SOFTWARE SECURITY AND RANDOMIZATION THROUGH PROGRAM PARTITIONING AND CIRCUIT VARIATION

MOVING TARGET DEFENSE (MTD) ’14
3 Nov 2014

Todd Andel
Lindsey Whitehurst
Todd McDonald

School of Computing
University of South Alabama

Work performed under NSF grants CNS-1305369, DUE-1241675, DUE-1303384
Partitioning security critical program sections to FPGAs may mitigate many software security risks that rely on jumping within a program’s address space.

Since we utilize reconfigurable hardware, our partition approach can be used to provide a dynamic and adaptive software layout, resulting in a continually changing target.
Overview

- Hardware/Software Paradigm and Program Partitioning
- Partitioning for Software Security
  - Where we’re at
- Transitioning Towards Dynamic Target
FPGA growth has allowed for:
Customized reconfigurable “software” onto a hardware device
• Increasing Speed and Efficiency of Applications
• Protecting from Side-Channel-Analysis
• Protecting Intellectual Property and Preventing Tampering
• Dynamically Monitoring Programs at Runtime
Partitioning idea has been used for speedup a.k.a a co-processor

Reconfigurable logic changes this from a manufacture time decision to a compile time decision
Our Quest

- Determine if program partitioning between an FPGA and GPP can increase software security
  - Previous works do not provide functional protection of the code
  - Investigate system resilience against buffer overflow attacks
    - Well known and documented
    - Initial indication that system will enhance security
  - Cost-Effective Study
    - Determine the additional overhead added because of new configuration
FPGAs Enhance Security via Partitioning

- FPGAs do not have a program counter
  - Can attacks that rely on addresses be mitigated by running the vulnerable portions on an FPGA?

- For Example:
  - Stack Overflow
  - Heap Overflow
  - Return-to-libc

Original Function Call
Attacked Function Call
<table>
<thead>
<tr>
<th>Goal</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implement Vulnerable Program, Demonstrate Vulnerabilities</td>
<td>✓</td>
</tr>
<tr>
<td>Partition and Implement Software on GPP and FPGA</td>
<td>✓</td>
</tr>
<tr>
<td>Test Partitioned System</td>
<td>✓</td>
</tr>
<tr>
<td>Determine Overhead Associated with System</td>
<td></td>
</tr>
</tbody>
</table>
Implement Vulnerable Program and Attack on GPP

• **Hardware**
  - Xilinx Virtex-5 LX50T FPGA on Diligent Genesis development board

• **Microblaze Processor**
  - Designed in Xilinx XPS Using Base System Builder
  - Acts as GPP
  - Uses GCC Compiler
  - Turned off Compiler Flags to Prevent Stack Protection
  - Simple C Program vulnerable to Buffer Overflows
Vulnerable as expected since sending in a larger license code than the buffer

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "platform.h"

int checkLicense(char **license)
{
    char license_buffer[16];
    int valid_flag[1] = {0};
    (strcpy)(license_buffer, *license);
    if((strcmp)(license_buffer,"validLicense")==0)
    {
        valid_flag[0] = 1;
    }
    return valid_flag[0];
}

int main()
{
    init_platform();
    char *myLicense = "notAValidLicenseButOverflowingTheBuffer";
    if(checkLicense(&myLicense))
    {
        printf("\n\n============================\n"
        "Correct License! Please Continue\n"
        "============================\n\n")
        ;
    }
    else
    {
        printf("\nIncorrect License, Access Denied.\n\n")
        ;
    }
    return 0;
}
```
• **Microblaze Designed in Xilinx XPS**
  • Includes dual-port BRAM
  • C program running on Microblaze
  • Attached to BRAM port A

• **User core implemented in VHDL**
  • `checkLicense` now a circuit
    • License key included
    • Attached to BRAM port B

• **Trigger and data both passed through BRAM**
Control Flow within System

- Control determined by value in base address of shared BRAM space
- Data located in next address location in BRAM
- While C program is in control, lock = 1
- While VHDL is in control, lock = 2

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x90000000</td>
<td>Base Address of Shared BRAM</td>
</tr>
<tr>
<td>0x90000004</td>
<td>Lock</td>
</tr>
<tr>
<td>0x90000008</td>
<td>Data</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
int main()
{
    char *p_data = "validLicense";
    int *p_lock;
    int *p_data_location;

    p_lock = 0x90000004;
    p_data_location = 0x90000020;

    memcpy(p_data_location, p_data, 12);
    *p_lock = 1;

    xil_printf("User Entered Data = %s\n", p_data);

    while(*p_lock != 2){
        if(*p_data_location != 0 )
        {
            xil_printf("\n\n========================\n");
            xil_printf("Correct License! Please Continue\n");
            xil_printf("========================\n\n");
        }
        else
        {
            xil_printf("\nIncorrect License, Access Denied.\n\n");
        }
    return 0; }
Partition Design Operates as Intended

- Unfinished
  - More testing, runtime input
  - Timing and Overhead
  - Repeat for real GPP Partition vs. Microblaze
Transitioning to Dynamic Adaptive Variation

- Reconfigurable hardware allows target to change:
  - Two thrusts
    1. Partitioning
    2. Equivalent circuits
Partitioning Variants

• Randomly select partition
  • Basic blocks, function, method, object

• Automate via HLL – HDL compiler
  • SystemC, Streams-C, Impulse C

• Challenges
  • How to select partition and how often
  • Changing trigger and data changes between variants
Circuit Variations

- Take partition and produce circuit variants via polymorphic generator
  - Variants with same I/O relationship
  - Possibly change I/O relationships with fake inputs/fake outputs
  - Essentially a form of *indistinguishability* obfuscation

- Preferably we would like variants that:
  - Are generated randomly and efficiently
  - Hide some form of abstract information (topology, signals, components, function)

- Current techniques:
  - Iterative subcircuit selection and replacement
  - Deterministic hiding algorithms (mainly component hiding)
Iterative Subcircuit Selection and Replacement

- Random Boolean logic expansion (using logic rules)
- Random circuit generation (generate random circuits until you find a match)
- Random function expansion (using BDD)
The Big Picture with Dynamic Partial Reconfiguration

- HLL functions, basic blocks
- How many? How often? How selected?
- Program changes to remaining software

- Equivalent circuit variants
- Generation and selection

- Runtime changes to logic
- Xilinx, Altera, & OpenPR tools

Dynamic Program Partitioning

Dynamic Polymorphism

Partial Dynamic Reconfiguration

SystemC/VHDL -> Netlist

Partition

Logic

FPGA
Summary

• Hardware/Software Paradigm and Program Partitioning

• Partitioning for Software Security
  • Where we’re at

• Transitioning Towards Dynamic Target