CyberShield

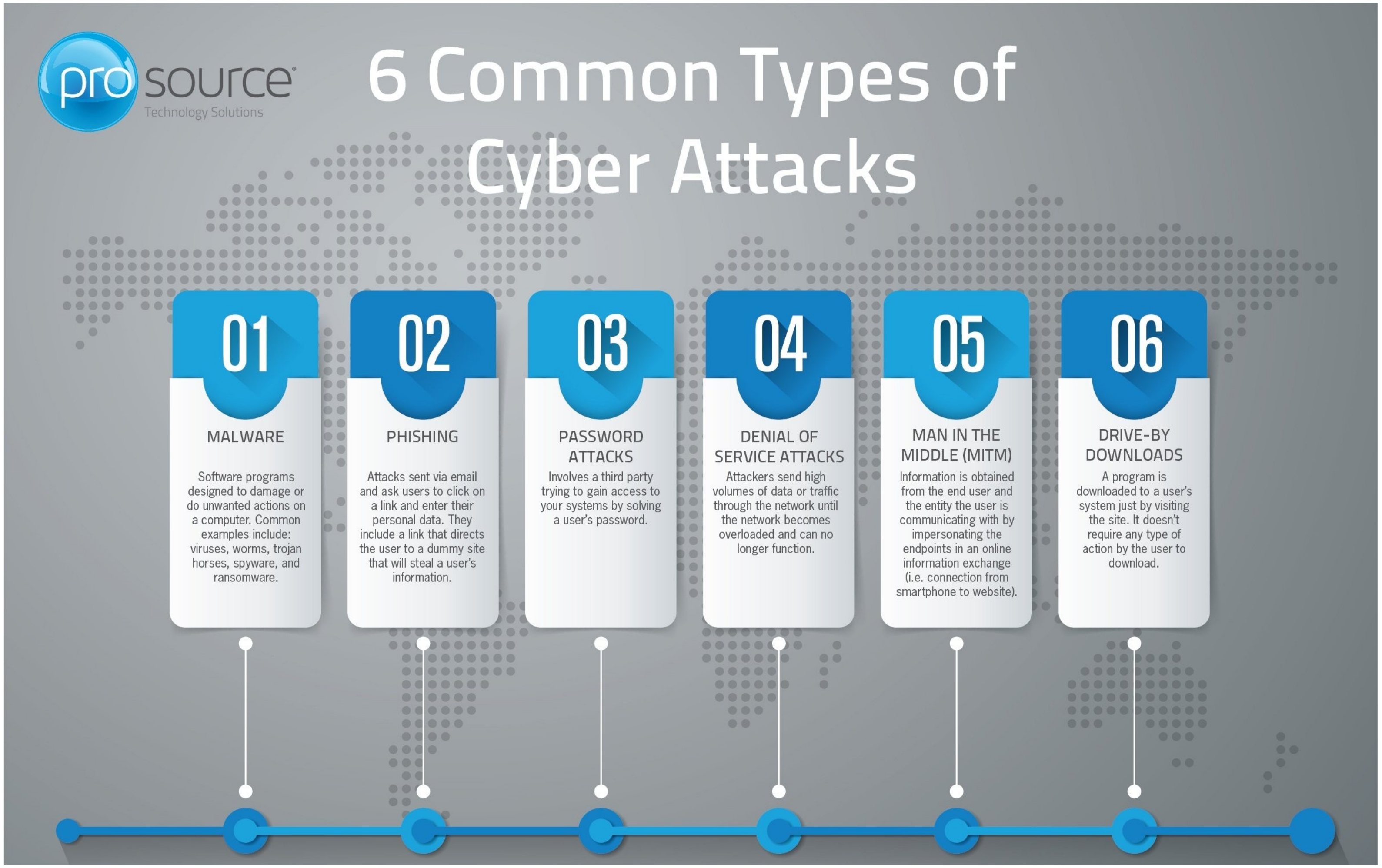
Intrusion Tolerant Flight Computing Through Hardware Obfuscation

CySER Summer Workshop

May 26, 2023

Chris Major
Research Assistant, Electrical and Computer Engineering

Types of Cybersecurity Attacks



prosource Technology Solutions

6 Common Types of Cyber Attacks

01 MALWARE
Software programs designed to damage or do unwanted actions on a computer. Common examples include: viruses, worms, trojan horses, spyware, and ransomware.

02 PHISHING
Attacks sent via email and ask users to click on a link and enter their personal data. They include a link that directs the user to a dummy site that will steal a user's information.

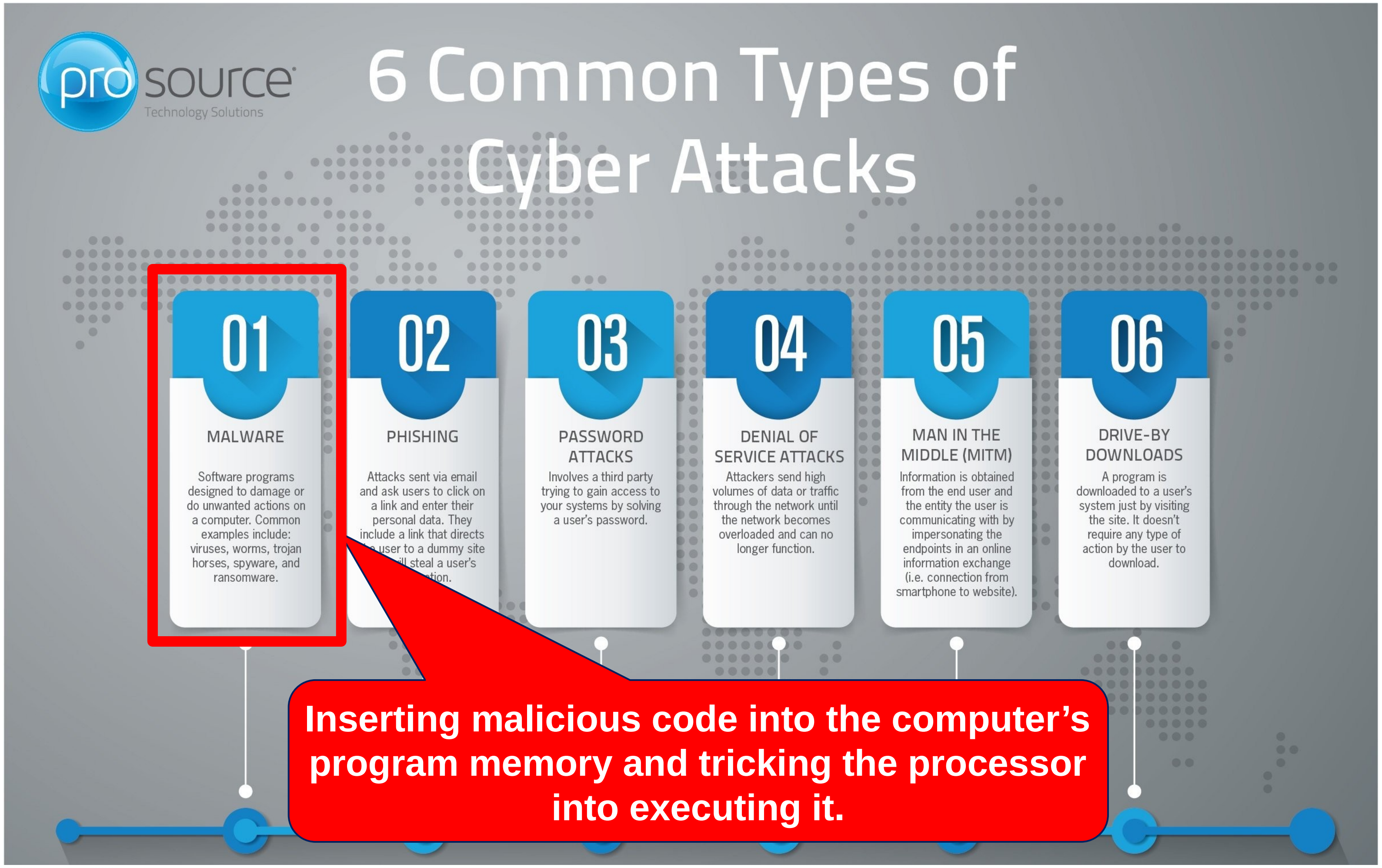
03 PASSWORD ATTACKS
Involves a third party trying to gain access to your systems by solving a user's password.

04 DENIAL OF SERVICE ATTACKS
Attackers send high volumes of data or traffic through the network until the network becomes overloaded and can no longer function.

05 MAN IN THE MIDDLE (MITM)
Information is obtained from the end user and the entity the user is communicating with by impersonating the endpoints in an online information exchange (i.e. connection from smartphone to website).

06 DRIVE-BY DOWNLOADS
A program is downloaded to a user's system just by visiting the site. It doesn't require any type of action by the user to download.

Types of Cybersecurity Attacks


pro source
Technology Solutions

6 Common Types of Cyber Attacks

01 MALWARE
Software programs designed to damage or do unwanted actions on a computer. Common examples include: viruses, worms, trojan horses, spyware, and ransomware.

02 PHISHING
Attacks sent via email and ask users to click on a link and enter their personal data. They include a link that directs ... user to a dummy site ...ill steal a user's ...tion.

03 PASSWORD ATTACKS
Involves a third party trying to gain access to your systems by solving a user's password.

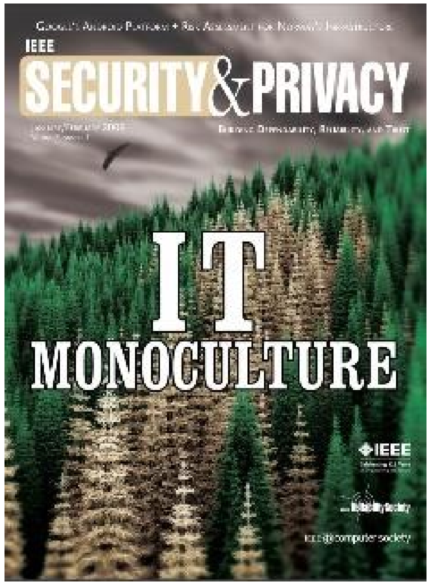
04 DENIAL OF SERVICE ATTACKS
Attackers send high volumes of data or traffic through the network until the network becomes overloaded and can no longer function.

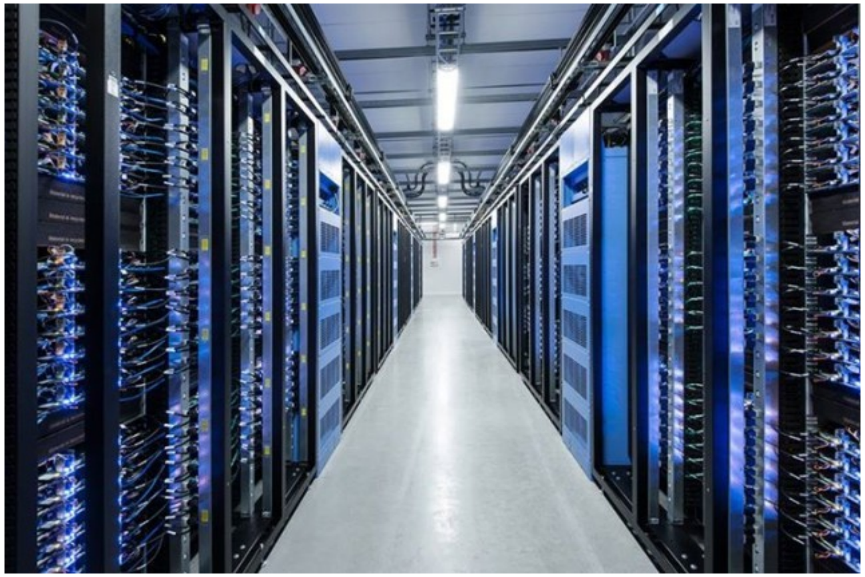
05 MAN IN THE MIDDLE (MITM)
Information is obtained from the end user and the entity the user is communicating with by impersonating the endpoints in an online information exchange (i.e. connection from smartphone to website).

06 DRIVE-BY DOWNLOADS
A program is downloaded to a user's system just by visiting the site. It doesn't require any type of action by the user to download.

Inserting malicious code into the computer's program memory and tricking the processor into executing it.

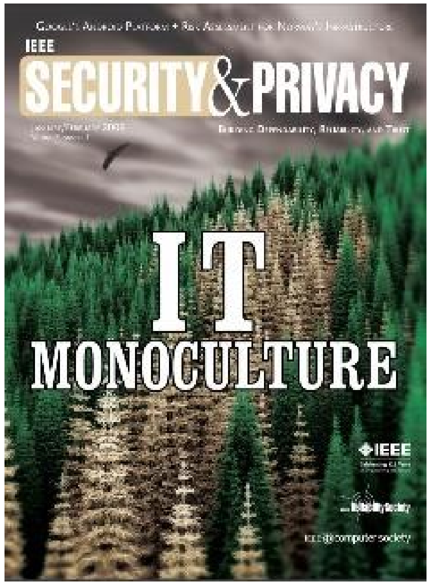
The Malware Challenge

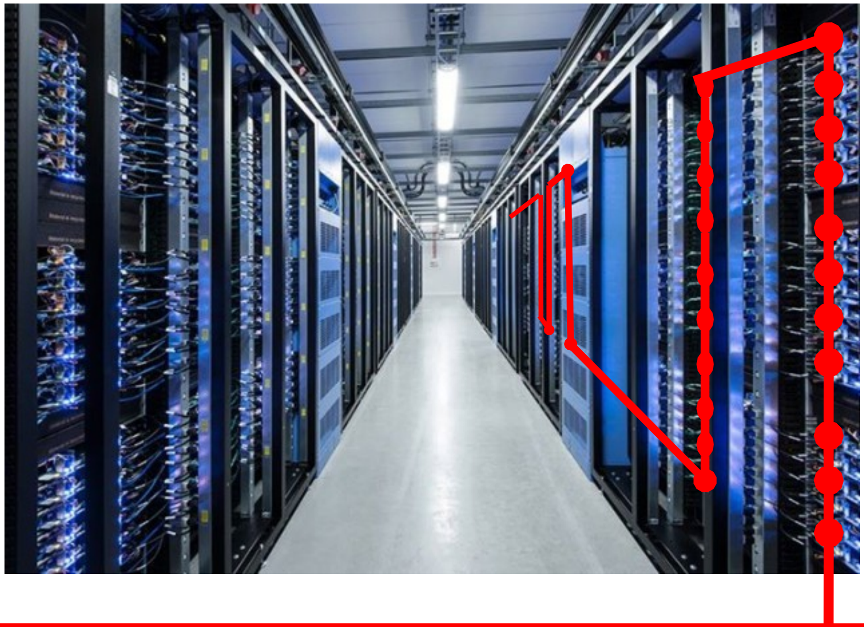




- The nation's cyber infrastructure consists of a massive number of **identical computer systems**.
- This **homogeneity** is advantageous because a single piece of software can be deployed across millions of systems to increase capacity.

The Malware Challenge





However, this gives an attacker a significant advantage in terms of effort relative to system defenders by re-using their attack across numerous systems.

The Embedded Advantage

Personal Computers
400M sold in 2018.

The Embedded Advantage

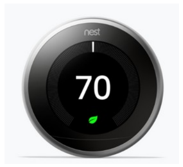
Personal Computers
400M sold in 2018.

Smart Phones
1.5B sold in 2018.

The Embedded Advantage

Personal Computers
400M sold in 2018.

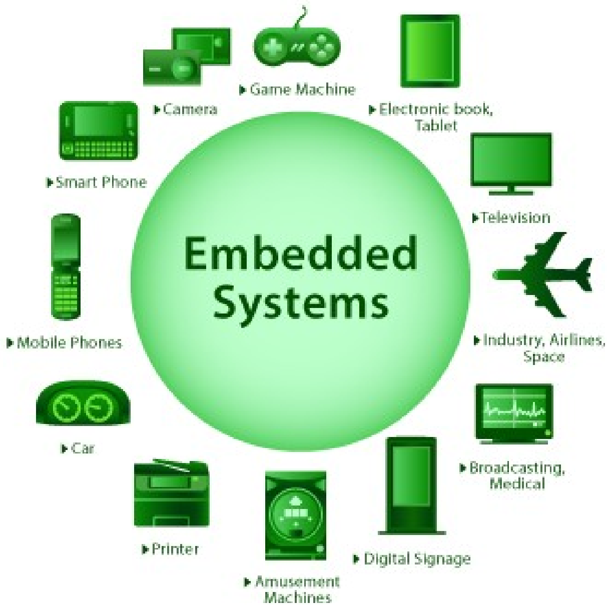
Smart Phones
1.5B sold in 2018.



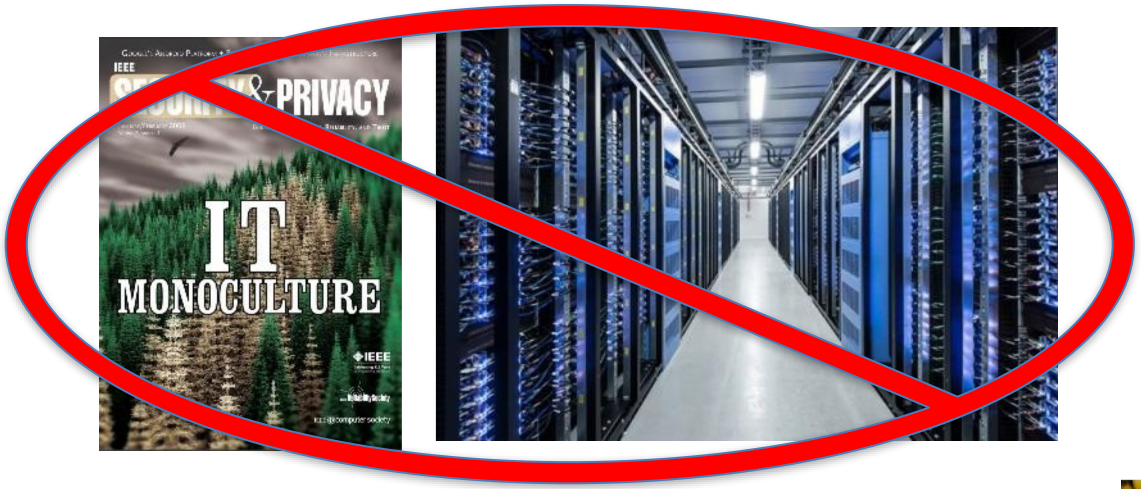
Embedded Computers
25B sold in 2018.

Our Approach

- Our focus is on diversifying **embedded computers** (not infrastructure).
- Embedded systems have ...
  - Smaller physical dimensions (sometimes 8-pin packages)
  - Lower Clock Frequencies (1MHz - 16MHz)
  - Smaller memories (256k to 1M)
  - Dedicated software, not general-purpose
  - Often no OS other than real-time scheduler
- Embedded systems are often **homogenous processor families.**

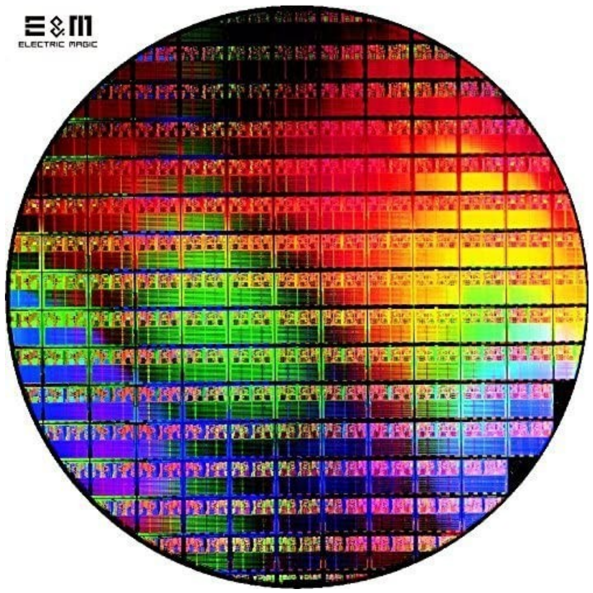


Removing The Advantage

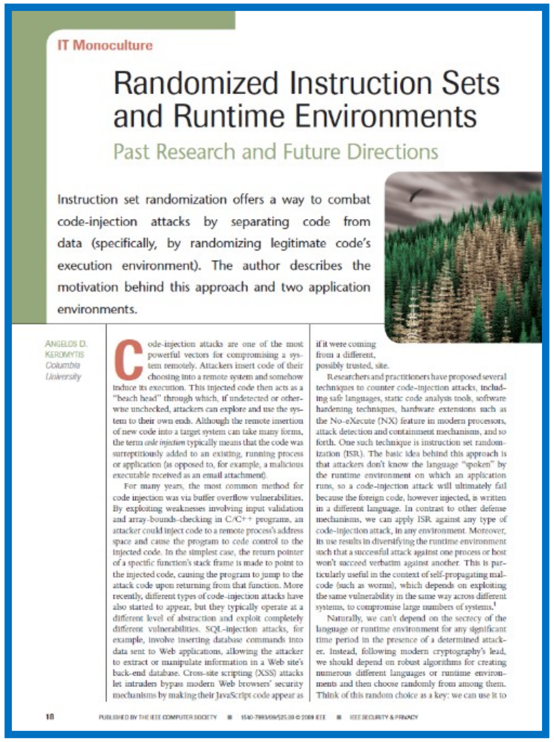


If **homogeneity** gives the attacker an advantage, **diversify the network** and **randomize the hardware**.

Diversifying Hardware



Most computer hardware is **fixed** and takes months/years to fabricate.



IT Monoculture

Randomized Instruction Sets and Runtime Environments

Past Research and Future Directions

Instruction set randomization offers a way to combat code-injection attacks by separating code from data (specifically, by randomizing legitimate code's execution environment). The author describes the motivation behind this approach and two application environments.

ANGELOS D. KEROMYTIS
Columbia University

Code-injection attacks are one of the most powerful vectors for compromising a system remotely. Attackers insert code of their choosing into a remote system and somehow induce its execution. This injected code then acts as a "beach head" through which, if undetected or otherwise unchecked, attackers can explore and use the system to their own ends. Although the remote insertion of new code into a target system can take many forms, the term *code injection* typically means that the code was surreptitiously added to an existing, running process or application (as opposed to, for example, a malicious executable received as an email attachment).

For many years, the most common method for code injection was via buffer overflow vulnerabilities. By exploiting weaknesses involving input validation and array-bounds-checking in C/C++ programs, an attacker could inject code to a remote process's address space and cause the program to code control to the injected code. In the simplest case, the return pointer of a specific function's stack frame is made to point to the injected code, causing the program to jump to the attack code upon returning from that function. More recently, different types of code-injection attacks have also started to appear, but they typically operate at a different level of abstraction and exploit completely different vulnerabilities. SQL-injection attacks, for example, involve inserting database commands into data sent to Web applications, allowing the attacker to extract or manipulate information in a Web site's back-end database. Cross-site scripting (XSS) attacks let intruders bypass modern Web browsers' security mechanisms by making their JavaScript code appear as if it were coming from a different, possibly trusted, site.

Researchers and practitioners have proposed several techniques to counter code-injection attacks, including safe languages, static code analysis tools, software hardening techniques, hardware extensions such as the No-eXecute (NX) feature in modern processors, attack detection and containment mechanisms, and so forth. One such technique is instruction set randomization (ISR). The basic idea behind this approach is that attackers don't know the language "spoken" by the runtime environment on which an application runs, so a code-injection attack will ultimately fail because the foreign code, however injected, is written in a different language. In contrast to other defense mechanisms, we can apply ISR against any type of code-injection attack, in any environment. Moreover, its use results in diversifying the runtime environment such that a successful attack against one process or host won't succeed verbatim against another. This is particularly useful in the context of self-propagating malcode (such as worms), which depends on exploiting the same vulnerability in the same way across different systems, to compromise large numbers of systems.[1]

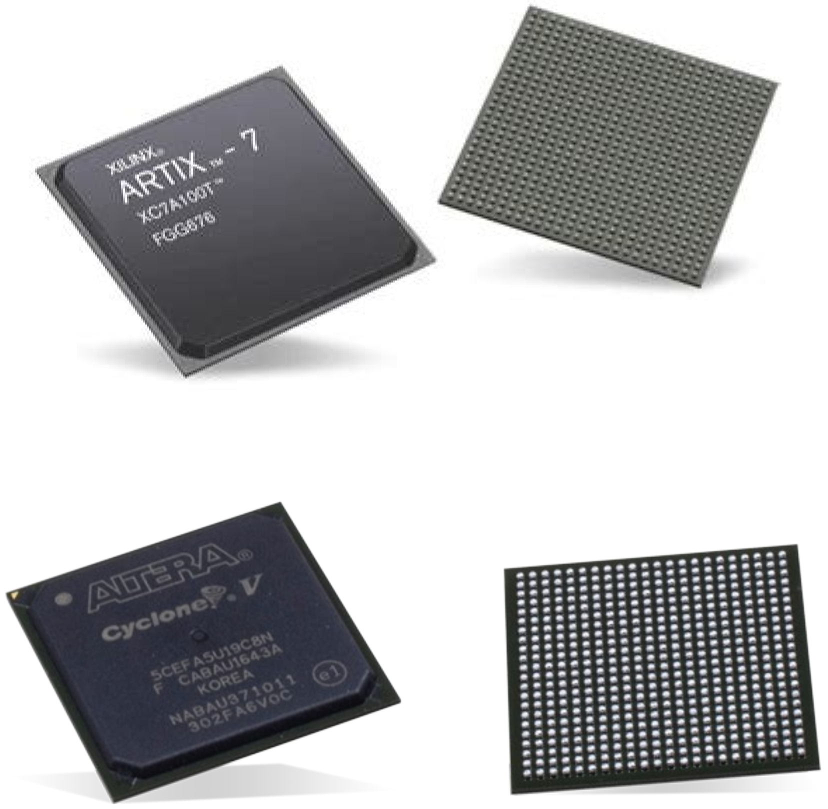
Naturally, we can't depend on the secrecy of the language or runtime environment for any significant time period in the presence of a determined attacker. Instead, following modern cryptography's lead, we should depend on robust algorithms for creating numerous different languages or runtime environments and then choose randomly from among them. Think of this random choice as a key: we can use it to

18 PUBLISHED BY THE IEEE COMPUTER SOCIETY ■ 1540-7993/09/$25.00 © 2009 IEEE ■ IEEE SECURITY & PRIVACY

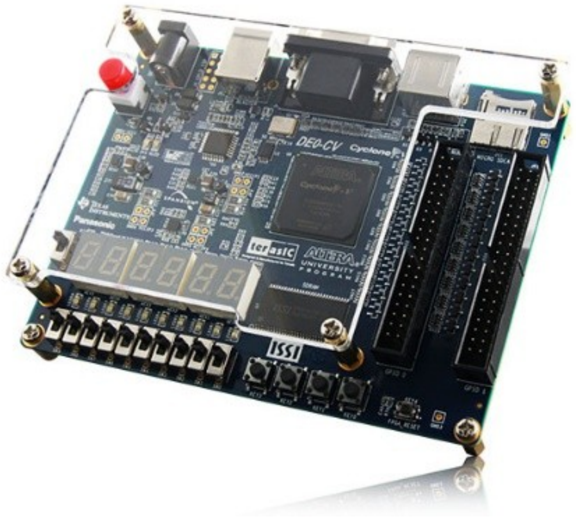
There has been some prior work in the area of randomization of instructions sets in **Virtual Machines**, with promising results.

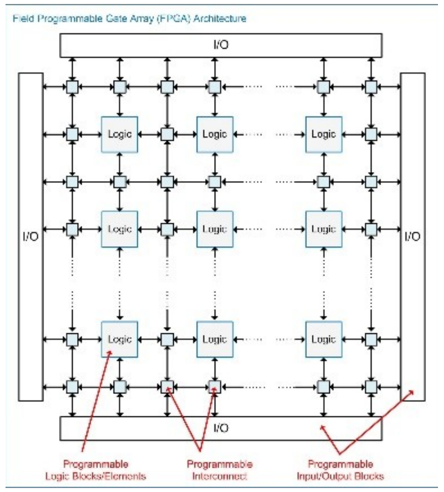
Field Programmable Gate Arrays (FPGAs)

- FPGAs are digital logic devices that can be **configured into any computational architecture.**
  - Logic Matrix
  - Reconfigurable
- FPGAs can take advantage of **parallelism and redundancy for hardware acceleration.**
- **Commercial availability** and extensive development support make FPGAs easy to access and implement.



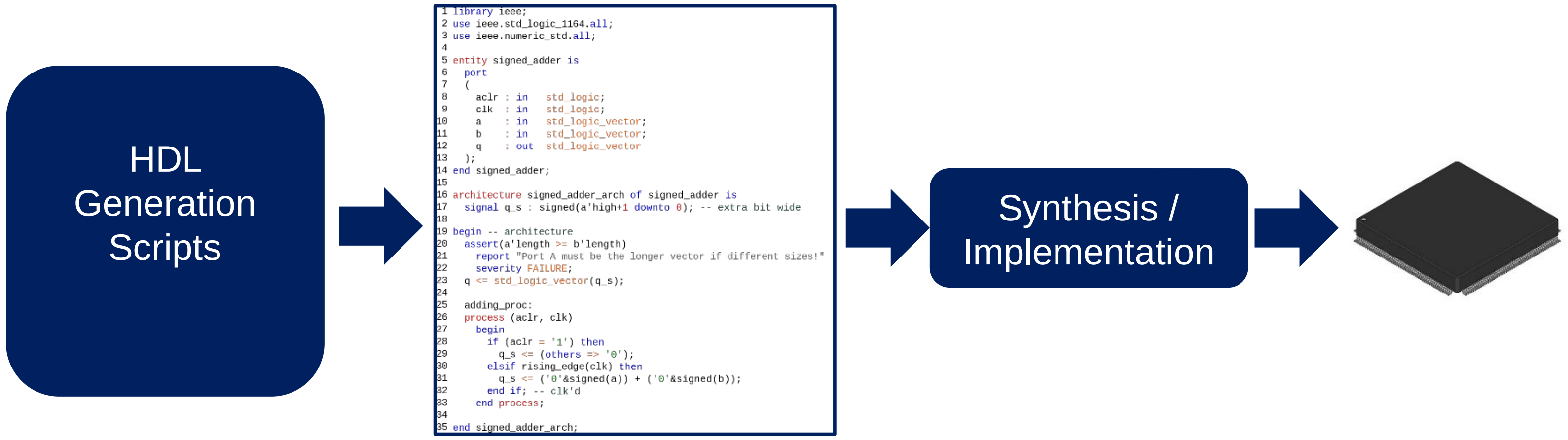
Field Programmable Gate Arrays (FPGAs)




Field Programmable Gate Array (FPGA) Architecture
I/O
Logic
Programmable Logic Blocks/Elements
Programmable Interconnect
Programmable Input/Output Blocks

- FPGA hardware is designed using a **Hardware Description Language (HDL)**.
- Once we have a design in an HDL, we can use scripts to create versions of it with alterations.
- The HDL can be created at **compile-time**.

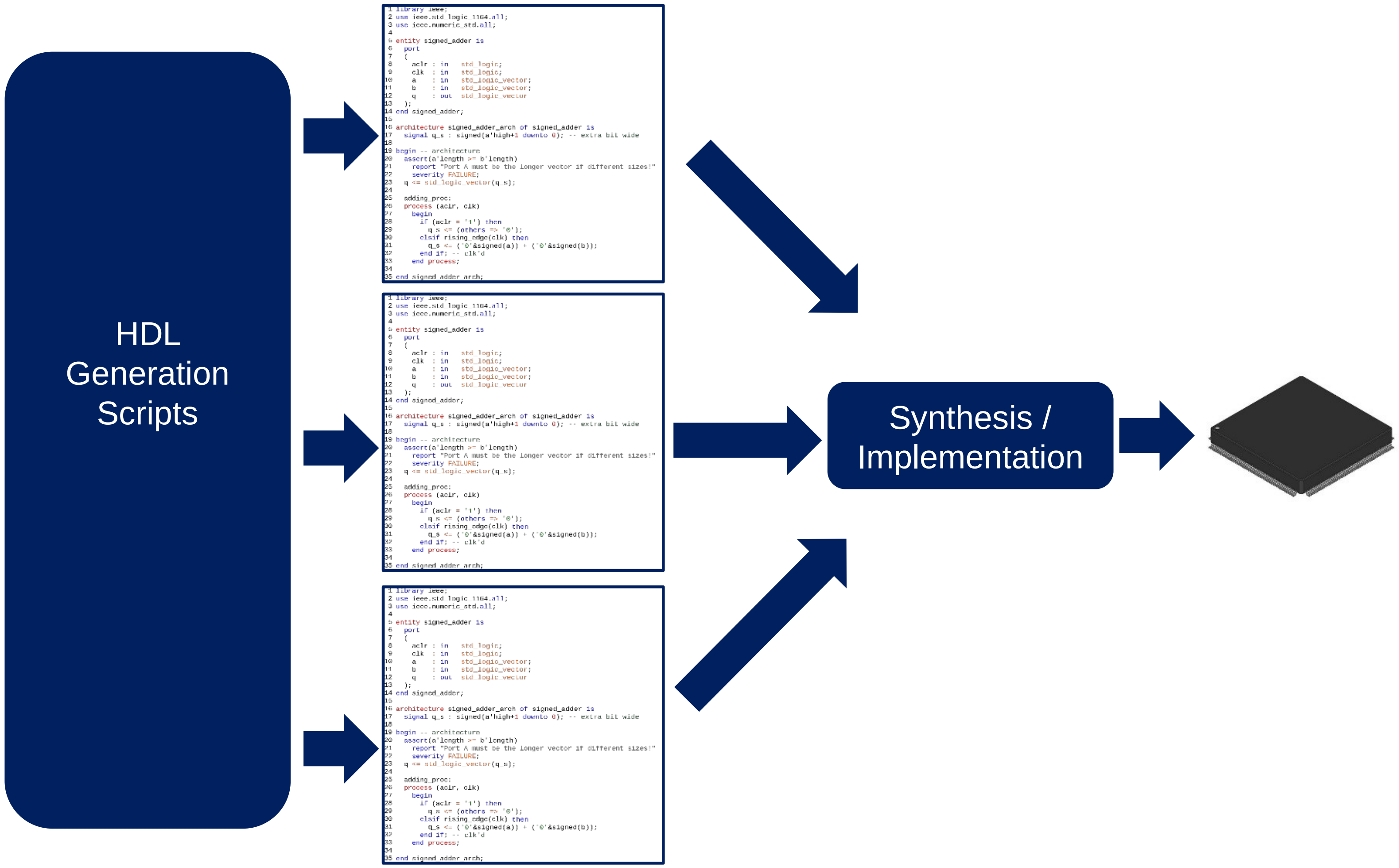
Field Programmable Gate Arrays



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

- HDL is generated into **Real-Time Logic (RTL)**
- RTL is generated through the FPGA vendor’s **synthesis and implementation suite.**
- A bitstream is generated that can translate the RTL into logic block placement within the FPGA.

Heterogenous Cores



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

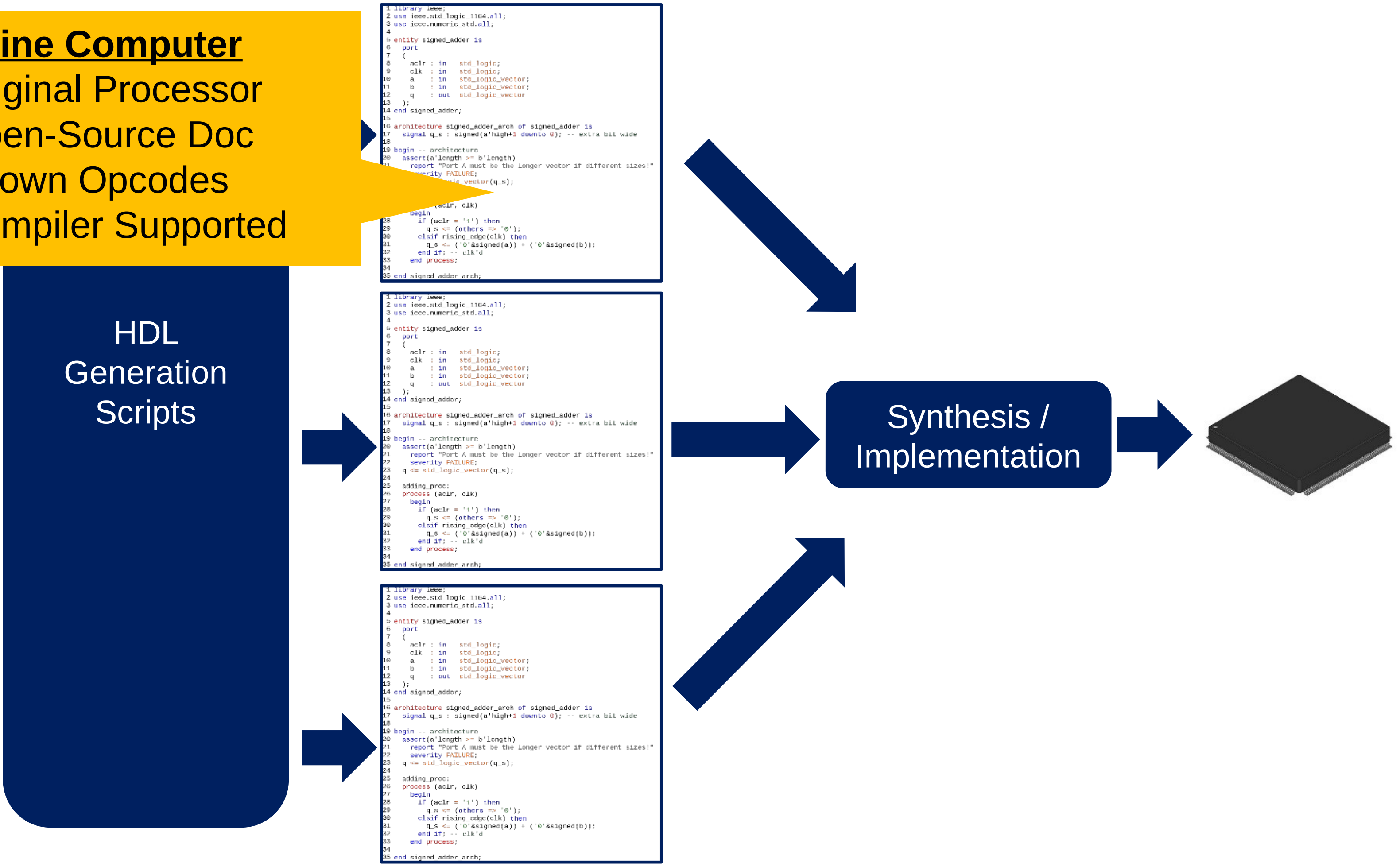
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

Heterogenous Cores

<u>Baseline Computer</u>

- Original Processor
- Open-Source Doc
- Known Opcodes
- Compiler Supported



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    verity FAILURE;
    ic_vector(q_s);

      (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

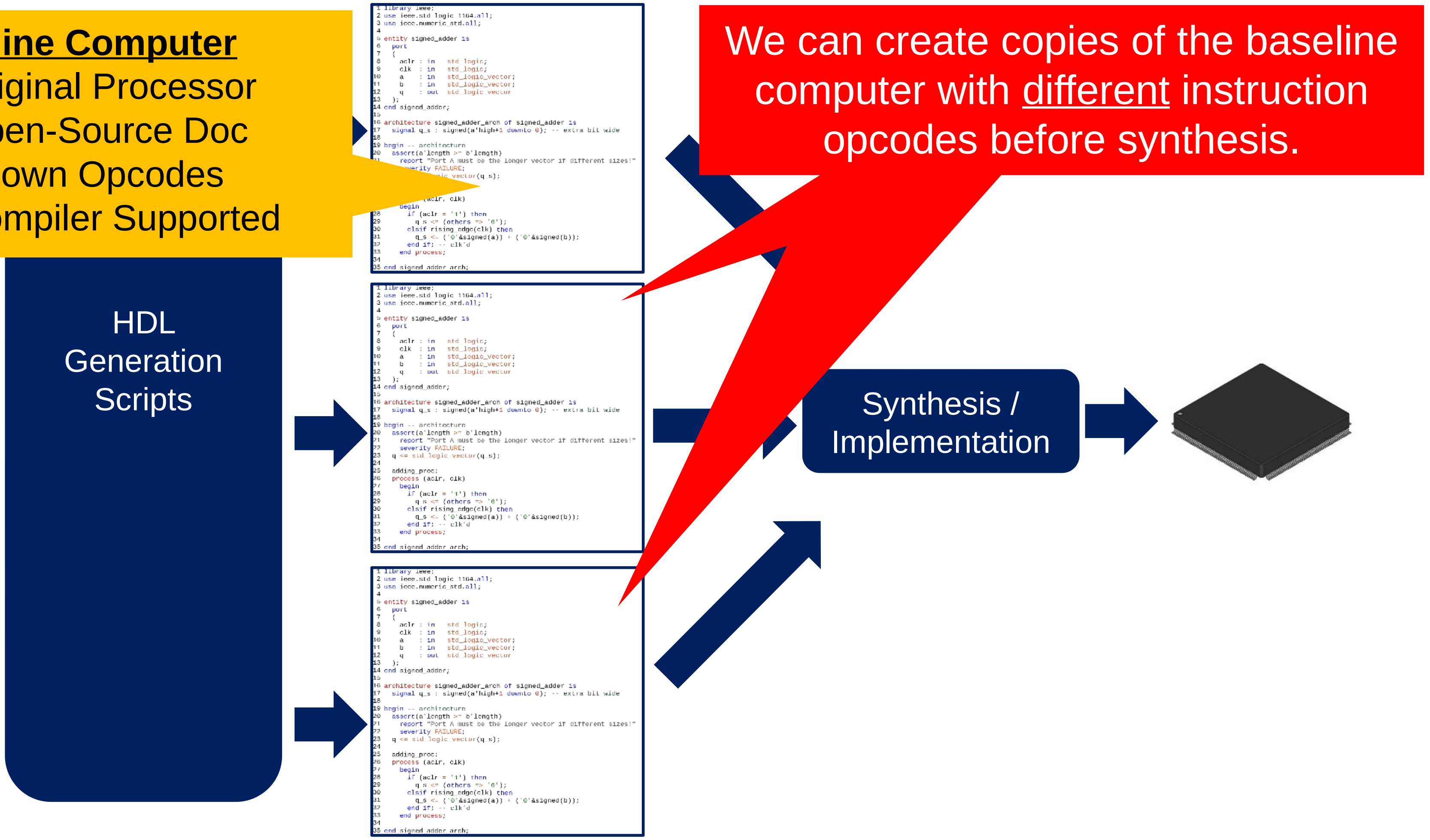
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

Heterogenous Cores

Baseline Computer

- Original Processor
- Open-Source Doc
- Known Opcodes
- Compiler Supported

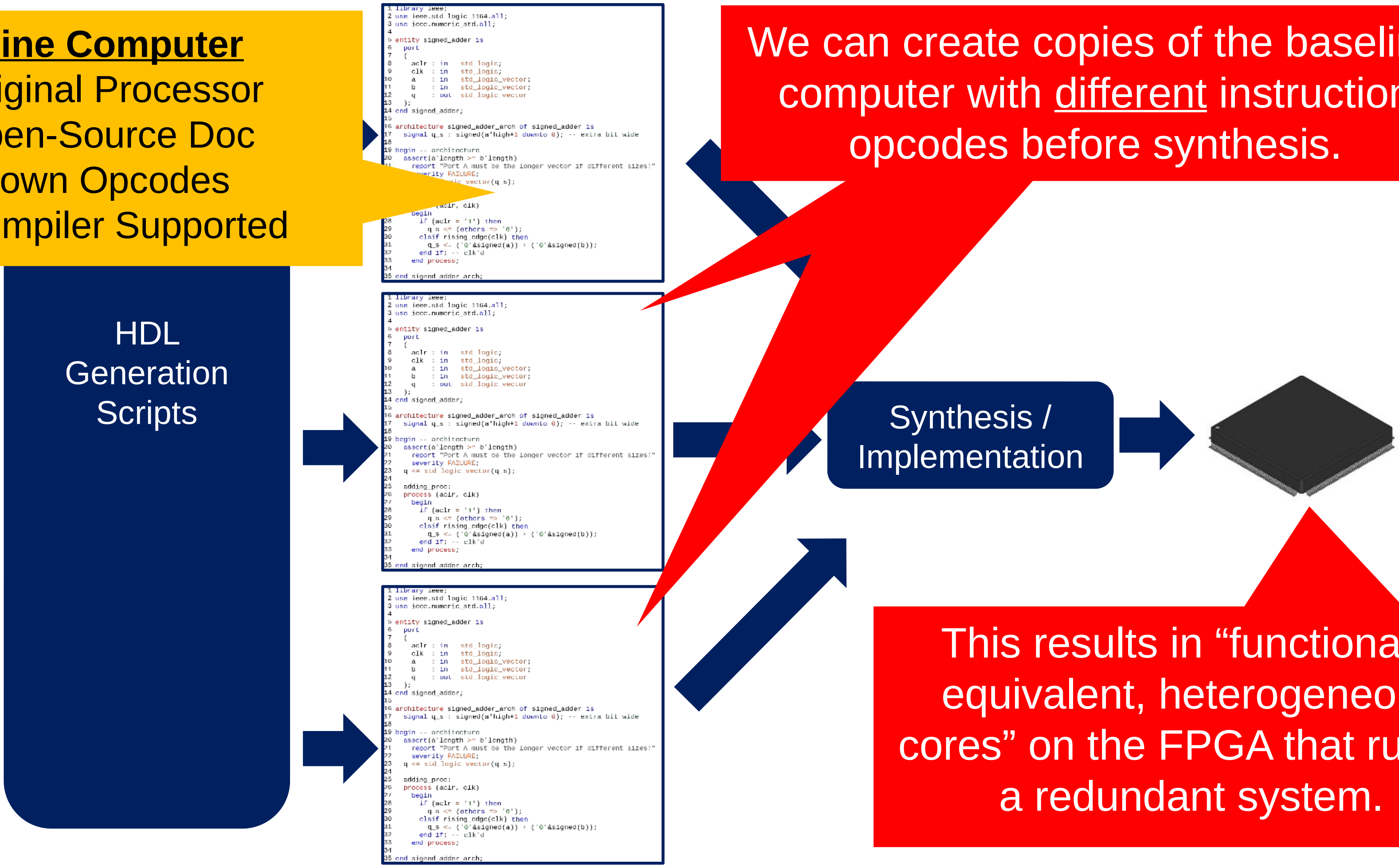
We can create copies of the baseline computer with <u>different</u> instruction opcodes before synthesis.



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

Heterogenous Cores

Baseline Computer
- Original Processor
- Open-Source Doc
- Known Opcodes
- Compiler Supported



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

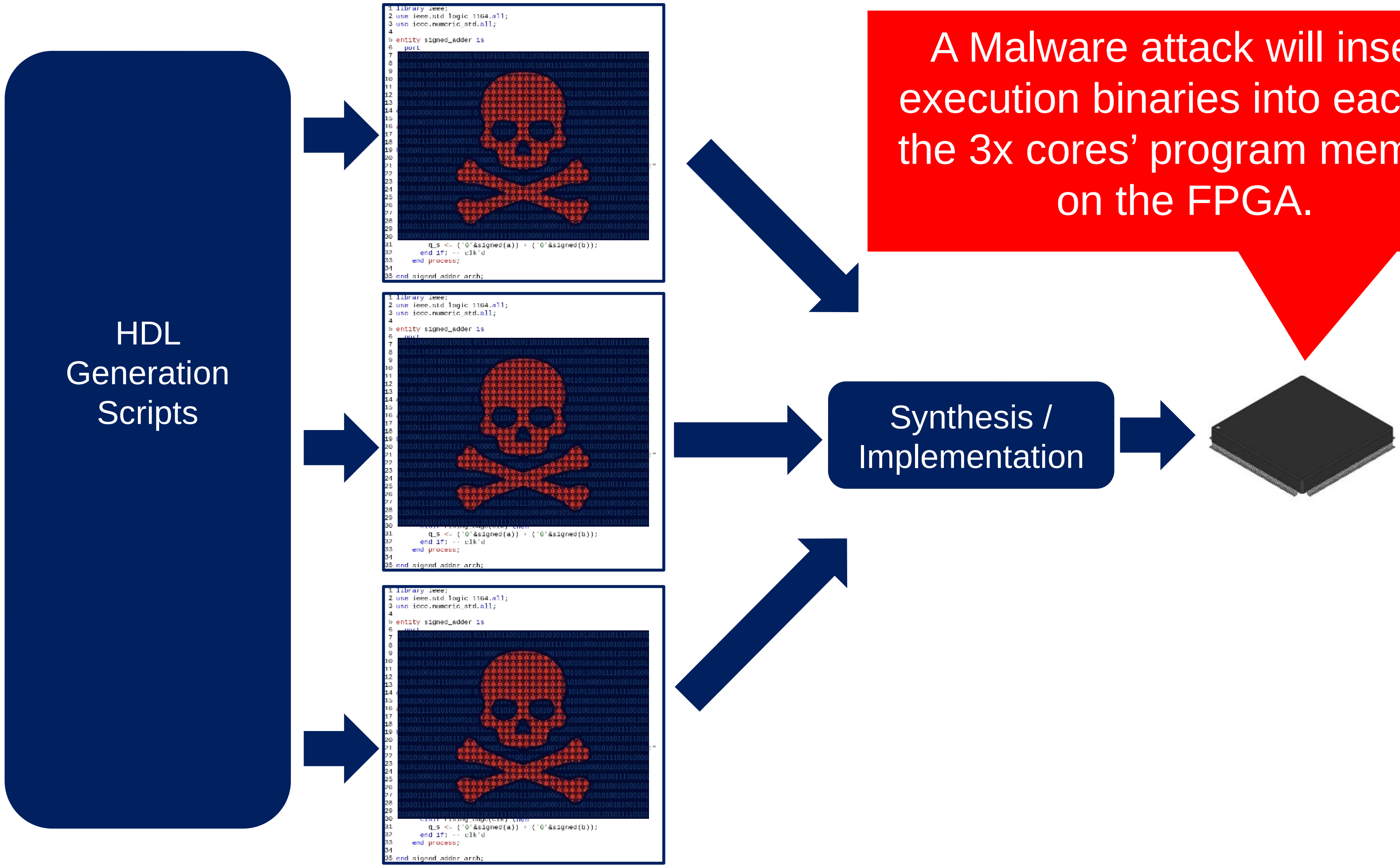
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

We can create copies of the baseline computer with <u>different</u> instruction opcodes before synthesis.

This results in “functionally equivalent, heterogeneous cores” on the FPGA that run as a redundant system.

Heterogenous Cores

A Malware attack will insert execution binaries into each of the 3x cores' program memory on the FPGA.

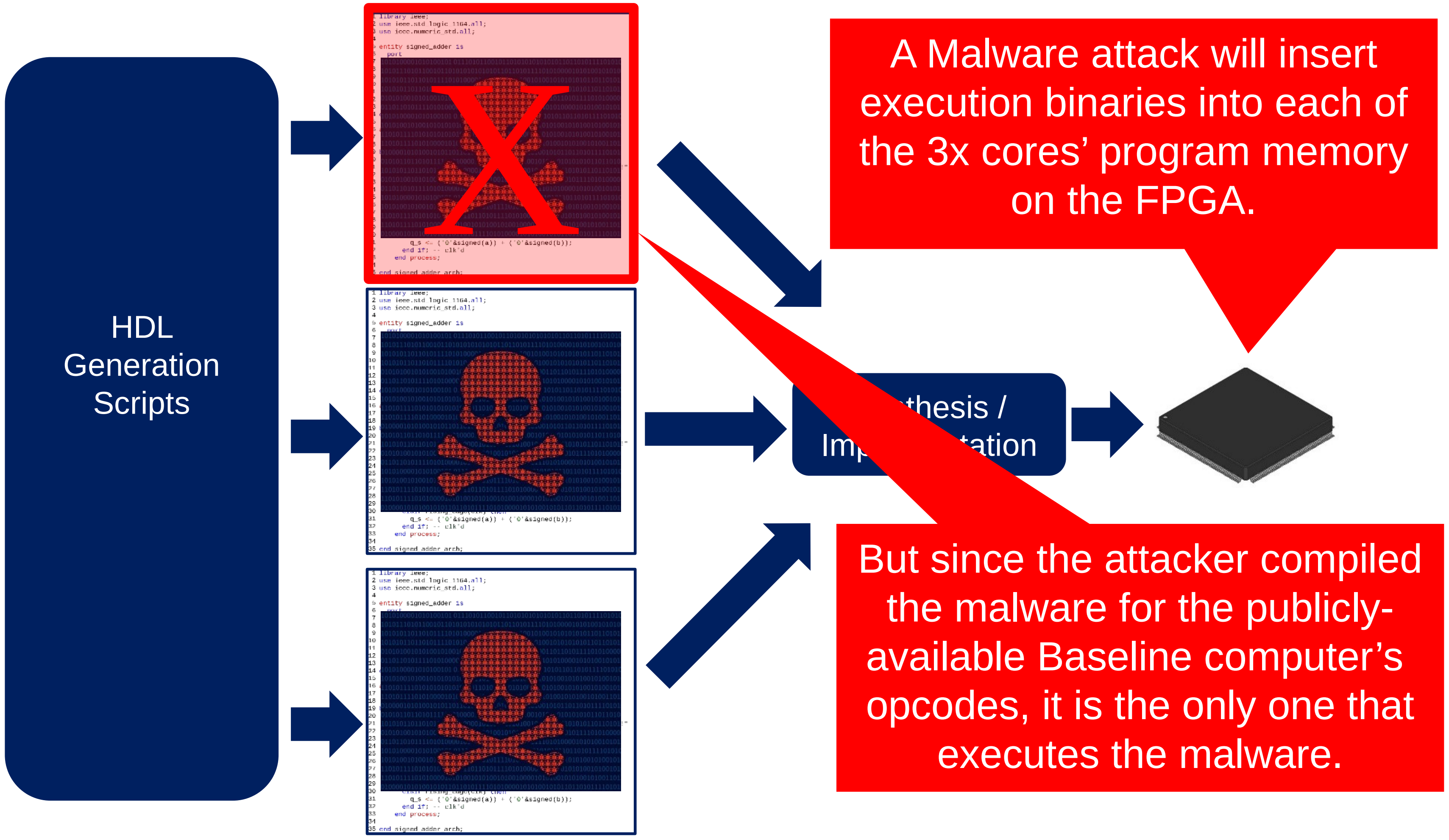


```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

Heterogenous Cores



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
      q_s <= ('0'&signed(a)) + ('0'&signed(b));
    end if; -- clk'd
  end process;
end signed_adder_arch;
```

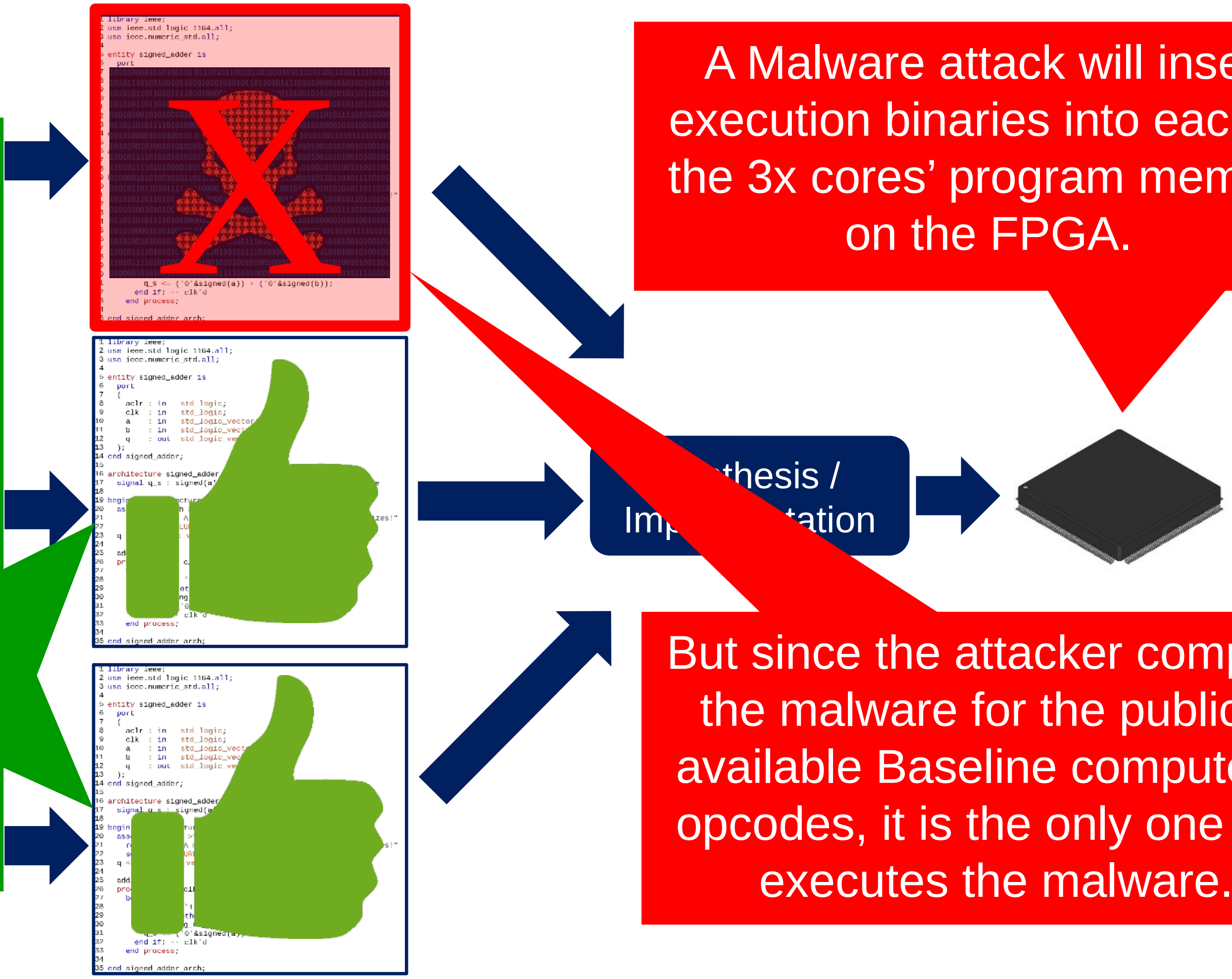
A Malware attack will insert execution binaries into each of the 3x cores' program memory on the FPGA.

But since the attacker compiled the malware for the publicly-available Baseline computer's opcodes, it is the only one that executes the malware.

Heterogenous Cores

The computers with randomized opcodes don't recognize the malware.

We can either throw an exception or run a pre-defined routine to remove the malware.

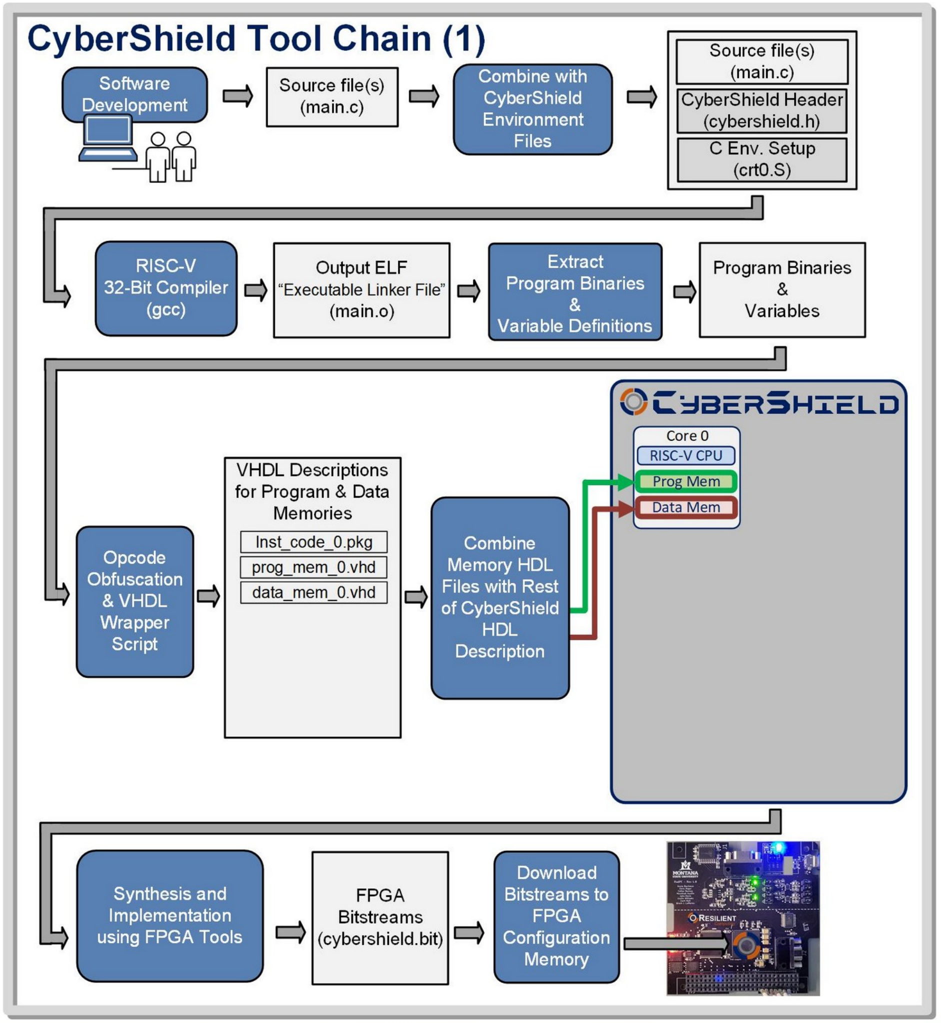


A Malware attack will insert execution binaries into each of the 3x cores' program memory on the FPGA.

But since the attacker compiled the malware for the publicly-available Baseline computer's opcodes, it is the only one that executes the malware.

The CyberShield Concept

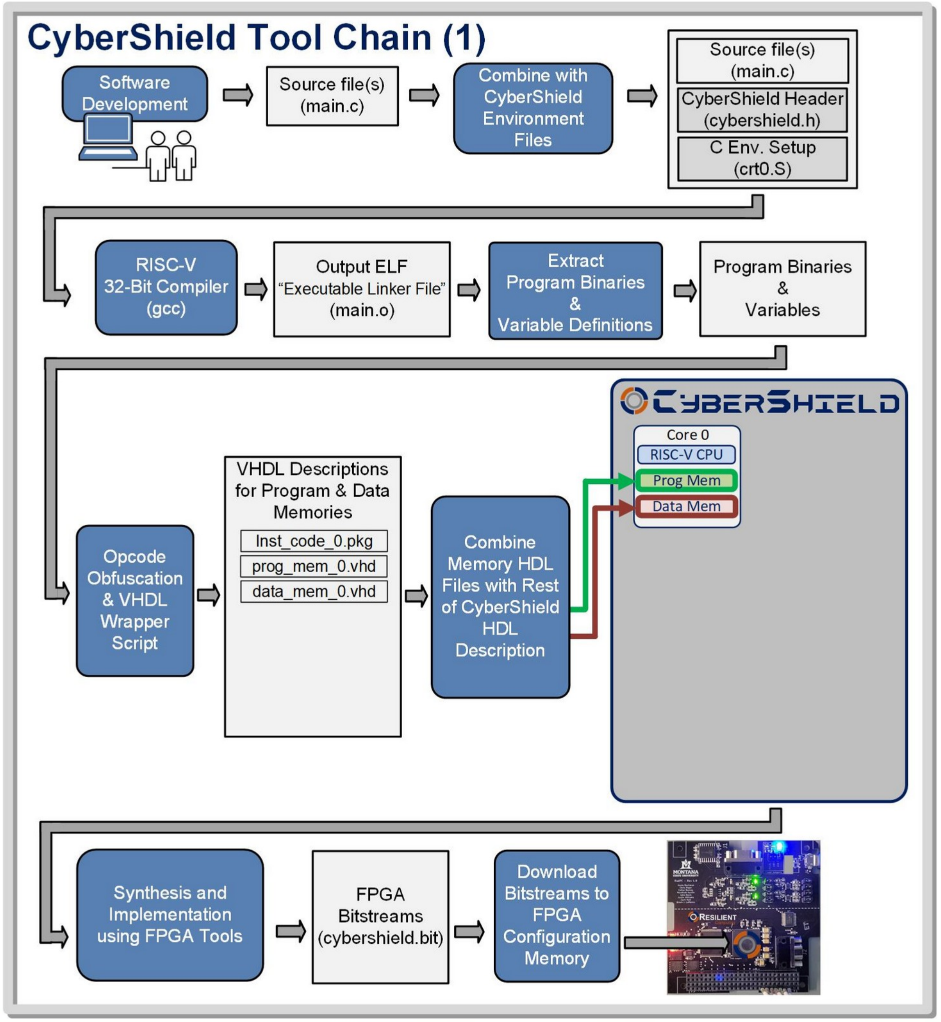
- Compile-time creation of obfuscated computing hardware.



CyberShield Tool Chain (1)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
CyberShield
Core 0
RISC-V CPU
Prog Mem
Data Mem
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory

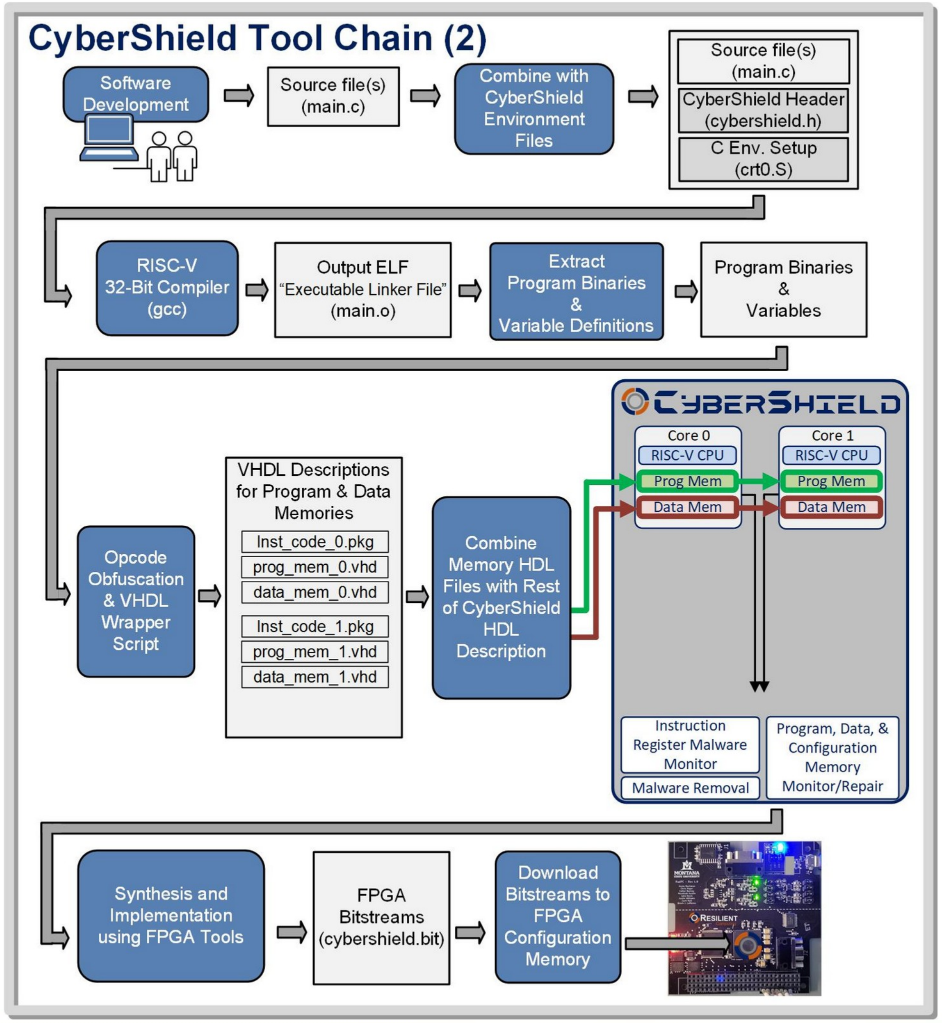
The CyberShield Concept

- Compile-time creation of obfuscated computing hardware.
- But what if the attacker guesses our instruction codes?


CyberShield Tool Chain (1)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
CYBERSHIELD
Core 0
RISC-V CPU
Prog Mem
Data Mem
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory

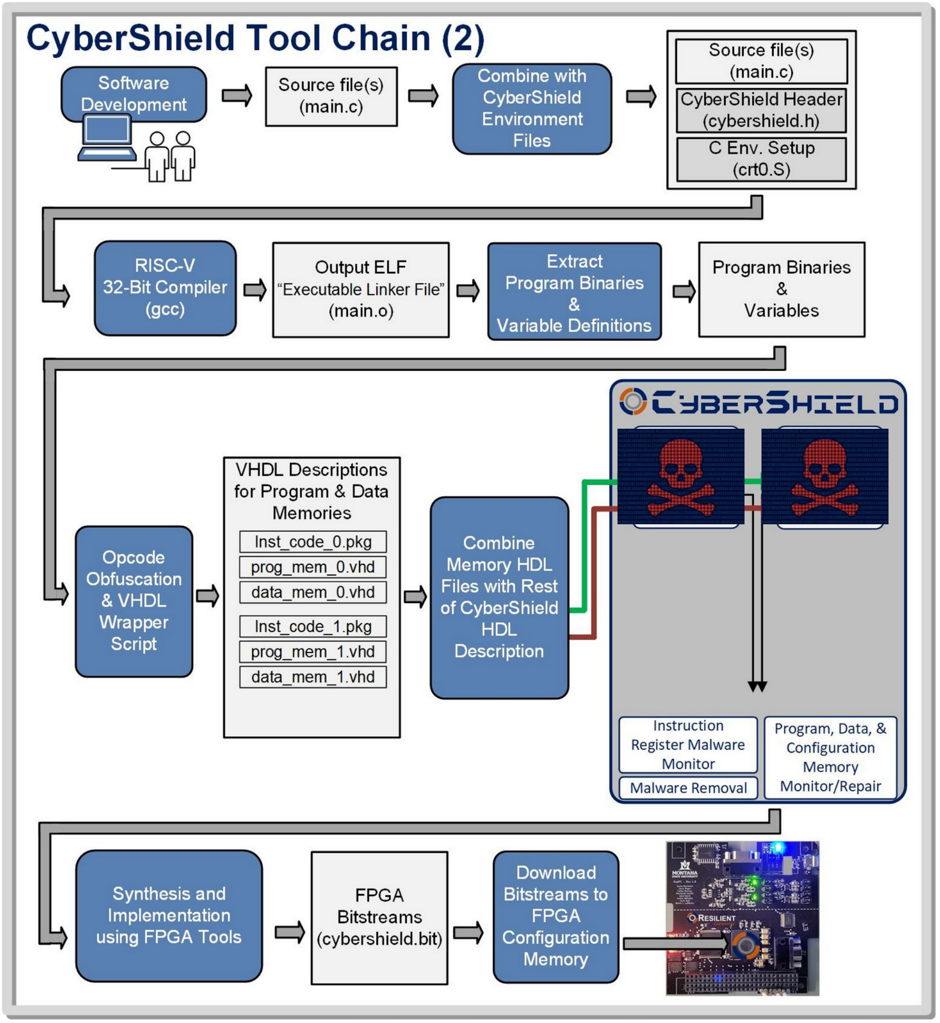
The CyberShield Concept

- Compile-time creation of obfuscated computing hardware.
- But what if the attacker guesses our instruction codes?
- What if we had **TWO** sets of different instruction code assignments?


CyberShield Tool Chain (2)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
CYBERSHIELD
Core 0
RISC-V CPU
Prog Mem
Data Mem
Core 1
RISC-V CPU
Prog Mem
Data Mem
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Inst_code_1.pkg
prog_mem_1.vhd
data_mem_1.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
Instruction Register Malware Monitor
Malware Removal
Program, Data, & Configuration Memory Monitor/Repair
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory

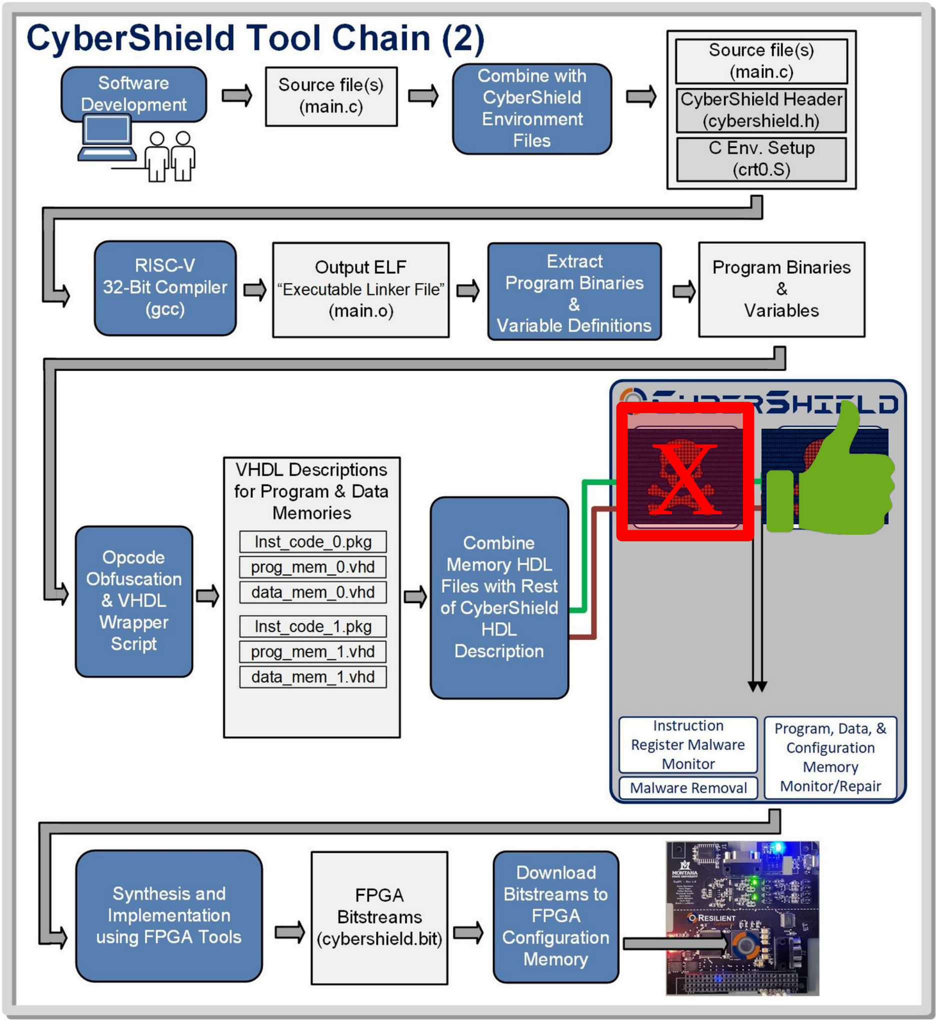
The CyberShield Concept

- Compile-time creation of obfuscated computing hardware.
- But what if the attacker guesses our instruction codes?
- What if we had **TWO** sets of different instruction code assignments?
- Then we could compare opcodes between the computers and if the are ever the SAME, it is malware.


CyberShield Tool Chain (2)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
CYBERSHIELD
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Inst_code_1.pkg
prog_mem_1.vhd
data_mem_1.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
Instruction Register Malware Monitor
Malware Removal
Program, Data, & Configuration Memory Monitor/Repair
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory
MONTANA
RESILIENT

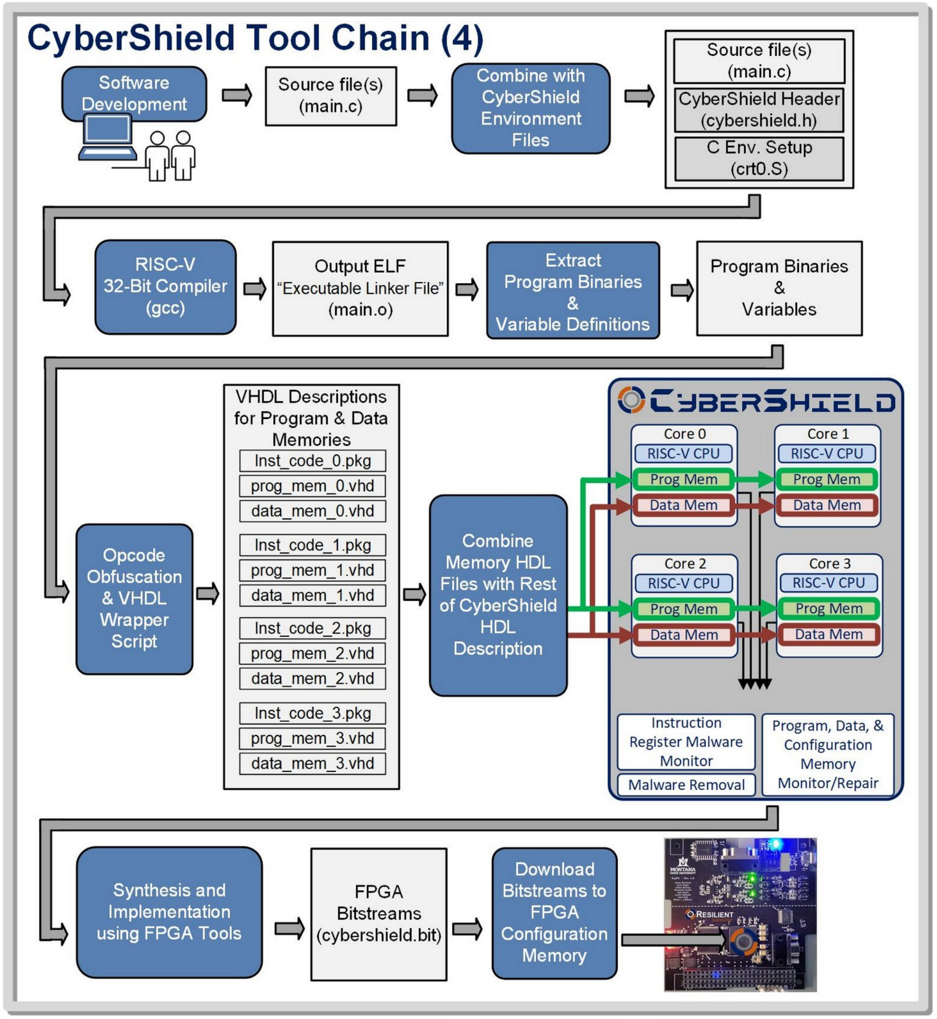
The CyberShield Concept

- Compile-time creation of obfuscated computing hardware.
- But what if the attacker guesses our instruction codes?
- What if we had **TWO** sets of different instruction code assignments?
- Then we could compare opcodes between the computers and if the are ever the SAME, it is malware.


CyberShield Tool Chain (2)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Inst_code_1.pkg
prog_mem_1.vhd
data_mem_1.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
CYBERSHIELD
X
Instruction Register Malware Monitor
Malware Removal
Program, Data, & Configuration Memory Monitor/Repair
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory
RESILIENT

The CyberShield Concept

- Compile-time creation of obfuscated computing hardware.
- But what if the attacker guesses our instruction codes?
- What if we had **TWO** sets of different instruction code assignments?
- Then we could compare opcodes between the computers and if the are ever the SAME, it is malware.
- We could add **MULTIPLE** sets of instruction codes to **MULTIPLE** computers.



CyberShield Tool Chain (4)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Inst_code_1.pkg
prog_mem_1.vhd
data_mem_1.vhd
Inst_code_2.pkg
prog_mem_2.vhd
data_mem_2.vhd
Inst_code_3.pkg
prog_mem_3.vhd
data_mem_3.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
CYBERSHIELD
Core 0
RISC-V CPU
Prog Mem
Data Mem
Core 1
RISC-V CPU
Prog Mem
Data Mem
Core 2
RISC-V CPU
Prog Mem
Data Mem
Core 3
RISC-V CPU
Prog Mem
Data Mem
Instruction Register Malware Monitor
Malware Removal
Program, Data, & Configuration Memory Monitor/Repair
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory

The CyberShield Concept

- Why add more than two computing cores?
- That’s the flight computing component.

The CyberShield Concept

- Why add more than two computing cores?
- That’s the flight computing component.

UAS (RIS GPS OCX)

USA (RMD Patriot)

Missiles (RMD Patriot)

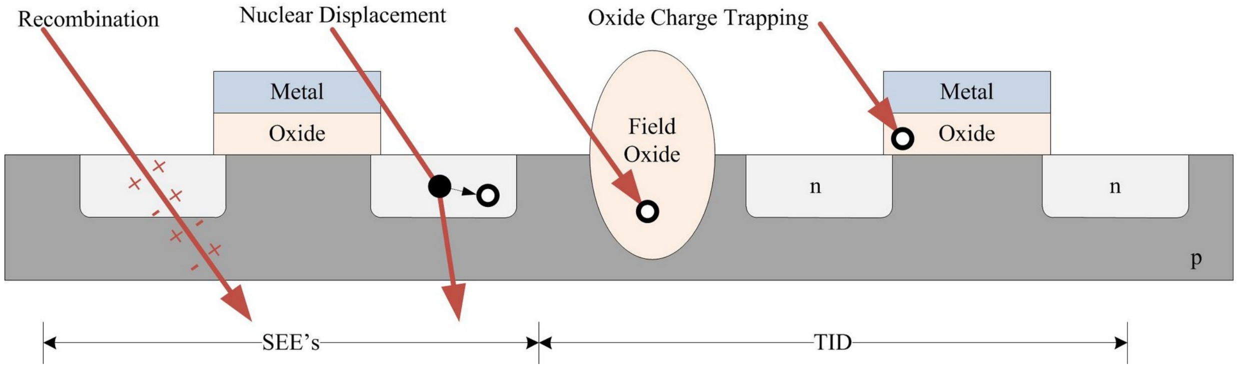
The CyberShield Concept

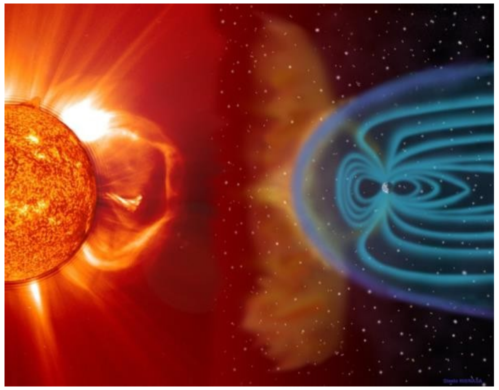
- Why add more than two computing cores?
- That's the flight computing component.

UAS (RIS GPS OCX)

USA (RMD Patriot)

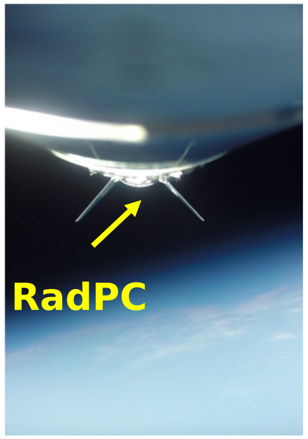
Missiles (RMD Patriot)

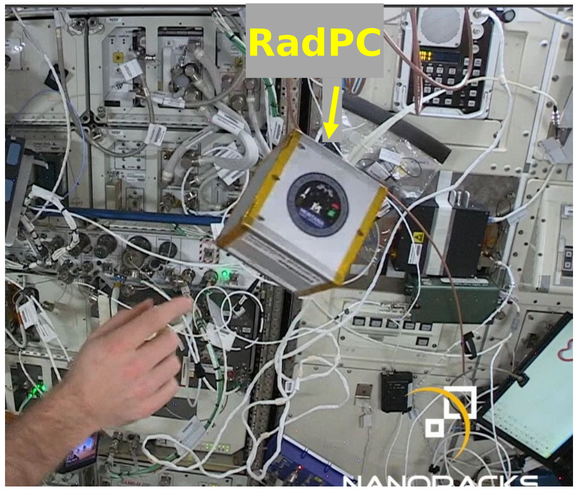


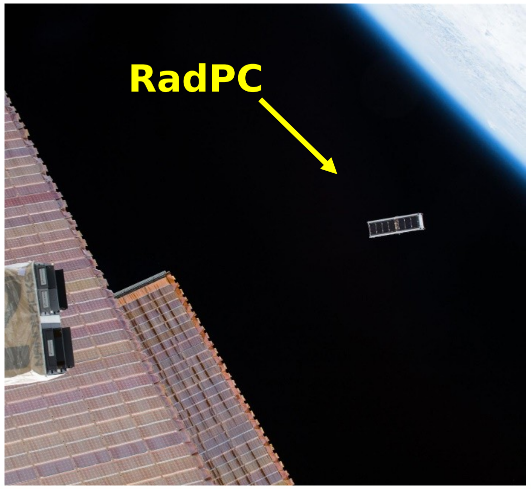


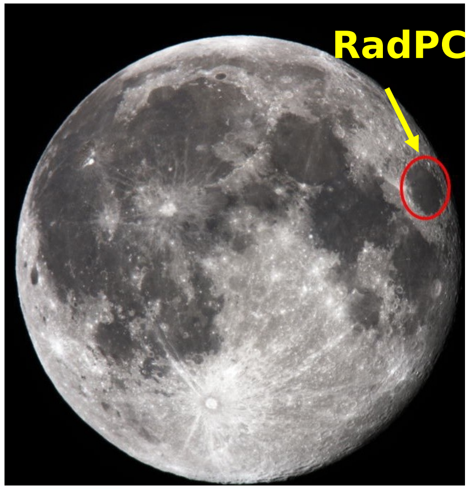
Spaceflight Computing

MSU has a history of building and deploying space computers.

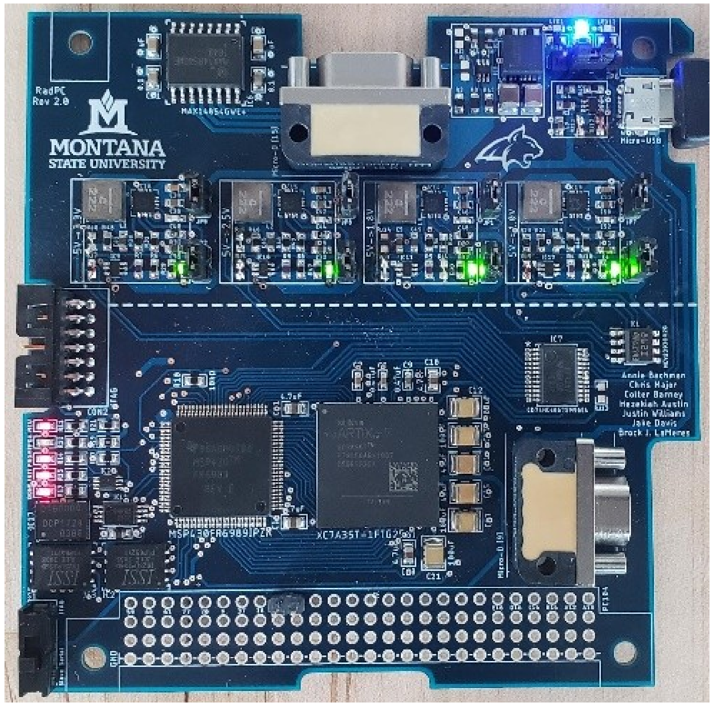


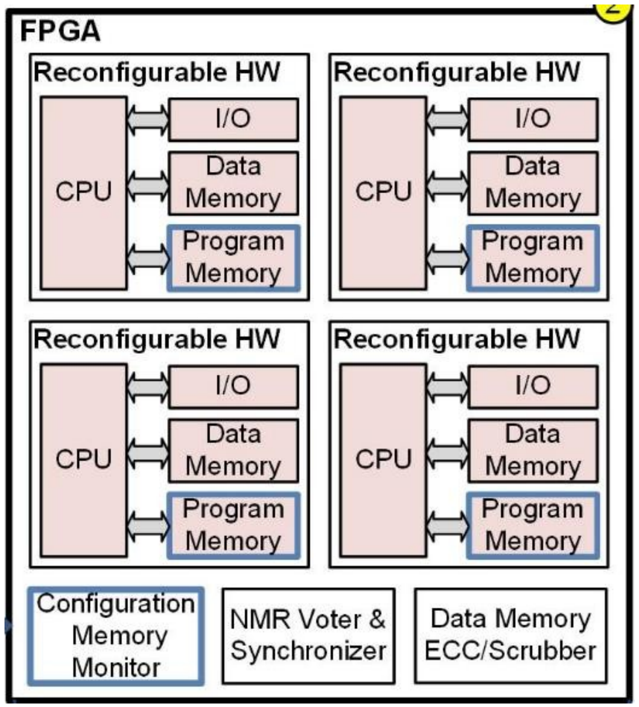




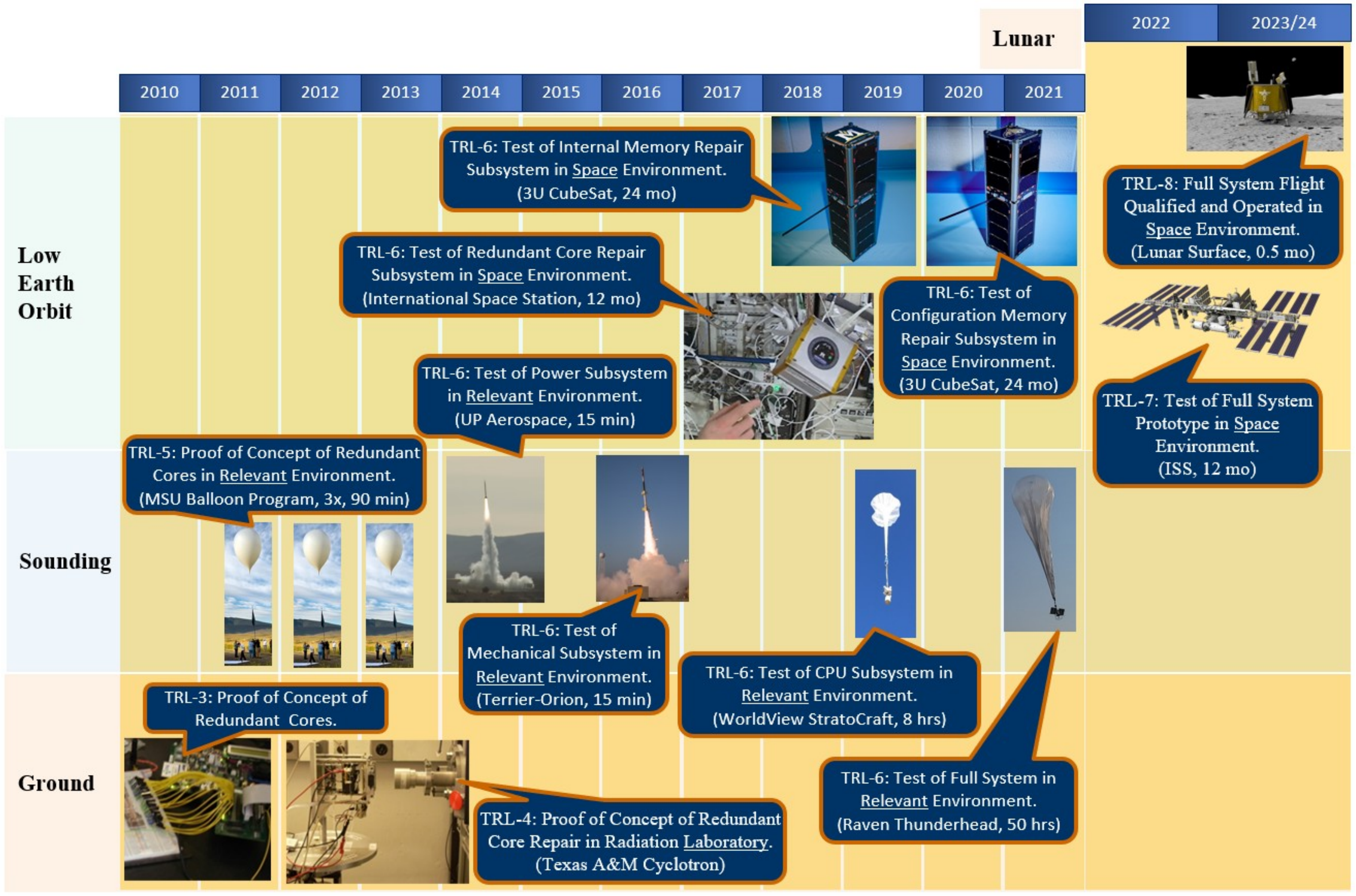


The RadPC Architecture





RadPC combines **reconfigurable logic** and **redundancy mechanisms** to protect program execution from radiation faults

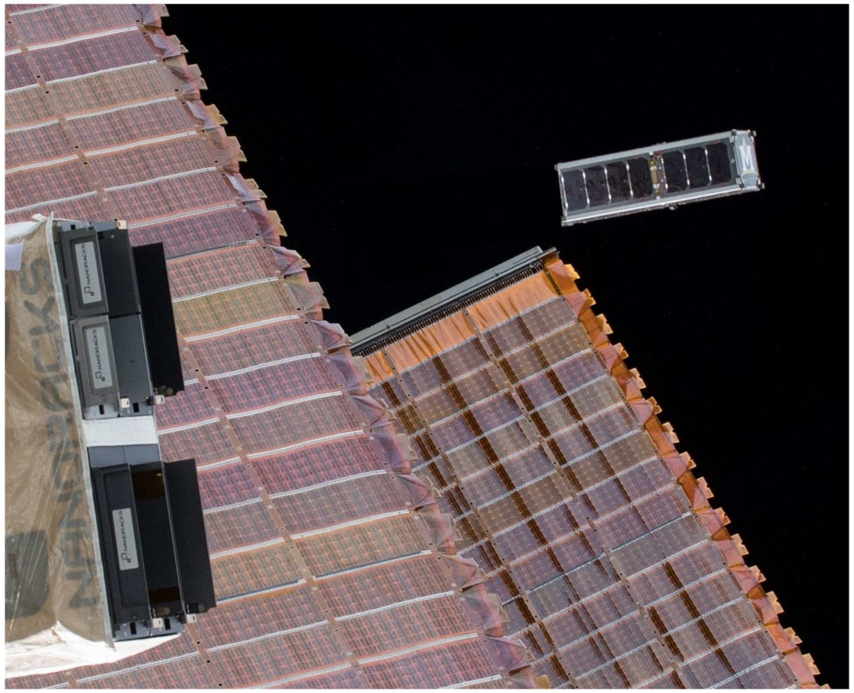

Lunar
2022
2023/24
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
Low Earth Orbit
Sounding
Ground
TRL-6: Test of Internal Memory Repair Subsystem in Space Environment. (3U CubeSat, 24 mo)
TRL-6: Test of Redundant Core Repair Subsystem in Space Environment. (International Space Station, 12 mo)
TRL-6: Test of Power Subsystem in Relevant Environment. (UP Aerospace, 15 min)
TRL-5: Proof of Concept of Redundant Cores in Relevant Environment. (MSU Balloon Program, 3x, 90 min)
TRL-6: Test of Configuration Memory Repair Subsystem in Space Environment. (3U CubeSat, 24 mo)
TRL-8: Full System Flight Qualified and Operated in Space Environment. (Lunar Surface, 0.5 mo)
TRL-7: Test of Full System Prototype in Space Environment. (ISS, 12 mo)
TRL-6: Test of Mechanical Subsystem in Relevant Environment. (Terrier-Orion, 15 min)
TRL-6: Test of CPU Subsystem in Relevant Environment. (WorldView StratoCraft, 8 hrs)
TRL-3: Proof of Concept of Redundant Cores.
TRL-4: Proof of Concept of Redundant Core Repair in Radiation Laboratory. (Texas A&M Cyclotron)
TRL-6: Test of Full System in Relevant Environment. (Raven Thunderhead, 50 hrs)

RadPC in Space

Timeline of RadPC missions and demonstrations

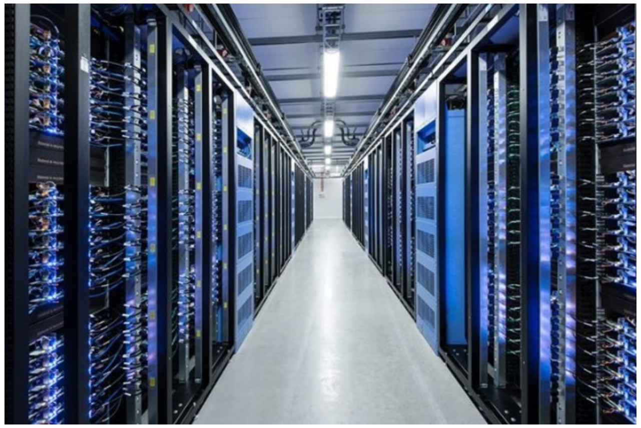
RadPC – Flight Heritage

- RadPC has extensive mission history and **flight heritage**.
  - 8 high altitude balloons
  - 2 sounding rockets
  - 3 International Space Station missions
  - 2 Cubesat satellites
  - 1 upcoming lunar mission
- Flight heritage determines the aerospace/defense industry's **confidence in implementing hardware and software**.



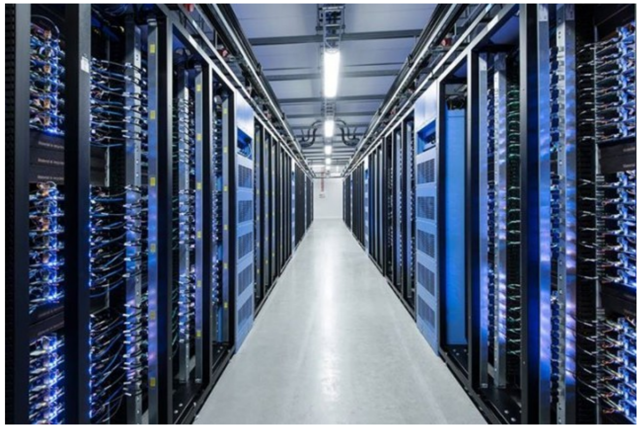
Fault Resilience

- What is the connection between **space radiation** and **malware**?



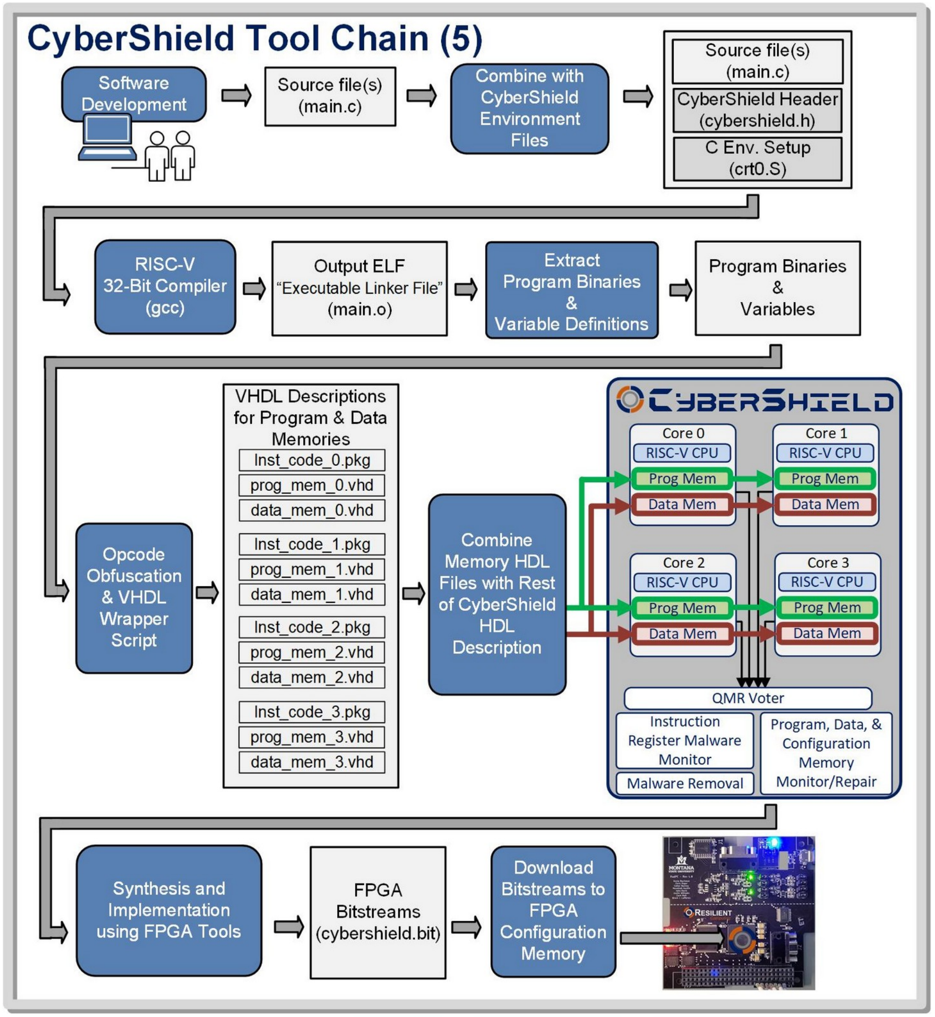
Fault Resilience

- What is the connection between **space radiation** and **malware**?
  - Unexpected error injection
  - Manipulation of program data
  - Mitigation through redundancy



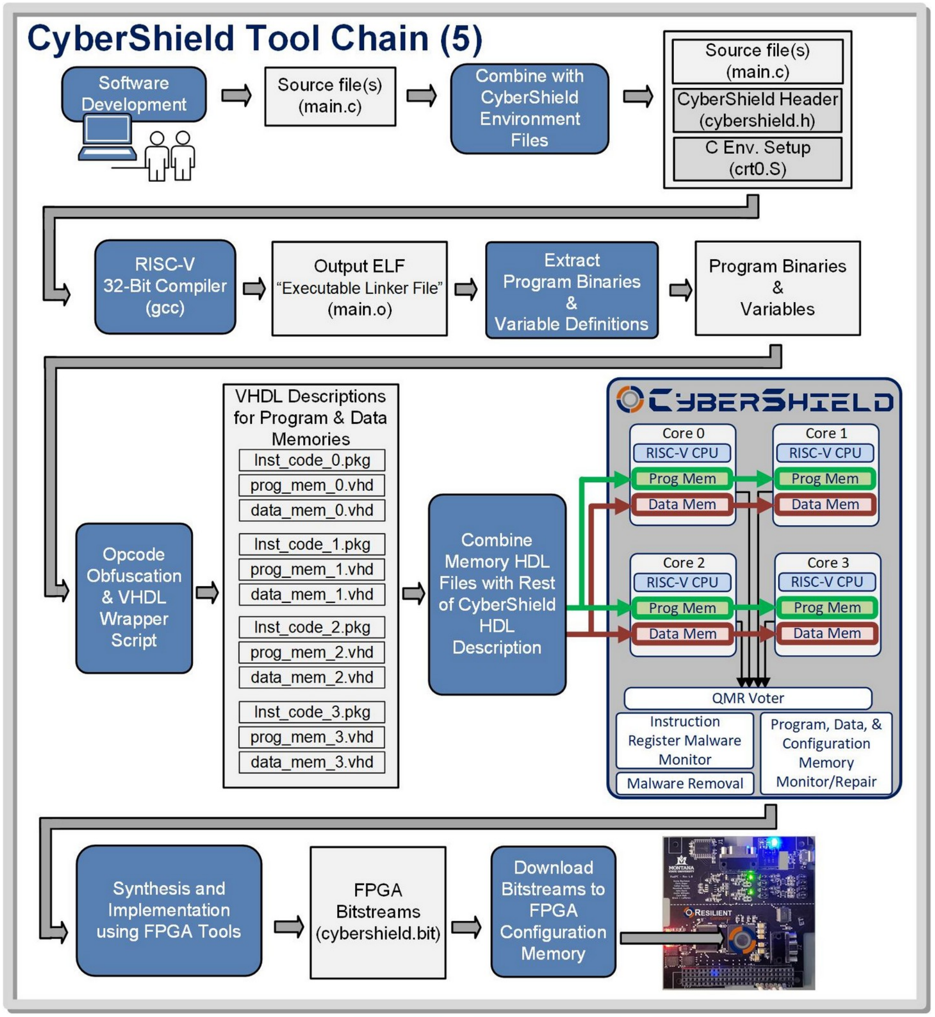
CyberShield + RadPC

- How do we leverage RadPC’s fault recovery mechanisms?

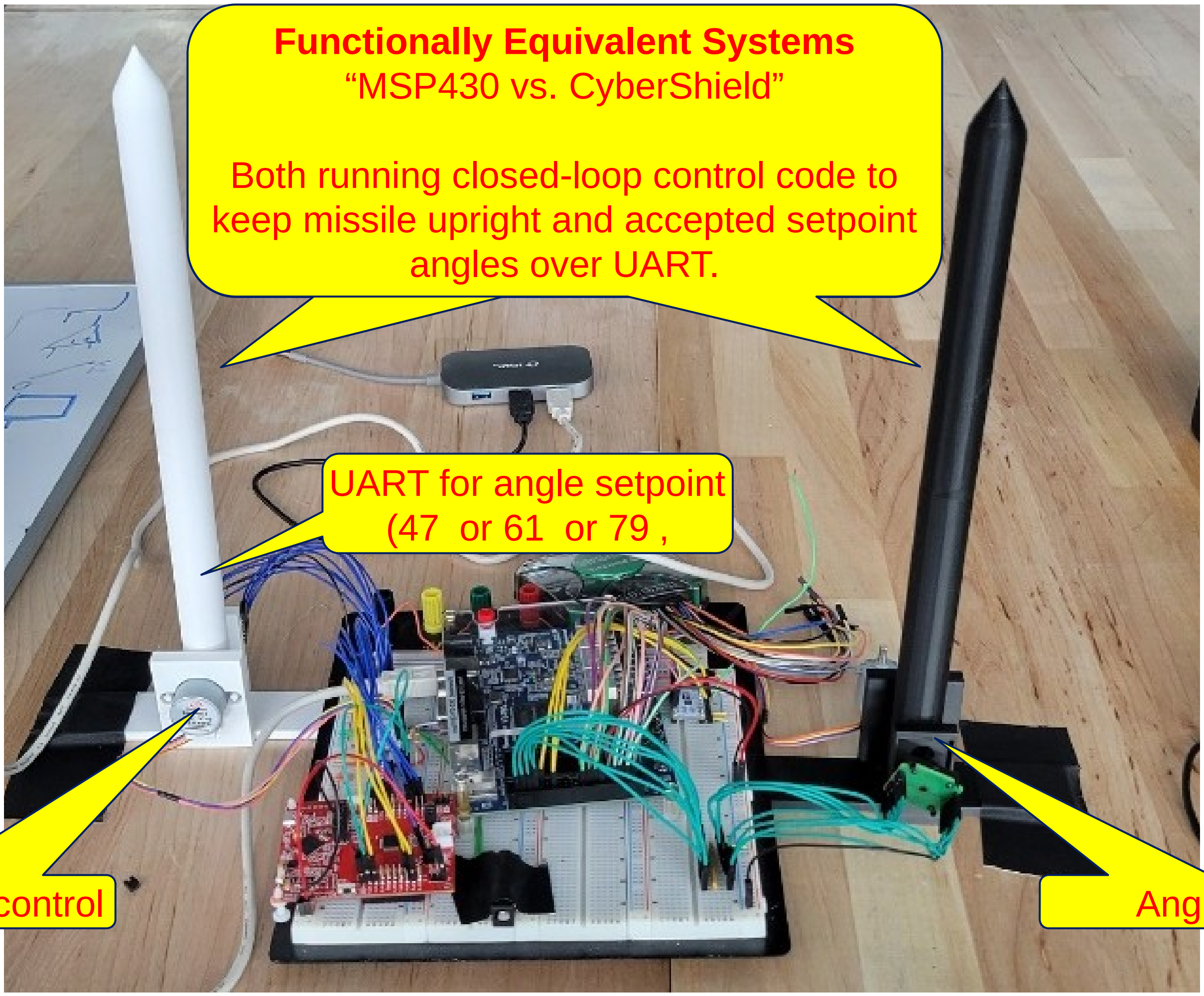

CyberShield Tool Chain (5)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Inst_code_1.pkg
prog_mem_1.vhd
data_mem_1.vhd
Inst_code_2.pkg
prog_mem_2.vhd
data_mem_2.vhd
Inst_code_3.pkg
prog_mem_3.vhd
data_mem_3.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
CYBERSHIELD
Core 0
Core 1
Core 2
Core 3
RISC-V CPU
Prog Mem
Data Mem
QMR Voter
Instruction Register Malware Monitor
Malware Removal
Program, Data, & Configuration Memory Monitor/Repair
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory

CyberShield + RadPC

- How do we leverage RadPC's fault recovery mechanisms?
  - Use the **Quad Modular Redundancy (QMR)** standard from RadPC to create 4 cores.
  - Ensure each core only understands its own opcodes.
  - Add a **voting mechanism** to determine which core has been attacked.
  - Add a **recovery mechanism** to refresh the faulted core in hardware.


CyberShield Tool Chain (5)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
Opcode Obfuscation & VHDL Wrapper Script
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Inst_code_1.pkg
prog_mem_1.vhd
data_mem_1.vhd
Inst_code_2.pkg
prog_mem_2.vhd
data_mem_2.vhd
Inst_code_3.pkg
prog_mem_3.vhd
data_mem_3.vhd
Combine Memory HDL Files with Rest of CyberShield HDL Description
CYBERSHIELD
Core 0
RISC-V CPU
Prog Mem
Data Mem
Core 1
RISC-V CPU
Prog Mem
Data Mem
Core 2
RISC-V CPU
Prog Mem
Data Mem
Core 3
RISC-V CPU
Prog Mem
Data Mem
QMR Voter
Instruction Register Malware Monitor
Malware Removal
Program, Data, & Configuration Memory Monitor/Repair
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory
MONTANA
RESILIENT

Malware Recovery Demonstration



Functionally Equivalent Systems
"MSP430 vs. CyberShield"

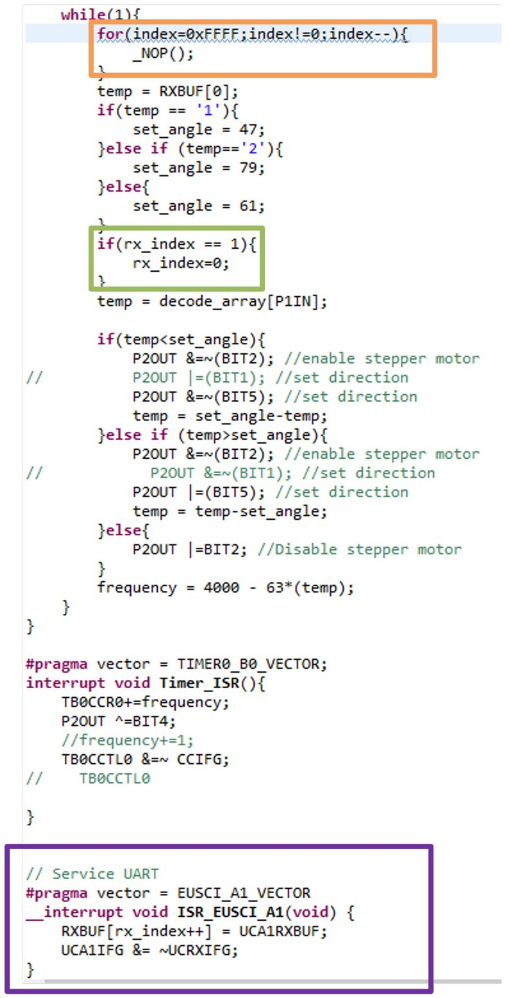
Both running closed-loop control code to keep missile upright and accepted setpoint angles over UART.

UART for angle setpoint (47 or 61 or 79 ,

Stepper motor for control

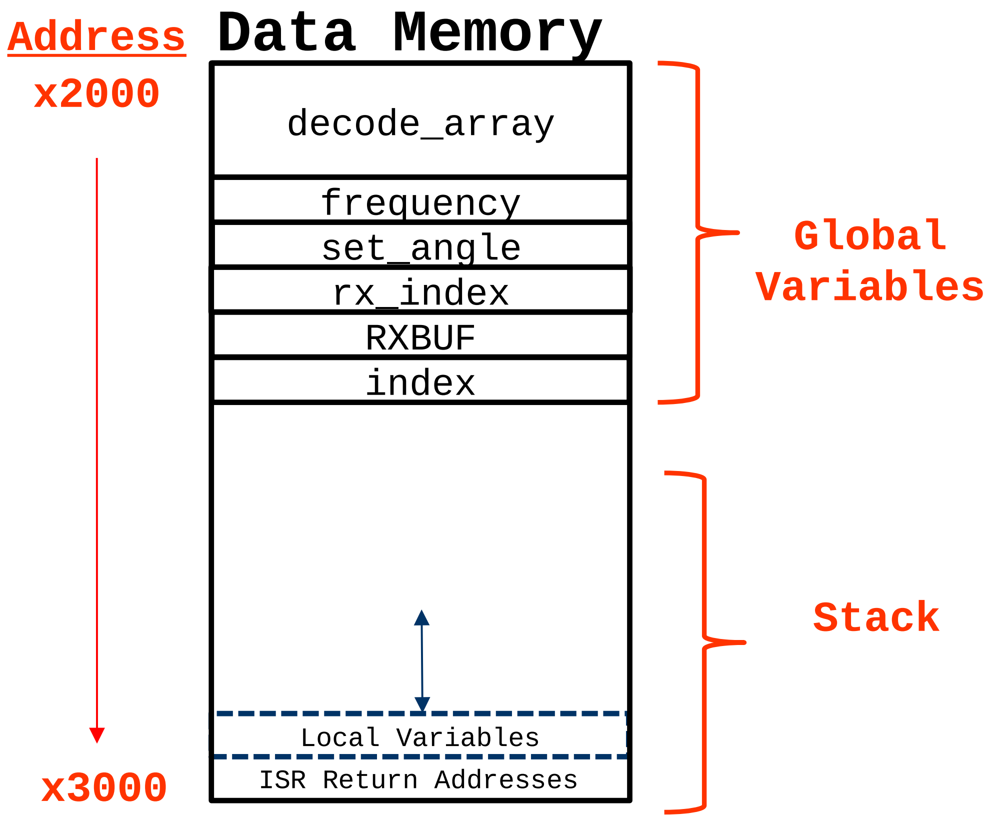
Angle sensor

Within The Memory Map ...

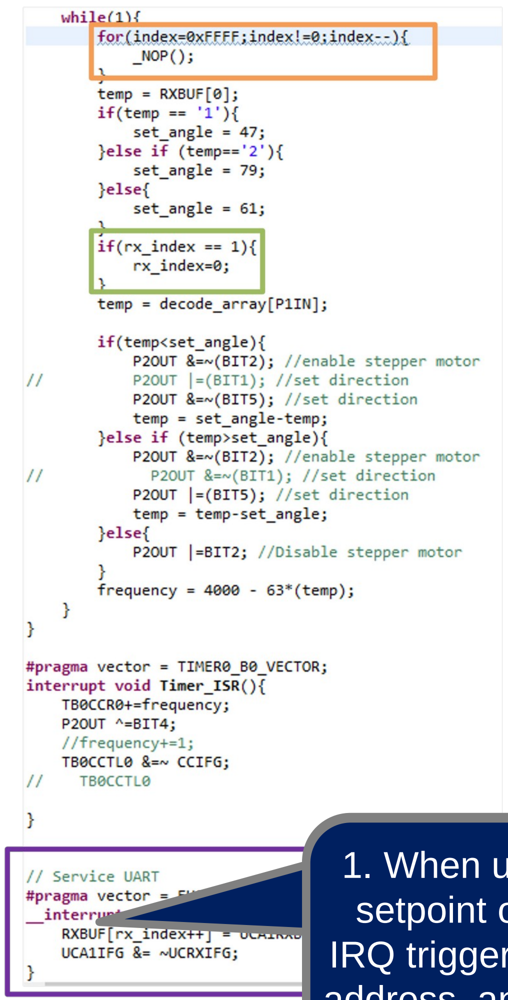


```
    while(1){
        for(index=0xFFFF;index!=0;index--){
            _NOP();
        }
        temp = RXBUF[0];
        if(temp == '1'){
            set_angle = 47;
        }else if (temp=='2'){
            set_angle = 79;
        }else{
            set_angle = 61;
        }
        if(rx_index == 1){
            rx_index=0;
        }
        temp = decode_array[P1IN];

        if(temp<set_angle){
            P2OUT &=~(BIT2); //enable stepper motor
//          P2OUT |=(BIT1); //set direction
            P2OUT &=~(BIT5); //set direction
            temp = set_angle-temp;
        }else if (temp>set_angle){
            P2OUT &=~(BIT2); //enable stepper motor
//            P2OUT &=~(BIT1); //set direction
            P2OUT |=(BIT5); //set direction
            temp = temp-set_angle;
        }else{
            P2OUT |=BIT2; //Disable stepper motor
        }
        frequency = 4000 - 63*(temp);
    }
}

#pragma vector = TIMER0_B0_VECTOR;
interrupt void Timer_ISR(){
    TB0CCR0+=frequency;
    P2OUT ^=BIT4;
    //frequency+=1;
    TB0CCTL0 &=~ CCIFG;
//    TB0CCTL0

}

// Service UART
#pragma vector = EUSCI_A1_VECTOR
__interrupt void ISR_EUSCI_A1(void) {
    RXBUF[rx_index++] = UCA1RXBUF;
    UCA1IFG &= ~UCRXIFG;
}
```

Program Vulnerabilities
(Classic Buffer Overflow Attack)



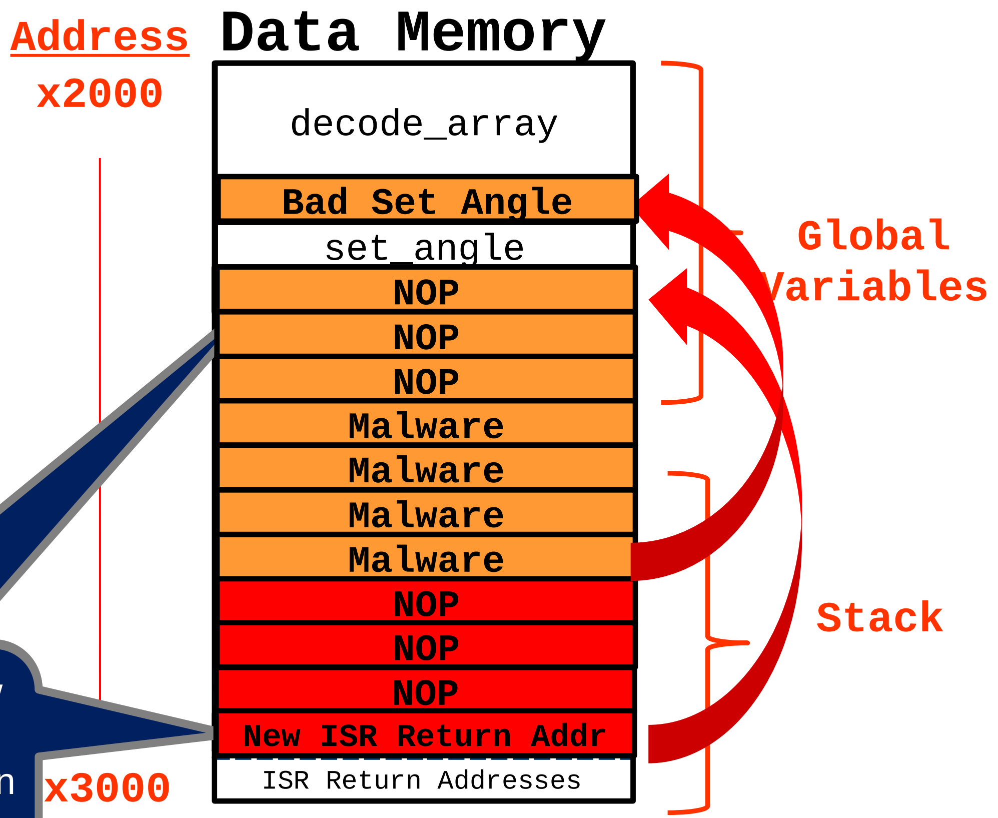
Within The Memory Map ...



```
    while(1){
        for(index=0xFFFF;index!=0;index--){
            _NOP();
        }
        temp = RXBUF[0];
        if(temp == '1'){
            set_angle = 47;
        }else if (temp=='2'){
            set_angle = 79;
        }else{
            set_angle = 61;
        }
        if(rx_index == 1){
            rx_index=0;
        }
        temp = decode_array[P1IN];

        if(temp<set_angle){
            P2OUT &=~(BIT2); //enable stepper motor
//          P2OUT |=(BIT1); //set direction
            P2OUT &=~(BIT5); //set direction
            temp = set_angle-temp;
        }else if (temp>set_angle){
            P2OUT &=~(BIT2); //enable stepper motor
//            P2OUT &=~(BIT1); //set direction
            P2OUT |=(BIT5); //set direction
            temp = temp-set_angle;
        }else{
            P2OUT |=BIT2; //Disable stepper motor
        }
        frequency = 4000 - 63*(temp);
    }
}

#pragma vector = TIMER0_B0_VECTOR;
interrupt void Timer_ISR(){
    TB0CCR0+=frequency;
    P2OUT ^=BIT4;
    //frequency+=1;
    TB0CCTL0 &=~ CCIFG;
//    TB0CCTL0

}

// Service UART
#pragma vector =
__interru
    RXBUF[rx_index++] = UCA1RXB
    UCA1IFG &= ~UCRXIFG;
}
```

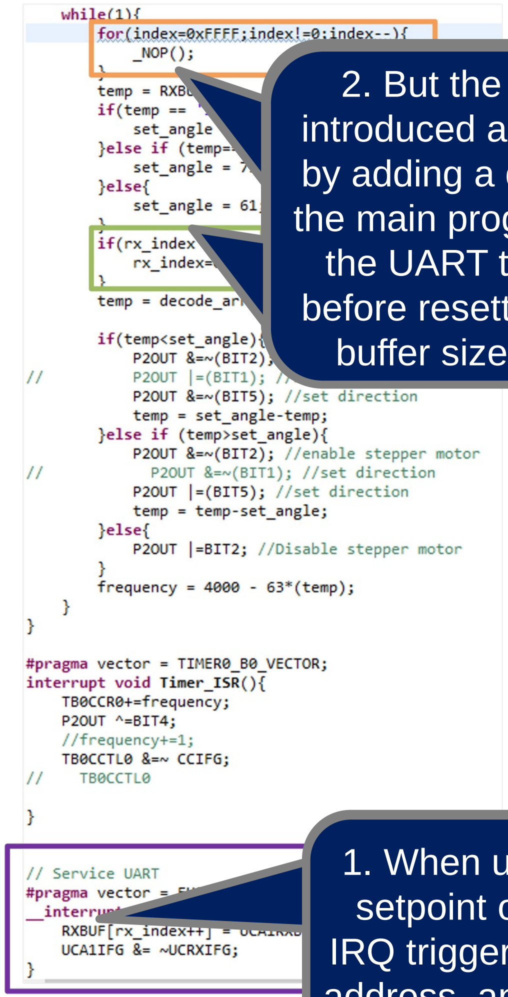
Program Vulnerabilities

<u>(Classic Buffer Overflow Attack)</u>



1. When user sends new setpoint over UART, an IRQ triggers, stacks return address, and retrieves new value for RXBUF.

Within The Memory Map ...

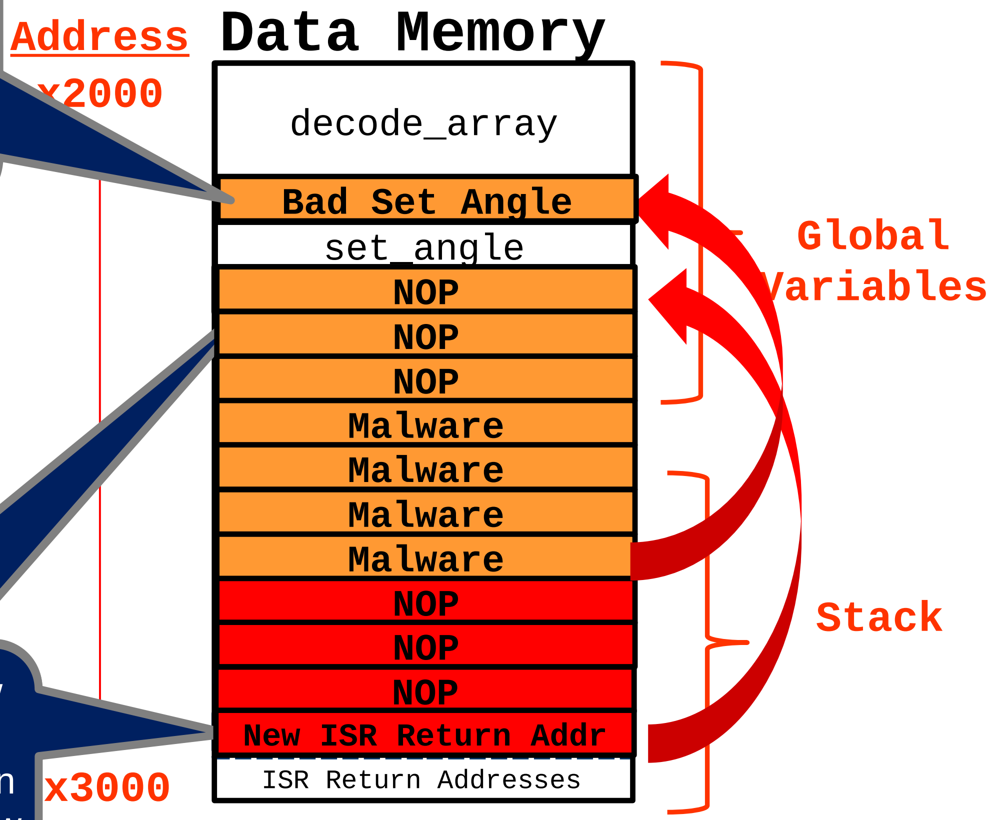


```
    while(1){
        for(index=0xFFFF;index!=0;index--){
            _NOP();
        }
        temp = RXBU
        if(temp == '
            set_angle
        }else if (temp==
            set_angle = 7
        }else{
            set_angle = 61
        }
        if(rx_index
            rx_index=
        }
        temp = decode_ar

        if(temp<set_angle){
            P2OUT &=~(BIT2);
//          P2OUT |=(BIT1); //
            P2OUT &=~(BIT5); //set direction
            temp = set_angle-temp;
        }else if (temp>set_angle){
            P2OUT &=~(BIT2); //enable stepper motor
//            P2OUT &=~(BIT1); //set direction
            P2OUT |=(BIT5); //set direction
            temp = temp-set_angle;
        }else{
            P2OUT |=BIT2; //Disable stepper motor
        }
        frequency = 4000 - 63*(temp);
    }
}

#pragma vector = TIMER0_B0_VECTOR;
interrupt void Timer_ISR(){
    TB0CCR0+=frequency;
    P2OUT ^=BIT4;
    //frequency+=1;
    TB0CCTL0 &=~ CCIFG;
//    TB0CCTL0

}

// Service UART
#pragma vector = EU
__interrupt
    RXBUF[rx_index++] = UCA1RXB
    UCA1IFG &= ~UCRXIFG;
}
```

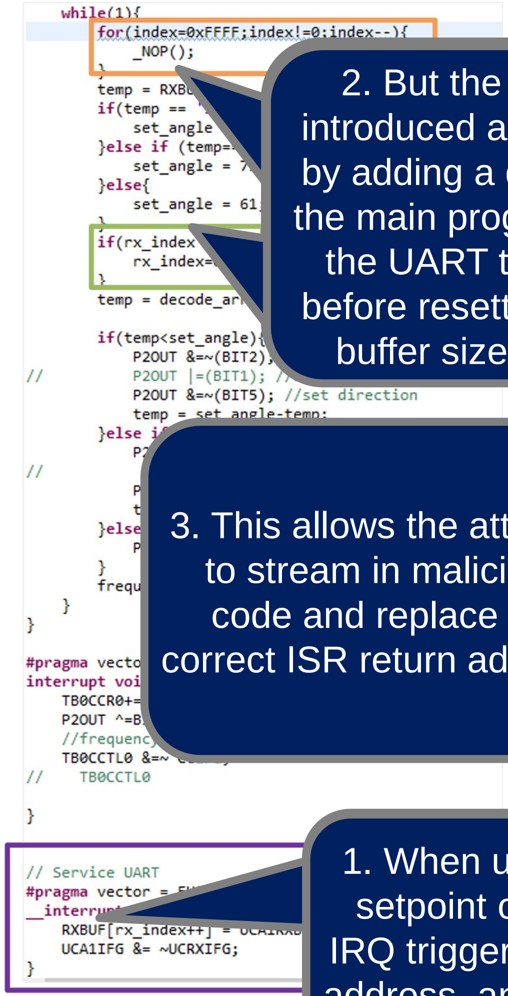
2. But the developer introduced a vulnerability by adding a delay loop in the main program to allow the UART to complete before resetting the input buffer size back to 0.

1. When user sends new setpoint over UART, an IRQ triggers, stacks return address, and retrieves new value for RXBUF.

Program Vulnerabilities
<u>(Classic Buffer Overflow Attack)</u>



Within The Memory Map ...



```
    while(1){
        for(index=0xFFFF;index!=0;index--){
            _NOP();
        }
        temp = RXBU
        if(temp == '
            set_angle
        }else if (temp==
            set_angle = 7
        }else{
            set_angle = 61;
        }
        if(rx_index
            rx_index=
        }
        temp = decode_ar

        if(temp<set_angle){
            P2OUT &=~(BIT2);
//          P2OUT |=(BIT1); //
            P2OUT &=~(BIT5); //set direction
            temp = set_angle-temp;
        }else i
            P2
//
            P
            t
        }else
            P
        }
        frequ
    }
}

#pragma vecto
interrupt voi
    TB0CCR0+=
    P2OUT ^=B
    //frequenc
    TB0CCTL0 &=~
//     TB0CCTL0

}

// Service UART
#pragma vector = 
__interru
    RXBUF[rx_index++] = UCA1RXB
    UCA1IFG &= ~UCRXIFG;
}
```

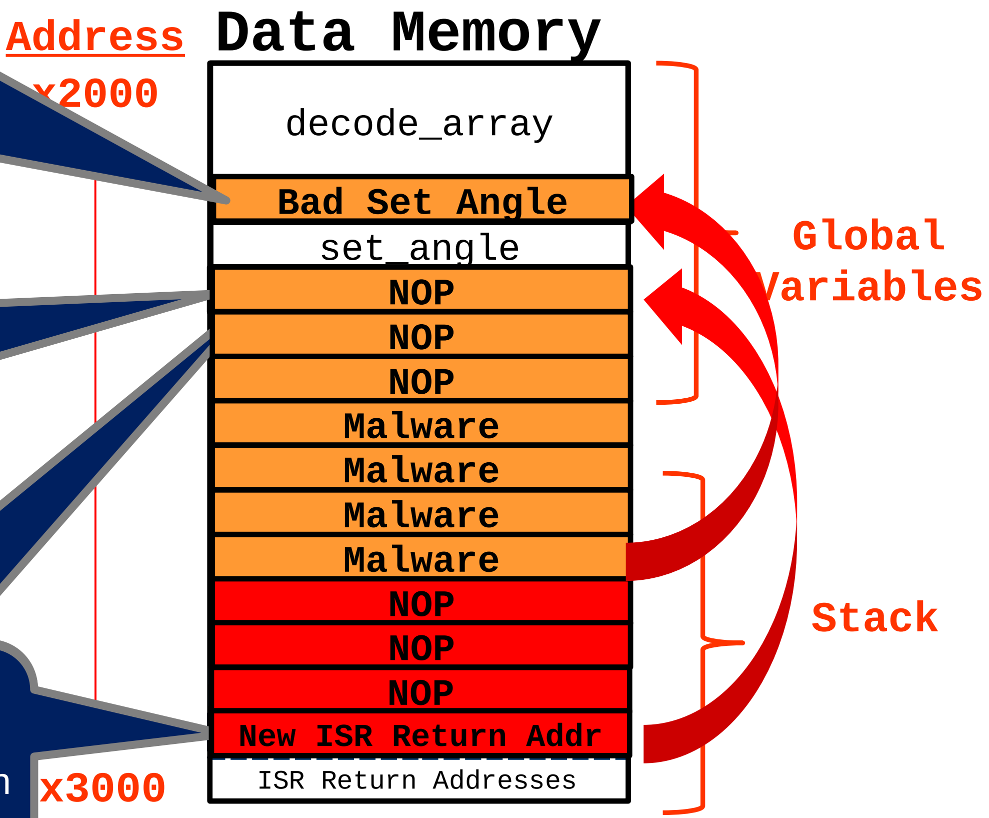
2. But the developer introduced a vulnerability by adding a delay loop in the main program to allow the UART to complete before resetting the input buffer size back to 0.

3. This allows the attacker to stream in malicious code and replace the correct ISR return address.

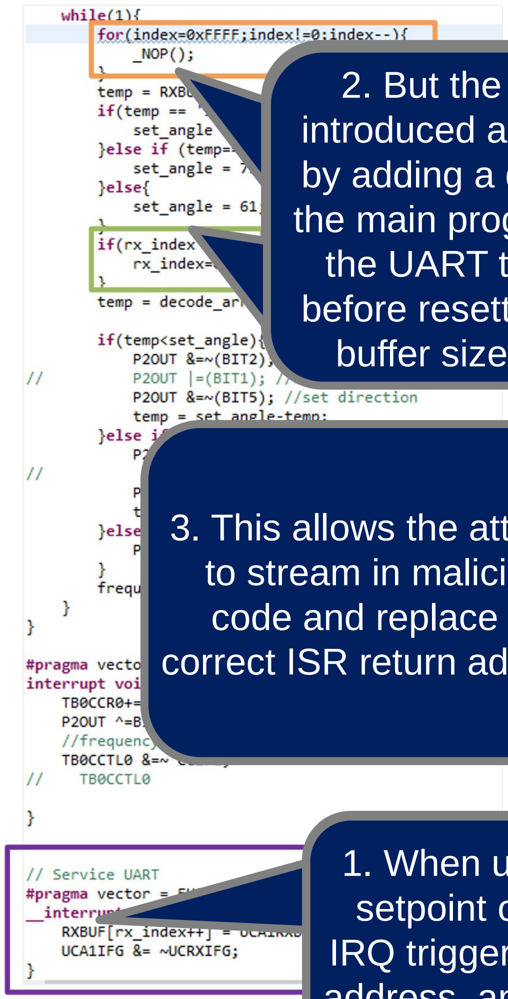
1. When user sends new setpoint over UART, an IRQ triggers, stacks return address, and retrieves new value for RXBUF.

Program Vulnerabilities

<u>(Classic Buffer Overflow Attack)</u>



Within The Memory Map ...



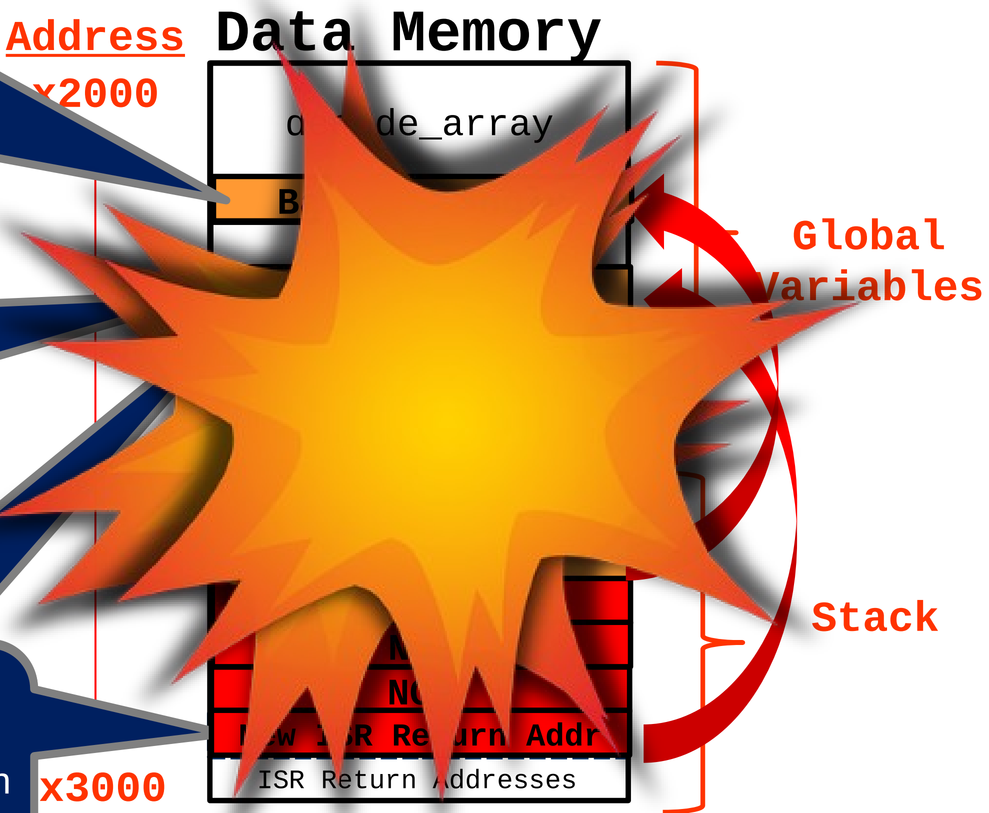
```
    while(1){
        for(index=0xFFFF;index!=0;index--){
            _NOP();
        }
        temp = RXBU
        if(temp ==
            set_angle
        }else if (temp=
            set_angle = 7
        }else{
            set_angle = 61;
        }
        if(rx_index
            rx_index=
        }
        temp = decode_ar

        if(temp<set_angle){
            P2OUT &=~(BIT2);
//          P2OUT |=(BIT1); //
            P2OUT &=~(BIT5); //set direction
            temp = set_angle-temp;
        }else i
            P2
//
            P
            t
        }else
            P
        }
        frequ
    }
}

#pragma vecto
interrupt voi
    TB0CCR0+=
    P2OUT ^=B
    //frequency
    TB0CCTL0 &=~
//    TB0CCTL0

}

// Service UART
#pragma vector =
__interrupt
    RXBUF[rx_index++] =
    UCA1IFG &= ~UCRXIFG;
}
```

2. But the developer introduced a vulnerability by adding a delay loop in the main program to allow the UART to complete before resetting the input buffer size back to 0.

3. This allows the attacker to stream in malicious code and replace the correct ISR return address.

1. When user sends new setpoint over UART, an IRQ triggers, stacks return address, and retrieves new value for RXBUF.

Program Vulnerabilities
(Classic Buffer Overflow Attack)



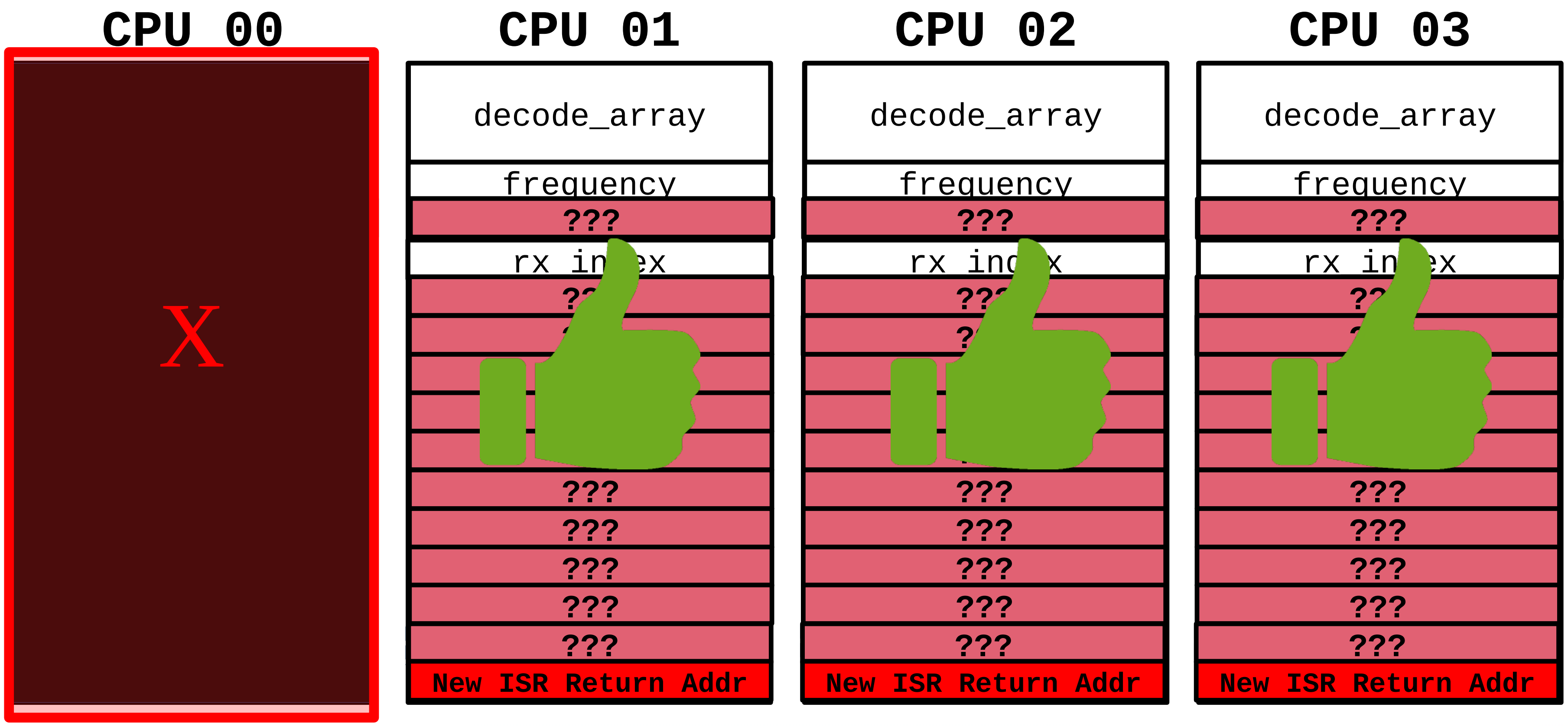
Redundancy Advantage

- Each CPU’s memory has the **exact same malware opcodes** ...

CPU 00	CPU 01	CPU 02	CPU 03
decode_array	decode_array	decode_array	decode_array
frequency	frequency	frequency	frequency
Bad Set Angle	**???**	**???**	**???**
rx_index	rx_index	rx_index	rx_index
NOP	**???**	**???**	**???**
NOP	**???**	**???**	**???**
NOP	**???**	**???**	**???**
Malware	**???**	**???**	**???**
Malware	**???**	**???**	**???**
Malware	**???**	**???**	**???**
Malware	**???**	**???**	**???**
NOP	**???**	**???**	**???**
NOP	**???**	**???**	**???**
NOP	**???**	**???**	**???**
New ISR Return Addr	**New ISR Return Addr**	**New ISR Return Addr**	**New ISR Return Addr**

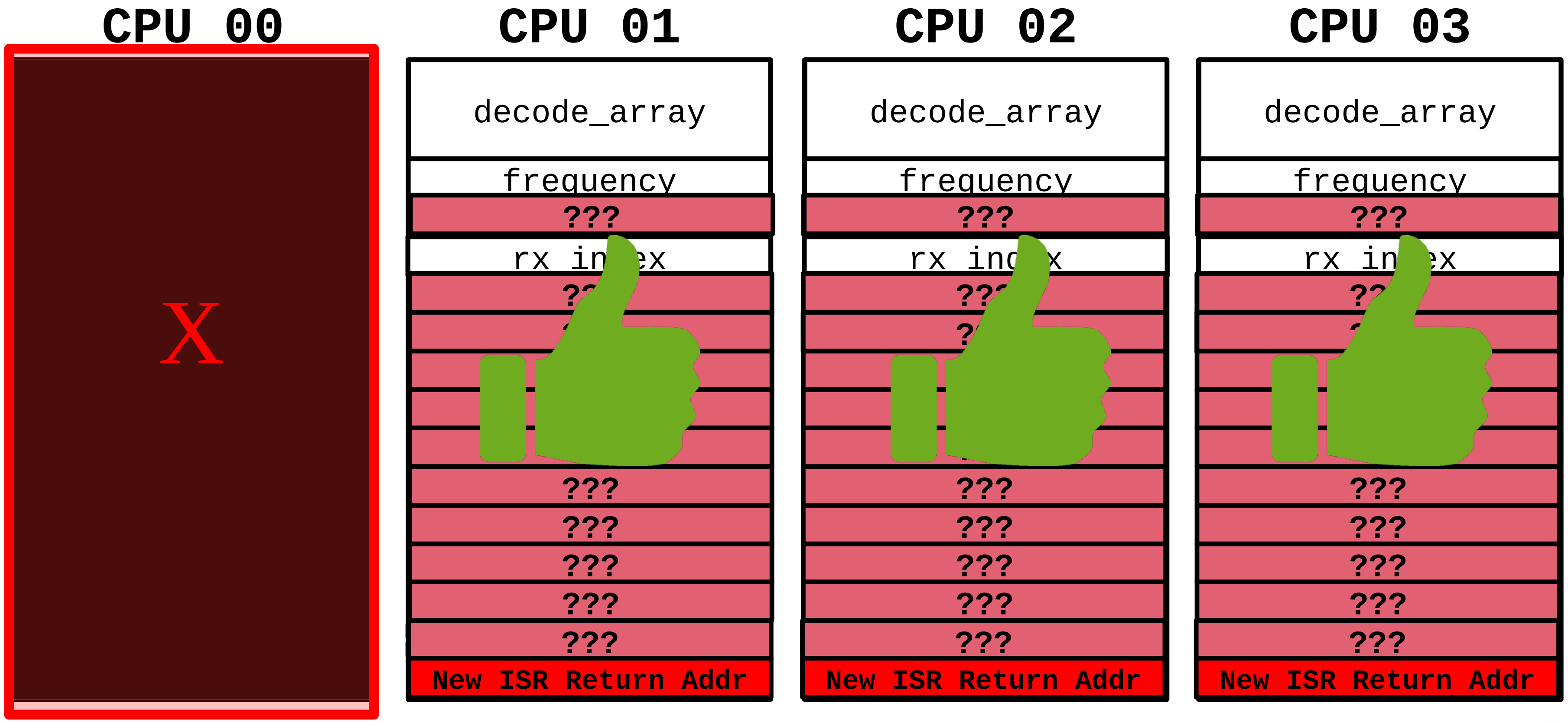
Redundancy Advantage

- … but only **one core** can actually run the malware.



Redundancy Advantage

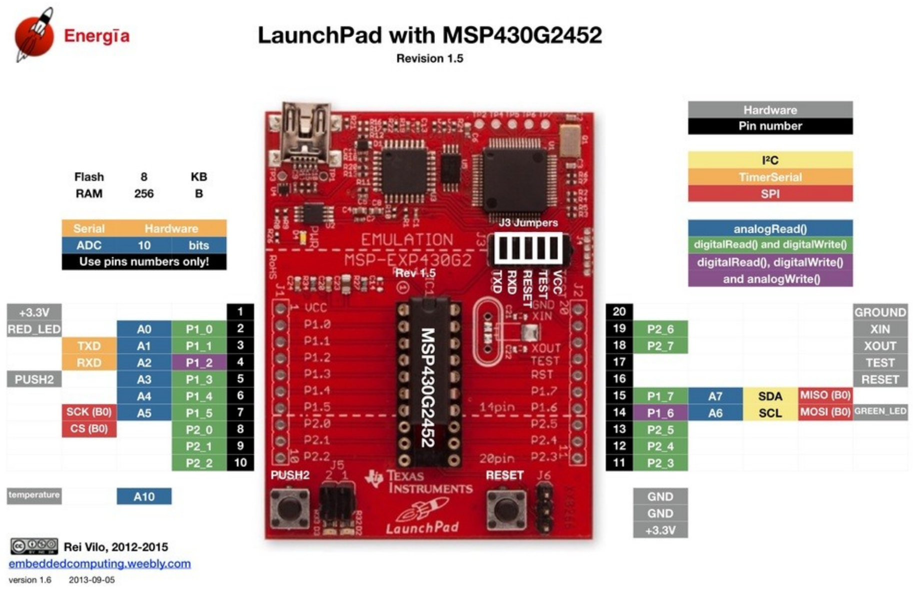
- So what kind of processor core should be used for this strategy?



MSP430

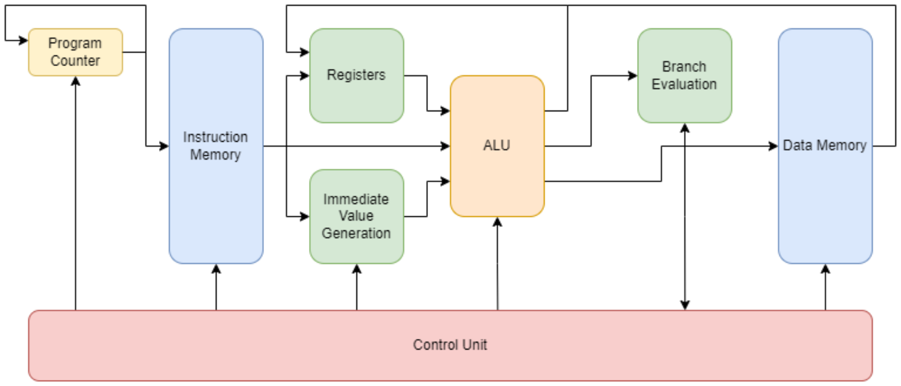
- The **MSP430** is a 16-bit microcontroller provided by Texas Instruments.
- The opcodes are widely available due to public documentation.
- The processor's functions are proprietary and difficult to replicate without insider access.
- No open-source softcore versions for FPGAs exist.

TEXAS INSTRUMENTS



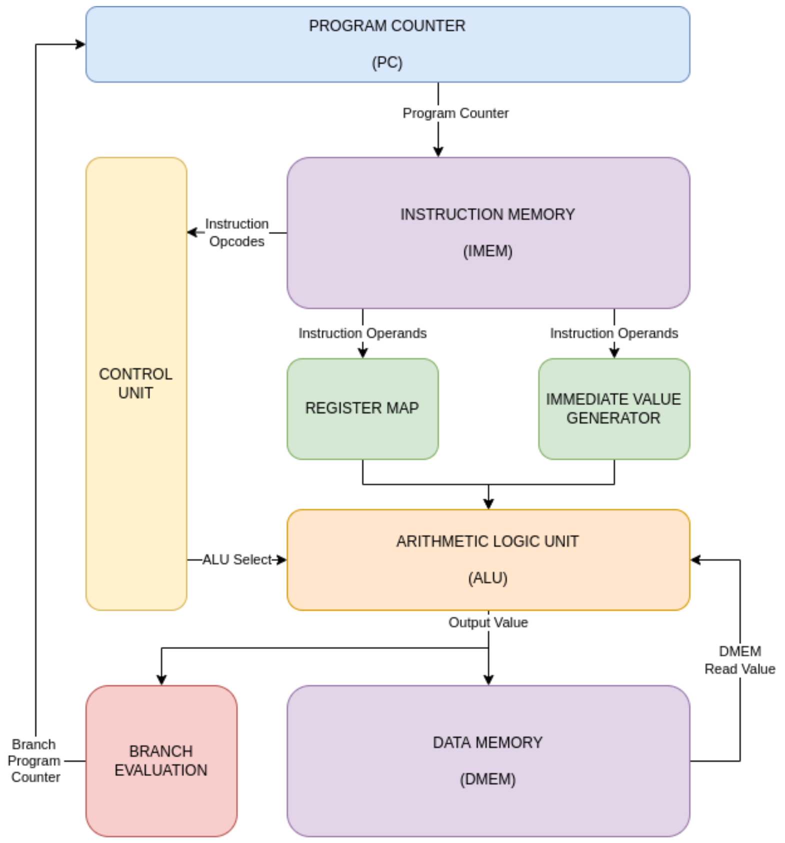
The RISC-V Processor

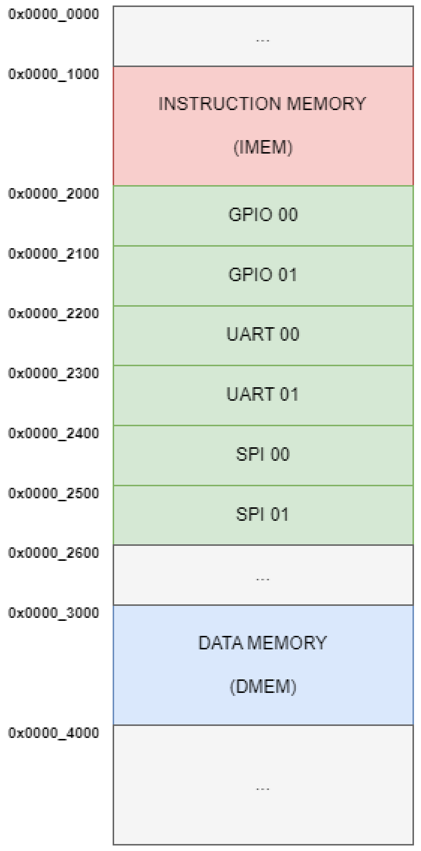
- RISC-V is an **open-source Industry Standard Architecture (ISA).**
- Base form is a 32-bit integer-based core **(RV32I)**.
- RISC-V cores can be implemented onto FPGAs with **easy access to signals.**
- A design must conform to the RISC-V ISA to be considered functional.



Our RISC-V HDL Core

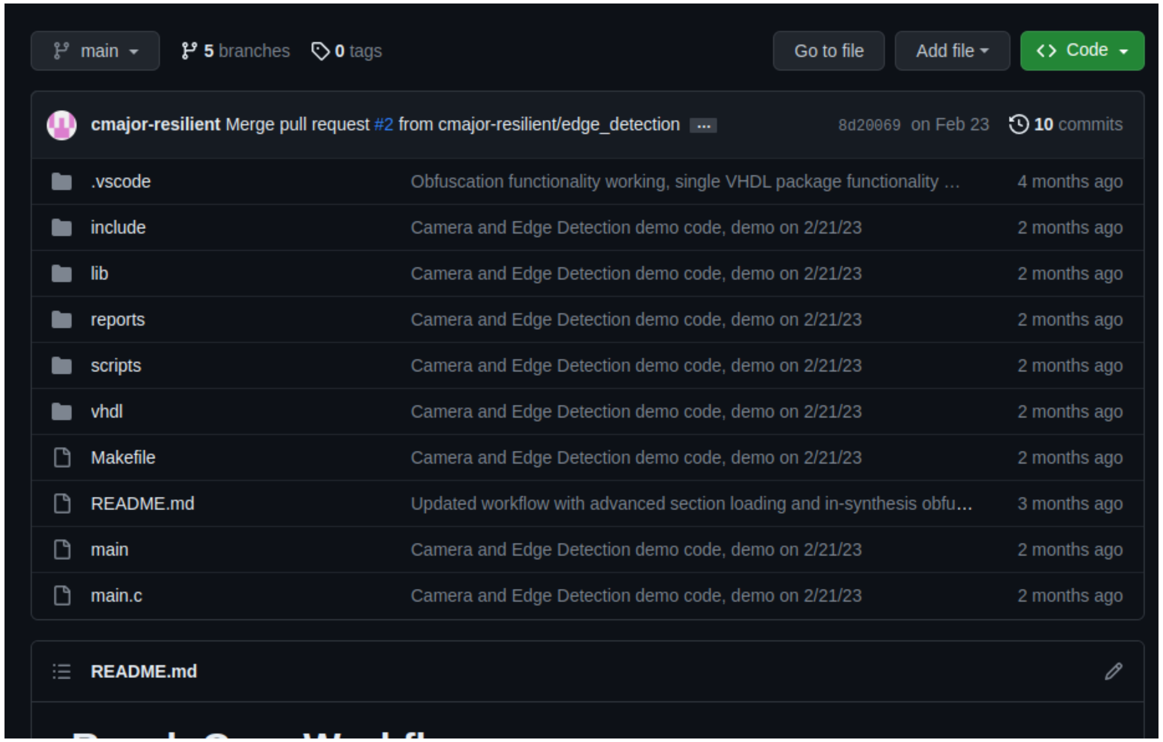
- RISC-V RV32I ISA
  - 32-bit registers
  - Integer Operations
- VHDL Components
- 4k IMEM, 4k DMEM
- 2 GPIO, 2 UART, 2 SPI





Software Development

- The **RV32I compiler** is provided by the RISC-V Collaboration
- Our custom **Makefile** compiles C into binaries, then exports binaries to VHDL.
- Minimal standard libraries are currently supported (stdint.h) in the CyberShield system.



cmajor-resilient Merge pull request #2 from cmajor-resilient/edge_detection ... 8d20069 on Feb 23 10 commits

.vscode	Obfuscation functionality working, single VHDL package functionality ...	4 months ago
include	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago
lib	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago
reports	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago
scripts	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago
vhdl	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago
Makefile	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago
README.md	Updated workflow with advanced section loading and in-synthesis obfu...	3 months ago
main	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago
main.c	Camera and Edge Detection demo code, demo on 2/21/23	2 months ago

README.md

Image Processing Demonstration

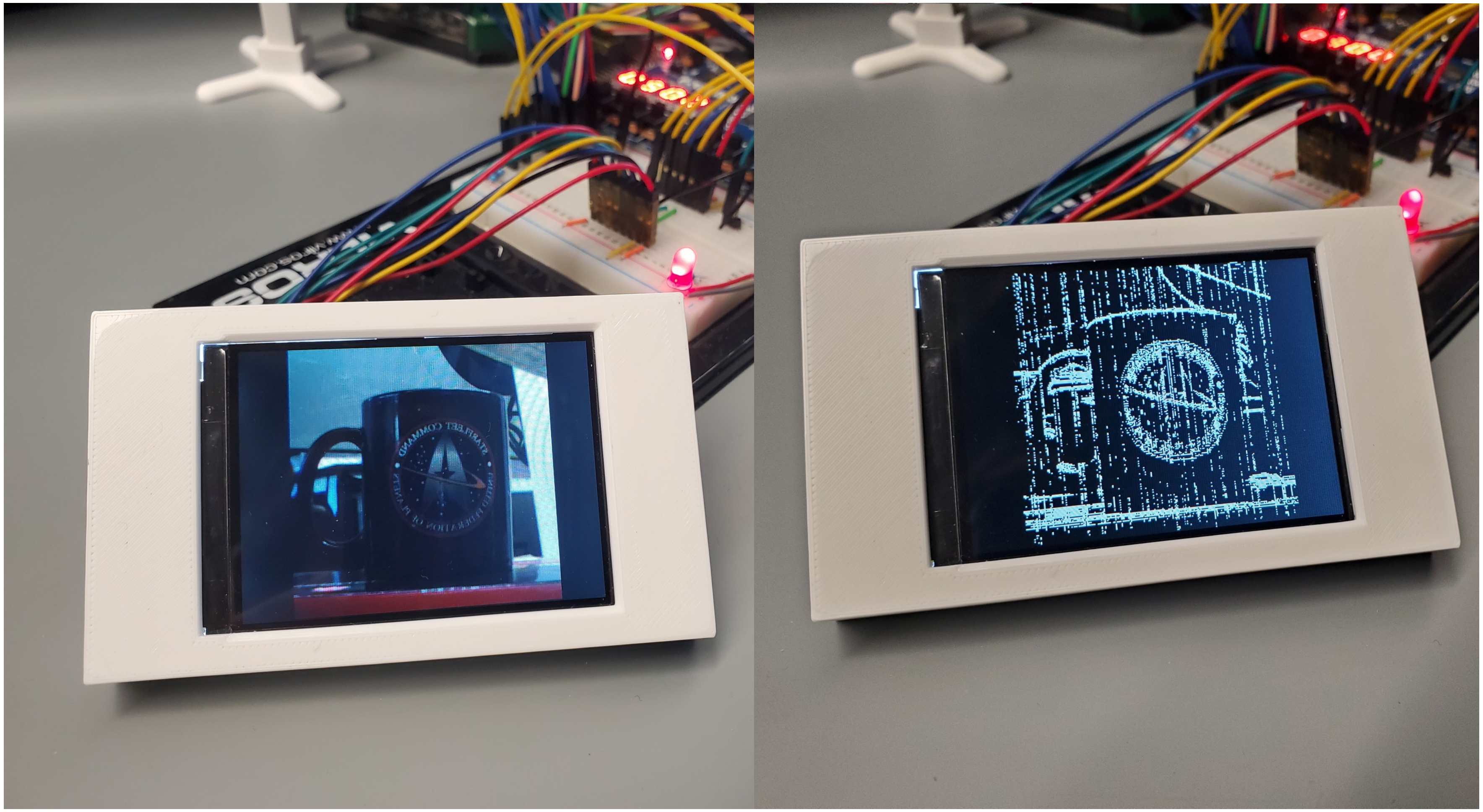
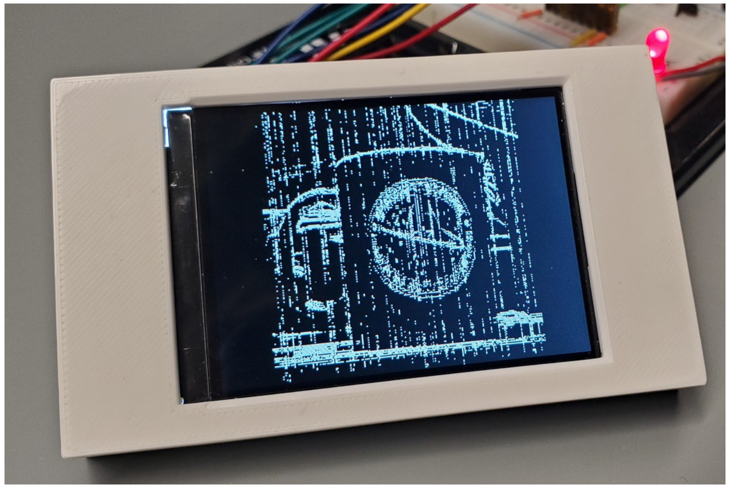


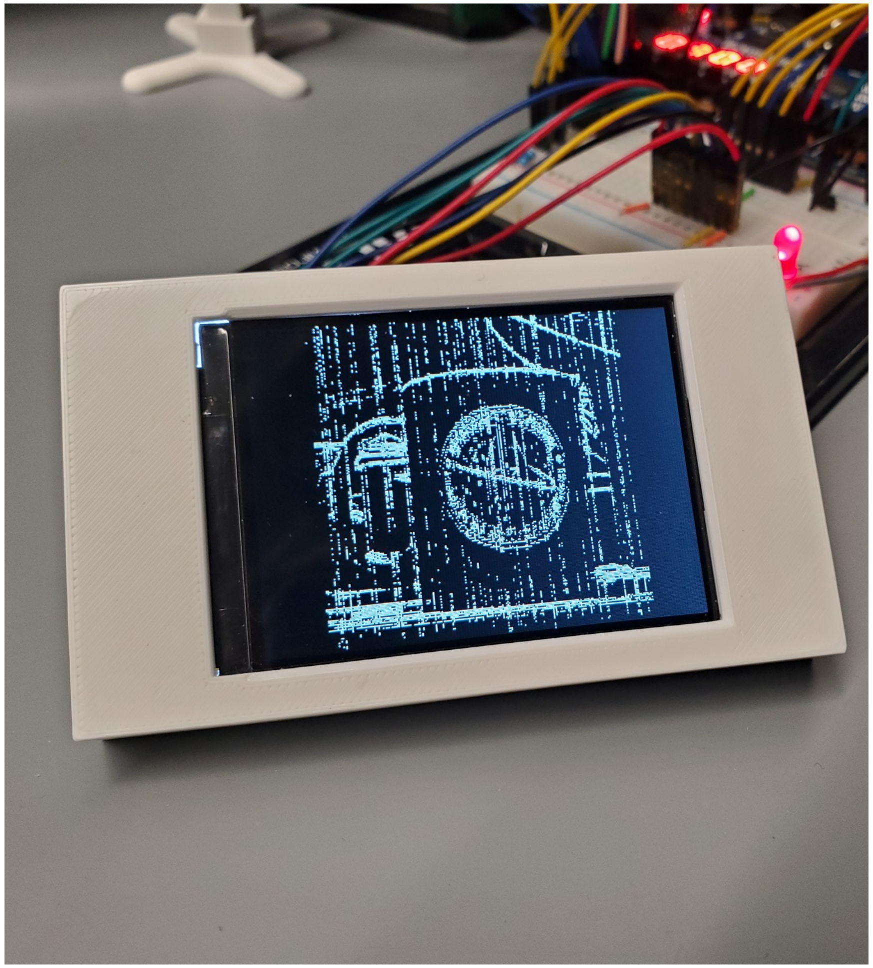
Image Processing Demonstration

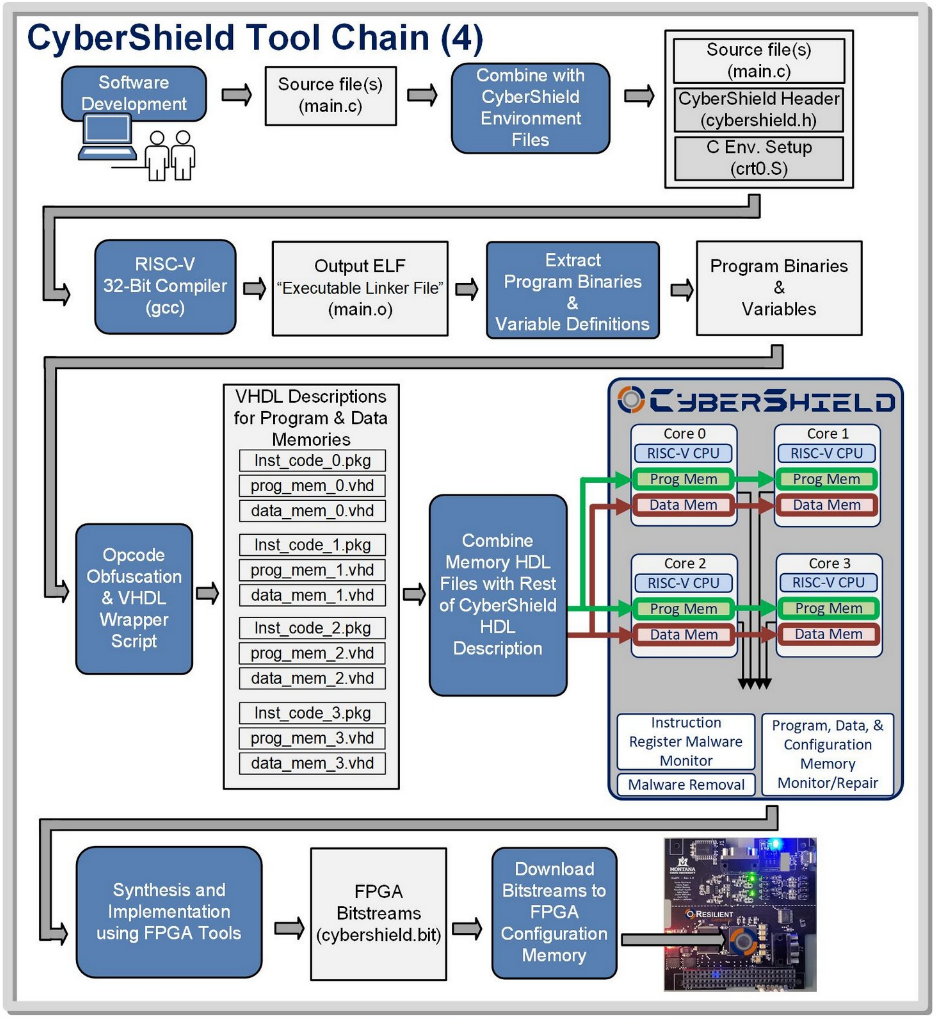
- We use the **RISC-V** architecture to process an image from a camera.
- A **UART command** specifies if the screen should show the image or the edge detected version.
- The UART is our **attack vector** for our buffer overflow attack.
- Each processor core has **different opcodes** but the same attack vector.



Demonstration Video

Questions?





CyberShield Tool Chain (4)
Software Development
Source file(s) (main.c)
Combine with CyberShield Environment Files
Source file(s) (main.c)
CyberShield Header (cybershield.h)
C Env. Setup (crt0.S)
RISC-V 32-Bit Compiler (gcc)
Output ELF "Executable Linker File" (main.o)
Extract Program Binaries & Variable Definitions
Program Binaries & Variables
VHDL Descriptions for Program & Data Memories
Inst_code_0.pkg
prog_mem_0.vhd
data_mem_0.vhd
Inst_code_1.pkg
prog_mem_1.vhd
data_mem_1.vhd
Inst_code_2.pkg
prog_mem_2.vhd
data_mem_2.vhd
Inst_code_3.pkg
prog_mem_3.vhd
data_mem_3.vhd
Opcode Obfuscation & VHDL Wrapper Script
Combine Memory HDL Files with Rest of CyberShield HDL Description
CYBERSHIELD
Core 0
RISC-V CPU
Prog Mem
Data Mem
Core 1
RISC-V CPU
Prog Mem
Data Mem
Core 2
RISC-V CPU
Prog Mem
Data Mem
Core 3
RISC-V CPU
Prog Mem
Data Mem
Instruction Register Malware Monitor
Malware Removal
Program, Data, & Configuration Memory Monitor/Repair
Synthesis and Implementation using FPGA Tools
FPGA Bitstreams (cybershield.bit)
Download Bitstreams to FPGA Configuration Memory