• Insight
  • Components can be fabricated using different technology nodes without impacting performance and functionality
  • Take advantage by splitting the design to create a more robust and secure architecture

• Assumption
  • Availability of a secure and provisioned fab for older technology nodes
SECURE FABRICATION OF UNTRUSTED CHIPS

• Split the components of the sensor node into trusted and untrusted domains
• Trusted domain contains modules that can be fabricated using older technology nodes
• Untrusted domain contains modules that require cutting edge technology facilities
• After deployment, the untrusted domain acts as a continuous prover of correct operation to the trusted verifier domain
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
CONCLUSIONS

• Present multiple approaches for detecting run-time software and hardware malicious modifications in medical devices
  • Analog-, digital- and physiological-based signatures
  • Generate hardware signatures at run-time and compare them to securely stored golden signatures
• Detect extremely small hardware Trojans and hardware errors in embedded medical devices
• Detect malicious modification to application code running on an embedded processor in medical devices
• Simulation and synthesis results showed that the developed architecture successfully detects the targeted types attacks with minimal area and performance overhead
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Discussion and Conclusions
• List of Publications
• References
• Journal papers
  


• Conference papers
  


PUBLICATIONS – OTHERS

• Journal papers

• Conference papers
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Discussion and Conclusions
• List of Publications
• References
REFERENCES


REFERENCES – CONTINUED


REFERENCES – CONTINUED


© Georgia Institute of Technology, 2018
REFERENCES – CONTINUED


THANK YOU

Presented by:

Taimour Wehbe, Ph.D. Candidate
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, Georgia, USA

taimour.wehbe@gatech.edu
http://www.prism.gatech.edu/~twehbe3/
BACKUP SLIDES
HARDWARE-BASED SIGNATURE GENERATION USING MISRS – ARCHITECTURE AND APPROACH

- Chip 1: A/D & Signature Generation
  - Using FPGAs and commercial off-the-shelf (COTS) components
  - Using ASICs
- Chip 2: Signature Test & Sensor Data Encryption
- Novelty: addresses HT attacks on input values as they initially appear on a digital chip
HARDWARE-BASED SIGNATURE GENERATION USING MISRS

• Chip 1: A/D & Signature Generation

• Chip 2: Signature Test & Sensor Data Encryption
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
  • **Hardware Trojan Attack and Error Detection in Medical Devices**
    • An Example Scenario
    • Architecture Overview
    • Digital-based Signature Generation
    • **Physiological Features-based Signature Generation**
    • Analog-based Signature Generation
    • Combined Architecture
    • HT Attacks
    • Experimental Results
    • Run-time Code Integrity in Embedded and Medical Devices
    • Impact on Embedded Systems Security
    • Conclusions
    • List of Publications
    • References
PHYSIOLOGICAL FEATURES-BASED SIGNATURE GENERATION

- Added physiological feature extraction hardware to test for anomalies
- Novelty: helps distinguish HT attacks and hardware errors from health problems
PHYSIOLOGICAL FEATURES ALARM SIGNAL SEVERITY

• Alarm Severity:
  • No anomalies
  • Anomaly (with high fidelity)
  • Gray zone (possibility of an anomaly)

• Heart Rate (HR) Alarm Signal:
  • No anomaly: 45 bpm < HR < 110 bpm
  • Anomaly: HR < 30 bpm or HR > 150 bpm
  • Gray zone: 30 bpm < HR < 45 bpm or 110 bpm < HR < 150 bpm

• R-J Interval Alarm Signal:
  • No Anomaly: variation < 15 %
  • Anomaly: variation > 50 %
  • Gray zone: 15 % < variation < 50 %
<table>
<thead>
<tr>
<th>Physiological-based signature</th>
<th>MISR-based Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Match</td>
</tr>
<tr>
<td>No Anomaly</td>
<td>Correct operation</td>
</tr>
<tr>
<td>Gray Zone</td>
<td>Potential health problem (person is recommended to seek medical help)</td>
</tr>
<tr>
<td>Anomaly</td>
<td>Person is asked to seek immediate help</td>
</tr>
</tbody>
</table>
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models

• Hardware Trojan Attack and Error Detection in Medical Devices
  • An Example Scenario
  • Architecture Overview
  • Digital-based Signature Generation
  • Physiological Features-based Signature Generation
  • Analog-based Signature Generation
  • Combined Architecture
  • HT Attacks
  • Experimental Results

• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
• Novelty:
  • Further complicate the job of the attacker by incorporating both analog and digital expertise
  • Provide a faster detection mechanism at a much smaller area overhead
ANALOG-BASED SIGNATURE GENERATION – CHIP 2

Comparison Logic

if \(|\text{Signature 1} - \text{Signature 2}| \leq \text{threshold}\) declare a match

if \(|\text{Signature 1} - \text{Signature 2}| > \text{threshold}\) declare a mismatch

© Georgia Institute of Technology, 2018
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models

• Hardware Trojan Attack and Error Detection in Medical Devices
  • An Example Scenario
  • Architecture Overview
  • Digital-based Signature Generation
  • Physiological Features-based Signature Generation
  • Analog-based Signature Generation
  • Combined Architecture
  • HT Attacks
  • Experimental Results

• Run-time Code Integrity in Embedded and Medical Devices
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
COMBINED ARCHITECTURE FOR DETECTING AND DISTINGUISHING HT ATTACKS FROM HEALTH PROBLEMS

- Combined analog, digital and physiological based signature generation and testing
SINGLE ATTACKS (1&2)

Attack type 1
- Very similar to our threat model
- Payload directly attacks input data
- Other types of signature-based HT detection techniques do not cover this type of attack as their signature generation relies on the input data

Attack type 2
- Very similar to our threat model
- Payload attacks data at the outputs of internal modules
- Other HT detection techniques including signature-based ones were proven to be effective
• HT simultaneously targets data and signature

• Attacks on low order bits might pass undetected

• It is unclear what advantage would the attacker have
CHALLENGES

• Linux memory management and virtual memory implementation \([53]\)
• Details of the exact locations of the executable code including the library code
• Static vs. dynamic linking
  • Ways of incorporating a security architecture that seamlessly allows for both types of linking
• Implementing our technique on embedded systems with Address Space Layout Randomization (ASLR) including caches and their associated coherency protocols \([47]\)
• Applicability with systems protected by Kernel Address Space Layout Randomization (KASLR)
UNDERSTANDING ELF STRUCTURE

• ELF Header
  • Specifies targeted platforms, CPU wordlength
  • Offset of Program Header Table and Section Header Table
  • Use `readelf -h heartrate` command for more info.

• Program Header Table
  • Always comes after ELF Header
  • Specifies all different segments
  • Each segment might contain one or more sections
  • Different segments can be partially overlapped in some sections
  • Use `readelf -l heartrate`.

• Section Header Table
  • Usually comes at the end of ELF.
  • Specifies all different sections (smallest grouped unit in ELF)
  • Use `readelf -S heartrate`.

ELF File Structure

This is 1 Segment

These are 2 Sections
ADDRESSING UNMAPPED REGIONS DURING SIGNATURE GENERATION

- Analyze the paging process by the OS
- Extract unmapped regions within a page
- Define granularity and feed unmapped regions to signature generation algorithm

Unmapped region index

Page

Page size = 4KB

Unmapped region index

granularity = 4B
ADDRESSING UNMAPPED REGIONS DURING SIGNATURE GENERATION

- Example: Indices 1 and 3 have unmapped code
- Might want to consider initializing and periodically setting memory space to zero

<table>
<thead>
<tr>
<th>Unmapped region index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>1023</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mask</td>
<td>0x11111111</td>
<td>0x00000000</td>
<td>0x11111111</td>
<td>0x00000000</td>
<td>0x11111111 ... 1111</td>
<td>0x11111111</td>
</tr>
</tbody>
</table>

Result applied to the hash algorithm
SIGNATURE / HASH GENERATION

• Lightweight hash / Message Authentication Code (MAC) implementations
  • Currently using SHA256

• Hash every page in the text segment of a process
  • Page size: 4 KB → Hash output: 256 bits

• At compile-time, use a software SHA256 hash generator after the applications executable is created

• At run-time, use a hardware SHA256 hash engine to hash every page (currently present in physical memory) of a monitored process
PROOF-OF-CONCEPT EXPERIMENT

• Conduct experimental simulations to validate the design
• Perform attacks through real-time code modification
• Extensively test and validate the design requirements
• A beta-test ZedBoard implementation is performed with a heart rate monitoring application
• Sample malware targeting the heart rate monitoring application is developed
• Conduct performance comparison with the baseline non-monitoring OS version
IMPLEMENTING KERNEL-LEVEL MONITORING

• In our current implementation, the kernel is loaded into the same location in the memory at boot time

• Directly after kernel boot, the hardware monitor starts assessing the code integrity of some modules in the kernel
  • Kernel module responsible for the memory management of user-level task and applications (task_mem) starts to be monitored as soon as the kernel boots
  • Our kernel-level driver responsible for interacting between the kernel and the hardware monitor is started and the driver waits for the user-level application to start running (heartrate)
  • The hardware monitor starts monitoring the code of our kernel-level driver too
  • Once the user-level application (heartrate) starts running, the hardware monitor starts assessing the integrity of application’s code too
OUTLINE

• Introduction
• Research Overview
• Background and Prior Work
• Hardware and Software Threat Models
• Hardware Trojan Attack and Error Detection in Medical Devices
• Run-time Code Integrity in Embedded and Medical Devices
  • Motivation and Example Scenario
  • Methodology
  • Assumptions and Challenges
  • Kernel-level Code Integrity Assessment
  • Experimental Platform and Results
  • Current Limitations and Future Research
• Impact on Embedded Systems Security
• Conclusions
• List of Publications
• References
CURRENT ARCHITECTURAL AND IMPLEMENTATION LIMITATIONS

• Architectural:
  • Run-time compilation where code within pages is modified or new pages are added at run-time
    • Run-time code modification of the native application is not currently supported
    • Currently, dynamic linking of libraries is supported but not addition of new libraries or library code at run-time

• Implementation:
  • Part of the DMA controller is still implemented in software
    • To access memory contents, the hardware goes through an indirection using a user-level driver
    • Ideally, a kernel-level driver should be used even if the control remains in software
  • Fully implement an on-chip memory controller to read contents of DDR at run-time
    • Currently limited by the address space we are able to access using Xilinx IPs
    • Control the AXI DMA from hardware by creating a custom IP
UNADDRESSED CHALLENGES

• Just-in-time (JIT) compilation and run-time code relocation [54]
  • Regenerate hashes at runtime
  • Procedure has to be secure so probably could not be implemented in software
  • Need compiler expertise to explore the dynamic linking process including the symbol table information for resolution of Jump-and-Link and Return addresses
  • The worst case might be that all hashes need to be regenerated because of links in the pages and this worst case may be impractical
    • Might be resolved by the Procedure Linkage Table (PLT) implemented in Linux ELF files

• Return-Oriented Programming (ROP) attacks are currently not covered
  • They use the addresses in the PLT and we currently don't check the integrity of the PLT section
Two most important entries with \texttt{p\_type} “PT\_LOAD”:

1. Load Code Segment (from offset 0x00 w/ size 0x4FC in ELF) to Virt. Mem (from Addr. 0x8000)
2. Load Static Data Segment (from offset 0x1000 w/ size 0x138 in ELF) to Virt. Mem (from Addr. 0x11000).
UNDERSTANDING ELF STRUCTURE

• At compile time, natural ELF is NOT dynamically linked w/ libraries
• At load time, they are linked.

• 3 regions for each ELF (or DLL):
  - r-xp: Code
  - r--p: Static Data
  - rw-p: Dynamic Data

• Program at the lowest Virt. Mem Addr.

• Code and Data from different processes are scattered in Physical Memory and not consecutive for use. [3]
• Smallest Unit for Mapping: 1 Page = 4KB

/proc/<pid>/maps to find V.Addr. for all pages of natural ELF and DLLs.
HARDWARE SIGNATURE GENERATION (ALTERNATIVE)

- Multiple Input Signature Register (MISR) implementations [8]
  - Typically used in Built-In-Self-Test (BIST)

Built-In-Logic-Block-Observer (BILBO) configured to operate as a MISR