# On the Design and Misuse of Microcoded (Embedded) Processors — A Cautionary Note

Nils Albartus[*†], Clemens Nasenberg[*†], Florian Stolz[*], Marc Fyrbiak[†], Christof Paar[*†] and Russell Tessier[‡]

[*]Ruhr University Bochum, Germany
[†]Max Planck Institute for Security and Privacy, Germany
[‡]University of Massachusetts, Amherst, USA

## Abstract

Today's microprocessors often rely on microcode updates to address issues such as security or functional patches. Unfortunately, microcode update flexibility opens up new attack vectors through malicious microcode alterations. Such attacks share many features with hardware Trojans and have similar devastating consequences for system security. However, due to microcode's opaque nature, little is known in the open literature about the capabilities and limitations of microcode Trojans.

We introduce the design of a microcoded RISC-V processor architecture together with a microcode development and evaluation environment. Even though microcode typically has almost complete control of the processor hardware, the design of meaningful microcode Trojans is not straightforward. This somewhat counter-intuitive insight is due to the lack of information at the hardware level about the semantics of executed software. In three security case studies we demonstrate how to overcome these issues and give insights on how to design meaningful microcode Trojans that undermine system security. To foster future research and applications, we publicly release our implementation and evaluation platform[1].

## 1 Introduction

Embedded systems are the technology behind the Internet of Things (IoT) and many other existing and emerging applications, ranging from smart appliances and medical implants to self-driving cars [12]. Since the life-span of embedded systems commonly stretches over years or even decades, they must offer flexibility with respect to both function and security. Even though most of today's embedded systems provide a mechanism to update software, some security problems cannot be handled with software updates alone and require changes to the underlying hardware [25]. To this end, hardware updates in the form of new microcode have been common practice in desktop and server systems for many years [7, 15, 29]. Microcode [36] can be viewed as an interpreter between the software-visible Instruction Set Architecture (ISA) and the internal hardware realization of the Central Processing Unit (CPU). Updated microcode provides a mechanism for efficient in-field hardware changes. The countermeasures for high-profile micro-architectural attacks, such as Spectre [20], are impressive examples of the security benefits offered by microcode [17]. Attack mitigation was possible via microcode updates for deployed hardware. However, microcode is not restricted to popular desktop/server CPUs. Some embedded processors incorporate updatable microcode, *e.g.*, the Intel Atom processor family. It is reasonable to assume that embedded microcode architectures will become increasingly common in the future, given the growing complexity and safety/security requirements of embedded systems, e.g., cyber-physical systems and the IoT.

Microprocessors, and other integrated circuits, are in almost all systems considered trusted, which has the unfortunate consequence that malicious low-level manipulations, e.g., through hardware Trojans, can lead to devastating security failures, cf., e.g., [13]. Hardware Trojans in Application Specific Integrated Circuits (ASICs) are static and lack post-manufacturing versatility, limiting their usefulness (from an attacker's perspective) in several ways. First, they cannot be erased once they are implemented, which imposes strict requirements on their stealthiness since they have to stay undetectable during the entire lifetime of the application. Second, typically all ASICs of a series are Trojan-equipped, which increases the risk that they will eventually be detected. Third, it is difficult to distribute the affected ASICs selectively: It can be attractive for an adversary to distribute weakened hardware only to a certain user population, e.g., only in government systems in a specific country. In contrast to hardware Trojans, microcode Trojans overcome these "drawbacks" (again, from an adversarial perspective) due to their adaptive nature. Microcode Trojans combine low-level hardware access with software-level flexibility, which results in two powerful key features: (1) they are dynamically programmable, and (2) they can be dynamically injected and removed via updates. At the same time, they share the potential to undermine system in-

[1]https://github.com/emsec/riscv-ucode

tegrity and security in the same devastating way as classical hardware Trojans, while simultaneously being extremely difficult to detect by current software defense measures [22]. These features make microcode Trojans attractive for large-scale adversaries such as nation-state actors.

In general, microcode architecture details are proprietary, and microcode updates are typically secured using strong cryptography. Update keys and the microcode implementation itself are among the most guarded secrets of CPU vendors. However, researchers recently demonstrated that both microcode cryptographic keys [10] and microcode implementations details [22] can be disclosed from commercial off-the-shelf (COTS) CPUs. In particular, the former work [10] demonstrated the successful extraction of decryption keys for Intel Atom, Celeron, and Pentium CPUs, so that in case of physical device access, custom microcode updates can be issued. The latter work [22] reverse-engineered significant parts of the microcode structure and microcode capabilities of AMD K8/K10 CPUs. Generally, there is no straightforward way to analyze microcode nor to identify the potential for malicious microcode updates.

**Goals and Contributions.** In this paper, we focus on the design and security implications of microcoded CPUs in embedded systems. Our goal is to assess the effectiveness, capabilities, and limitations of malicious microcode with respect to system security and cryptographic implementations. We must overcome the major challenge that even though a microcode Trojan designer has seemingly total control over a platform, he/she faces the conundrum of having limited information about the application and/or system-level software under execution.

Since, to the best of our knowledge, no suitable open-source implementation of a contemporary microcoded CPU and *associated* microcode development tools are available, we developed a microcoded architecture for the RISC-V ISA for experimentation and evaluated it on a Field Programmable Gate Array (FPGA)-based platform. This microcoded processor implementation is synthesizable, designed in an embedded system context with additional peripherals, and supports the entire RISC-V base instruction set (RV32I). It provides a realistic embedded platform for our microcode Trojan experimentation. Based on this platform, we make the following contributions:

- **Microcode Trojans.** We introduce a realistic adversary model for microcode attacks on modern (embedded) systems. We demonstrate the workflow of a Trojan designer and show how to overcome the “unlimited capabilities versus limited information” situation.

- **Capabilities and limitations of microcode Trojan exploits.** We describe representative microcode Trojans that circumvent a secure boot mechanism and two examples of symmetric crypto subversion through side-channel analysis and trigger-word based key leakage. Their threat potential and countermeasures are discussed.

- **Real-world relevance.** We maintain high practical relevance by injecting Trojans into widely-used software and firmware. We manipulate the verification check of the Chrome OS bootloader, insert an exploitable timing side-channel in popular constant-time AES implementations such as openSSL, and show how to leak the key for an architecture-specific implementation by inserting targeted faults.

- **Microcoded RISC-V evaluation platform.** We present the design and implementation of a microcoded RISC-V (RV32I) microprocessor implemented on an FPGA evaluation system. Our platform supports numerous tasks tailored to security engineering (e.g., prototyping for ISA extensions). To foster research and education in hardware security and computer architecture, our evaluation platform is publicly available.

## 2 Technical Background

In this section, we provide a systematic overview of the mechanics of microcode and its (mis-)uses in security applications. Moreover, we provide a brief background on (classical) hardware Trojans to highlight similarities and differences to malicious microcode.

### 2.1 Microcode

**Microcode Overview** Since microcode serves as an abstraction layer between static hardware and user-visible ISA instructions, hardware manufacturers have utilized microcode in Complex Instruction Set Computer (CISC) processors for improved efficiency and diagnostics for several decades [7, 15, 29]. Microcode is generally used in CISC architectures (most notably x86) for instructions that can not easily be directly implemented in hardware based on Reduced Instruction Set Computer (RISC) paradigms. In particular, a complex instruction, a.k.a. *macroinstruction*, is translated into a sequence of simple microinstructions [36] to perform computation. Although microcode was initially implemented in a read-only fashion [22, 30], manufacturers introduced an update mechanism to handle complex design errors for in-field hardware (e.g., Intel Pentium fdiv bug [39]) and install changes late in the design process. Typically, a microcode update is uploaded to a CPU during boot processes via motherboard firmware (e.g., BIOS or UEFI) or the operating system. Since an update is stored in low-latency, volatile CPU RAM, microcode updates are non-persistent. In addition, contemporary CISC processors leverage microcode to deploy security measures (e.g., Intel SGX [8]) or mitigations against micro-architecture attacks (e.g, Spectre, Meltdown, . . . ) [17].

**Microcode Encoding** Microprocessors have tight space requirements. Microcode instructions must be stored in an integrated ROM, which requires significant space on the die depending on the spaciousness of microcode instructions. In general, two formats for microcode encoding exist [23]:

- *Horizontal* microcode is minimally encoded. Each bit of the microcode instruction steers exactly one control signal inside the CPU. This approach allows for parallelism as one instruction can perform many tasks at once. However, this format is verbose and wastes Read-Only Memory (ROM) space because some signals may be mutually exclusive and will never be activated at the same time.
- *Vertical* microcode is maximally encoded and resembles traditional RISC instruction sets. In this format, multiple control signals are encoded into compressed bit fields leading to a more compact microcode. The designer, therefore, trades ROM space for additional decoders which are usually cheaper to implement than larger ROMs.

**Microcode Hooks** CPUs manufactured by major vendors Intel and AMD support a series of *match registers* which are used to update faulty microcode instructions. These registers redirect microcode execution from ROM to update RAM for specific ISA opcodes. Details from these microcoded architectures have been reverse engineered from patents [8, 22] and device delayering [21]. Thus, all ISA instructions can potentially be hooked.

**Microcode Scratch Registers** Koppe *et al*. wrote a specification of AMD's microcode based on their findings during reverse engineering [22]. AMD's microcode has access to internal registers which are hidden from the software programmer and the general-purpose registers of the x86 ISA. Some internal registers have special functions that, for example, help to implement branches. Others can be used to store temporary values, thus we refer to them as *scratch registers*.

**Microcode Security Aspects** This paper examines microcode-based Trojans that can leak cryptographic information from the processor. Previous work on microcode Trojans [22] examined different attack vectors enabled by malicious microcode updates. The paper mainly focused on privilege escalation or gaining system control. Previous work by Koppe *et al.* [22] only provided necessary primitives for cryptographic attack vectors on public-key cryptographic systems, but did not provide details on end-to-end attacks.

**Microcode Update** Microcode updates can either be applied by the BIOS or UEFI that is installed on the motherboard or by the operating system itself. Linux and Windows both offer functionality to update microcode automatically during their respective boot processes. To prevent attackers from issuing malicious microcode updates, Intel implements an RSA signature scheme that verifies update integrity with the microcode update being encrypted [7].

### 2.2 Hardware Trojans

In 2005, the US Department of Defense published a report about hardware trustworthiness, which sparked extensive research on the offensive and defensive aspects of malicious hardware manipulations [6, 11, 13, 16, 40]. A hardware Trojan typically consists of a *payload*, realizing the malicious functionality, and a *trigger*, activating the Trojan payload. Trigger logic implements the activation condition of the Trojan and usually depends on a set of trigger inputs. Generally, the trigger is designed to avoid detection during testing and is often only activated on rare conditions [16].

Hardware Trojan research has been mainly focused on injecting Trojans at the hardware description level or in supply-chain processes [6]. Confirmed real-world hardware Trojans have not been seen with the exception of Bloomberg's *Big Hack* [28], and even that Trojan allegedly involved PCB-level modifications rather than malicious circuit manipulations.

## 3 Designing Microcode Trojans: Seemingly Unlimited Capabilities vs. Limited Information

In this section, we describe the attack model and discuss the principal capabilities and limitations of malicious microcode.

### 3.1 Adversary Model

The high-level goal of the adversary is to undermine system security (e.g., by extraction of cryptographic keys) with the help of malicious microcode updates. In particular, malicious microcode subverts the general trust model assumptions - namely that the hardware is trustworthy and behaves correctly.

We assume that the adversary has knowledge about the microcode design and its implementation details, cf. Section 4, and is capable of deploying microcode updates on a target system. Even though microcode updates are (cryptographically) secured in practice, several works have already demonstrated how to bypass security measures [10, 22] and deploy custom microcode updates, with physical access to the device. To issue microcode updates remotely, the attacker needs access to the signing keys. Thus, in addition to adversaries who target aforementioned vulnerable hardware architectures, possible adversaries include nation-state adversaries, who can influence the CPU vendors, or even malicious vendors themselves. In this work, we assume that the attacker can issue arbitrary microcode updates since we are analyzing the impact of malicious microcode and not the security of the update mechanism.

### 3.2 Microcode and Software Semantics

As microcode represents an abstraction layer between the hardware implementation and the software, it possesses some unique traits that can be leveraged for malicious intent. Adversarial-controlled microcode enables fine-granular control of the CPU data path, including registers and memory. Once a malicious update is deployed, an adversary can replace any of the original ISA instructions with an arbitrary sequence of microcode instructions. Even though this characteristic appears to enable unlimited capabilities with respect to the Trojan payload (i.e., the malicious action executed) due to direct hardware access, critical information about the high-level software constructs is missing in this context. This poses a problem for the design of the Trojan trigger. Even answering seemingly simple questions such as "Is the TLS protocol currently executed?" — an attractive trigger condition for a Trojan which is straightforward on the system level — is non-trivial on the microcode level. These questions are particularly difficult to answer for an adversary if he can only observe individual ISA instructions, which is the default case since microcode by itself is stateless. In order to enable complex Trojan trigger conditions that lead to more stealthy and, thus, meaningful Trojans, instruction, or (input/output) data *sequences* must be evaluated, as discussed in the following paragraphs.

## 3.3 Microcode Trojan Design Strategies

Based on the discussion above, it is useful to categorize malicious microcode into two broad classes. We distinguish Trojan design strategies that use (1) stateless triggers and (2) stateful triggers.

**Stateless Trigger** We define a stateless microcode Trojan trigger as a mechanism that only checks operands of the individual assembly instruction, e.g., for a specific magic word. For example, an adversary may modify the `ADD` instruction iff one of the operand takes the value `0xf0f0 f0f0` — or any other 32 bit pattern.

Furthermore, a stateless trigger can combine multiple values from the CPU state, such as the program counter value, to identify an assembly instruction used in a specific part of the software. However, stateless Trojans are limited to one macroinstruction's current execution and cannot directly share information between different instructions.

**Stateful Trigger** In contrast, we define a stateful trigger as a mechanism that processes state across multiple instructions. For example, the Trojan is not only triggered by the operand value `0xf0f0 f0f0` of the `ADD`, but also checks if a specific instruction sequence has been executed beforehand. This approach enables conclusions about high-level software (e.g., by matching specific instruction or data sequences over a certain period) with which complex, extremely targeted trigger conditions can be realized. For instance, the Trojan can check for a specific known instruction signature, e.g., a sequence of assembly code of a cryptographic primitive used in a known library. If desired, the operands used within these signature sequences can also be evaluated, which allows the use of additional magic trigger values. Even the implementation of an entire on/off mechanism is possible, increasing the overall stealthiness.

**Payload** Microcode Trojan payloads are generally application-specific but versatile since microcode enables software-like flexibility with access to low-level hardware features.

**Adding new (malicious) instructions** Not only can existing instructions in the architecture be manipulated, but new (hidden) instructions can be added. This option is of concern if an attacker has access to the system and can execute arbitrary code. Depending on the system, custom instructions could change the privilege-level (*e.g.*, from user-mode to system-mode in RISC-V) or allow access to protected memory. Since the insertion and execution of new instructions are controlled by the attacker, the possibility of accidental triggering by legitimate code is limited. Hidden instructions could contain routines whose functionality is only limited by microcode storage size. However, even small tasks require many microcode steps, limiting potential use cases. To support custom instruction addition, the decode unit of the CPU must be modifiable, otherwise the insertion of new instructions is not possible.

## 3.4 Microcode Trojan Building Blocks

To design microcode Trojans we make use of microcode building blocks that perform specific operations including (1) reads/writes with general-purpose registers (including microcode scratch registers that hold temporary data), (2) arithmetic and logic computation, (3) conditional branches (and loop) operations, and (4) operations to hook microcode execution via updates, since not every instruction might be microcoded for performance reasons. Using these capabilities, we show how an attacker is able to design powerful microcode triggers and payloads. Koppe *et al.* showed the existence of general building blocks to perform these operations in the AMD K8/K10 [22]. The blocks provide Turing-complete capabilities.

## 4 Microcoded Processor Evaluation Platform

To foster and enable research, we have developed our own microcoded embedded processor evaluation platform. Our platform allows for fast microcoded instruction prototyping. Our supporting tools enable the straightforward generation of microcoded instructions and serve as a basis for our security analysis.

- **Microcoded RISC-V CPU:** The system is built to support the RISC-V base specification RV32I. This enables integration with existing RISC-V compiler toolchains and rapid software development and reuse. The approach profits from the established RISC-V ecosystem.

- **Microcode Language & Generation:** We developed a high-level descriptive microcode language that allows for fast instruction prototyping and the automatic generation and deployment of microcode. The language facilitates instruction modification and the addition of new instructions.

- **Evaluation Platform:** We integrated the CPU and microcode in a framework to conduct our security analysis. The architecture is modeled as a simple embedded system/micro-controller with all memory and peripherals on-chip.

An alternative microcoded RISC-V CPU implemented in the Chisel language [5] was previously developed. Instead of using this core, we opted to develop a new custom implementation that includes the microcode building blocks found in commercial systems (*e.g.*, scratch registers, cf. Section 3.4). Our new core allows for deep coupling with our microcode generation framework used for rapid prototyping.

## 4.1 CPU Overview

An overview of the microcoded RISC-V architecture is depicted in Figure 1. The following discussion explains how the microcode control unit interacts with the data path.

**Microcode Control Unit** The complete microprocessor instruction cycle is controlled by a microcode sequencer comparable to a modifiable finite state machine. A microprocessor instruction (macroinstruction) is broken into a set of microinstructions executed by the microcode sequencer. The current location in the *μCode ROM* is determined by the Microprogram Counter (μPC). Each microinstruction contains all control signals for the data path, which are stored in the Microinstruction Register (μIR) register. The microcoded control signals manage the enables and operations of the data path and determine the next step of the microcode sequencer. The *μPC Mux* selects the next step. The sequencing options are: increment the μPC, take a conditional branch, fetch the next instruction, or jump to an instruction start address. The conditional branch can jump to an arbitrary location in the *μCode ROM*. The next instruction is decoded in the *instr dec* block.

**Data path** The main CPU data path is designed to require multiple cycles per macroinstruction using a von Neumann architecture. Only one participant can internally transfer data in a cycle using the single bus. This limitation results in multiple cycles for the execution of most macroinstructions but allows for sequential modeling and low scheduling overhead. Operations performed in the Arithmetic Logic Unit (ALU) use the operand registers *A* and *B*. The Register File (RF) holds the 32 internal registers per the RV32I specification. Additionally, four scratch registers can be used to store temporary values for the macroinstructions. The implementation of microcode accessible scratch registers is typical, even in a modern desktop CPU [22]. All external memory and peripherals are accessed through the Random Access Memory (RAM) interface.

**Limitations** To keep the system extensible as a proof of concept for our attacks, the architecture does not implement instruction-level parallelism. Since a system-on-chip architecture with memory included is modeled, memory access does not require caches or pre-fetch engines.

## 4.2 Microcode

In the following, the microcode features of our architecture are described.

### 4.2.1 Microcode Language

A microcode language was developed and used to facilitate fast instruction prototyping, manipulation, and generation of microcode. Listing 1 shows the general structure of a CPU instruction in our microcode language. A sequence of one or multiple microinstructions is declared line by line. One line translates exactly to one bus cycle. One microstep consists of an identifier and the microcode command. The microinstruction identifier can also be used to feature jumps to microinstructions in the microcode ROM, enabling conditional branches in a microcode instruction.

```
def <instruction_name>
    micro_step_0: command_0;
    micro_step_1: ...;
```

Listing 1: Microcode Instruction Definition Prototype

The macroinstruction can contain a combination of different data path operations. In one cycle, a single data bus transaction can occur, and the next microcode sequencing step can be determined. Thus, in one command, data can be moved, the ALU controlled, branches evaluated and executed, and instructions sequenced.

**Sequential Microcode Modeling** In the following, the implementation of an instruction defined in the RISC-V ISA using our microcode language is illustrated. The *ADD* instruction adds the values from the register file locations specified by *rs1* and *rs2* and stores the result at location *rd*. The following text shows the mnemonic for the `ADD` instruction:

```
ADD rd, rs1, rs2
```

Our implementation of the ADD instruction in our microcode language is shown in Listing 2. Since all data transfers must be mapped to different cycles, this instruction is broken down into three microinstructions.

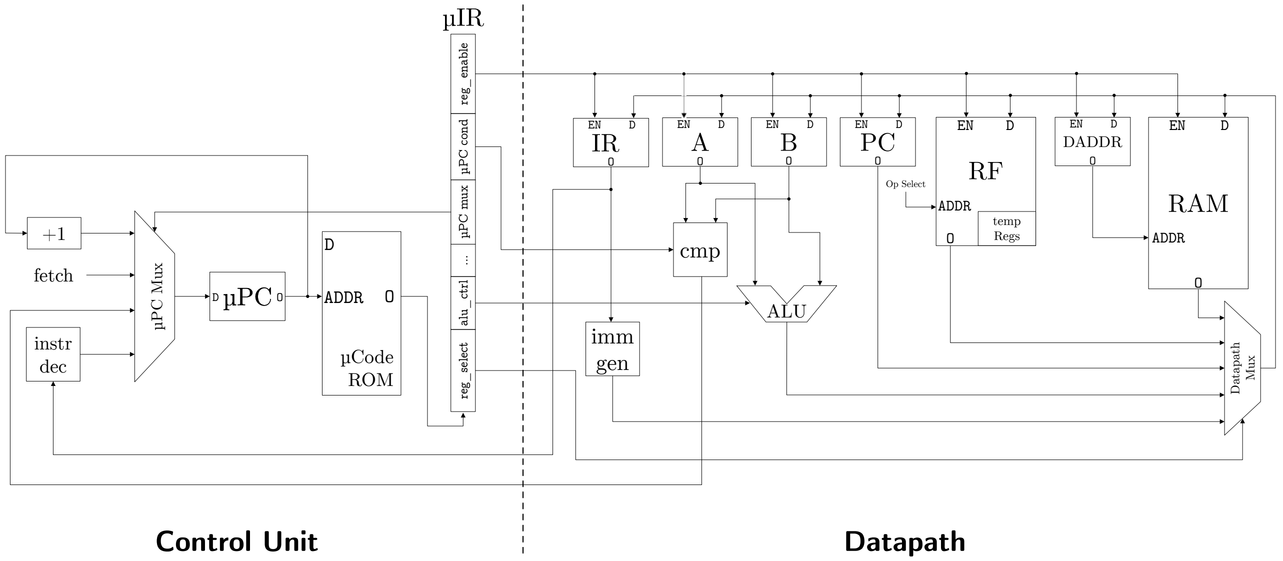


Figure 1: Overview of our microcoded RISC-V architecture

```
def add
    add0: a <- rf[rs1];
    add1: b <- rf[rs2];
    add2: rf[rd] <- alu(a + b); fetch;
```

Listing 2: Microcode Instruction *ADD*

In the first step, *add0*, the data at location *rs1* is stored in arithmetic register a, in the second step *add1* the value from *rs2* is stored in register b, and subsequently, in a third step *add2*, the ALU is instructed to add both values together and store the result in location *rd* in the register file.

Our scripts generate the microcode instructions and provide memory files for the $\mu$Code ROM. The microcode has been implemented in a horizontal encoding scheme. We implemented the entire RV32I base instruction in microcode.

### 4.2.2 Microcode Update Mechanism

Our architectures allows for in-field microcode updates during application execution. First, microcode data is stored in an internal structure. A flush operation then halts the CPU and copies the microcode content from internal memory into the *µCode ROM* — technically turning it into an adaptable microcode RAM in our proof-of-concept. To update the lookup of microinstruction addresses, the dispatch table is realized in memory. This enables the modification, addition, and removal of instructions in-field.

## 4.3 Implementation & Setup

Our microcoded RISC-V CPU (as seen in Figure 1), including a memory/peripheral bus, was embedded in an FPGA.

Peripheral components are treated as memory-mapped devices accessed through the RAM interface. A custom bootloader, stored in RAM, initializes the hardware registers, including the stack pointer in the register file. An attached workstation (PC) is used to provide microcode updates and firmware on startup via a UART. The bootloader then executes this firmware once the initialization has been completed. A timer (TIM) is used to measure system clock cycle counts.

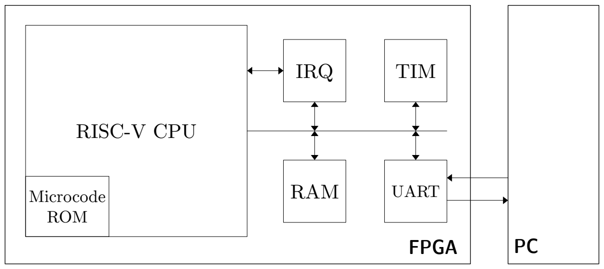


Figure 2: Setup overview

**Synthesis & Implementation** The processor system was implemented in Verilog. The core was tested on a NexysA7 development board manufactured by Digilent. The design runs at 100 MHz and is constrained by a critical path through on-FPGA RAM. Table 1 shows resource utilization on the NexysA7 development board, which features an Artix-7 100T with 32 KB RAM used as memory.

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 1917 | 63,400 | 3.02 |
| LUTRAM | 248 | 19,000 | 1.31 |
| FF | 924 | 126,800 | 0.73 |
| BRAM | 8 | 135 | 5.93 |

Table 1: Utilization of the entire system for an XC7A100T-1CSG324C Xilinx FPGA

**Software Tools** The *RISC-V GNU toolchain* is used to cross-compile applications. We developed *drivers* for the `UART`, `TIM` and `LED` interfaces to handle interactions. For simulation, we designed a testbench using *verilator* [31]. The environment was used to simulate the boot up and execution processes, including the execution of applications and microcode updates.

## 5 Case-Study: (In-)Secure Boot

In order to prevent the execution of malicious or modified firmware, many real-world embedded devices deploy secure boot [24]. However, industry has not standardized implementation strategies for secure boot systems.

From a high-level perspective, a secure boot system loads the firmware (including its cryptographic signature) and then performs a signature verification before code execution to ensure its integrity. If verification fails, i.e., unauthorized firmware is about to be loaded, the boot process is halted. Otherwise, the boot process continues (e.g., firmware execution or verification of the next boot process stage).

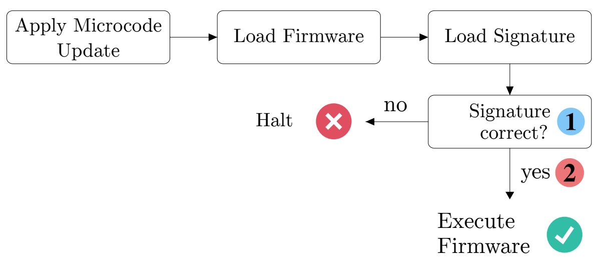


Figure 3: Boot process overview with secure boot (simplified)

To ensure that the secure boot software, a.k.a. the *bootloader*, itself has not been subjected to tampering, it is deployed using a secure ROM that is protected from hardware-based attacks [24]. This implementation implies that the bootloader cannot be updated.

In the following discussion, we assume that the microcode update is applied before firmware verification since a microcode update typically implements hardware bug fixes that should be loaded as early as possible in the boot process. Note that the impact of a permuted order, i.e., microcode update after firmware verification, is discussed in Section 5.3.

**High-Level Attack Idea.** To bypass secure boot and load unauthorized firmware, we focus on disarming critical cryptographic integrity checks without attacking the cryptographic implementation itself. In particular, we leverage (1) the static nature of the bootloader and (2) the handling of the verification result. Even though the bootloader is stored in a secure (read-only) memory, it can be read-out and analyzed using software reverse-engineering [32].

Once the cryptographic integrity checks are understood, a microcode Trojan can be crafted that patches semantics of the

```
if (verify(firmware, signature)){
  asm volatile("jal ra, 0x7000");
} else {
  while (1) {} // trap CPU
}
```

```
0x230: jal   ra,280 <verify>
0x234: addi  a5,a0,0
0x238: beq   a5,zero,240 <main+0x240>
0x23c: jal   ra,7000 <_isatty+0x20>
0x240: jal   zero,240 <main+0x240>
```

Listing 3: Example: signature verification result handling in *vboot* [14, 18] (Chrome OS verified boot system). Above C code and below assembly code after compilation (gcc 9.2.0 with optimization level `O0`).

instruction that handles control flow based on the verification result. Note that we can target a specific instruction based on the unique address (by checking the program counter value) ① in Figure 3. Moreover, we can change instruction semantics ( ② in Figure 3) in a way so that control transfer is always redirected to the valid signature program path or conditionally so that the valid signature program path is only taken on an additional trigger condition.

### 5.1 Microcode Trojan Design

We now detail our microcode Trojan trigger and payload design that bypasses the *vboot* [18] secure boot implementation.

① **Trigger** We identified the `BEQ` instruction at address `0x238` in Listing 3 as the relevant instruction for the Trojan trigger since it handles verification result processes. More precisely, if the return value of the `verify()` function is zero (verification fails), a branch is made to the memory address `0x240` where the boot process is trapped (because the instruction at this address always jumps back to itself). Thus we added a microcode check to the `BEQ` instruction semantics that checks for the address `0x23c` (since the program counter value always points to the next address after instruction fetch). We optimized our microcode Trojan trigger code, so it is only executed in case the result comparison yields equal (so there is no performance impact on the `BEQ` instruction when the result is not equal, cf. `beq2` in Listing 4).

The designed trigger performs the desired functionality since the branch is only taken if the verification failed (e.g., 0 is returned), and the address is not `0x23c`. The branch is not taken if the address is `0x23c`, and thus the next instruction is fetched. Note that the Trojan does not influence other *BEQ* instructions, since it is only executed at one specific point when the program counter (PC) is at address `0x23c`.

**2 Payload** The payload in this case is the act of not taking the branch. The payload enables the attacker to load tampered firmware, resulting in further attack options.

```
def beq
    beq0: a <- rf[rs1];
    beq1: b <- rf[rs2];
    beq2: b <- ig(imm_b); if a != b fetch;

    # Trojan
    beqt0: a <- 0xC;
    beqt1: b <- 0x3;
    beqt2: b <- b << 4;
    beqt3: a <- alu(a | b);
    beqt4: b <- 0x2;
    beqt5: b <- b << 4;
    beqt6: b <- b << 4;
    beqt7: a <- alu(a | b);
    beqt8: b <- pc;
    beqt9: if a == b fetch;

    beq3: b <- ig(imm_b);
    beq4: a <- pc;
    beq5: a <- alu(a - 4);
    beq6: pc <- alu(a + b);
    beq7: fetch;
```

Listing 4: Microcode Trojan for BEQ instruction to bypass secure boot. Due to limited capabilities of microcode, first we need to construct higher level primitives. We first load the constants and assemble address trigger *0x23C*, by subsequent shift and or operations (beqt0-beqt7). We then load the *PC* into the ALU operand registers and compare the results (beqt8-beqt9). If the results match, our Trojan payload is executed and the branch is not taken (beqt9), regardless the signature verification result.

## 5.2 Evaluation

**Setup** For this case-study, we implemented a secure boot process using a standard signature scheme relying on public-key cryptography. To this end, we used the signature verification implementation extracted from the Chrome OS verified boot system *vboot* [18] that implements RSA-based signatures, with 2048-bit keys (in combination with SHA-256).

**Unauthorized Firmware Execution** We successfully executed our attack by subverting the microcode and manipulating the firmware that is supposed to be verified. Even though the verification fails and the boot process *should* halt, our unauthorized firmware is successfully executed due to the microcode Trojan.

## 5.3 Discussion

This case study demonstrated that *even* a single assembly instruction of an application can be targeted and equipped with additional (malicious) semantics. Even though we limited our evaluation to the *vboot* bootloader code, it is obvious that the attack itself can be transferred to other critical code. If an additional conditional trigger should be deployed (e.g., the instruction at address 0x23c has to be executed three times before the Trojan is activated), the trigger could be adjusted to be stateful. The basic approach of the Trojan can be leveraged in implementations that feature defenses against physical fault injections [9, 38] for an adversary who can control instruction semantics using malicious microcode. Common countermeasures against physical fault injection attacks include redundancy, random delays, or monitoring, which can be easily bypassed in this adversary model.

**System Impact** Table 3 shows the performance overhead of our Trojan case-studies. The results indicate a relatively low-performance overhead across the Embench benchmark suite with the inserted Trojan. For benchmarks that have a high amount of data-dependent control flow, such as edn and nbody, the overhead is higher. Benchmarks without a significant number of data-dependent branches, such as crc32, aes and sha256, demonstrate negligible overhead. In our system, the Trojan does not influence control or data flow since the triggered address only occurs once in the static bootloader address space. If there is an Memory Managment Unit (MMU) or Memory Protection Unit (MPU) present, the address may occur at different points in the code and thus change control flow uncontrollably. This issue could be circumvented by applying a second (non-malicious) microcode update that removes the Trojan from the targeted instruction or by checking for instruction bytes near the target address, minimizing the chance of accidental triggers. However, applying run-time microcode updates is a non-trivial challenge that must be supported by the hardware system. Special mechanisms are needed to save and restore the CPU's internal datapath state, *e.g.* general-purpose registers.

## 6 Case-Study: (In-)Constant Time

To implement security properties such as confidentiality and integrity, it is required that deployed cryptographic implementations cannot be compromised [13]. Most real-world cryptographic implementations offer constant-time encryption and decryption processes, rendering classical timing attacks impossible. In this case study, we use a malicious microcode update to introduce an exploitable timing anomaly in widely-deployed open-source Advanced Encryption Standard (AES) implementations that ultimately leaks the utilized key.

**High-Level Attack** To implant an exploitable timing leakage, we require a (measurable) conditional execution (or, in other words, a timing dependency) from secret key data and (attacker-controlled) public plaintext data. Hence, we focus on the first *KeyAddition* layer. We thus modify the XOR instruction to artificially prolong execution timing (payload 2) based on operand values, i.e. key and plaintext (trigger

1), see Figure 4. Based on execution time measures and subsequent statistical analysis, we recover the deployed key.

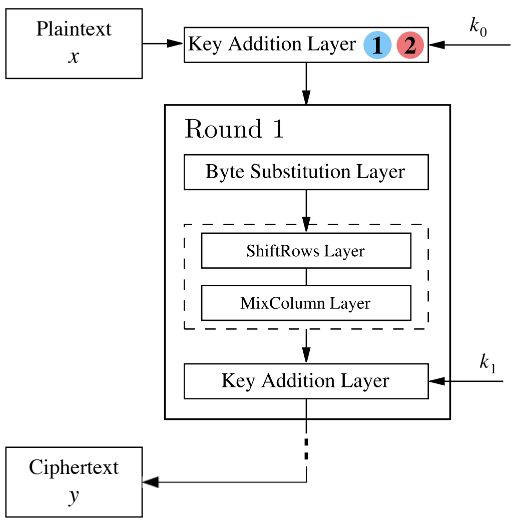


Figure 4: AES Timing Trojan Overview [26]

We want to highlight that our Trojan is designed with the goal of being independent of a concrete AES implementation. However, as noted before, a CPU can neither automatically distinguish whether an AES encryption nor which exact AES operation is currently being executed. Hence our Trojan cannot prolong just the `XOR` instruction of the first *KeyAddition* operation, but rather any `XOR` instruction. Since this results in additional noise for accidental Trojan triggers, we require statistical analysis to filter noise.

## 6.1 Microcode Trojan Design

Even though the high-level idea can be straightforwardly translated to a microcode Trojan, see Listing 5, an attack would require the measurement of $2^{32}$ encryption operations (e.g., for a register size of 32 bits) to trigger the Trojan at least once. However, this amount of measurements is impractical for a realistic attack.

```
1  def xor
2      xor0: a <- rf[rs1];
3      xor1: b <- rf[rs2];
4      xor2: rf[rd] <- alu(a ^ b);
5      xort1: if a != b; fetch;
6      xortnop: nops();
7      xort2: fetch;
```

Listing 5: Exemplary microcode Trojan for `XOR` instruction to implant exploitable timing leakage. In case operands match (line 5), a timing dependency is added (line 6).

To this end, we designed and evaluated multiple microcode triggers, i.e., based on 4-bit, 8-bit, and 16-bit operand checks. However, these straightforward strategies require a large number of measurements since a 4-bit check almost always trigger and thus introduces a significant amount of noise, while a 16-bit check requires at least $2^{16}$ measurements to trigger at least once. Note that this does not imply that we require $2^{16}$ measurements to recover the key, but rather 500K to 1M measurements are necessary to recover the key with such a trigger.

We optimized our final Trojan trigger design to combine the best of both worlds, i.e., checking the whole 32-bit operation but in a byte-by-byte wise fashion where each byte is checked if the previous byte matched, see Listing 6.

```
1  def xor
2    xor0: a <- rf[rs1];
3    xor1: b <- rf[rs2];
4    xor2: rf[rd] <- alu(a ^ b);
5
6    xort0: if a[ 7: 0] != b[ 7: 0]; goto xort1;
7    xortnop: nop;
8    xort1: if a[15: 8] != b[15: 8]; goto xort2;
9    xortnop: nop;
10   xort2: if a[23:16] != b[23:16]; goto xort3;
11   xortnop: nop;
12   xort3: if a[31:24] != b[31:24]; fetch;
13   xortnop: nop;
14   xort4: fetch:
```

Listing 6: Pseudocode of our optimized microcode Trojan for `XOR` instruction to implant exploitable timing leakage in a byte-by-byte fashion. `NOP`s are added for every matching operand byte (lines 6-13). We refer the interested reader to Appendix A.1 for the detailed microcoded description.

## 6.2 Evaluation

**Setup** To evaluate the effectiveness of our microcode Trojan, we selected several widely-deployed open-source constant-time AES implementations, namely openSSL [1], gnuTLS/Nettle [4], and Käsper-Schwabe [3, 19].

Table 2: Impact of optimization level for AES implementation

| | Implementation | XOR count | | Source |
|---|---|---|---|---|
| | | -Os | -O3 | |
| **openSSL** | T-table | 197 | 330 | [1] |
| **gnuTLS/Nettle** | T-table | 63 | 64 | [4] |
| **Käsper-Schwabe** | bitslicing | 222 | 572 | [3] |

We compiled the sources using the gcc RISC-V toolchain and evaluated both compiler optimizations `Os` (memory-optimized) and `O3` (performance-optimized). Even though the optimization level has an impact on the total amount of `XOR` instructions (e.g., caused by `IPA-CP` and `Pool-loop` settings for `O3`), see Table 2, the general attack is independent of the optimization level. In the following, we report evaluation

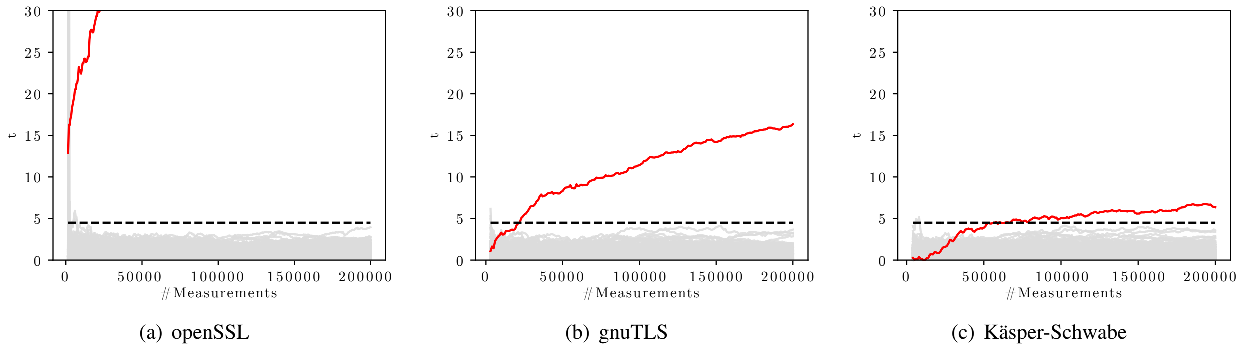


Figure 5: $t$-test values for the last key byte of each implementation using byte-by-byte microcode trigger as shown in Listing 6.

for the Os setting. To measure the execution time, we used the available hardware timer in our evaluation platform, see Section 4.3.

**Exploitable Leakage Analysis** In order to evaluate whether leakage information can be exploited, we use the standard Welch's $t$-test [37]. To this end, we measured the execution time of 10000 fixed and 10000 random plaintexts encryption operations both (1) without microcode Trojan trigger, and (2) with microcode Trojan trigger deployed. As expected, all implementations without the Trojan trigger exhibit a (dimension-free) $t$-value of 0 (= no leakage), whereas, with Trojan trigger, each implementation exhibits a $t$-value of $> 4.5$ (= indicator value whether an implementation leaks information): OpenSSL (40.81), gnuTLS/Nettle (48.61), and Käsper-Schwabe (9.53).

**Key Retrieval** Figure 5 depicts our $t$-test evaluation results for the choosen AES implementations. In particular, the correct key candidate (marked in red) was clearly separated after $\sim$50k measurements for both openSSL and gnuTLS, while $\sim$200k measurements were necessary for Käsper-Schwabe implementation, to reliably guess *all* key bytes. Note that the latter implementation requires more measurements due to its gate-logic implementation for the *SubBytes* layer, which uses various XOR instruction that introduce additional statistical noise.

## 6.3 Discussion

In this case study, we demonstrated that we are able to implant sophisticated microcode Trojan triggers that can introduce exploitable leakage across several allegedly secure AES implementations (and compiler optimizations). We designed and optimized our trigger to minimize the number of required timing measurements. Even though we conducted our measurements in a low-noise system (e.g., no out-of-order execution or multiple cache hierarchy), the principle of our Trojan can be adapted to *noisier* systems by adapting the amount of NOP instructions in the payload to increase the timing dependency.

**System Impact** Table 3 shows that the performance impact is negligible for general-purpose applications. However, software that utilizes numerous XOR operations, such as CRC and cryptographic implementations, exhibits a significant performance impact (*e.g.*, up to 30%). Note that the Trojan does not alter the control or data flow and thus is stealthy for the majority of benchmarks.

## 7 Case-Study: (In-)Secure Cryptography

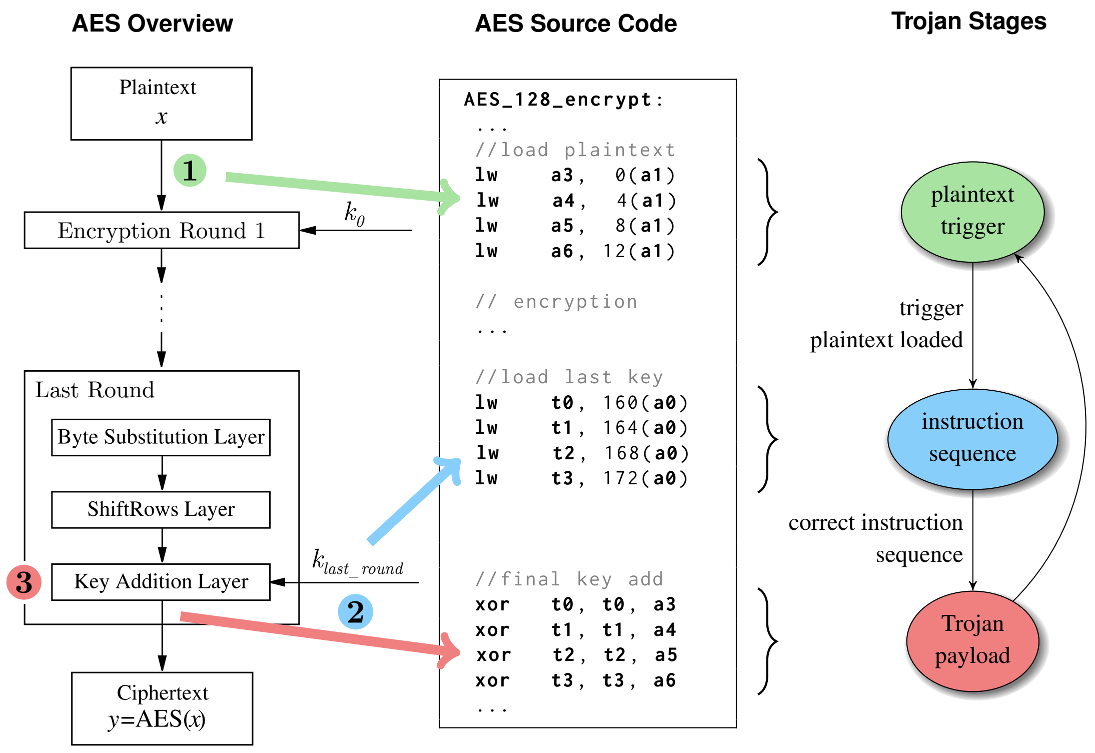
Most real-world cryptographic software libraries provide specialized assembly implementations to enable fast and secure implementations. More precisely, assembly implementations enable security engineers to control and reason about implementation security on specific architectures. In this case study, we leverage the static nature of the (rarely-changing) assembly code by designing a stealthy microcode Trojan that can leak the cryptographic key only for a single attacker-controlled *magic* plaintext.

**High-Level Attack Idea** To leak cryptographic key material, we focus on a specific fault injection during cryptographic processing. In particular, we leverage the static nature of cryptographic assembly implementations combined with a multi-stage microcode trigger mechanism for an AES implementation. For a specific *magic* plaintext, we insert a fault in the last *KeyAddition* operation so that the ciphertext is always equivalent to the last round key by setting the state after the last *ShiftRows* operation to zero. Based on the last round key, we can compute the main key [34].

Our Trojan leverages a microcoded instruction matching state machine that spans multiple instructions and, depending on its state, executes specific trigger functionality or payload. Figure 6 shows the high-level attack idea of our Trojan.

## 7.1 Microcode Trojan Design

We now detail our multi-stage microcode Trojan trigger and payload design to insert sophisticated faults during cryptographic processing to leak its keys. We build a sophisticated state machine-based Trojan trigger to share informa-



```
AES_128_encrypt:
 ...
 //load plaintext
 lw    a3,  0(a1)
 lw    a4,  4(a1)
 lw    a5,  8(a1)
 lw    a6, 12(a1)

 // encryption
 ...

 //load last key
 lw    t0, 160(a0)
 lw    t1, 164(a0)
 lw    t2, 168(a0)
 lw    t3, 172(a0)


 //final key add
 xor   t0, t0, a3
 xor   t1, t1, a4
 xor   t2, t2, a5
 xor   t3, t3, a6
 ...
```

Figure 6: High-Level Attack Overview of the AES Fault Trojan: On the left, the general structure of AES is shown. The arrows match the associated steps of the AES block diagram [26] to the code taken from [33]. On the right, the associated code snippets are matched to our Trojan stages, responsible for either the trigger or payload.

tion through multiple cycles and between different instructions. Note that we store the state using internally-available scratch registers for this purpose. Our Trojan modifies only two instructions, namely the LW (load word) instruction, for the *magic* plaintext trigger and identification of the last *KeyAddition*, and the XOR instruction to subsequently insert the payload.

The state machine keeps track of the current state of our multi-stage Trojan. The finite state machine shown in Figure 7, shows the transitions in one stage, as well as between different stages, which are explained in the following sections. The state machine is implemented in microcode with conditional jumps based on the current state stored in the scratch registers and different handlers for the associated state operations. The state is updated in the scratch register afterward, thus allowing for *communication* between instructions and over time.

We provide details based on a RISC-V implementation by Stoffelen et al. [33], but the general operating principle of our Trojan is independent of the underlying ISA and implementation, as discussed in Section 7.3. Currently, none of the major real-world cryptographic software libraries provide highly optimized RISC-V assembly implementations. Still, the availability of implementations is only a matter of time due to the rising popularity of RISC-V.

### 7.1.1 Trigger Design

To identify a magic plaintext and the instructions responsible for the last *KeyAddition* operation, our multi-stage trigger operates in two stages, as depicted in Figure 6:

1 ***Magic* Plaintext Trigger**. To insert faults only for a specific plaintext, we first have to check for the plaintext during the load from memory (via the LW instruction). Therefore, we alter the LW instruction semantics using an optimized finite state machine that checks a predefined sequence of four magic 32-bit words, see Figure 7. The current state is stored in a scratch register. Once the *magic* plaintext trigger has been received, we move to the next stage.

2 **Last *KeyAddition* Operation Trigger.** To identify the execution of the last *KeyAddition* operation, we check for the specific instruction sequence that loads the last round key using the state machine approach from the previous step. Since this load is also implemented via the LW instruction, as seen in the assembly snippet in Figure 6, we also change LW instruction semantics accordingly. Note that we only enter this second trigger stage once the magic plaintext is detected, see Figure 7.

**LW Instruction Semantics** Since the LW instruction semantics must be armed with both the plaintext trigger 1 and the instruction sequence trigger 2, its microcode becomes significantly complex.

The state check in microcode is performed by comparing constant values for the associated state encoding (*e.g.*, 0x0 for the initial state, 0x1 for the second state, ...) with the current state value held in the associated scratch register. Based on the current state, we go to the associated microinstruction that handles the execution of the expected behavior of the state — to be more precise, checking the specific word or offset, depending on the stage. We refer the interested reader to Appendix B.1, where the microcode implementation of the LW Trojan is detailed.

#### 7.1.2 Payload Design

To inject faults in the XOR operation of the last *KeyAddition* operation, we simply set its second operand to zero. Hence, the last round key is stored in the state registers. This payload is executed in the last four states as the lowest segment of the state machine depicted in Figure 7 shows:

3 ***KeyAddition* Payload**. To identify if the *XOR* payload is to be executed, first, the microcode checks if the payload execution stage has been reached. After passing this check, the malicious *XOR* implementation is executed and leads to a leakage of the first operand, which is the last round key. This happens for each 32-bit part of the key. The internal stage counter is incremented for each *XOR* operation and subsequently returns to the initial reset state after four executions of 128-bit operations on the AES state.

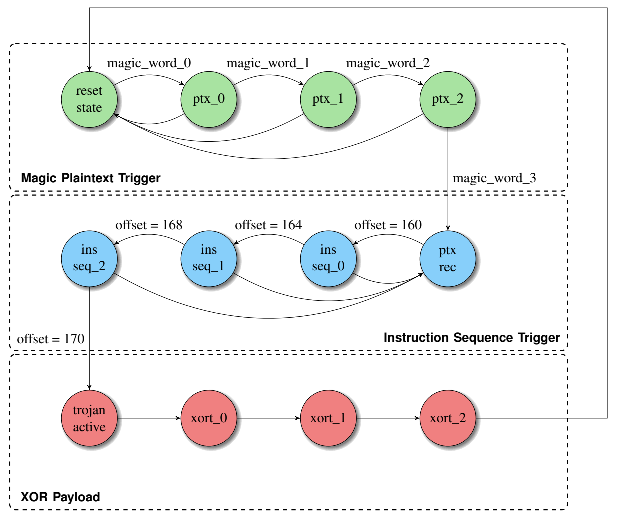


Figure 7: Detailed AES Fault Trojan FSM, depicting the trigger conditions in each stage.

**XOR Instruction Semantics** We armed the XOR instruction semantics with both state machine checks and payload. First, we check whether the Trojan has been activated so that the payload can be inserted in the correct position. Afterward, the result of the XOR operation is modified by setting the second operand to zero and storing the last round key in the register file instead of the actual ciphertext of the encryption.

Since the state machine only transfers the payload state for a 128-bit plaintext match (and the correct instruction sequence has been identified), the probability of accidental fault injection is negligible. We refer the interested reader to Appendix B.2, where we detail the exact microcode implementation of the Trojan.

## 7.2 Evaluation

We successfully executed our attack on the RISC-V AES implementation by Stoffelen [33] by maliciously subverting the microcode as detailed. The microcode leaks the utilized cryptographic key if and only if a specific magic plaintext is presented to the AES code for encryption.

## 7.3 Discussion

We demonstrated that complex microcode Trojan triggers can be implemented that span different instructions.

For implementations that incorporate countermeasures against fault injection attacks, the Trojan may be equipped with an additional stage that bypasses subsequent error detection. If scratch registers cannot be utilized (e.g., high scratch register pressure) to store the global state across different instructions, we may simply use reserved RAM addresses that are typically available in embedded system memory maps.

**Crypto Rarely Changes** Our Trojan design requires detailed knowledge of the deployed assembly code, i.e., to match a certain instruction sequence. We now support our claim that highly optimized assembly implementations in cryptographic libraries rarely change, enabling the design of such Trojans. As an example, OpenSSL provides 30 assembly implementations for a variety of different platforms [1]. We took the x86 implementation of the major crypto library OpenSSL as an example and analyzed its commit history on GitHub [2].

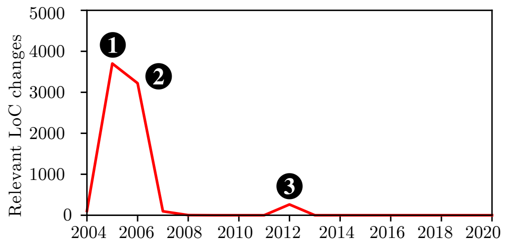


Figure 8: Graph displaying changes to the optimized OpenSSL x86 AES implementation since 2004

As shown in Figure 8, the frequency of relevant assembly code changes is limited. Code changes are connected to cache attack mitigations for the peaks at 1 and 2, as well as the addition of new support for architectures (in this case the Intel Atom) at peak ❸. The instructions that perform the actual cryptographic calculation stayed mostly untouched. At the time of writing, the assembly code has not changed for over eight years. Hence, it is safe to assume that an adversary can make certain assumptions about the code layout of the deployed cryptographic library.

**System Impact** Table 3 shows that even though the Trojan implementation was optimized by improving state checks and efficiently handling most probable cases (e.g., no part of the Trojan has been triggered, and the state machine is in the first state most of the time), the overall Trojan overhead is relatively high (up to 50%). However, the risk of an accidental Trojan trigger is negligible due to the use of a sophisticated trigger condition.

**Custom Key Extraction Instruction** Instead of leaking the key by manipulating the last key addition, a custom instruction can be added to extract the key at a later time. Once the loading of the key has been identified ②, the key can be stored in microcode scratch registers. At a later time, the attacker can execute his/her custom malicious instruction to extract the key from the registers. This method would make the Trojan even more stealthy since no data manipulation takes place and only performance overhead is added. However, the attacker needs access to the system, and storing values for an extended period of time risks the scratch registers being overwritten.

## 8 Discussion

In the following, we discuss the implications of microcode Trojans and provide insights into possible mitigations.

### 8.1 Generality and Portability

Even though our case studies focus on embedded systems and the RISC-V ISA, our microcode Trojans are transferable to other hardware and software platforms since the necessary building blocks can be found across modern CPUs (cf. Section 3.4). For the constant-time case-study, noise caused by complex system execution can be addressed in several ways: (1) increase the number of NOP instructions to amplify timing dependence, or (2) perform a higher number of measurements. Instruction level parallelism may have a negative impact on the Trojans that rely on specific instruction sequences, especially if the sequences are complex and long. For example, interrupts and out-of-order execution could disturb the sequence and thus prevent Trojan triggering or lead to false payload execution. This issue should be analyzed in future research, as microcode Trojans must be specifically tailored for a target architecture.

Complex systems will enable the exploration of additional kinds of microcode Trojan triggers and payloads. For example, in pipelined systems, triggering could be enhanced since multiple instructions are present in the CPU's data path at the same time, allowing for more accurate assessments of an application under execution.

Microcode Trojans will likely require updates when new versions or general firmware updates are issued. However, microcode is typically updated on a regular basis, making it adjustable, a key strength compared to hardware Trojans.

### 8.2 Security Implications

Our case studies demonstrate the severe ramifications for system security that are introduced by malicious microcode updates. Even on our resource-limited embedded system, we demonstrated powerful Trojans that undermine system security. The obscure nature of currently deployed microcoded CPUs hinders proper security analysis by the general public since implementation and updates are one of the best-kept secrets by the vendors. We, as users, must trust our hardware and its integrity blindly. Even though microcode offers several advantages, the concept of updatable hardware comes with significant security risks that must be further addressed.

### 8.3 Mitigations

Open (readable) microcode enables system users to apply traditional measures from malicious software analysis to identify malicious behavior. However, since microcode is kept opaque, we, as a research community, must develop mitigations to detect anomalies and develop defenses even in such a strong adversary model.

**Fingerprinting Legitimate Microcode Behaviour** We observed that our Trojans introduce behavioral timing changes, see Table 3. Hence, a potential mitigation could involve a vendor measuring the correct timing behavior of each instruction and publicly reporting the values for each microcode update. This mitigation would enable end-user checking without the disclosure of detailed information about the microcoded architecture from the microprocessor vendor. Generally, the timing information could form an *official* fingerprint released by the vendor for each instruction. The information could be extended with additional features, such as power behavior and general input/output behavior. Users could then fingerprint each instruction and compare measured results to the vendor-provided fingerprint. However, this approach assumes that the information provided by the vendor is trustworthy.

Note that this approach might not work for every instruction type since several instructions have input-dependent execution time. Furthermore, the analysis may be manipulated if hardware features enabled by microcode can be leveraged to detect that such measurement processes are being performed.

**Malware Analysis of Microcode** If microcode is readable from software, it can be evaluated using malware-like analyses. To enable in-depth analyses, vendors must provide details of the microcode implementation and structure. A microcode-

| Benchmark | Without Trojan | Secure Boot Trojan | | Timing Trojan | | AES Fault Trojan | |
|---|---|---|---|---|---|---|---|
| | Cycles | Cycles | Overhead | Cycles | Overhead | Cycles | Overhead |
| crc_32 | 1027194 | 1027194 | 0.00% | 1217883 | 15.66% | 1396490 | 26.44% |
| edn | 21140442 | 23295837 | 9.25% | 21140640 | 0.00% | 23533786 | 10.17% |
| huffbench | 13371606 | 13472850 | 0.75% | 13371711 | 0.00% | 23249383 | 42.49% |
| minver | 554736 | 587142 | 5.52% | 562419 | 1.37% | 660001 | 15.95% |
| nshneu | 554736 | 587142 | 5.52% | 562419 | 1.37% | 660001 | 15.95% |
| statemate | 27258 | 28347 | 3.84% | 27258 | 0.00% | 30118 | 9.50% |
| st | 26762280 | 28760991 | 6.95% | 26889999 | 0.47% | 28344001 | 5.58% |
| ud | 196194 | 200154 | 1.98% | 196860 | 0.34% | 306914 | 36.08% |
| wikisort | 80726326 | 83308015 | 3.10% | 81195517 | 0.58% | 110816657 | 27.15% |
| matmult-int | 9698424 | 9962424 | 2.65% | 9698634 | 0.00% | 12959764 | 25.17% |
| mont64 | 831484 | 836962 | 0.65% | 832435 | 0.11% | 1518872 | 45.26% |
| nbody | 222240058 | 244234943 | 9.01% | 223143562 | 0.40% | 233888566 | 4.98% |
| nettle-sha256 | 270012 | 270078 | 0.02% | 402918 | 32.99% | 527593 | 48.82% |
| nettle-aes | 2157660 | 2157792 | 0.01% | 2826786 | 23.67% | 4258054 | 49.33% |

Table 3: Cycle overhead comparison of our Trojan case-studies using the Embench benchmark suite [27]

independent hardware path for reading microcode must be implemented to ensure that the readout process has not been subjected to tampering, *e.g.* by a malicious microcode update.

**Towards Resilient Microcode Architectures** Currently, microcode updates have Turing-complete computation model capabilities limited only by hardware storage size. Even though microcode updates provide hardware designers with powerful capabilities, we have demonstrated that this power can be leveraged by adversaries as well. To build a resilient architecture under the assumption of malicious microcode updates, future interdisciplinary (security) research may focus on whether microcode updates may be restricted in a way that is powerful enough for hardware designers to patch erroneous CPU behavior, but simultaneously limit the capabilities of various microcode Trojan classes.

## 8.4 Comparison to Classical Malicious Hardware and Malicious Software

Malicious microcode is a distinct class of attack vectors since it possesses traits from both software and hardware worlds, in particular with respect to flexibility and stealth.

**Malicious Hardware** Unlike hardware Trojans, microcode Trojans do not lack post-manufacturing versatility, since microcode Trojans can be inserted as easily as they can be removed from a target system. As both hardware Trojans and malicious microcode are custom-tailored to target a system and its applications, the flexibility of microcode enables scalability, while providing similar stealthiness capabilities. With respect to current defenses, detection methods for hardware Trojans exist (cf. [35]), however, such analyses typically require specialized equipment to investigate hardware implementation chip details. Here, the analysis of malicious microcode has traits that are similar to the analysis and subsequent removal of malicious software (e.g., low-level root-kits), however, as long as implementation details and the microcode structure are not fully reverse-engineered or published by vendors, malicious microcode provides similar stealth capabilities as malicious hardware.

**Malicious Software** Malicious microcode is inflexible when compared to traditional malicious software due to required low-level hardware access and the limited availability of information from running applications. Malicious software is typically seamlessly portable to new architectures and software versions, while microcode requires custom-tailoring to the CPU architecture and targeted application. As noted in the previous paragraph, microcode Trojans generally possess traits that are similar to sophisticated low-level malicious software. However, considering the limited information available for COTS CPUs and the current state of defenses, malicious microcode enables significantly improved stealth capabilities.

Modern Trusted Execution Environments (TEEs), such as Intel SGX, are partially, if not completely, implemented with microcode [8]. Since microcode provides the Trusted Computing Base (TCB) foundation, any malicious microcode update invalidates the security properties of the TCB. Hence, adversaries with the ability to issue malicious microcode updates could unleash devastating attacks on a spectrum of modern computing systems.

Thus, microcode Trojans provide a balance between flexibility and stealthiness. Since no mechanisms to analyze microcode semantics are available yet for commercially available CPUs, microcode Trojans constitute a dangerous affair that has been sparsely discussed in the scientific community.

## 9 Conclusion

In this paper we explored the threat posed by malicious microcode, with a focus on embedded CPUs. We showed that by using stateful trigger conditions, the adversary can design targeted Trojans that will rarely — if ever — be triggered by mistake. Similarly, we showed that there is a large design space for the Trojan payload, i.e., the actual malicious action executed. Through three case studies, we demonstrated that Trojans that lead to major security violations can be realized. We also showed that there is a trade-off between stealthiness and trigger complexity — complex triggers come with considerable run time costs. This observation gives rise to detection and mitigation strategies. Even though our experiments were done on a RISC-V platform, they carry over in principle to other CPUs, both for embedded and desktop applications.

## Acknowledgements

We would like to thank Iryna Schwindt for her initial research into commercial microcode, Felix Wegener for supporting us with the timing side-channel attack and Jérémie Crenne for his input regarding microcoded RISC-V architectures. We would also like to thank our shepherd, Ramya Jayaram Masti, for her great input, as well as the anonymous reviewers for their helpful comments. This work was supported in part by DFG Excellence Strategy grant 39078197 (EXC 2092, CASA), ERC grant 695022 and NSF grant CNS-1563829.

## References

[1] OpenSSL - Cryptography and SSL/TLS Toolkit. https://github.com/openssl/openssl. Accessed: 2020-10-10.

[2] OpenSSL AES x86 implementation. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aes-586.pl. Accessed: 2020-10-10.

[3] Simple C module for constant-time AES encryption and decryption. https://github.com/bitcoin-core/ctaes. Accessed: 2020-10-10.

[4] The GnuTLS Transport Layer Security Library. https://gitlab.com/gnutls/gnutls/. Accessed: 2020-10-10.

[5] Berkley, U. The sodor processor collection (on github). [Online]. Available: https://github.com/ucb-bar/riscv-sodor.

[6] Bhunia, S., Hsiao, M. S., Banga, M., and Narasimhan, S. Hardware trojan attacks: Threat analysis and countermeasures. *Proc. IEEE 102*, 8 (2014), 1229–1247.

[7] Chen, D. D., and Ahn, G.-J. Security analysis of x86 processor microcode, 2014.

[8] Costan, V., and Devadas, S. Intel SGX explained. *IACR Cryptol. ePrint Arch. 2016* (2016), 86.

[9] Daemen, J., Dobraunig, C., Eichlseder, M., Gross, H., Mendel, F., and Primas, R. Protecting against statistical ineffective fault attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020*, 3 (2020), 508–543.

[10] Dan Goodin Oct 28, . . p. U. In a first, researchers extract secret key used to encrypt intel cpu code, Oct 2020.

[11] DC, D. S. B. W. Report of the Defense Science Board Task Force on High Performance Microchip Supply, 2005.

[12] Fyrbiak, M., Strauss, S., Kison, C., Wallat, S., Elson, M., Rummel, N., and Paar, C. Hardware reverse engineering: Overview and open challenges. In *IEEE 2nd International Verification and Security Workshop, IVSW 2017, Thessaloniki, Greece, July 3-5, 2017* (2017), IEEE, pp. 88–94.

[13] Fyrbiak, M., Wallat, S., Swierczynski, P., Hoffmann, M., Hoppach, S., Wilhelm, M., Weidlich, T., Tessier, R., and Paar, C. HAL-The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. *IEEE Transactions on Dependable and Secure Computing* (2018).

[14] Google. Verified boot - the chromium projects. [Online]. Available: https://sites.google.com/a/chromium.org/dev/chromium-os/chromiumos-design-docs/verified-boot.

[15] Heller, L. C., and Farrell, M. S. Millicode in an ibm zseries processor. *IBM Journal of Research and Development 48*, 3.4 (2004), 425–434.

[16] Hicks, M., Finnicum, M., King, S. T., Martin, M. M. K., and Smith, J. M. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. *login Usenix Mag. 35*, 6 (2010).

[17] Intel Corporation. Intel issues updates to protect systems from security exploits. [Online]. Available: https://newsroom.intel.com/news-releases/ intel-issues-updates-protect-systems-security-exploits/., 2017.

[18] jallennk. Signature verification for embedded systems (on github). [Online]. Available: https://github.com/jhallen/rsa-verify.

[19] Käsper, E., and Schwabe, P. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings* (2009), C. Clavier and K. Gaj, Eds., vol. 5747 of *Lecture Notes in Computer Science*, Springer, pp. 1–17.

[20] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. *CoRR abs/1801.01203* (2018).

[21] Kollenda, B., Koppe, P., Fyrbiak, M., Kison, C., Paar, C., and Holz, T. An exploratory analysis of microcode as a building block for system defenses. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, pp. 1649–1666.

[22] Koppe, P., Kollenda, B., Fyrbiak, M., Kison, C., Gawlik, R., Paar, C., and Holz, T. Reverse engineering x86 processor microcode. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017* (2017), E. Kirda and T. Ristenpart, Eds., USENIX Association, pp. 1163–1180.

[23] Mark Smotherman. A Brief History of Microprogramming. http://ed-thelen.org/comp-hist/MicroprogrammingABriefHistoryOf.pdf. Accessed: 2020-10-10.

[24] Matrosov, A. Modern secure boot attacks: Bypassing hardware root of trust from software. *Blackhat Asia* (2019).

[25] Narayanasamy, S., Carneal, B., and Calder, B. Patching processor design errors. In *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA* (2006), IEEE, pp. 491–498.

[26] Paar, C., and Pelzl, J. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.

[27] Patterson, D., Bennett, J., Dabbelt, P., Garlati, C., Madhusudan, G. S., and Mudge, T. Embenchtm: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative recruiting for the long overdue and deserved demise of dhrystone. *RISC-V Workshop 2019* (jun 2019).

[28] Robertson, J., and Riley, M. The big hack: How china used a tiny chip to infiltrate u.s. companies, Oct 2018.

[29] Sarangi, S. R., Narayanasamy, S., Carneal, B., Tiwari, A., Calder, B., and Torrellas, J. Patching processor design errors with programmable hardware. *IEEE Micro 27*, 1 (2007), 12–25.

[30] Shirriff, K. Reverse engineering the ARM1 processor's microinstructions . [Online]. Available: `http://www.righto.com/2016/02/reverse-engineering-arm1-processors.html`, 2016.

[31] Snyder, W. verilator. [Online]. Available: `https://github.com/verilator/verilator`.

[32] Steil, M. 17 mistakes microsoft made in the xbox security system. In *22nd Chaos Communication Congress* (2005).

[33] Stoffelen, K. Efficient cryptography on the RISC-V architecture. In *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings* (2019), P. Schwabe and N. Thériault, Eds., vol. 11774 of *Lecture Notes in Computer Science*, Springer, pp. 323–340.

[34] Swierczynski, P., Fyrbiak, M., Koppe, P., and Paar, C. FPGA trojans through detecting and weakening of cryptographic primitives. *IEEE Trans. on CAD of Integrated Circuits and Systems 34*, 8 (2015), 1236–1249.

[35] Tehranipoor, M., and Koushanfar, F. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers 27*, 1 (2010), 10–25.

[36] Ward, S. A., and Jr., R. H. H. *Computation structures*. MIT electrical engineering and computer science series. MIT Press, 1990.

[37] Welch, B. L. The generalization of student's' problem when several different population variances are involved. *Biometrika 34*, 1/2 (1947), 28–35.

[38] Werner, M., Schilling, R., Unterluggauer, T., and Mangard, S. Protecting RISC-V processors against physical attacks. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019* (2019), J. Teich and F. Fummi, Eds., IEEE, pp. 1136–1141.

[39] Wolfe, A. For Intel, it's a case of FPU all over again. EE-Times [Online]. Available: `http://www.fool.com/EETimes/1997/EETimes970516d.htm`, 1997.

[40] Wolff, F. G., Papachristou, C. A., Bhunia, S., and Chakraborty, R. S. Towards trojan-free trusted ics: Problem analysis and detection scheme. In *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008* (2008), D. Sciuto, Ed., ACM, pp. 1362–1365.

## A Implementation of AES Timing Trojan in Microcode

### A.1 Trojanized *XOR* instruction

Listing 1 shows the Trojanized XOR instruction for the case-study in Section 6. The Trojan adds an overhead of 28 microinstructions (excluding the additional payload `NOP`s) to each XOR execution. To check if the operands were equal at a specific byte, we check whether the result of a byte comparison is zero. For this, we combine left and right shifts to mask the targeted byte locations. The masked result is checked against the RISC-V zero register that always contains zeros. The comparison in, *e.g.*, line 25, determines whether a jump is taken. Note that microcode generally has limited functionality and basic comparisons need to be implemented with a significant overhead.

```
1  def xor
2      # AES Trojan Timing
3      xor0: a <- rf[rs1];
4      xor1: b <- rf[rs2];
5      xor2: rf[rd] <- alu(a ^ b);
6      # 32-24
7      xort_check_32_24_0: a <- rf[rd];
8      xort_check_32_24_1: b <- 0x8;
9      xort_check_32_24_2: a <- alu(a >> b);
10     xort_check_32_24_3: a <- alu(a >> b);
11     xort_check_32_24_4: a <- alu(a >> b);
12     xort_check_32_24_5: b <- rf[x0];
13     xort_check_32_24_6: if a != b goto
           xort_check_24_16_0;
14     xorttnop: nop;
15     xorttnop: nop;
16
17     # 24-16
18     xort_check_24_16_0: a <- rf[rd];
19     xort_check_24_16_1: b <- 0x8;
20     xort_check_24_16_2: a <- alu(a << b);
21     xort_check_24_16_2: a <- alu(a >> b);
22     xort_check_24_16_3: a <- alu(a >> b);
23     xort_check_24_16_4: a <- alu(a >> b);
24     xort_check_24_16_5: b <- rf[x0];
25     xort_check_24_16_6: if a != b goto
           xort_check_16_8_0;
26     xorttnop: nop;
27     xorttnop: nop;
28
29     # 16-8
30     xort_check_16_8_0: a <- rf[rd];
31     xort_check_16_8_1: b <- 0x8;
32     xort_check_16_8_2: a <- alu(a << b);
33     xort_check_16_8_2: a <- alu(a << b);
34     xort_check_16_8_2: a <- alu(a >> b);
35     xort_check_16_8_3: a <- alu(a >> b);
36     xort_check_16_8_4: a <- alu(a >> b);
37     xort_check_16_8_5: b <- rf[x0];
38     xort_check_16_8_6: if a != b goto
           xort_check_8_1_0;
39     xorttnop: nop;
40     xorttnop: nop;
41
42     # 8-1
43     xort_check_8_1_0: a <- rf[rd];
44     xort_check_8_1_1: b <- 0x8;
45     xort_check_8_1_2: a <- alu(a << b);
46     xort_check_8_1_2: a <- alu(a << b);
47     xort_check_8_1_2: a <- alu(a << b);
48     xort_check_8_1_5: b <- rf[x0];
49     xort_check_8_1_5: if a != b fetch;
50     xorttnop: nop;
51     xorttnop: nop;
52     xortreturn: fetch;
```

Listing 1: Trojanized `XOR` instruction for the AES timing Trojan

## B Implementation of AES Fault Trojan in Microcode

Our AES fault Trojan, described in Section 7, modifies two instructions. The stateful Trojan makes use of scratch register *t4*, which holds the current state. The state machine shown in Figure 7 is encoded in the order the states occur, starting with *0x0* for the initial state and counting up for 12 states in total (*0x1, 0x2, .., 0xB).*

## B.1 Trojanized *LW* instruction

We implemented the Trojan's trigger to have minimal performance impact by optimizing for the state that occurs most often, the reset state, in trigger stage 1 (see Section 7), since it is unlikely that a word (with $2^{32}$ possibilites) is triggered by accident. We first check to see if the magic word check has already been passed as it requires significant overhead (identifier `l_ck_temp_gt_4`). Afterwards, a check is made to determine if the magic word has been received or not (identifier `l_check_magic`). If not, the next instruction is fetched and the Trojan is reset. In our case, 19 cycles are added to *LW*, although the amount varies depending on the trigger word since the constant check must be constructed in microcode, which can take additional cycles for words with fewer zeros. We trigger for the 128-bit magic word `0x0000dead 0000dead 0000dead 0000dead`. This approach allows for the use of the same magic word check for all four states in the first stage of the Trojan, otherwise an additional penalty would occur.

During the second stage (the instruction sequence stage — starting from identifier `l_t4_0x4_0`), the current state is checked and a jump is made to the associated offset check.

Every time the expected magic word or instruction offset is loaded, a transition to the next state in the FSM is made by incrementing the counter (see `l_inc_cnt`).

```
def lw
    l0: a <- rf[rs1];
    l1: b <- ig(imm_i);
    l2: daddr <- alu(a + b);
    l3: nop;
    l4: rf[rd] <- dmem[daddr] word;

    #check magic word
    #0x0000dead
    l_ck_temp_gt_4_0: a <- t4;
    l_ck_temp_gt_4_1: b <- 0x4;
    l_ck_temp_gt_4_2: if a u>= b goto
        l_t4_0x4_0;
    l_ck_magic_00: a <- 0xD;
    l_ck_magic_01: b <- 0xA;
    l_ck_magic_02: b <- b << 4;
    l_ck_magic_03: a <- alu(a | b);
    l_ck_magic_04: b <- 0xE;
    l_ck_magic_05: b <- b << 4;
    l_ck_magic_06: b <- b << 4;
    l_ck_magic_07: a <- alu(a | b);
    l_ck_magic_08: b <- 0xD;
    l_ck_magic_09: b <- b << 4;
    l_ck_magic_10: b <- b << 4;
    l_ck_magic_11: b <- b << 4;
    l_ck_magic_12: a <- alu(a | b);
    l_ck_magic_13: b <- rf[rd];
    l_ck_magic_14: if a == b goto l_inc_cnt_0;

    # reset t4
    l_reset_phase1: t4 <- 0x0; fetch;
    l_reset_phase2: t4 <- 0x4; fetch;


    l_inc_cnt_0: a <- t4;
    l_inc_cnt_1: b <- 0x1;
    l_inc_cnt_2: t4 <- alu(a + b); fetch;


    # t4 == 0x4
    l_t4_0x4_0: a <- t4;
    l_t4_0x4_1: b <- 0x4;
    l_t4_0x4_2: if a == b goto l_ck_offset_0_0;
    l_t4_0x5_0: a <- t4;
    l_t4_0x5_1: b <- 0x5;
    l_t4_0x5_2: if a == b goto l_ck_offset_1_0;
    l_t4_0x6_0: a <- t4;
    l_t4_0x6_1: b <- 0x6;
    l_t4_0x6_2: if a == b goto l_ck_offset_2_0;
    l_t4_0x7_0: a <- t4;
    l_t4_0x7_1: b <- 0x7;
    l_t4_0x7_2: if a == b goto l_ck_offset_3_0;

    l_else: fetch;

    # if offset == 0xA0; set t4 to 0x5
    l_ck_offset_0_0: a <- 0xA;
    l_ck_offset_0_1: a <- a << 4;
    l_ck_offset_0_2: b <- 0x0;
    l_ck_offset_0_3: a <- alu(a | b);
    l_ck_offset_0_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;
    # if offset == 0xA4; set t4 to 0x6
    l_ck_offset_1_0: a <- 0xA;
    l_ck_offset_1_1: a <- a << 4;
    l_ck_offset_1_2: b <- 0x4;
    l_ck_offset_1_3: a <- alu(a | b);
    l_ck_offset_1_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;
    # if offset == 0xA8; set t4 to 0x7
    l_ck_offset_2_0: a <- 0xA;
    l_ck_offset_2_1: a <- a << 4;
    l_ck_offset_2_2: b <- 0x8;
    l_ck_offset_2_3: a <- alu(a | b);
    l_ck_offset_2_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;
    # if offset == 0xAC; set t4 to 0x8
    l_ck_offset_3_0: a <- 0xA;
    l_ck_offset_3_1: a <- a << 4;
    l_ck_offset_3_2: b <- 0xC;
    l_ck_offset_3_3: a <- alu(a | b);
    l_ck_offset_3_4: b <- ig(imm_i); goto
        l_ck_offset_x_5;

    l_ck_offset_x_5: if a != b goto
        l_reset_phase2;
    l_ck_offset_x_6: goto l_inc_cnt_0;
```

Listing 2: Trojanized `LW` instruction for the AES fault Trojan

## B.2 Trojanized *XOR* instruction

The modified `XOR` instruction has been crafted to minimize overhead. It serves as the payload execution macroinstruction. In non-triggered states, only three additional cycles are executed, namely the check for the current state (see xort_t4_trig_ck). If the Trojan has not been triggered, the standard *XOR* operation is executed and the next instruction is fetched (see identifier *xor*). If the Trojan has been triggered, it is necessary to check if the code is in the last *XOR* payload states, located in xort_t4_trig. In all four of the final states, the executing payload disregards the second *XOR* operand, which leads to a transparent output of the first *XOR* operand to the return register (see xort_payload). For the first three payload executions, the counter is incremented, until the Trojan execution is finished and the state set to zero (see xort_zeroize).

```
def xor
    # check if t4 >= 0x5
    xort_t4_trig_ck_0: a <- t4;
    xort_t4_trig_ck_1: b <- 0x8;
    xort_t4_trig_ck_2: if a u>= b goto
        xort_t4_trig_0;

    # if not triggered execute regular xor
    xor0: a <- rf[rs1];
    xor1: b <- rf[rs2];
    xor2: rf[rd] <- alu(a ^ b); fetch;

    # t4 == 0x8
    xort_t4_trig_0: a <- t4;
    xort_t4_trig_1: b <- 0xB;
    xort_t4_trig_2: if a == b goto
        xort_zeroize_0;

    # increase by one
    xort_inc_0: a <- t4;
    xort_inc_1: b <- 0x1;
    xort_inc_2: t4 <- alu(a + b); goto
        xort_payload_0;

    # zeroize
    xort_zeroize_0: t4 <- 0x0;

    # malicious payload
    xort_payload_0: a <- rf[x0];
    xort_payload_1: b <- rf[rs1];
    xort_payload_2: rf[rd] <- alu(a ^ b); fetch;
```

Listing 3: Trojanized XOR instruction for the AES fault Trojan