Field Stripping a Weapons System: Building a Trustworthy Computer

Gabriel L. Somlo, Ph.D.
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

Software
(e.g., Linux, BSD, seL4)

Hardware
(e.g., x86, ARM, RISC-V)

Applications (incl. compiler)

System Runtime Libraries

Kernel

Hypervisor (optional)

CPU ISA & I/O Registers

Microarchitecture

Register Transfer Level (RTL)

ASICs

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!!!

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></td>
</tr>
<tr>
<td>Microarchitecture</td>
<td>Hardware (e.g., x86, ARM, RISC-V)</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  
    end  
endmodule: alu_mod
```

Elaboration

![Elaboration Diagram]

Synthesis, Optimization

Technology Mapping, Place-and-Route

Mask (ASIC) or
Bitstream (FPGA)

Research Review 2019

Field Stripping a Weapons System: Building a Trustworthy Computer

© 2019 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.
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

Builder Sources

Build Tool

Fielded System

System Sources (blueprint, design, etc.)
Anchoring Trust for Fielded Systems

System Sources (blueprint, design, etc.)

Builder Sources

C Compiler

Build Tool

Fielded System
Anchoring Trust for Fielded Systems

System Sources (blueprint, design, etc.)

Builder Sources

C Compiler

Build Tool

Fielded System
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

Diagram showing the components:
- Memory (DRAM) Controller
- CPU (Rocket Chip)
- Peripheral Bus (MMIO)
- UART
- Eth.
- µSD

Diagram visualizes the connections between these components.
Next Steps

<table>
<thead>
<tr>
<th>NEAR</th>
<th>MID</th>
<th>FAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Optimizations</td>
<td>Formal Analysis &amp; Verification</td>
<td>Hardware Assurance BCPs</td>
</tr>
<tr>
<td>• Early prototype HDL is a target-rich environment for further performance improvements, e.g.,:</td>
<td>• Starting from a bounded set of sources, 100% as trustworthy as the fielded system.</td>
<td>• Cyber weapons as trustworthy as kinetic, despite supply chain complications!</td>
</tr>
<tr>
<td>• 64bit AXI system bus</td>
<td>• Goal: measure <em>actual</em> ability to trust the system by conducting source code analysis!</td>
<td></td>
</tr>
<tr>
<td>• separate RAM and MMIO data paths</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Demo: Linux booting on Rocket+LiteX on ECP5 FPGA
Backup Slides
Related Topics

Diminishing distinction between civilian and military/industrial security:

- Bruce Schneier blog post: [https://www.lawfareblog.com/myth-consumer-security](https://www.lawfareblog.com/myth-consumer-security)
- DoD ability to source trustworthy microchips drowned out by consumer market
- [https://youtu.be/1uCy-T22el8?t=132](https://youtu.be/1uCy-T22el8?t=132)

Right To Repair:

- [automobiles, electronics, agricultural machinery](https://www.lawfareblog.com/myth-consumer-security)
- issues of ownership, control, trust: all aspects of security
C Compiler vs. “Trusting Trust”: Problem and Workaround

• **self-propagating C compiler hack** (Ken Thompson)
  - malicious compiler inserts Trojan during compilation of a *victim program*
    • clean source → malicious binary
      • including compiler’s own sources!
      • compiler source hack *no longer needed* after 1st iteration!

• David A. Wheeler’s defense: **Diverse Double Compilation**
  - suspect compiler A: sources $S_A$, binary $B_A$
  - trusted compiler T: binary $B_T$
    $$S_A \rightarrow B_A \rightarrow X \quad S_A \rightarrow B_T \rightarrow Y$$
    • X and Y are functionally identical, but different binaries
    $$S_A \rightarrow X \rightarrow X_1 \quad S_A \rightarrow Y \rightarrow Y_1$$
    • $X_1$ and $Y_1$ must be identical binaries (since X, Y were functionally identical)!