# Hybrid Obfuscation to Protect against Disclosure Attacks on Embedded Microprocessors

Marc Fyrbiak, Simon Rokicki, Nicolai Bissantz, Russell Tessier, Christof Paar

# Hybrid Obfuscation to Protect against Disclosure Attacks on Embedded Microprocessors

Marc Fyrbiak, Simon Rokicki, Nicolai Bissantz, Russell Tessier, Christof Paar, *Fellow, IEEE*

**Abstract**—The risk of code reverse-engineering is particularly acute for embedded processors which often have limited available resources to protect program information. Previous efforts involving code obfuscation provide some additional security against reverse-engineering of programs, but the security benefits are typically limited and not quantifiable. Hence, new approaches to code protection and creation of associated metrics are highly desirable. This paper has two main contributions. We propose the first hybrid diversification approach for protecting embedded software and we provide statistical metrics to evaluate the protection. Diversification is achieved by combining hardware obfuscation at the microarchitecture level and the use of software-level obfuscation techniques tailored to embedded systems. Both measures are based on a compiler which generates obfuscated programs, and an embedded processor implemented in an FPGA with a randomized ISA encoding to execute the hybrid obfuscated program. We employ a fine-grained, hardware-enforced access control mechanism for information exchange with the processor and hardware-assisted booby traps to actively counteract manipulation attacks. It is shown that our approach is effective against a wide variety of possible information disclosure attacks in case of a physically present adversary. Moreover, we propose a novel statistical evaluation methodology that provides a security metric for hybrid-obfuscated programs.

**Index Terms**—Computer architecture, ISA randomization, software obfuscation, reverse-engineering.

## 1 Introduction

Embedded microprocessors are vital resources in a wide array of low-end computing platforms. With the continuing growth of the Internet of Things (IoT), simple processors are widely found in vehicles, appliances, health sensors, and infrastructure monitors, among other systems [1]. Often, these processors must operate in severely constrained environments with stringent power and performance demands. However, in many cases, predictable software security and reliability must be maintained.

Most previous efforts at providing software protection at the hardware level have included secure processors [2], [3], [4]. Although provably secure, these implementations focus more on preventing illegal execution of code and data security than on obscuring software control flow and algorithm implementation. For many real-world embedded systems, the storage of program code in external, untrusted memories and the knowledge of the Instruction Set Architecture (ISA) provide an unfortunate attack vector [5] regarding reverse-engineering and intellectual property protection. Hence, as soon as the ISA can be concealed from an adversary, the effort level needed to disclose critical information from the program code, such as an algorithm implementation, rises.

Increasingly, microprocessors and other circuitry are implemented in devices which include field-programmable logic [6] providing an attractive opportunity to customize a processor's control logic. Instruction decoding implemented in field-programmable logic offers a flexible solution to randomize instruction encoding among numerous embedded systems. Furthermore, such hardware-level alteration is agnostic to other processor architecture features.

In this paper, we focus on a hybrid approach to security via hardware-level obfuscation on the microarchitecture level and the use of software-level obfuscation techniques suited for embedded systems. Our approach tackles the shortcomings of existing state-of-the art ISA randomization defenses for an adversary with physical access to the target device. Particularly, we address various generic disclosure attacks for embedded systems that can be exploited to extract critical instruction encoding information. We categorize these information disclosure sources as ranging from general hardware access capabilities to program specific characteristics to the employed instruction encoding format. We then introduce our hybrid obfuscation design which defeats the different attacks and thus prevents reverse-engineering and software execution on an illegitimate platform. Our main contributions are:

- **Hardware-level Obfuscation.** We introduce an ISA randomization scheme which prevents information disclosure of a broad range of attacks. The scheme alters the microprocessor decode unit using hardware-efficient transformations. Furthermore, we restrict memory accesses using a fine-grained, hardware-level policy and actively counteract manipulation attacks using hardware-level booby traps.
- **Software-level Obfuscation.** In concert with the augmented hardware, we employ software-level obfuscation techniques including Control Flow Graph (CFG)-level and instruction-level obfuscation to remove diverse program characteristics and thus overcome the general limitations of our cost-efficient ISA randomization.
- **Coverage of Dynamic Adversaries.** We provide a detailed analysis of various information disclosure sources for a physical adversary with dynamic access to the instruction bus. We also discuss the generic


- *M. Fyrbiak, N. Bissantz, and C. Paar are with the Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany. E-mail: {marc.fyrbiak,nicolai.bissantz,christof.paar}@rub.de.*
- *S. Rokicki is with the IRISA, University of Rennes, France. Email: simon.rokicki@irisa.fr.*
- *R. Tessier and C. Paar are with the University of Massachusetts Amherst, USA. Email: tessier@umass.edu.*

shortcomings of state-of-the-art ISA randomization defenses in our adversary model.
- **Novel Evaluation Methodology.** We present a novel metric for our hybrid obfuscation approach to express the effects of different obfuscation transformations. In particular, we incorporate statistical analysis of the instruction-level distributions and similarity analysis of the dynamic CFGs.

## 2 Background and Related Work

Our work builds on previous research in secure and attack-resistant processor design. The relationship between these works and our approach is detailed below:

**Secure Processors.** Over the past fifteen years, a number of secure processors have been developed. XOM [2] protects against piracy and manipulation of application software by only allowing an application to execute code from specific memory locations. AEGIS [3] uses instruction set extensions to protect execution and access to segments of memory. The OASIS processor [4] also uses ciphered data with cryptographic keys provided by Physical Unclonable Functions (PUFs). Other processors [7] do not include cryptographic cores but instead use PUFs to obfuscate portions of processor function, e.g., instruction opcodes. In order to mask the memory accesses, Oblivious Random Access Memory (ORAM) can be utilized [8]. While these approaches provide provable security, the presence and use of cryptographic cores adds significant area and performance overheads and thus non-negligible costs [9] for the overall system. Our processor architectural changes are considerably more modest compared to cryptographic approaches and do not impact processor performance.

**Side Channel Attacks.** As a consequence of physical adversarial access to embedded devices, Side-Channel Analysis (SCA) using power consumption or electromagnetic emanation can be exploited. For example, SCA attacks on market-dominating Xilinx and Altera Static Random Access Memory (SRAM)-Field Programmable Gate Arrays (FPGAs) families [10], [11] have been shown. SCA is not only limited to cryptographic implementations. It can be also leveraged to extract the code of embedded processors based on the electromagnetic emanation [12], [13]. However, the attack technique has various limitations such as imperfect recognition rates and a restriction to opcode and not operand detection.

**Reverse-Engineering and Obfuscation.** A variety of software obfuscation and deobfuscation approaches have been developed over decades. Software security to hamper reverse-engineering has yielded various transformations to restrict static and dynamic analysis [14], [15], [16]. These transformations include code flattening, data encoding, on-demand code decryption, and virtual-machine based techniques [17]. To automatically reverse-engineer programs equipped with obfuscation transformations, automatic deobfuscation techniques have been developed [18], [19]. Note that these approaches generally require knowledge of the ISA to emulate or statically analyze the targeted program.

Statistical analyses of the targeted program are employed for malware detection and classification. For example, frequency analysis, entropy, and hidden Markov models of instructions are able to classify malware among several families [20], [21], [22]. Also, several algorithms have been proposed to measure the similarity between CFGs with the goal of malware detection [23]. In particular, different approaches were developed based on subgraphs [24] and graph edit distance [25]. Software exploitation [26] and software diversity [27] are related topics, but focus on different adversary models. In our evaluation, we merge the concepts of statistical analysis and CFG similarity to demonstrate the influence of obfuscation techniques and to define a security metric.

**ISA Randomization.** As software obfuscation suffers from the fundamental limitation of a known ISA, various approaches were developed to randomize the ISA with the goal of code injection mitigation [28], [29], [30], [31], [32], [33], [34]. These transformations modify the original instruction encoding and employ additional hardware circuitry to retrieve the original instruction prior to the decode phase. Contrary to our work, these related works focus on a different adversary model without physical access capabilities.

## 3 System and Adversary Model

In the following, we specify the assumptions for our hybrid obfuscation defense and identify the generic disclosure attacks in this setup.

### 3.1 System Model

We assume a simple and low-cost Reduced Instruction Set Computer (RISC)-based processor architecture. In particular, we assume the use of secure internal Random Access Memory (RAM) for stored data and insecure in-system (external) Read-Only Memory (ROM) for program memory. Due to the processor's simplicity, we preclude the use of caches. We further assume Memory Mapped Input/Output (MMIO) to perform communication with external peripheral devices.

Overall, we assume that the underlying Central Processing Unit (CPU) hardware is trustworthy. In contrast, the external ROM and all external bus interfaces are untrustworthy. Hardware debugging features (e.g., JTAG) that reveal values of internal registers or RAM are excluded from the device or are disabled.

### 3.2 Adversary Model

We suppose that the adversary has physical access to the target device. His main goal is the reverse-engineering of high-level information from the program such as protocols, cryptographic keys, or the algorithm(s) itself. Based on the physical access, the adversary can read and write arbitrary values in the untrusted program memory. Furthermore, the adversary is capable of dynamic read/write access to the external bus interfaces via probing and tampering with low-speed and high-speed buses. Passive side-channel analysis can be leveraged by the adversary, however, invasive attacks such as transistor level modification are outside the scope of this work.

### 3.3 Adversarial Disclosure Attacks

Instruction execution results in a variety of actions such as register and memory read/writes and status flag register updates. Hence, the attacker can obtain execution information even without knowledge of the instruction encoding.

**Control Flow Attack.** The adversary obtains the address of the next fetched instruction and thus the instruction pointer by passive access to external interfaces. Hence, all control flow instructions and thus the Dynamic Control Flow Graph (DCFG) are immediately revealed and the type (unconditional, conditional, call/return) of each executed control flow instruction is disclosed. Even in the case of an obfuscated instruction encoding, the adversary can exploit control flow instructions to reveal the encoding by manipulation of the instruction operand and observation of the next fetched instruction. Similarly, branch instructions can be exploited to check whether two register values meet a condition (depending on the specific branch opcode) at runtime through modification of the source data in the instruction operand.

**Input/Output Attack.** The adversary can observe communication from the CPU to peripheral devices through passive access to the external interfaces. Even if the instruction encoding is obfuscated, the adversary can exploit Input/Output (I/O) instructions to disclose internal, dynamic values by manipulation of the source data encoding in the instruction operand.

The following three attacks may not directly reveal the instruction encoding as they depend on the underlying CPU and instruction set, but they can aid the adversary's reverse engineering.

**System Configuration Attack.** In general, the adversary can deduce valuable system information by means of system configuration. Any displayed errors can be used to deduce sensitive information [35]. Even if the instruction encoding is obfuscated, a manipulated system configuration instruction supports the adversary's reverse-engineering efforts to understand the system, e.g., sleep activation, interrupt deactivation, or timer unit configuration. For example, if a manipulated instruction leads to the CPU entering sleep mode, the adversary can deduce which bits in the obfuscated encoding belong to the opcode field. Similarly, any displayed error discloses valuable information regarding the instruction encoding to the adversary. For example, if a manipulated instruction leads to an *invalid opcode error* the adversary can deduce which bits belong to the opcode field.

**Instruction Timing Attack.** Different instruction groups can include instructions which consume a varying number of clock cycles. For example, arithmetic instructions may require more cycles than logical instructions. Based on precise measurements of the consumed clock cycles per instruction, the adversary can disclose the targeted instruction's group [35]. For example, if a manipulated instruction leads to a different execution time after modification, the adversary can deduce which bits in the obfuscated encoding belong to the opcode.

**Correctness Attack.** The deterministic behaviour of a targeted part of a program may be determined by examining the program's output. Even if the instruction encoding is obfuscated, the adversary can gain information. In particular, if a deterministic part of program is altered in such a way that its semantic is preserved, the program output remains the same, see Sect. 9 for a detailed attack description.

The related works in Tab. 1 do not assume a physical and dynamic adversary. Thus, all publicly known ISA randomization schemes can be circumvented by use of the attacks. For example, the key $k$ of XOR-based schemes can be obtained via the control flow attack as follows: an unconditional control flow instruction $i \oplus k$ is executed and the adversary obtains the next fetched instruction and hence obtains the operand of the instruction $i$. For permuted instructions, the adversary simply toggles each bit per trial to obtain the manipulated next instruction address and thereby reveals the permutation.

| Approach | ISA Randomization Technique | CF Attack | I/O Attack |
|---|---|---|---|
| [28] | XOR | √ | √ |
| [29] | XOR / Instr. Permutation | √ | √ |
| [30] | XOR | √ | √ |
| [36] | XOR / Instr. Permutation | √ | √ |
| [34] | Instr. Perm. and Operand Subst. | √ | √ |
| Ours | see Sect. 5 | – | – |

Table 1: Overview of ISA randomization schemes and their susceptibility to control flow and I/O attacks.

## 4 Instruction Set Architecture

### 4.1 Assembly Language

The assembly language format of a microprocessor is a crucial component for an obfuscation scheme. Lst. 1 defines a generic assembly language in Backus-Naur Form (BNF). This language can be mapped to virtually any assembly language.

```
⟨Program⟩ ::= ⟨Program⟩ ⟨Inst⟩
⟨Inst⟩    ::= ⟨M⟩
            | ⟨CF⟩ ⟨imm⟩
            | ⟨DT⟩ ⟨reg⟩ ⟨reg⟩ ⟨imm⟩
            | ⟨DTC⟩ ⟨imm⟩
            | ⟨AL⟩ ⟨reg⟩ ⟨reg⟩ ⟨val⟩
⟨M⟩       := 'nop' | 'sleep'
⟨CF⟩      ::= 'call' | 'ret' | 'jmp' | 'beq' | 'bne'
⟨DT⟩      ::= 'ld' | 'st'
⟨DTC⟩     ::= 'ldc' | 'stc'
⟨AL⟩      ::= 'and' | 'or' | 'not' | 'xor' | 'sll' | 'srl' | 'slt'
            | 'add' | 'sub' | 'mul' | 'div'
⟨val⟩     ::= ⟨imm⟩ | ⟨reg⟩
```

Listing 1: Generic RISC-based assembly language in BNF clustered into instruction groups.

In general, the instructions of the assembly language can be grouped into four distinct groups:

- **Control Flow** ⟨**CF**⟩: Instructions which cause an unconditional control flow change (`jmp`), branch instructions (`beq`, `bne`), and function calls (`call`, `ret`).
- **Data Transfer** ⟨**DT**⟩,⟨**DTC**⟩: Instructions which transfer data via load (`ld`) and store (`st`) between the different memories such as the register file and the RAM.
- **Arithmetic/Logical** ⟨**AL**⟩: Instructions that modify the data contents by means of logical and arithmetic functions such as `add` and `xor`.
- **Miscellaneous** ⟨**M**⟩: All remaining instructions such as `nop` and `sleep`.

Note that the rationale for the missing registers in the branch instructions and the additional `ldc` and `stc` instructions is stated in Sect. 5.3.

### 4.2 Instruction Format

We employ the Microprocessor without Interlocked Pipeline Stages (MIPS) instruction format due to its wide use in embedded systems and its simplicity. The latter characteristic is particularly advantageous for security analysis and implementation.

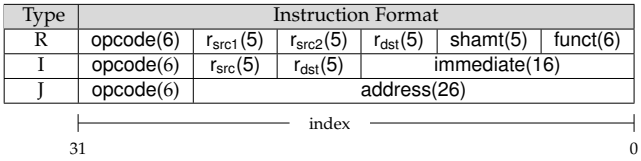



Figure 1: MIPS instruction format with 32-bit width. The bit-width of each instruction field is denoted by the number put in brackets after the field name.

Each MIPS instruction is encoded in a 32-bit vector with a 6-bit opcode. There are three distinct types of instruction formats: R-type instructions encode two source registers $r_{src1}$, $r_{src2}$ and a destination register $r_{dst}$, each 5-bit wide, a 5-bit shift amount field `shamt`, and a 6-bit function field `funct`. The `funct` field further specifies instruction operation beyond the opcode. I-type instructions employ a source register $r_{src}$, a destination register $r_{dst}$, and a 16-bit `immediate` value. The operand of J-type instructions only consists of a 26-bit `address` field.

## 5 Hardware-level Obfuscation

A fundamental limitation of software-level obfuscation techniques is the adversary's knowledge of the instruction encoding. Our hardware-level obfuscation targets this encoding knowledge with simultaneous consideration of the disclosure attacks.

### 5.1 Opcode Substitution

The principle idea of our proposed *opcode substitution* transformation is to employ a randomized encoding of the opcode field so that the adversary does not know which opcode maps to which operation. To enhance the effect of the substitution, we consider homophonic ciphers.

**Homophonic Substitution Cipher.** A homophonic substitution cipher maps each plaintext symbol to one or more ciphertext symbols, called *homophones*, thereby flattening the ciphertext symbol distribution and obstructing ciphertext symbol frequency analysis compared to simple substitution ciphers. Since their appearance, homophonic substitution ciphers have been successfully attacked by exploiting inherent characteristics of human language [37], [38], [39]. In contrast to the nature of human language, an instruction sequence can be arbitrarily altered to hide relevant statistical information[1]. Based on the concept of homophones, we informally define our employed opcode substitution as follows:

> The opcode substitution transformation randomly replaces the native instruction opcode by a pre-defined relation. Particularly, the relation is right-total and left-unique with respect to the native ISA encoding.

To obtain the native opcode during execution, the decode unit of the CPU implements the inverse mapping of the right-total, but not right-unique, and left-unique opcode substitution relation[2]. This transformation is scalable as the number of relations from one native opcode to the codomain elements can be chosen freely. For an implementation, the upper bound depends on the employed instruction format and number of supported instructions. In case of the MIPS instruction format, the 6-bit `funct` field is also affected by opcode substitution as it encodes opcode information (jointly with the 6-bit `opcode` field).

**Example.** We assume an ISA with a 2-bit `opcode` that only employs the two opcodes `00` and `01`. Opcode substitution maps the original opcodes as follows: opcode `00` is related to `10` and `01`, and opcode `01` is related to `11` and `00`. Thus, all 2-bit values are employed through opcode substitution (right-total), and any substituted opcode relates to only one original opcode (left-unique). Since opcode `00` relates to more than one value, this relation is not right-unique.

### 5.2 Operand Permutation

The instruction operand field(s) encode the quantities of the operation and thus enables data flow analysis. For example, immediate values can define branch decisions, memory accesses, or constants. Even without knowledge of the opcode (which defines the interpretation of the operand), the operand field can yield meaningful information. Therefore, we apply an *operand permutation* transformation to the instruction operand field as follows:

> The operand permutation transformation permutes the bit-indices of the operand field by a pre-defined random bit permutation. Program memory is split into chunks and each chunk of instructions uses its own randomly chosen permutation.

To reveal the native operand during execution in the CPU, the inverse permutation is selected based on the instruction address, see Fig. 2. This transformation is scalable as the number of permutations can be chosen freely. For the employed MIPS format, the whole 26-bit operand is permuted independent of the instruction type. As a result, the 6-bit `funct` field for R-type instructions is spread across the operand.

**Unobfuscated Instructions.** As a consequence of the disclosure attacks, we do not obfuscate the whole instruction set. The following instructions are neither affected by the opcode substitution nor the operand permutation.

- Control flow instructions: `jmp`, `call`, `ret`, `beq`, `bne`.
- System configuration instructions: `sleep`.
- Data transfer instructions: `ldc`, `stc`.

1. For example, the instruction `add r1, r2, 2` can be split up into an arbitrary combination of multiple `add`, `sub`, or shift instructions so that the semantic is preserved.

2. Properties of a binary relation R between the sets S and T:
Right-Total: $\forall t \in T \exists s \in S\colon (s,t) \in R$
Right-Unique: $\forall s \in S \forall t, t' \in T\colon (s,t) \in R \wedge (s,t') \in R \Rightarrow t = t'$
Left-Unique: $\forall s, s' \in S \forall t \in T\colon (s,t) \in R \wedge (s',t) \in R \Rightarrow s = s'$

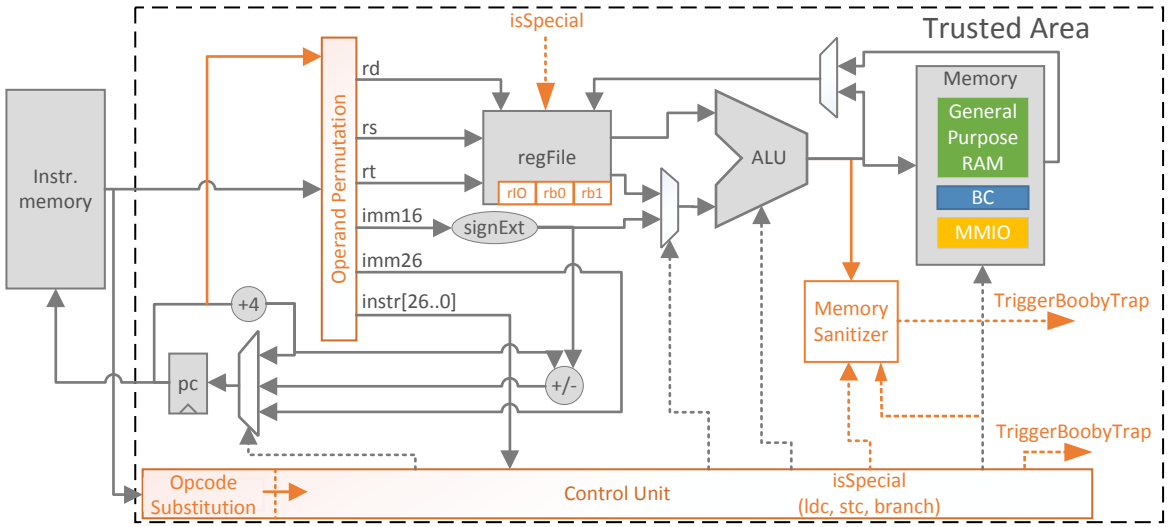



Figure 2: CPU datapath augmented with the hardware-level obfuscation features (marked in orange).

### 5.3 Hardware-enforced Access Control

Hardware-level obfuscation of the instruction encoding does not prevent instruction tampering via the I/O or control flow attacks described in Sect. 3.3. The lack of hardware-level access control on the the data transfer interface between the CPU and peripheral devices creates the vulnerability. To protect against I/O and branch instruction manipulation, we propose a lightweight hardware-based access control mechanism. Techniques and conditions to detect and respond to tampering are described in Sect. 5.4.

Hardware-enforced access control restricts the location of operand registers and memory addresses in I/O and branch instructions to a certain memory area rather than to general purpose registers. The unobfuscated data flow (`ldc`, `stc`) and branch instructions (`beq`, `bne`) are used in conjunction with three interface registers $r_{I/O}$, $r_{b0}$, $r_{b1}$ to transfer data to or from insecure memory areas (e.g., memory addresses used for I/O). These registers are isolated from the general purpose registers and cannot be accessed as general purpose register operands for hardware-level obfuscated instructions.

**Access Control Policies.** To realize access control, several hardware-level policies are employed. Internal addressable memory in the microprocessor, which includes the three interface registers, is divided into three parts: the general-purpose GP, the border control BC, and the MMIO areas.

- Hardware-level obfuscated instructions are forced to operate on locations in the GP and BC areas, but not the MMIO area.
- Unobfuscated data-flow and branch instructions are forced to operate in the BC and MMIO area, but not the GP area.
- The interface register $r_{I/O}$ (accessible via an address in the BC area) is implicitly employed by `ldc` and `stc` instructions[3]. The register is accessed by the CPU using its memory address or implicitly by the `ldc` and `stc` instructions.
- The interface registers $r_{b0}$ and $r_{b1}$ (accessible via addresses in the BC area) are implicitly utilized by branch instructions. The registers are accessed by the CPU using memory addresses or implicitly by branch instructions.

**I/O Instructions.** To transfer data between the CPU and peripheral devices, the following steps are used:

- **Read from MMIO.** To transfer data from peripheral devices to the CPU, the data is first read via an `ldc` instruction and placed in $r_{I/O}$. The data is then read from $r_{I/O}$ and placed in the internal register file via an `ld` instruction.
- **Write to MMIO.** To transfer data from the CPU to the peripheral devices, the data is first written to $r_{I/O}$ via an `st` instruction. The data is then written from $r_{I/O}$ to a MMIO address via a `stc` instruction.

**Control Flow Instructions.** Before a branch instruction is executed, the two values that are compared are loaded by the CPU into $r_{b0}$ and $r_{b1}$ via `st` instructions. Afterwards, the branch instruction that implicitly accesses the registers $r_{b0}$ and $r_{b1}$ is executed.

For the employed MIPS ISA, we augment the CPU by a further dedicated register for the call/return mechanism. The MIPS return instruction `jr ra` jumps to the value in the register `ra` which could be exploited to reveal dynamic values in the general purpose register file. Hence, the additional register is hardware-enforced to only operate on `jal`/`jr` instructions that store and restore the program counter upon execution of a call and return, respectively.

By use of hardware-level access control, the goal of the adversary is to find an appropriate obfuscated `ld`/`st` instruction to construct an attack. The key idea is that the obfuscated `st` and `ld` (that write and read data to and from the memory-mapped registers) can be hidden by software-level obfuscation techniques described in Sect. 6.

### 5.4 Hardware-level Booby Traps

In addition to hardware-enforced access control, a hardware-level tamper response mechanism is needed to prevent

3. For example, a `ldc 0xabcd` instruction loads the value from address `0xabcd` to the register $r_{I/O}$. A `stc 0xabcd` instruction stores the value from the register $r_{I/O}$ to address `0xabcd`.

disclosure of the ISA encoding. This response mechanism is added to our defense arsenal to protect against a variety of the attacks, see Sect. 9.

A booby trap is an active defense that is directly triggered by a detected attack. Such approaches have been developed for software protection [40] and for protection of high-security hardware devices. Hardware systems with tamper detection and response include hardware firewalls in smart cards [41] and the erasure of cryptographic material in response to an attack [42].

**Booby Trap Triggers.** For our system, we employ the following triggers to detect an attack:

- **Invalid Memory Access.** The access control unit triggers a booby trap once an invalid memory access by the `ld` / `st` or `ldc` / `stc` instructions is executed.
- **Dedicated Opcodes.** Several dedicated opcodes (particularly R-type instructions) are reserved to trigger a booby trap on execution.
- **Malformed Operands.** The instruction format is leveraged to detect malformed operands and subsequently trigger a booby trap on execution. For example, a non-shift R-type instruction that encodes a non-zero `shamt` value triggers a booby trap.

**Booby Trap Tamper Response.** To prevent the disclosure of the instruction encoding, the instruction decode unit can be cleared in a non-volatile manner to prevent further operation. For practical purposes, a counter could be employed to detect multiple attack attempts prior to the burning of a fuse to disable the unit.

## 6 Software-level Obfuscation

A limitation of hardware-level obfuscation is the preservation of the program structure. Despite an obfuscated instruction encoding, the CFG and instruction sequencing provide valuable information sources as demonstrated in Sect. 10. Hence, we employ well-established software obfuscation transformations to hide these information sources.

**Control Flow Graph-Level Obfuscation.** The control flow of a call graph and associated basic block topology provide viable information for a reverse-engineer. Looping and conditional structures can be exploited to deduce information regarding the high-level algorithm implementation. To obscure these traces, *code flattening* and *procedure merging* are employed [43]. *Code flattening* inserts a `switch` statement and a dispatcher that controls the basic block execution sequence. *Procedure merging* scrambles both function and interprocedural level control by *inlining* functions within each other.

**Basic Block-Level Obfuscation.** CFG-level obfuscation effectively disrupts control flow assumptions and splits large basic blocks, however small basic blocks remain unchanged. To hide small basic blocks, we utilize *basic block normalization*. The number of instructions per basic block are normalized by filling space with *garbage code* that does not affect the semantic behavior of the basic block. A range of basic block sizes are used and basic block positioning is permuted in memory to hinder static analysis.

**Instruction-Level Obfuscation.** CFG and basic block level transformations do not remove all program characteristics. Dedicated instruction sequences and memory accesses are not affected by these obfuscation techniques. To remove potential code structure assumptions such as the stack clean-up after a function call, *instruction substitution* is used which replaces specific instruction sequences by other functional equivalent but more complicated instruction sequences [43]. This technique is a vital component for our statistical evaluation.

Instruction substitution is used to hide critical `ld/st` instructions employed for hardware-level access control, see Sect. 5.3. For example, an instruction substitution rule can be easily defined for (virtually) any architecture which replaces a `ld` instruction by a stack pointer displacement, a `pop`, and a subsequent original stack pointer recovery. The stack pointer displacement can be further constituted of multiple instructions so that the basic block normalization spreads this information among diverse basic blocks. Note that the register value that is added to the immediate value in the `ld/st` instructions can be arbitrarily changed while the sum of both values (the target address) remains the same. This effect greatly increases the number of distinct representations.

## 7 Implementation

To demonstrate our hybrid obfuscation approach we have developed a soft microprocessor generation framework to automatically implement protected CPUs. This framework is supported with a full compilation flow which performs our structural transformations.

**Hardware Implementation.** Our prototype hardware implementation employs the SPREE soft processor generation framework [6]. Processors generated with the framework use the MIPS instruction format. A Register Transfer Level (RTL) description of specific processor instantiations are generated from textual descriptions of the ISA and processor data path. An RTL generator was written to include hardware for a randomized ISA encoding and the insertion of the hardware units described in Sect. 5. The number of opcode homophones and operand permutations in the processor hardware implementation are user-defined with respect to the instruction format. Similarly, the number of booby trap triggering opcodes and the memory layout are also user-defined.

**Software Implementation.** Our prototype compiler implementation leverages the LLVM/Clang compiler and the Obfuscator-LLVM environment developed by Junod *et al.* [43]. The latter framework provides transformations such as the code flattening and instruction substitution. We extended the Obfuscator-LLVM environment to implement basic block normalization.

To allow for late code address determination, we modified the GNU assembler to generate two non-linked `.o` object files. Each object file is created with different encoded ISAs: ISA A and ISA B. The object files are then linked to their libraries, which were also encoded with ISAs A and B. Relocatable instructions are then identified by examining instructions that differ in the two files after obfuscation is removed. Once all relocatable instructions have been processed, an object file is linked and address dependent operand permutation

is applied. The final executable file is fully linked and ISA encoding is employed.

## 8 Performance Evaluation

In this section we present area and run-time performance results for the hardware and software obfuscation transformations.

**Area Overhead.** We evaluated the area overheads for varying hardware-level obfuscation parameters in contrast to dedicated CPU units based on the SPREE processor generation framework. Hardware designs were synthesized to an Altera Cyclone V A2 FPGA. All designs were tested via simulation to determine their accurate behavior.

| Design | LUTs | FFs | Mem. Bits |
|---|---|---|---|
| SPREE + BT | 915 | 204 | 2048 |
| SPREE + BT + OS + OP(1) | 933 | 204 | 2304 |
| SPREE + BT + OS + OP(2) | 936 | 204 | 2304 |
| SPREE + BT + OS + OP(4) | 976 | 204 | 2304 |
| SPREE + BT + OS + OP(8) | 1015 | 204 | 2304 |
| SPREE | 889 | 202 | 2048 |
| **Available Resources** | 18868 | 112960 | 1802000 |

Table 2: Hardware area overhead for the additional obfuscation elements.

Tab. 2 lists the utilized hardware resources in terms of Look-up tables (LUTs) and Flip Flops (FFs) required for the different hardware units. The booby trap (**BT**) unit realizes the logic to trigger a booby trap based on incorrect memory accesses, malformed instructions, and for dedicated opcodes as described in Sect. 5.4. We considered two dedicated opcode triggers and one MMIO address. The opcode substitution (**OS**) and the operand permutation (**OP**) implement the hardware-level transformations described in Sect. 5.1 and Sect. 5.2, respectively. The **OS** designs include a homophonic substitution which employs each possible value for the opcode and funct fields, the extended register file, and support for `ldc/stc` instructions. **OP(x)** denotes that $x$ distinct operand permutations are implemented.

The hardware overhead of the **BT** circuitry is lightweight in terms of additional LUTs and FFs and the overhead of the **OS** consists only of several LUTs. Increasing the number of operand permutations **OP** results in a hardware overhead of up to 14% for LUTs compared to the original SPREE processor. The 36 registers (= 32 general purpose registers + $r_{I/O}$ + $r_{b0}$ + $r_{b1}$ + 1 call/ret register) are implemented in the memory blocks of the FPGA. Similarly, the internal 64 kB RAM is implemented in the memory blocks of the FPGA.

The processor speed is not affected by the hardware-level augmentations listed in Tab. 2. The additional elements do not affect the critical path of the design in the execute stage of the three-stage processor (ALU and regFile). Our modifications are restricted to the instruction fetch and memory stages.

**Performance and Memory Overhead.** We evaluated the performance overhead for the different obfuscation transformations on the SPREE embedded benchmark suite [6]. The obfuscation strategies were applied to the programs and the clock cycle counts for 100 versions of each program were measured. In the following discussion, the employed obfuscation transformations are abbreviated as follows: opcode substitution (**OS**), opcode permutation (**OP**), instruction substitution (**IS**), code flattening (**CF**), and basic block normalization (**BBN**). Procedure merging was not considered for the evaluation as the targeted benchmark programs generally consist of only one function.

Tab. 3 lists the performance results for the different obfuscation strategies for the SPREE benchmark programs. The average performance slowdown of **IS** ranges between approx. 1.1× and 2× for all programs except `des`. For `des`, the influence of **IS** is significant as the cryptographic algorithm employs numerous `xor` instructions that are swapped for more complex representations. The **CF** transformation leads to performance slowdown based on the underlying program structure. Since the implementation of the `des` program is loop-free, **CF** does not affect performance. For non-loop-free programs, the slowdown depends on high-level program structure. Measured slowdown ranges between approx. 2× and 6×. For the combined **OS+OP+CF+IS** strategy the effect of both transformations is additive. The **CF** transformation adds a switch statement to the target program and uses a variable to control execution order. This transformation is not affected by the **IS** implementation. The combined obfuscation approaches **OS+OP+CF+IS+BBN** affect performance loss more significantly. Transformation **CF** splits up basic blocks which are then padded by **BBN**. Hence, performance is slowed down by a factor of approximately 4× to 39× compared to the unobfuscated version.

Tab. 4 states program size for the different obfuscation strategies. For the **IS** transformation, most targeted programs are only increased by several hundred bytes. For reasons similar to those given for performance evaluation, `des` is strongly influenced by **IS**. The **CF** transformation affects program size depending on the underlying program structure. Since the targeted programs in the suite contain a small number of loops and branches, program size increases by less than 1 kB compared to the **OS+OP** obfuscated programs. The combined obfuscation strategy **OS+OP+CF+IS+BBN** increases the binary size as each basic block is padded with instructions. A binary size overhead of up to 3 kB is seen for the targeted programs compared to the **OS+OP** obfuscated programs. The `des` program size increases by 4× compared to the **OS+OP** obfuscated program.

In summary, software protection naturally affects code performance and binary size depending on the degree of obfuscation. Our program slowdown and binary size results are in line with several other related works in this field, see Sect. 2. The performance and binary size effects of hardware obfuscation are much more limited.

## 9 Security Analysis

In this section, a security analysis is provided for the diverse adversary accessible attacks presented in Sect. 3.3.

**Hardware-level Booby Traps.** The booby trap mechanism provides a crucial anchor to prevent the disclosure of processor's instruction encoding in response to an attack. To the best of the authors' knowledge, current SRAM-FPGA families only support fuses which can be programmed via external FPGA access. Relying on this type of fuse programming risks disruption by the adversary. Alternatively, a booby trap can be used in which the adversary triggers the device to reload its entire configuration before the soft microprocessor

| | bubbl. | crc | CRC32 | des | fact. | fft | fir | iquant | quant |
|---|---|---|---|---|---|---|---|---|---|
| **OS+OP** | 8012 | 24611 | 568886 | 1097 | 139 | 3067 | 1085 | 3401 | 3697 |
| **OS+OP+IS** | 8308 | 45930 | 1085654 | 4428 | 143 | 3784 | 1286 | 4485 | 4425 |
| **OS+OP+CF** | 36687 | 41514 | 1165379 | 1097 | 454 | 18187 | 7519 | 11217 | 10020 |
| **OS+OP+CF+IS** | 38105 | 61237 | 1641889 | 4429 | 455 | 19659 | 7801 | 12710 | 11854 |
| **OS+OP+CF+IS+BBN** | 48293 | 191189 | 6609719 | 4439 | 1732 | 119486 | 22160 | 35451 | 42932 |
| **Unobfuscated** | 8012 | 24611 | 568886 | 1097 | 139 | 3067 | 1085 | 3401 | 3697 |

Table 3: Software performance evaluation for the obfuscation strategies. Each result indicates the number of cycles arithmetically averaged over 100 programs.

| | bubbl. | crc | CRC32 | des | fact. | fft | fir | iquant | quant |
|---|---|---|---|---|---|---|---|---|---|
| **OS+OP** | 1.42 | 1.40 | 1.56 | 5.92 | 1.35 | 2.04 | 1.94 | 1.93 | 1.98 |
| **OS+OP+IS** | 1.44 | 1.57 | 1.92 | 20.62 | 1.36 | 2.20 | 2.07 | 2.12 | 2.15 |
| **OS+OP+CF** | 1.87 | 1.73 | 2.03 | 5.92 | 1.86 | 2.95 | 2.68 | 2.62 | 2.71 |
| **OS+OP+CF+IS** | 1.89 | 1.90 | 2.34 | 20.50 | 1.86 | 3.15 | 2.83 | 2.96 | 3.03 |
| **OS+OP+CF+IS+BBN** | 2.39 | 2.77 | 3.33 | 20.69 | 2.33 | 4.83 | 3.73 | 4.11 | 4.33 |
| **Unobfuscated** | 1.42 | 1.40 | 1.56 | 5.92 | 1.35 | 2.04 | 1.94 | 1.93 | 1.98 |

Table 4: Software size evaluation for the obfuscation strategies. Each result indicates the program memory size in kB arithmetically averaged over 100 programs.

is reactivated. This action requires a time-consuming process. For example, the smallest Cyclone V device (A2) requires 21,061,028 configuration bits [44]. Configuration can be performed 16 bits at a time at 125 MHz. Thus, configuration requires at least 10.5 ms per booby trap trigger.

**Control Flow Attack.** `jmp`, `call`, and `ret` instructions cannot be exploited to disclose instruction encoding information as the instructions are not affected by hardware-level obfuscation. We illustrate our resistance to the control flow attack via an example. In a possible attack, the attacker would write values from the register file to interface registers $r_{b0}$ and $r_{b1}$ to compare the values using an unobfuscated `beq` instruction. Therefore, the attacker must craft two store instructions `st r0,r0,imm0` and `st r0,r0,imm1`, where `imm0` and `imm1` are the addresses of $r_{b0}$ and $r_{b1}$, respectively. To accomplish the attack, the attacker must guess the opcode ($2^6 - k$), where $k$ is the number of known opcodes. In addition, he must guess the operand permutation for the 16-bit immediate value ($\binom{26}{16}$ steps) and the two addresses ($2^{16} \cdot (2^{16} - 1)$ steps). To algorithmically verify the guessed instruction encoding, the attacker could attempt to transfer values from all 32 registers and check for equality. Even without consideration of the booby trap mechanism, the worst-case attack complexity for $k = 10$ known opcodes is approximately $2^{65}$. Note that for the MIPS architecture register $r0$ always holds a constant zero.

**Input/Output Attack.** Similar to the branch instruction attack, the adversary could leverage writes to MMIO addresses to reveal the instruction encoding, although the amount of time required to perform the attack is prohibitive. He must craft a store instruction `st r0,r0,imm`, where `imm` is the address of $r_{I/O}$. Hence, the attacker must guess the opcode ($2^6 - k$ steps), the operand permutation for the 16-bit immediate value ($\binom{26}{16}$ steps), and the address value ($2^{16}$ steps). Even without consideration of the booby trap mechanism, the worst-case attack complexity for $k = 10$ known opcodes is approximately $2^{49}$ (including the 31 checks to determine if the written values are distinct).

Furthermore, the evaluation of all possible 32-bit instructions will trigger $520,093,696 \approx 2^{28.95}$ booby traps due to the invalid `shamt` field (for 8 non-shift R-type instructions). Hence, the reconfiguration time for an A2 FPGA is approx. 63 days as each reconfiguration requires at least 10.5 ms. Even for a randomly chosen instruction value, the probability that it triggers the `shamt` field booby trap is around 12%.

**Correctness Attack.** As a consequence of the homophones in the opcode substitution, the adversary can leverage the correctness attack to reveal parts of the instruction encoding. The homophones are mainly implemented in the funct field of R-type instructions. For simplicity, we assume that the targeted program is deterministic so that the same inputs compute the same output values and the adversary is able to observe both.

The attacker would like to substitute homophones for an instruction with just one representative value to reveal parts of the instruction encoding. If the adversary alters the funct field to a correct homophone opcode, the output of a deterministic algorithm will not change. An incorrect homophone yields an incorrect opcode or operand and hence a different output. To complete the attack, the attacker must guess the opcode ($2^6 - k$ steps), the operand permutation for the 6-bit immediate value ($\binom{26}{6}$ steps), and the funct value ($2^6$ steps). Even without consideration of the booby trap mechanism, the worst-case attack complexity for $k = 10$ known opcodes is approximately $2^{29}$ deterministic program executions.

**System Configuration / Instruction Timing Attack.** Since system configuration instructions are not affected by the hardware-level obfuscation and all obfuscated instructions consume the same number of clock cycles, both attacks cannot be exploited.

**Cautionary Note.** All the attack strategies noted above require the use of attacker-controlled values to perform a hypothesis test. If the targeted program deliberately contains an `st ra,rb,imm` instruction in which `imm` is the address of an interface register and `rb` is attacker controlled ($r0$ in the case of MIPS), the attack complexities are significantly reduced. The adversary only must guess the `ra` encoding in the operand ($\binom{26}{6} \approx 2^{17}$ steps) and analyze the register file ($2^5 - 1$ steps). Hence, the system designer must analyze whether such an `st ra,rb,imm` instruction exists and

obfuscate the instruction accordingly.

In summary, hardware-level obfuscation increases the adversary's efforts. A booby trap is likely to be triggered via an invalid memory access for a guessed `st` instruction or a dedicated opcode for R-type instructions. To increase the probability that the average-case attack also triggers a booby trap, the system designer can adjust the parameters for the booby trap triggers prior to processor generation.

## 10 Security Metrics for Obfuscation

Obfuscation deters algorithm reverse-engineering and analysis of the algorithm's internal architecture. A current limitation of obfuscation is the lack of a metric to measure the degree of obfuscation for different approaches. The generic concept of indistinguishability provides a formal treatment and offers provable arguments for obfuscation from the theoretic point of view. However, cryptographic program obfuscation schemes are still far away from being deployable [45], especially for embedded systems with constrained resources. A practical measure of obfuscation degree for software-only obfuscation has been limited by the attacker's knowledge of the targeted ISA and, thus, an ability to emulate and analyze the targeted program. This situation changes for hardware-level obfuscation systems due to the concealed ISA encoding. In the following, we propose a novel evaluation methodology to provide an obfuscation metric for hybrid obfuscated systems.

### 10.1 Similarity Metric

A key characteristic of obfuscation is to (virtually) destroy any correlation between an obfuscated and an unobfuscated program. For example, suppose there are two distinct programs $\mathsf{P}_1$, $\mathsf{P}_2$ and their obfuscated versions $\mathcal{O}(\mathsf{P}_1)$, $\mathcal{O}(\mathsf{P}_2)$. If there exists a significant correlation of a characteristic between $\mathsf{P}_1$ and $\mathcal{O}(\mathsf{P}_1)$, but no significant correlation for this feature between $\mathsf{P}_1$ and $\mathcal{O}(\mathsf{P}_2)$, the obfuscated program $\mathcal{O}(\mathsf{P}_1)$ can be matched to its unobfuscated counterpart. For example, the similarity of the DCFG or the entropy of the instruction opcodes could be used as such characteristics. Thus, a goal of obfuscation is to make obfuscated programs as similar as possible so that their measurable quantities do not allow a distinction between them

**Methodology.** To examine the program similarity created by obfuscation strategies and, thus, the degree of obfuscation, we implement the following: First, a set of programs $\mathcal{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ is selected. Obfuscated versions for each program in $\mathcal{P}$ are then generated. For an obfuscation strategy $\mathcal{O}$, multiple versions per program are generated to increase the coverage of the assessment. Then, a set of similarity comparison algorithms $\mathcal{A} = \{\mathbb{A}_1, \ldots, \mathbb{A}_m\}$ are employed and the similarities between the obfuscated and unobfuscated programs are examined.

We assume that the adversary can implicitly obtain the function call graph and, hence, some functions of the program. For example, several program functions may be provided by an adversary-accessible open-source or closed-source library. Hence, the adversary is able to examine the similarity of numerous functions in a targeted program (instead of the whole program) to break the obfuscation. This examination allows for testing of function-level similarity that is more fine-grained than similarity analysis for a larger program. Furthermore, this evaluation methodology uses a concept which is similar to computational indistinguishability. Nevertheless, without the use of strong cryptographic primitives, the property of computationally indistinguishability cannot be guaranteed. Our goal is not indistinguishability from a uniform distribution, but rather eliminating similarity between obfuscated programs and their unobfuscated counterparts, which is sufficient for practical purposes.

**Advantages.** Our proposed evaluation methodology is particularly beneficial for hardware-level obfuscation schemes as the adversary cannot emulate and thus reverse-engineer the obfuscated program without the corresponding ISA encoding. Furthermore, this assessment roadmap is generic in the sense that new algorithms can be developed and added to the set of similarity comparison algorithms. In this way we can (automatically) examine obfuscation benefits against a defined set of attacks and provide a measure of the degree of obfuscation for a specific obfuscation strategy applied to certain programs. Notably, we can identify programs for which the selected obfuscation strategy might not be sufficient to bring the program set to a specific measurable obfuscation level.

**Limitations.** Despite various advantages, we acknowledge that the measurability approach does have certain limitations. Similar to ORAM [8] and cryptographic obfuscation [46], the I/O behaviour cannot be modeled. However, embedded systems are generally equipped with less I/O than general-purpose systems with a rich-featured operating system. It can be said that we cannot conclude from the statistics that a particular obfuscation strategy avoids successful attack. However, the statistics do indicate that at least certain global properties can be successfully obfuscated and certain strategies generally lead to a poor obfuscation. A further arguable issue is the selection of our target program set. In our case, we exclude specially crafted programs that would still have measurable similarity after the obfuscation since we want to provide a measure for programs more typically deployed by users.

Similarity measures of external data memory are outside the scope of our metrics. Data randomization schemes [27] could be utilized to dynamically encode/decode the internal, trusted RAM before it is stored to/loaded from the external, untrusted data memory.

### 10.2 Case Study – SPREE Benchmark Suite

We evaluated the programs of the SPREE benchmark suite [6] with the evaluation methodology described in the previous section. The statistical distributions of the instruction memory (Sect. 10.2.2) and the dynamic CFG (Sect. 10.2.3) for the different obfuscation strategies were investigated. It should be noted that almost all programs in this benchmark suite consist of only one vital function making the program set an ideal candidate suite for our evaluation methodology.

#### *10.2.1 Statistical Background*

To allow the reader to better interpret our results, we provide a concise summary of our measures and the rationale behind their use. Our evaluation measures are based on the number of appearances of a certain 6-bit opcode $o \in \{0,1\}^6$ in a

program $\mathsf{P}$ denoted by $\mathcal{N}_\mathsf{P}(o)$. Similarly, opcode triples and the number of their appearances are denoted by $o^3 \in \{0,1\}^6 \times \{0,1\}^6 \times \{0,1\}^6$ and $\mathcal{N}^3_\mathsf{P}(o^3)$, respectively. Moreover, $\mathcal{F}_\mathsf{P}(o)$ is the empirical distribution for the opcode $o$ of a program $\mathsf{P}$, i.e.

$$\mathcal{F}_\mathsf{P}(o) = \frac{\mathcal{N}_\mathsf{P}(o)}{\sum_i \mathcal{N}_\mathsf{P}(i)}, \quad i \in \{0,1\}^6$$

For completeness, we performed instruction operand analysis for our statistical measures. However, the 26-bit operand distributions did not provide meaningful results. Subsequently, we evaluated hashes of the operands using the measures. However, the results for the least significant 6-/7-/8-bit of the operand hashes were similar to the opcode results, hence we omit these results from the following discussion.

**Entropy.** The Shannon entropy is a measure of the information content of a random variable. In our case, we are interested in the entropy of the opcode and operand distributions as we expect that a larger entropy hints at better obfuscation (due to less pronounced peaks).

***Definition 1 (Shannon Entropy).*** The Shannon entropy of a program is determined by:

$$\mathcal{E}(\mathsf{P}) = -\sum_o \mathcal{F}_\mathsf{P}(o) \cdot \log_2(\mathcal{F}_\mathsf{P}(o))$$

**Standard Deviation.** The standard deviation is a measure of the inhomogeneity of a random variable. In our case, we evaluate the standard deviation for the frequency of opcode triples rather than a single opcode value as triplet distributions are also employed for the frequency analysis of simple cryptographic ciphers. The larger the difference in frequency of certain triples, the larger the standard deviation can be. This metric indicates which unobfuscated programs relate to an obfuscated one.

***Definition 2 (Adapted Standard Deviation).*** The adapted standard deviation $\mathrm{sd}_3(\mathsf{P})$ is determined by

$$\mathrm{sd}_3(\mathsf{P}) = \sqrt{\frac{1}{n_+ - 1} \sum_{o^3} (\mathcal{N}^3_\mathsf{P}(o^3) - \nu(\mathsf{P}))^2}$$

$$\text{for } \mathcal{N}^3_\mathsf{P}(o^3) > 0, \quad n_+ := |\{o^3\}|, \quad \nu(\mathsf{P}) := \frac{1}{n_+} \sum_{o^3} \mathcal{N}^3_\mathsf{P}(o^3)$$

We additionally performed analysis for the adapted standard deviations $\mathrm{sd}_2$ and $\mathrm{sd}_4$ based on pairs and 4-tuples of consecutive opcodes. However, these results were similar to the results of $\mathrm{sd}_3$, hence we omit these results from the following discussion.

**$\mathcal{E}$-$\mathrm{sd}_3$ Information.** The combined information of entropy $\mathcal{E}$ and $\mathrm{sd}_3$ was also considered. For two distinct program $\mathsf{P}_1$ and $\mathsf{P}_2$, the marginal distributions of the $\mathcal{E}$ and $\mathrm{sd}_3$ may strongly overlap so that the programs cannot be told apart. However, $\mathcal{E}$ and $\mathrm{sd}_3$ may be strongly positively correlated for $\mathsf{P}_1$, but anticorrelated for $\mathsf{P}_2$. As a result, the programs form distinct clusters of points in an $\mathcal{E}$-$\mathrm{sd}_3$ diagram and hence offer a distinction between the programs $\mathsf{P}_1$ and $\mathsf{P}_2$.

**Correlation.** The correlation of the distributions between the unobfuscated and obfuscated programs was also considered as a statistical measure. The Spearman correlation was employed as this correlation measures the amount by which two variables are connected by a monotonous trend. This measure contrasts with the more restrictive assumption of linearity in case of the Pearson correlation. The Spearman correlation also achieves increased robustness against possible outliers which heavily impact the Pearson correlation, i.e., opcodes having a (close to) zero frequency. The Spearman correlation uses the ranks of the observations as opposed to the observations themselves (Pearson's correlation). In our case, we examined the correlation between the ranked opcode frequency distribution for obfuscated and unobfuscated programs.

***Definition 3 (Spearman Correlation).*** The Spearman correlation for two program $\mathsf{P}_x$ and $\mathsf{P}_y$ is determined by

$$\rho(\mathcal{N}_{\mathsf{P}_x}, \mathcal{N}_{\mathsf{P}_y}) = 2^{-6} \cdot \sum_{o \in \{0,1\}^6} \left(\frac{\mathrm{rk}()\mathcal{N}_{\mathsf{P}_x}(o)) - \hat{\mu}_{\mathsf{P}_x}}{S_{\mathsf{P}_x}}\right) \cdot \left(\frac{\mathrm{rk}(\mathcal{N}_{\mathsf{P}_y}(o)) - \hat{\mu}_{\mathsf{P}_y}}{S_{\mathsf{P}_y}}\right)$$

for the mean $\hat{\mu}_\mathsf{P} := 2^{-6} \cdot \sum_{o \in \{0,1\}^6} \mathrm{rk}(\mathcal{N}_\mathsf{P}(o))$ and the standard deviation $S_\mathsf{P} := \sqrt{2^{-6} \cdot \sum_{o \in \{0,1\}^6} (\mathrm{rk}(\mathcal{N}_\mathsf{P}(o)) - \hat{\mu}_\mathsf{P})^2}$, where the pairs $(\mathrm{rk}(\mathcal{N}_{\mathsf{P}_x}(o)), \mathrm{rk}(\mathcal{N}_{\mathsf{P}_y}(o)))$ are the ranks of the observed numbers of appearances $\mathcal{N}_{\mathsf{P}_y}(o)$ which are determined separately for each program.

### 10.2.2 Statistical Analysis of the Instruction Memory

Using the above statistical measures, evaluation results for the instruction memory were generated. Hybrid obfuscation was applied to each program. A total of 100 different ISA encodings per obfuscation technique were applied. The obfuscation transformations in the following discussion are abbreviated as in Sect. 8.

**Entropy.** Fig. 3 depicts the entropy of the instruction opcodes distribution for the increasingly sophisticated obfuscation strategies (a) - (d). The blue points below 3.0 depict the entropy of the unobfuscated programs in each figure (a) - (d). The boxes depict a sketch of the entropy distribution of the obfuscated programs, where the thick horizontal line within the box is the median and the box extends from the lower $25\%$ to the upper $75\%$ quantile, i.e., $\approx 25\%$ of the programs have entropies below the lower edge and $\approx 25\%$ have entropies above the upper edge of the box. The whiskers extend to the smallest and largest entropy and the circles outside of the region covered by the whiskers cover even more extreme entropies (only contained in some figures).

**OS+OP** performs the poorest with many outliers, small boxes, and a range of medians covering a range of approximately $4.1 - 5.0$ in entropy. For **OS+OP+IS** and **OS+OP+CF+IS**, there are sharp distributions without significant outliers (visible as the isolated points below and above the boxes). **OS+OP+CF+IS+BBN** combines the best of both groups: the boxes are nearly as large as **OS+OP+CF+IS** and there is approximately the same homogeneity of box positions (approximately 50% less variable for a large fraction of programs). This obfuscation strategy generates a significant number of outliers which complicates the determination of which program is under inspection.

The entropy provides an effective measure to quantify the extent of program information loss due to obfuscation. Here, we use the degree of homogeneity of the box locations as well as the percentage of entropy values which are far away from the typical values, i.e., the outliers.

**Standard Deviation.** Fig. 4 shows the boxplots of the standard deviation $\mathrm{sd}_3$ of the instruction opcode distribution for the increasingly sophisticated obfuscation strategies (a) - (d). Note that a favourable obfuscation strategy should result

in boxes which overlap for different programs and not allow for programs to be uniquely distinguished.

This goal is best achieved by the **OS+OP+CF+IS+BBN** obfuscation which produces the largest boxes and the greatest overlap in comparison to the other obfuscation methods. In contrast to the entropy metric, $\mathrm{sd}_3$ considers the variability in the distribution of opcode triples. A large variability in $\mathrm{sd}_3$ for different obfuscations of the same program illustrates a large variability in the homogeneity of the frequencies in which certain opcodes appear in consecutive order rather than the homogeneity of the frequencies in which certain opcodes appear overall in the code. Hence, $\mathrm{sd}_3$ provides important supplementary information to entropy to quantify the extent of the information loss due to obfuscation and also to specify the variability of different obfuscations of the same program.

**$\mathcal{E}$-sd$_3$ Information.** Fig. 5 combines the entropy information and the $\mathrm{sd}_3$ values for the opcodes by displaying a point in an entropy-$\mathrm{sd}_3$ coordinate system for each program. A small dot is shown for each obfuscated program and a star is shown for an unobfuscated program. The colors for the dots indicate the program from which the obfuscated program was generated.

We see that the point clouds and the overlap between clouds for different programs are much larger for the **OS+OP+CF+IS+BBN** obfuscation technique than for all others. While this effect is also discernible in the boxplots of the marginal distributions of the entropies and the $\mathrm{sd}_3$, the **$\mathcal{E}$-sd$_3$** diagram demonstrates that these quantities are not correlated or anticorrelated in a way which would allow the determination of a program under consideration. Such a correlation would show that a large entropy value coincides with a large $\mathrm{sd}_3$. This correlation would result in disjoint clusters of points (alignment parallel to the main diagonal in the diagram). Thus, the **$\mathcal{E}$-sd$_3$** diagram provides essential information of the combined measures to depict the information loss due to obfuscation.

**Correlation.** Fig. 6 depicts the results of the correlations for the opcodes of the `fft` program, a typical case. Each panel shows the correlations between the opcode frequencies for a specific obfuscation. The boxes illustrate the distribution of the correlations between the obfuscated programs and the unobfuscated `fft` and the stars show the correlations between the unobfuscated programs and the unobfuscated program `fft`.

The correlations between the obfuscated programs and the true underlying program `fft` are not significantly different across all obfuscation approaches. Hence, the correlations do not identify the true underlying program since the hardware-level obfuscation is sufficient to hamper program distinction.

#### 10.2.3 Dynamic Control Flow Graph Similarity

Since the adversary has access to the DCFG, the similarity between obfuscated and unobfuscated DCFGs was evaluated for the SPREE benchmark suite. For each obfuscation strategy, we generated 100 programs per benchmark and extracted each DCFG. As described in Sect. 2, several algorithms have been proposed to measure the CFG similarity. In a recent evaluation, Chan *et al.* [23] demonstrated that the graph edit distance algorithm proposed by Hu *et al.* [25] is most efficient in terms of accuracy and run time. This approach was used to determine the similarity score of our recorded DCFGs.

Fig. 7 shows the results of our DCFG similarity evaluation. Obfuscated programs are compared to their unobfuscated versions in (a) and (b). A similarity score close to 1 implies that the graphs are similar, whereas a score close to 0 implies the opposite. The figure shows that a targeted program can be uniquely distinguished among the set of obfuscated programs if the DCFG is not affected by the obfuscation (Fig. 7 (a)). DCFG similarity between obfuscated and unobfuscated programs decreases as more obfuscation techniques are combined, hampering unique identification (Fig. 7 (b)). For example, the `factorial` program in Fig. 7 (b) cannot be distinguished from the set of programs.

Based on the statistical and DCFG evaluation results, it is apparent that the information characteristic for the `des` cryptographic algorithm stands out compared to other general-purpose embedded programs. Overall, we see that just using hardware-level obfuscation is not sufficient to hide crucial program characteristics. It must be combined with software-level transformation in order prevent unique distinction by the various measures.

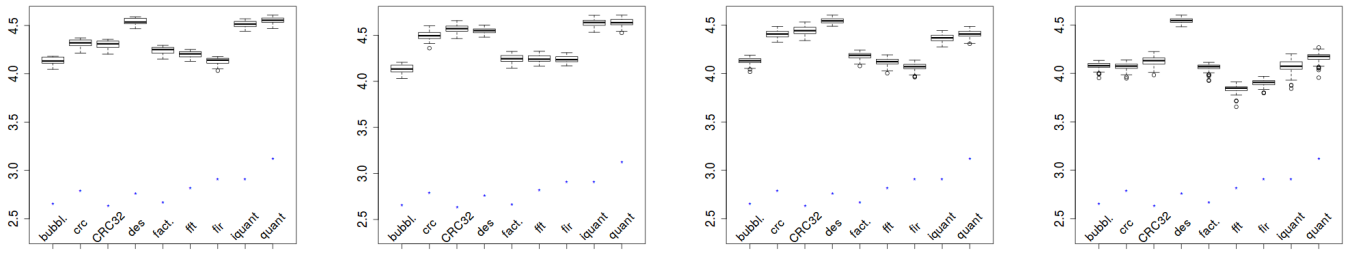
## 11 Discussion

In the following we analyze the diverse properties of our hybrid obfuscation scheme and discuss its security.

In general, hardware I/O mechanisms and instruction encoding format have crucial impacts on the security of hardware-level obfuscation. Our proposed lightweight processor augmentations mitigate generic adversary accessible attacks to hide the vital ISA encoding from a physical adversary. Furthermore, we are able to detect and respond to tampering attempts by the use of a booby trap mechanism. We have demonstrated how ISA encoding diversification can be implemented so that processors augmented with hardware-level features can be automatically generated. As a consequence of the randomized ISA encoding, the adversary is not able to directly disassemble and reverse-engineer a targeted program. Nevertheless, the ISA encoding itself is not sufficient in our adversary model as crucial program characteristics and hence we employ diverse software-level obfuscation transformations ranging from the CFG-level to the instruction-level.

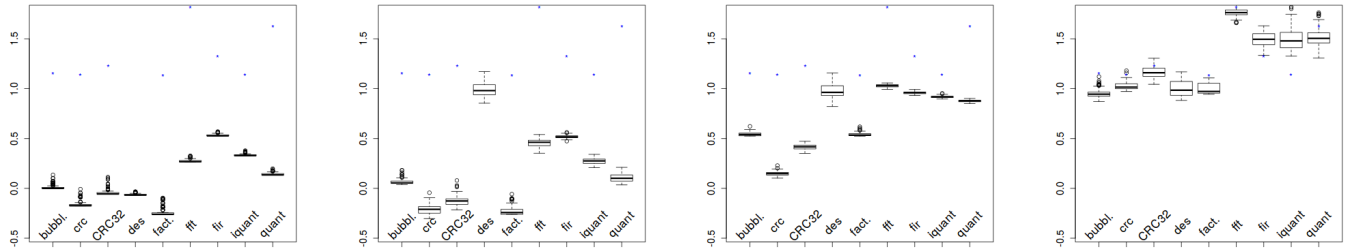
Note that this approach does not affect testability during development as the obfuscation can be selectively turned off, so that general-purpose user code can be debugged. Since we employ an integrated and automated compilation flow for the hardware-level and software-level obfuscation, a developer has full access to all compiler log files as well as the hardware-level instruction encoding mapping.

Perhaps most importantly, the benefits of hybrid obfuscation transformations have been evaluated with statistical evaluation metrics. It has been demonstrated that it is not possible to match an obfuscated program to one of a group of unobfuscated ones by considering a selection of statistical metrics (Sect. 10.1). The hardware overhead of our approach is about 14% of processor logic area.



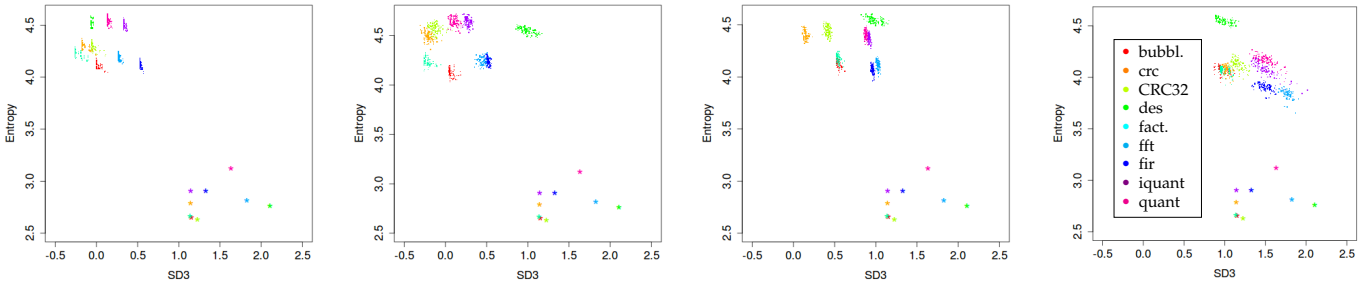

(a) **OS+OP**. (b) **OS+OP+IS**. (c) **OS+OP+CF+IS**. (d) **OS+OP+CF+IS+BBN**.

Figure 3: Entropy of the opcode distributions for the different obfuscation strategies.



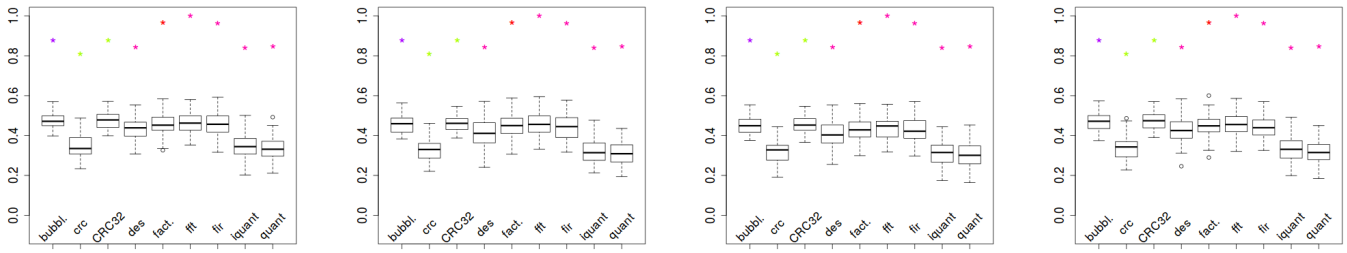

(a) **OS+OP**. (b) **OS+OP+IS**. (c) **OS+OP+CF+IS**. (d) **OS+OP+CF+IS+BBN**.

Figure 4: $\mathrm{sd}_3$-distributions for the different obfuscation strategies.



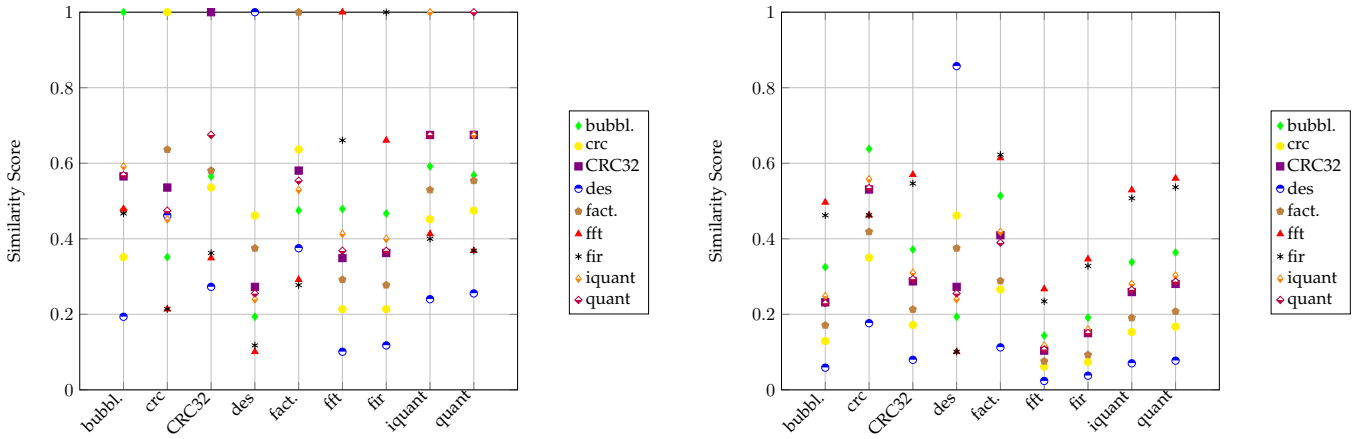

(a) **OS+OP**. (b) **OS+OP+IS**. (c) **OS+OP+CF+IS**. (d) **OS+OP+CF+IS+BBN**.

Figure 5: $\mathcal{E}$-$\mathrm{sd}_3$-diagrams for the different obfuscation strategies. The colour legend for all figures is specified in (d).




(a) **OS+OP**. (b) **OS+OP+IS**. (c) **OS+OP+CF+IS**. (d) **OS+OP+CF+IS+BBN**.

Figure 6: Correlations of the opcodes for `fft` program for the different obfuscation strategies.




(a) DCFG similarity for the **OS+OP** obfuscation strategy compared to the unobfuscated programs.

(b) DCFG similarity for the **OS+OP+CF+IS+BBN** obfuscation strategy compared to the unobfuscated programs.

Figure 7: Dynamic control flow graph similarity evaluation for the benchmark programs and different obfuscation strategies.

## 12 Future Work

**Hardware Issues.** The underlying concept of an obfuscated ISA encoding could potentially be applied to Application Specific Integrated Circuits (ASICs) that provide several field-programmable hardware elements, e.g., the WISC concept [47]. For example, such a device could include a user-defined ISA encoding, memory layout, and access control. In particular, an ASIC could be generated by using just one mask to reduce the manufacturing costs. Thus, a fleet of embedded systems could be diversified to counteract the "break one break all" principle. A further interesting direction is dynamic instruction encoding update. This approach would be particularly attractive as a *moving target defense*. The disabling of an FPGA-based soft processor in response to an attack is challenging since current SRAM-FPGAs do not offer the ability to permanently set one or more non-volatile fuses at run-time. The addition of this feature would help in preventing deobfuscation attacks.

The security analysis of our approaches for processors with dedicated cache memories is left for future research.

**Software Issues.** The performance overhead of software-level obfuscation is significant for embedded systems. However, our evaluation results are in line with reported results from other work published in this area, e.g. [43]. The analysis of further obfuscation transformations such as anti-emulation, code tamper proofing, and self-modifying code [16], [43] in combination with Application Specific Instruction-Set Processor techniques to decrease software performance overhead are left for future research.

**Security Metric Issues.** Our security metrics could be expanded to include new statistical tests that evaluate hybrid obfuscation systems. A new metric which considers the limited I/O of embedded systems would be particularly advantageous.

## 13 Conclusion

ISA randomization provides a viable approach for obfuscation and exploit mitigation for embedded processors. However, for embedded systems, various disclosure sources can be leveraged to reveal crucial ISA information. Once the ISA is revealed, the targeted software can be reverse-engineered. This issue is particularly worrisome for low-cost IoT systems with limited cryptographic protection.

In this work, we have presented a hybrid obfuscation scheme consisting of hardware-level and software-level obfuscation transformations to prevent a variety of disclosure attacks. We combined the obfuscation transformations with dedicated hardware booby traps to detect and respond to manipulation attempts. Finally, we demonstrated a novel evaluation methodology to assess the twofold diversification. This methodology provides a quantitative method to qualify the benefits of our approaches. The lack of quantitative metrics has been a long-standing issue in the software obfuscation domain. A performance evaluation of our prototype implementation demonstrates a lightweight hardware overhead of up to 14% for a simple, low-cost embedded processor.

## Acknowledgement

The authors would like to thank Philipp Koppe, Andre Pawlowski, Georg Becker and Ulrich Rührmair for the fruitful discussions and valuable suggestions throughout this work. Furthermore, we would like to thank the anonymous reviewers for their valuable comments. The research was supported in part through NSF grants CNS-1318497 and CNS-1421352, SFB823 (sub-project C4), INRIA Associate Team HARDIESSE, and ERC Advanced Grant 695022.

## References

[1] J. Chase, "The evolution of the internet of things," Texas Instruments, Tech. Rep., 2013.
[2] D. Lie et al., "Architectural support for copy and tamper resistant software," in *ASPLOS*, 2000, pp. 168–177.
[3] G. E. Suh et al., "Aegis: A secure single-core processor," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 570–580, 2007.
[4] E. Owusu et al., "OASIS: on achieving a sanctuary for integrity and secrecy on untrusted platforms," in *ACM CCS*, 2013, pp. 13–24.
[5] J. Zaddach et al., "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *NDSS*, 2014.
[6] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *CASES*, 2005, pp. 202–212.

[7] J Zheng, "A secure and unclonable embedded system using instruction-level PUF authentication," in *FPL*, 2014, pp. 1–4.

[8] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[9] C. Fletcher et al., "A Low-Latency, Low-Area Hardware Oblivious RAM Controller," in *FCCM*, 2015, pp. 215–222.

[10] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, "On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from Xilinx Virtex-II FPGAs," in *ACM CCS*, 2011, pp. 111–124.

[11] A. Moradi et al., "Side-channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II: Facilitating Black-box Analysis Using Software Reverse-engineering," in *ACM/SIGDA FPGA*, 2013, pp. 91–100.

[12] D. Strobel, D. Oswald, B. Richter, F. Schellenberg, and C. Paar, "Microcontrollers as (in)security devices for pervasive computing applications," *IEEE*, vol. 102, no. 8, pp. 1157–1173, 2014.

[13] D. Strobel, F. Bache, D. Oswald, F. Schellenberg, and C. Paar, "Scandalee: a side-channel-based disassembler using local electromagnetic emanations," in *DATE*, 2015, pp. 139–144.

[14] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley Professional, 2009.

[15] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *IH*, 2011, pp. 270–284.

[16] C. Willems and F. C. Freiling, "Reverse code engineering - state of the art and countermeasures," *it - Information Technology*, vol. 54, no. 2, pp. 53–63, 2012.

[17] B. Anckaert et al., "Proteus: virtualization for diversified tamper-resistance," in *Workshop on Digital Rights Management*, 2006, pp. 47–58.

[18] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *ACM CCS*, 2011, pp. 275–284.

[19] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *IEEE Symposium on Security and Privacy*, 2015, pp. 674–691.

[20] N. Runwal et al., "Opcode graph similarity and metamorphic detection," *J. Comput. Virol.*, vol. 8, no. 1-2, pp. 37–52, 2012.

[21] I. Sorokin, "Comparing files using structural entropy," *J. Comput. Virol.*, vol. 7, no. 4, pp. 259–265, 2011.

[22] S. Attaluri, S. McGhee, and M. Stamp, "Profile hidden markov models and metamorphic virus detection," *Journal in Computer Virology*, vol. 5, no. 2, pp. 151–169, 2008.

[23] P. P. F. Chan and C. S. Collberg, "A method to evaluate CFG comparison algorithms," in *IEEE QS*, 2014, pp. 95–104.

[24] C. Krügel et al., "Polymorphic worm detection using structural information of executables," in *RAID*, 2005, pp. 207–226.

[25] X. Hu, T. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *ACM CCS*, 2009, pp. 611–620.

[26] L. Szekeres et al., "SoK: eternal war in memory," in *IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[27] P. Larsen, S. Brunthaler, and M. Franz, "Automatic software diversity," *IEEE Security & Privacy*, vol. 13, no. 2, pp. 30–37, 2015.

[28] E. G. Barrantes et al., "Randomized instruction set emulation to disrupt binary code injection attacks," in *ACM CCS*, 2003.

[29] G. S. Kc, "Countering code-injection attacks with instruction-set randomization," in *ACM CCS*, 2003, pp. 272–280.

[30] B. Fechner, J. Keller, and A. Wohlfeld, "Web server protection by customized instruction set encoding," in *IPDPS*, 2006.

[31] E. G. Barrantes et al., "Randomized instruction set emulation," *ACM Trans. Inf. Syst. Secur.*, vol. 8, no. 1, pp. 3–40, 2005.

[32] G. Portokalidis, "Fast and practical instruction-set randomization for commodity systems," in *ACSAC*, 2010, pp. 41–48.

[33] J. Danger, S. Guilley, and F. Praden, "Hardware-enforced protection against software reverse-engineering based on an instruction set encoding," in *ACM SIGPLAN PPREW*, 2014, pp. 5:1–5:11.

[34] Z. Liu, W. Shi, S. Xu, and Z. Lin, "Programmable decoder and shadow threads: Tolerate remote code injection exploits with diversified redundancy," in *DATE*, 2014, pp. 1–6.

[35] D. Molnar et al., "The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks," in *ICISC*, 2006, pp. 156–168.

[36] S. Ichikawa, T. Sawada, and H. Hata, "Diversification of processors based on redundancy in instruction set," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E91-A, no. 1, pp. 211–220, 2008.

[37] T. Jakobsen, "A fast method for the cryptanalysis of substitution ciphers," *Cryptologia*, vol. 19, pp. 265–274, 1995.

[38] D. Oranchak, "Evolutionary algorithm for decryption of monoalphabetic homophonic substitution ciphers encoded as constraint satisfaction problems," in *GECCO*, 2008, pp. 1717–1718.

[39] A. Dhavare, R. Low, and M. Stamp, "Efficient cryptanalysis of homophonic substitution ciphers," *Cryptologia*, vol. 37, no. 3, pp. 250–281, 2013.

[40] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in *NSPW*, 2013, pp. 95–106.

[41] NXP Semiconductors, "NXP J3A040 and J2A040 Secure Smart Card Controller," 13 May 2011, rev. 01.03.

[42] ——, "i.MX 6Dual/6Quad Applications Processor Reference Manua," July 2015, rev. 3, Document Number: IMX6DQRM.

[43] P. Junod, "Obfuscator-LLVM – software protection for the masses," in *IEEE SPRO*, 2015, pp. 3–9.

[44] Altera Corporation, "Cyclone V Device Datasheet," December 2015.

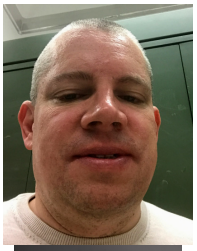
[45] D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff, "Implementing cryptographic program obfuscation," Cryptology ePrint Archive, Report 2014/779, 2014, http://eprint.iacr.org/.

[46] S. Garg, *TCC*. Springer, 2014, ch. Two-Round Secure MPC from Indistinguishability Obfuscation, pp. 74–94.

[47] P. Koopman, "Writable Instruction Set, Stack Oriented Computers: The WISC Concept," in *Proceedings of the 1987 Rochester Forth Conference*, 1987, pp. 49–71.


**Marc Fyrbiak** received his B.Sc. degree in computer science from TU Braunschweig, Germany in 2012 and his M.Sc. degree in IT security from Ruhr University Bochum, Germany in 2014. He is currently working towards the Ph.D. degree at the Chair for Embedded Security, under the supervision of C. Paar. His research interests include the implementation and reverse-engineering of embedded software.

**Simon Rokicki** received his B.Sc and M.Sc degrees from ENS Rennes, France in 2012 and 2014 respectively. He is currently working towards the Ph.D. degree in computer sciences in University of Rennes, under the supervision of S. Derrien and E. Rohou. His research interests include embedded systems architecture and compilation (both static and dynamic).



**Nicolai Bissantz** received Ph.D. degree in science at the University of Basel (Switzerland) 2001 and the habilitation in mathematics in 2015 at the Ruhr-University Bochum, respectively. He is currently director of studies at the Ruhr-University Bochum. His current research interests include statistical inverse problems and applied statistics in science and technology.

**Russell Tessier** (M'00-SM'07) received the B.S. degree in computer and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1989, and the S.M. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1992 and 1999, respectively. He is currently Professor of Electrical and Computer Engineering with the University of Massachusetts, Amherst, MA. His current research interests include computer architecture and FPGAs.

**Christof Paar (Fellow, IEEE)** received the M.Sc. degree from the University of Siegen and the Ph.D. degree from the Institute for Experimental Mathematics at the University of Essen, Germany. He holds the Chair for Embedded Security at Ruhr University Bochum, Bochum, Germany, and is an Affiliated Professor at the University of Massachusetts Amherst, Amherst, MA, USA. His research interests include highly efficient software and hardware realizations of cryptography, physical security, security evaluation of real-world systems, and cryptanalytical hardware.