Field Stripping a Weapons System: Building a Trustworthy Computer

Dr. Gabriel L. Somlo
Document Markings

Copyright 2019 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.


DM19-1107
Q: Could you ask a vendor for full software and hardware sources to any system or solution contracted by the DoD, today?

Myth: “It is no longer possible for a single person to fully understand how a computer works.”
Field Stripping

From dictionary.com:
To take apart (a weapon) for cleaning, lubrication, and repair or for inspection
Field Stripping: What About Modern Weapons Systems?

Embedded Computers with *exotic* enclosures and peripherals, e.g.:
- artillery
- navigation
- comms

Non-destructive testing & reverse engineering is relatively easy with *software*
- less so with *microchips*!
Hardware Attack Surface

• ASIC Fabrication (Malicious Foundry)
  - masks reverse engineered and modified to insert malicious behavior
    • privilege escalation CPU backdoor
    • compromised random number generator
  - problematic to test/verify after the fact!
  - mitigated by using FPGAs instead!

• Compilation (Malicious Toolchain)
  - generates malicious design from clean sources

• Design Defects (Accidentally or Intentionally Buggy HDL Sources)
  - Spectre
  - Meltdown
Field Stripping a Computer

- **Applications (incl. compiler)**
- **System Runtime Libraries**
- **Kernel**
- **Hypervisor (optional)**
- **CPU ISA & I/O Registers**
- **Microarchitecture**
- **Register Transfer Level (RTL)**
- **ASICS**
- **FPGAs**

<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e.g., Linux, BSD, seL4)</td>
<td>(e.g., x86, ARM, RISC-V)</td>
</tr>
</tbody>
</table>

Self-hosting:
- a system’s capability to produce new versions of itself, from bounded sources, without reliance on external third-party support*
- the software stack is self-hosting
  * Assuming the hardware can be trusted!!!
Field Stripping a Computer

<table>
<thead>
<tr>
<th>Applications (incl. compiler)</th>
<th>Software (e.g., Linux, BSD, seL4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Runtime Libraries</td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td></td>
</tr>
<tr>
<td>Hypervisor (optional)</td>
<td></td>
</tr>
<tr>
<td>CPU ISA &amp; I/O Registers</td>
<td>Hardware (e.g., x86, ARM, RISC-V)</td>
</tr>
<tr>
<td>Microarchitecture</td>
<td></td>
</tr>
<tr>
<td>Register Transfer Level (RTL)</td>
<td></td>
</tr>
<tr>
<td>ASICs</td>
<td>FPGAs</td>
</tr>
</tbody>
</table>

Self-hosting:
- a system’s capability to produce new versions of itself, from bounded sources, without reliance on external third-party support*
- the software stack is self-hosting
  * Assuming the hardware can be trusted!!!

Goal: Extend self-hosting property to encompass hardware, including hardware source-language (HDL) compiler!
Hardware Development and Compilation Stages

Source Code

```verilog
module alu_mod (  
  // operator:   
  input alu_op_t op,     
  // operands:  
  input logic [31:0] a, b,  
  // result:    
  output logic [31:0] res);  

always_comb begin   
  unique case (op)  
    ALU_ADD: res = a + b;  
    ALU_MUL: res = a * b;  
    ALU_XOR: res = a ^ b;  
    ALU_AND: res = a & b;  
    ALU_OR : res = a | b;  
    default: res = 32'b0;  
  endcase  
endmodule: alu_mod  
```

Elaboration

Synthesis, Optimization

Technology Mapping, Place-and-Route

- Mask (ASIC)
- Bitstream (FPGA)

...01010101...
ASICS vs. FPGAs

- Application Specific Integrated Circuits
- dedicated, optimized etched silicon
  - photolithographic masks
- “hard” IP cores

- Field Programmable Gate Arrays
- grid: programmable blocks, interconnect
  - bitstream
- “soft” IP cores
Anchoring Trust for Fielded Systems

System Sources (blueprint, design, etc.)

Build Tool

Fielded System
Anchoring Trust for Fielded Systems

System Sources (blueprint, design, etc.)

Builder Sources

Build Tool

Fielded System
Anchoring Trust for Fielded Systems

Builder Sources

C Compiler

Build Tool

System Sources (blueprint, design, etc.)

Fielded System
Anchoring Trust for Fielded Systems

System Sources (blueprint, design, etc.) ➔ Build Tool ➔ Fielded System

Builder Sources ➔ C Compiler
Bootstrapping a Trustworthy RISC-V Cleanroom System

• [x86/Linux]: Use DDC to verify we have a clean C compiler
  - including a rv64 cross-compiler
• [x86/Linux]: Build clean HDL compiler toolchain, for both x86 and rv64
• [x86/Linux]: Cross-compile target rv64 OS (kernel, libraries, utilities)
• [x86/Linux]: Build rv64 SoC FPGA bitstream, from HDL sources

• [rv64/Linux]: Boot up FPGA-based rv64 computer into cross-compiled OS
  - rv64/Linux system is self-hosting from this point forward!
• [rv64/Linux]: Natively rebuild FPGA bitstream, kernel, libraries, and applications
  - we now have a trustworthy cleanroom
  - guaranteed to “honestly” compile any imported sources (HDL and/or software)!
List of Ingredients

Physical Hardware: FPGA development board (based on Lattice ECP5 series chip):
• Versa-5G or TrellisBoard

Free/Open HDL toolchain (Verilog-to-bitstream):
• Yosys (compiler), Project Trellis (bitstream utilities), NextPNR (place-and-route tool)

Free/Open RISC-V 64-bit CPU:
• Rocket Chip

Free/Open system-on-chip (SoC) environment (e.g., system bus, peripherals):
• LiteX

Free/Open software stack (e.g., Linux kernel, glibc runtime, GCC compiler):
• Fedora-riscv64
LiteX + Rocket 64-bit FPGA-based Linux Computer

- Memory (DRAM) Controller
- CPU (Rocket Chip)
- Peripheral Bus (MMIO)
- UART
- Eth.
- μSD

Diagram shows the integration of LiteX with the CPU and peripheral buses, highlighting components such as memory, UART, Ethernet, and microSD.
## Next Steps

### Performance Optimizations
- Early prototype HDL is a target-rich environment for further performance improvements, e.g.,:
  - 64bit AXI system bus
  - separate RAM and MMIO data paths

### Formal Analysis & Verification
- Starting from a *bounded* set of sources, 100% *as trustworthy as* the fielded system.
- Goal: measure *actual* ability to trust the system by conducting source code analysis!

### Hardware Assurance BCPs
- Cyber weapons as trustworthy as kinetic, despite supply chain complications!
Demo: Linux booting on Rocket+LiteX on ECP5 FPGA

#

(© Copyright 2012-2019 Enjoy-Digital
(c) Copyright 2007-2015 M-Labs Ltd

BIOS built on Aug 31 2019 22:04:19
BIOS CRC passed (37844d20)

Migen git sha1: -------
LiteX git sha1: 24410f0b

-------------- SoC info ---------------
CPU: RocketRV64[imac] @ 60MHz
ROM: 32KB
SRAM: 4KB
L2: 8KB
MAIN-RAM: 131072KB

-------------- Peripherals init ---------------