# Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications

Christian Palmiero
*Politecnico di Torino, Italy*
christian.palmiero@studenti.polito.it

Giuseppe Di Guglielmo
*Columbia University, USA*
giuseppe@cs.columbia.edu

Luciano Lavagno
*Politecnico di Torino, Italy*
luciano.lavagno@polito.it

Luca P. Carloni
*Columbia University, USA*
luca@cs.columbia.edu

***Abstract*—Security for Internet-of-Things devices is an increasingly critical aspect of computer architecture, with implications that spread across a wide range of domains. We present the design and implementation of a hardware dynamic information flow tracking (DIFT) architecture for RISC-V processor cores. Our approach exhibits the following features at the architecture level. First, it supports a robust and software-programmable policy that protects bare-metal applications against memory corruption attacks such as buffer overflows and format strings, without causing false alarms when running real-world benchmarks. Second, it is fast and transparent, having a small impact on applications performances and providing a fine-grain management of security tags. Third, it consists of a flexible design that can be easily extended for targeting new sets of attacks. We implemented our architecture on PULPino, an open-source platform that supports the design of different RISC-V cores targeting IoT applications. FPGA-based experimental results show that the overall overhead is low, with no impact on the processor performance and negligible storage increase.**

## I. Introduction

While many forecasts differ on the specific numbers, the global Internet-of-Things (IoT) market is estimated to reach a size of the order of at least hundreds of billions of dollars by 2020 [1]. Many observers expect that *Open Source Hardware (OSH)* [2] can help kickstart semiconductor innovation and, in particular, innovation in IoT devices. Among the most promising OSH solutions, the RISC-V instruction set architecture (ISA) and the RISC-V foundation, which was founded in 2015 and now comprises more than 100 members, will likely play a major role in promoting a new era of processor innovation through open standard collaboration [3], [4]. In this context, robust security and privacy mechanisms are needed to protect personal data and monitor their flow from IoT devices to the cloud servers [5].

Among many techniques that provide reliable protection against a broad range of security attacks, *Dynamic Information Flow Tracking (DIFT)* is a versatile method that can be applied for both security and privacy purposes [6]. DIFT consists of mechanisms and policies that can be combined to protect vulnerable programs against a wide range of security exploits. For example, DIFT can be effectively applied to deter such software attacks as information leakage [7] and code injection [8]. The main idea of DIFT is to extend each data item of a program with a new field, called *tag*. The tag identifies whether a data item is certainly authentic, and therefore safe, or potentially malicious, and therefore unsafe. A DIFT protection scheme relies on three main concepts: tag initialization, tag propagation and tag check. During the *tag initialization* phase, all data items coming from potentially malicious channels are marked as spurious. In general, a potentially malicious channel is a legitimate I/O communication channel through which malicious inputs may be injected into the application by an attacker. During *tag propagation*, the processor, depending on the type of instruction that is being executed and on the authenticity of each input operand (as tracked with its corresponding tag), may decide to mark the result of a computation as spurious. This allows the processor to keep track of all information flows generated from spurious inputs at run-time. Finally, during *tag checking*, if the processor detects that a spurious data item is used in an unsafe manner, it raises a security exception. Thereafter, an exception handler determines whether the use of the spurious value is legitimate or not, i.e. whether the program should resume or terminate. A *security policy* defines the rules that specify how untrusted I/O channels are identified, how dependencies are tracked, and how restrictions on the use of spurious values are applied.

**Contributions.** In this paper, we present a low-overhead implementation of DIFT that is specialized for low-end embedded systems for IoT applications. The specific contributions of our work include:

- *The design of D-RI5CY, a DIFT-protected implementation of the RI5CY [9] processor core.* RI5CY is an optimized 4-stage in-order 32-bit RISC-V core that is based on PULPino, a state-of-the-art platform that supports different 32-bit RISC-V cores [10]. Our modifications implement the DIFT tag-propagation and tag-checking mechanisms in a way that is transparent to the execution of the regular instructions. In our design the manipulation of the DIFT tags does not add any latency overhead.
- *The realization of a prototype of D-RI5CY on a ZedBoard equipped with a Xilinx XC7Z020 FPGA*. The software applications running in a "bare-metal environment" are protected against memory-corruption attacks such as buffer overflows and format strings. We also show that D-RI5CY does not cause any false-positive alarms when running a set of non-malicious benchmark applications.
- *A comprehensive analysis of the performance and resource usage of D-RI5CY when running on the ZedBoard.* This analysis demonstrates that there is zero impact on the CPU time necessary to execute the software applications and that the overhead in terms of resource occupation is minimal.

Our current implementation of D-RI5CY is meant for IoT devices and was derived with the main goal of minimizing resource usage to prevent memory-corruption attacks. Its modular design, however, can be the basis to derive more complex implementations of protected RISC-V processors that support more complex DIFT policies.

## II. Related Work

The last fifteen years have seen a lot of research work on DIFT, including both hardware-based and software-based implementations, for different types of processor architectures.

One of the first hardware architectures for DIFT was developed by Suh *et al* for low-level security attacks [6]. They targeted both

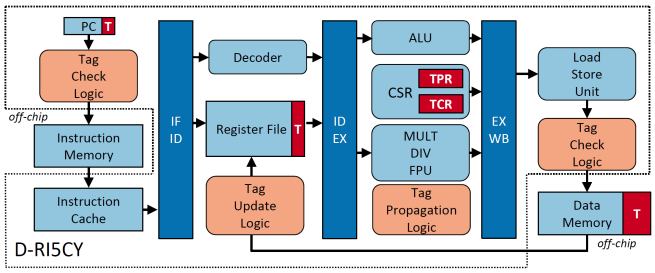


Fig. 1. Block diagram of the D-RI5CY processor. In red and pink the DIFT components.


Fig. 2. Register File and Data Memory.

control-data attacks, which alter the program control flow, and non-control-data attacks, which corrupt user identity data, configuration data, user input data, and decision-making data [11].

Chen *et al* proposed an architecture that can detect and stop stack buffer overflow attacks, format string attacks and heap corruption attacks, based on the notion of pointer taintedness [12]. A pointer is marked as tainted when its value is derived from a user supplied input. Attempting to dereference a tainted value during program execution triggers the detection of an attack.

With attackers focusing on the exploit of high-level semantic vulnerabilities, other than just memory-corruption bugs, a demand for new DIFT-based solutions arose. With Raksha, Dalton *et al* combined speed and fine-grain control over the rules of transparent hardware approaches with the flexibility and the adaptability to different types of exploits typical of robust software approaches [13].

More recently, Oszoy *et al* have proposed SIFT as a low-overhead DIFT architecture for simultaneous multithreading processors [14]. Chen *et al* have proposed SHIFT, which uses the speculative execution and deferred exceptions typical of some processor architectures (Titanium) [15]. Kannan *et al* have proposed a decoupled implementation of DIFT functionality in a separate co-processor [16]. Vachharajani *et al* have proposed RIFLE, which focuses on information leakage detection using a binary translation mechanism [17]. GLIFT is an implementation of DIFT at the gate level [18]. With WHISK, Porquet *et al* were the first to address the question of extending DIFT to support heterogeneous system-on-chip architectures featuring third-party intellectual-property components, such as hardware accelerators [19]. Finally, Song *et al* have proposed a hardware-assisted data-flow isolation (HDFI) technique to prevent attacks based on memory corruption [20].

In developing D-RI5CY, we kept in mind the lesson of many of these prior works. Our focus has been on deriving an implementation of DIFT for a RISC-V core that protects IoT applications against memory-corruptions attacks while presenting no performance overhead and minimal implementation costs.

## III. Background

**RISC-V Instruction Set Architecture.** RISC-V is an open and free instruction set architecture (ISA), which was originally developed at UC Berkeley [21] and now is managed and supported by the RISC-V Foundation [3], a non-profit corporation with over 100 members including such companies as Google, IBM and NVIDIA. The popularity of RISC-V continues to grow in academia, for both teaching and research purposes [22], as well as in the industry [23]. For the latter, RISC-V is expected to substantially accelerate the growth of Open Source Hardware [2], thereby fueling semiconductor innovation and market development, particularly in the areas of low-cost embedded systems and IoT devices. The RISC-V ISA defines a base integer ISA, which is mandatory for all RISC-V processor implementations, plus a set of optional standard extensions [4], [24]. The base RISC-V ISA has fixed-length 32-bit instructions, with variable length encoding conventions only permitted in custom extensions. The user-level base integer register state includes 32 general purpose registers (x0-x31) and the program counter (pc). Register x0 is hardwired to the constant value 0. According to the RISC-V calling convention, during a procedure call registers x1 and x2 host the return address and the stack pointer, respectively. Each particular RISC-V implementation can define an arbitrary collection of *Control and Status Registers (CSRs)* to manage and provide system functionalities.

**The PULPino Platform.** In recent years, there has been a flurry of activity in developing various RISC-V processor implementations and RISC-V-based systems-on-chip. In particular, researchers at ETH Zurich and Universita' di Bologna have created PULPino, a state-of-the-art platform that supports different RISC-V cores [10], [25]. PULPino is released under the Solderpad Hardware License and the source code can be freely accessed and adapted. The memory subsystem includes two single-port 32kB data and instruction RAMs and a boot ROM that contains a boot loader that can load a program via SPI from an external flash device. For communicating with the outside world, PULPino contains a broad set of peripherals, including GPIO, I2C, SPI, JTAG and UART. Among the different RISC-V cores supported by PULPino, we selected RI5CY [9] as the target processor to secure through the application of DIFT. RI5CY is a 4-stage in-order 32-bit RISC-V core that is optimized for low-power embedded systems and IoT application. RI5CY fully supports the base integer instruction set (RV32I), compressed instructions (RV32C) and the multiplication instruction set extension (RV32M) of the RISC-V ISA. In addition, it implements a set of custom extensions (RV32XPulp) that include supports for hardware loops, post-incrementing load and store instructions, ALU and MAC operations.

## IV. Securing RISC-V with DIFT

In this section, we describe the architectural choices for D-RI5CY. Our choices are general and can be easily applied to securing other IoT RISC-V implementations. In particular, we set the following guidelines for our protection scheme. The D-RI5CY must be able to detect and stop various known memory-corruption attacks; the protection must be flexible and extendable through software-programmable security policies to target future kinds of attacks; finally, the protection must provide a transparent and fine-grain management of security with no latency and small storage overhead.

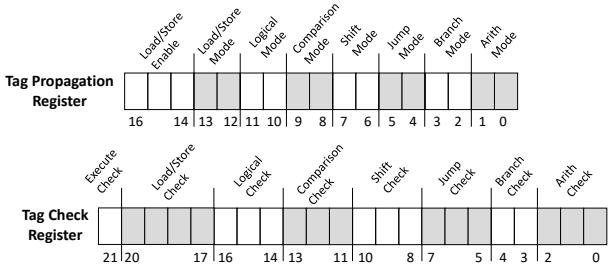


Fig. 3. Tag Propagation and Tag Check Registers.

### A. Protection Scheme Overview

In D-RI5CY, with the goal of the minimum hardware overhead, we use a one-bit tag to indicate whether the corresponding data block is authentic or spurious. It is straightforward to extend our scheme to multiple-bit tags to further distinguish the data values; for example, it may be helpful to distinguish the I/O inputs from the intermediate computation values. In the following discussion, tags with zero values indicate authentic data and tags with one values indicate spurious data.

Figure 1 illustrates the main components of the D-RI5CY architecture that we secured with DIFT. In the processor, we introduced additional components to propagate, check, and update the tags (in pink in Figure 1). We augmented the general-purpose registers and the program counter with a one-bit tag (marked as `T`). In the data memory, we added a one bit-tag to each byte because this is the smallest granularity that can be accessed by RISC-V processors. The use of four extra bits per 32-bit word introduces a quite acceptable storage overhead. In a future extension, the memory space overhead can be significantly reduced by using a more efficient tag management based on the concept of multi-granularity security tags, where all words in a chunk of data may be associated to the same tag value [6]. Figure 2 provides a detailed view of the tag-extended register file and data memory. Each data element is physically stored in memory with its associated tag. To access both data and tag, we use the same index (register id) for the register file and the same address for the data memory. The data and tag are always transmitted atomically and this required the extension of the data-memory bus from 32 bits to 36 bits.

We defined a library of routines to initializes the tags of the data coming from potentially malicious channels, while, at program start-up, D-RI5CY initializes the tags of the registers, program counter, and memory blocks to zero. In particular, we extended the RI5CY ISA with memory and register tagging instructions.

In addition, we modified the hardware implementation of every instruction in the base integer-instruction set (`RV32I`), the multiplication-instruction-set extension (`RV32M`), as well as the post-incrementing-load and -store instruction set belonging to the PULP custom extension (`RV32XPulp`). The tag propagation and check happen in the D-RI5CY pipeline in parallel with the standard behavior, without incurring any latency overhead.

Our architecture supports software-programmable security policies. These policies consist of rules, which have fine-grain control over tag propagation and check for different classes of instructions. The rules specify how the tags of the instruction operands are combined and checked. The propagation and check rules are stored in the *Tag Propagation Register* (TPR) and *Tag Check Register* (TCR) that we added to the D-RI5CY Control and Status Registers (CSRs). TPR and TCR require the highest privilege of access (machine mode). We program them through the RISC-V SYSTEM instruction, which is used to access system functionality and performs atomic read-modify-write operations on CSRs [26].

TABLE I
D-RI5CY INSTRUCTIONS PER SECURITY CLASSES.

| CLASS | INSTRUCTIONS |
|---|---|
| Load/Store | `LW`, `LH[U]`, `LB[U]`, `SW`, `SH`, `SB`, `LUI`, `AUIPC`, `XPulp Load/Store` |
| Logical | `AND`, `ANDI`, `OR`, `ORI`, `XOR`, `XORI` |
| Comparison | `SLTI[U]`, `SLT[U]` |
| Shift | `SLL`, `SLLI`, `SRL`, `SRLI`, `SRA`, `SRAI` |
| Jump | `JAL`, `JALR` |
| Branch | `BEQ`, `BNE`, `BLT[U]`, `BGE[U]` |
| Integer Arithmetic | `ADD`, `ADDI`, `SUB`, `MUL`, `MULH[U]`, `MULHSU`, `DIV[U]`, `REM[U]` |

TABLE II
TAG PROPAGATION REGISTER CONFIGURATION.

| FIELD | VALUE | RULE |
|---|---|---|
| Load/Store Enable | 001 | Source tag enabled |
| Load/Store Mode | 10 | Dest tag = Source tag |
| Logical Mode | 10 | Dest tag = Source1 tag OR Source2 tag |
| Comparison Mode | 00 | No Propagation |
| Shift Mode | 10 | Dest tag = Source1 tag OR Source2 tag |
| Jump Mode | 10 | JAL: New PC = Old PC<br>JALR: New PC = Source tag |
| Branch Mode | 00 | No propagation |
| Integer Arith Mode | 10 | Dest tag = Source1 tag OR Source2 tag |

### B. Tag-Propagation Mechanisms

We chose to specify the policy rules at the granularity of classes of instructions to balance flexibility and fine-grain control. We organized the `RV32I`, `RV32M`, and `RV32Xpulp` instruction set in seven classes: *Load/Store*, *Logical*, *Comparison*, *Shift*, *Jump*, *Branch* and *Integer Arithmetic*. All the instructions assigned to a certain class share the same tag-propagation and tag-check rules. Table I summarizes this organization for the D-RI5CY instructions.

The *tag-propagation rules* define how tag values must be propagated from the input operands to the output operand of an instruction. For example, a tag propagation rule may be "For an addition instruction, if one (or both) of the input operands are tagged as spurious then the output operand is tagged as spurious". This rule can be implemented as the Boolean OR between the addition input-operand tags.

The tag-propagation rules must be specified in the Tag Propagation Register (TPR), whose structure is shown in Figure 3. TPR is a 17-bit register organized in eight fields. There is a *Mode* field for each class of instructions and an additional *Enable* field for the Load/Store class.

Given an instruction, the corresponding *Mode* field specifies how to propagate the tags of the input operands to the output operand tag. There are four possibilities: the output tag keeps its old value (`00`); the output tag is set to one, if both the input tags are set to one (`01`); the output tag is set to one, if at least one input tag is set to one (`10`); finally, the output tag is set to zero (`11`).

The *Load/Store Enable* field introduces a finer-granularity rule to enable/disable the input operands before applying the propagation rule specified in the *Load/Store Mode* field. In particular, the three bits in the field allow the policy to enable the source, source-address, and destination-address tags, respectively.

### C. Tag-Checking Mechanisms

The *tag-check rules* restrict the operations that may be performed on tagged data. For example, a tag-check rule may be "If a tag of

a register is set to one then it cannot be used to address the data memory". If this happens, the system rises a security exception.

The tag-propagation rules must be specified in the Tag Check Register (TCR), whose structure is shown at the bottom of Figure 3. TCR is a 22-bit register and organized in eight fields. There is a *Check* field for each class of instructions and an additional *Check* field for the program counter.

Given an instruction, the corresponding *Check* field specifies which operands tags are checked in order to generate a security exception. If the check bit for an operand tag is set to one and the corresponding tag is equal to one, an exception is raised. For all the classes except Load/Store, there are three tags to consider: first input, second input, and output tags. For the Load/Store class there are four tags to take into account: source-address, source, destination-address, and destination tags.

Finally, the additional *Execute Check* field is associated with the program counter and specifies whether to raise a security exception when the program-counter tag is set to one.

### D. Security Policy for Memory Protection

The security policies specify how potentially malicious I/O channels are identified, how processor tracks spurious information (tag propagation), and when unsafe uses of spurious values result in a security exception (tag check). This section describes the security policy that we have defined to protect our IoT system against memory-corruption attacks. It is built on top of the mechanisms from Sections IV-B and IV-C.

Our policy marks all user-supplied inputs as spurious. This is a conservative choice, but based on the fact that IoT applications usually run in bare metal environments, without loading and executing an operating-system module that is capable of distinguishing between trusted and untrusted I/O channels.

To initialize the security tags of user-supplied inputs to one, we introduced four new instructions and extended the RI5CY version of the RISC-V toolchain. The new assembly instructions are:

- **p.set rd** sets to one the security tag of the destination register *rd*;
- **p.spsb x0, offset(rt)** sets to one the security tag of the memory byte at the address *value-stored-in-register-rt* + *offset*;
- **p.spsh x0, offset(rt)** sets to one the security tags of the memory half-word at the address *value-stored-in-register-rt* + *offset*;
- **p.spsw x0, offset(rt)** sets to one the security tags of the memory word at the address *value-stored-in-register-rt* + *offset*.

Table II and Table III report the tag-propagation and tag-check rules of our memory-protection policy. These are configurations of the TPR and TCR in Figure 3. The tag-propagation rules are quite conservative: for all the instructions except for comparison and branch, our rules establish that the destination tag is computed as the Boolean OR of the two sources tags. For comparisons and branch instructions, our rule avoids tag propagation because this would generate false-positive situations, as observed in the literature [6], [13]. For Load/Store instructions, the destination tag simply corresponds to the source tag. The tag-check rules prohibit tagged information from being used as a load address, store address, or program-counter value. Intuitively, this prevents the execution of malicious code.

TPR and TCR are programmed by a start-up routine at the beginning of each program, right before the main function is executed. The routine sets the content of the register according to the policy that the system operator is willing to enforce. The tag-propagation rules are applied on load and store operations at byte granularity: it is worth underlining that on half-word-store and word-store operations, the tag bit is extended to two or four bits. On half-word-load and word-load operations, the rule for merging many-bit tags is the OR-reduction.

TABLE III
TAG CHECK REGISTER CONFIGURATION.

| FIELD | VALUE | RULE |
|---|---|---|
| Load/Store Check | 1010 | Source address tag checked<br>Destination address tag checked |
| Logical Check | 000 | No check |
| Comparison Check | 011 | Source1 tag checked<br>Source2 tag checked |
| Shift Check | 000 | No check |
| Jump Check | 000 | No check |
| Branch Check | 00 | No check |
| Integer Arith Check | 000 | No check |
| Execute Check | 1 | Program Counter checked |

Finally, when a security exception occurs, the offending instruction is not committed and an exit routine is executed instead. The routine saves all caller-saved registers, loads back registers from the stack, and halts the program execution.

## V. EXPERIMENTS

We extended the RI5CY/PULPino implementation from ETH Zurich and Università di Bologna and defined a complete prototype system to evaluate the feasibility and cost of applying DIFT to a RISC-V core for IoT applications. To evaluate the effectiveness of our approach, we tested it with two main classes of attacks, which are based on buffer-overflow and format-string vulnerabilities, respectively. For the *buffer-overflow attacks*, we choose the suite developed by Wilander and Kamkar [27]. For the *format-string attacks*, we choose two well-known vulnerabilities from the TESO group [28]. Finally, we studied if D-RI5CY may incorrectly detect non-existing attacks, i.e. false positives.

### A. Experimental Setup

We synthesized D-RISCY/PULPino on a ZedBoard equipped with a Xilinx XC7Z020 FPGA. The width of the data-memory bus is 36 bits to accommodate four tag bits. The overall data memory of the system is 36KB (32KB data RAM and 4KB tag RAM). This introduces a 12.5% storage overhead. In our protection scheme instructions are not tagged because the application is loaded via SPI from an external flash device into the instruction RAM, that is afterwards used as a read-only memory. For this reason, instruction RAM, whose content is never spurious, is not tag extended. The integration of the DIFT support on RI5CY/PULPino requires an overall increase in the usage of LUT resources that does not exceed 0.82%. In addition, there is no impact on the processor performance because the tags are processed in parallel and independently from the instruction execution across all pipeline stages.

### B. Buffer-Overflow Attacks

Table IV summarizes the twenty classes of attacks in the Wilander suite and the protection results with D-RI5CY. The attacks are classified according to the buffer LOCATION, the attack TARGET, and the TECHNIQUE to overwrite the buffer. The possible RESULTS of an attack are: the attack is *detected* by our security mechanisms; the attack is not successful in a RISC-V environment and D-RI5CY does *not* generate a *false positive*; finally, the attack cannot be executed due to *portability* issues.

The attacks of the suite target *x86* architectures and are implemented in C language. We successfully ported the source code of thirteen attacks to the RISC-V calling convention [29] and compiled them with the RI5CY version of the RISC-V GNU toolchain [30].

TABLE IV
WILANDER'S BUFFER-OVERFLOW TEST SUITE AND RESULTS WITH D-RI5CY.

| ATTACK # | LOCATION | TARGET | TECHNIQUE | RESULT |
|---|---|---|---|---|
| 1 | Stack | Return Address | Direct | Detected |
| 2 | Stack | Base Pointer | Direct | No False Positive |
| 3 | Stack | Function Pointer (local variable) | Direct | Detected |
| 4 | Stack | Function Pointer (function parameter) | Direct | Detected |
| 5 | Stack | Longjmp Buffer (local variable) | Direct | Not portable |
| 6 | Stack | Longjmp Buffer (function parameter) | Direct | Not portable |
| 7 | Heap/BSS/Data | Function Pointer | Direct | Detected |
| 8 | Heap/BSS/Data | Longjmp Buffer | Direct | Not portable |
| 9 | Stack | Return Address | Indirect | Detected |
| 10 | Stack | Base Pointer | Indirect | No False Positive |
| 11 | Stack | Function Pointer (variable) | Indirect | Detected |
| 12 | Stack | Function Pointer (function parameter) | Indirect | Detected |
| 13 | Stack | Longjmp Buffer (variable) | Indirect | Not portable |
| 14 | Stack | Longjmp Buffer (function parameter) | Indirect | Not portable |
| 15 | Heap/BSS/Data | Return Address | Indirect | Detected |
| 16 | Heap/BSS/Data | Base Pointer | Indirect | No False Positive |
| 17 | Heap/BSS/Data | Function Pointer (variable) | Indirect | Detected |
| 18 | Heap/BSS/Data | Function Pointer (function parameter) | Indirect | Detected |
| 19 | Heap/BSS/Data | Longjmp Buffer (variable) | Indirect | Not portable |
| 20 | Heap/BSS/Data | Longjmp Buffer (function parameter) | Indirect | Not portable |

```
1  #define SIZE 16
2
3  void tag_words(u32 *data_ptr, u32 size) {
4      for(u32 i = 0; i < size; i++) {
5          /* p.spsw set to one the security tags
6             of each byte in a memory word */
7          asm volatile ("p.spsw x0, 0(%[offset]);"
8                        :
9                        :[offset] "r" (data_ptr));
10         data_ptr++;
11     }
12 }
13
14 void vuln_function(u32 input_1[SIZE], /*malicious*/
15                    u32 input_2[SIZE], /*malicious*/
16                    u32 input_3[SIZE]){/*non-malicios*/
17
18   /* Tag initialization phase */
19   tag_words(SIZE, input_1);
20   tag_words(SIZE, input_2);
21
22   /* Function body */
23   /* ... */
24 }
```

Listing 1. Tag routine.

For example, in all the buffer-overflow attacks based on the direct-overflow technique, we use the *frame-pointer register* (`x8`) to access the portion of the stack dedicated to the function arguments, because it is not always placed at the beginning of the stack frame as for *x86* architectures. We could not port the seven attacks that use `longjmp` buffer to RI5CY because of an incompatibility with the current version of the RISC-V GNU toolchain. We marked them with a darker background in the Table IV.

We noticed that all the RISC-V cores are intrinsically protected against the three buffer-overflow attacks that use the base pointer as an attack target (attacks number 2, 10, 16 in Table IV). When returning from a procedure call, the RISC-V calling convention assumes that all the accesses to local variables and the saved return value are relative to the *stack pointer*, and not to the *base pointer*. Indeed, in modern RISC architectures the stack-pointer register is enough to denote and address the current stack frame. Being these attacks not possible in a RISC-V environment, we investigated whether our protection scheme produces false positives on them. It turned out that the D-RI5CY does not incur false positives when it executes the attacks targeting the base-pointer register.

For the following experiments, we defined four tagging routines to mark the potentially malicious inputs of the applications that have to be protected. Each of these routines encapsulates a call to one of the assembly instructions that we introduced in the RI5CY ISA as described in Sec. IV-D. One of the routines and a simple usage example are shown in the Listing 1. At Line 3, `tag_words` is the tagging routines, which, given a pointer to a memory location (`data_ptr`), marks a certain number (`size`) of memory words as spurious. At Line 14, `vuln_function` is the vulnerable function. Because `input_1` and `input_2` are input parameters of the function and they can be malicious, we mark them as spurious. While we do nothing for `input_3` that is safe. Notice that since the applications run on the bare-metal D-RI5CY, we must explicitly mark the inputs that may be malicious at the beginning of the `vuln_function` execution. Conversely, with an operating system, the tag-initialization routine can be moved in a kernel module and be completely transparent to the user-space code.

D-RI5CY detects and stops all of the ten remaining classes of attacks that are a threat on a non-protected RI5CY core. A combination of Load/Store-source-propagation and program-counter-check rules detect the attacks using the *direct-overflow technique*. A combination of Load/Store-source-propagation and Load/Store-destination-address-check rules detect the attacks using the *indirect-overflow technique*.

For example, let us analyze the simplest class of buffer-overflow attacks as it is shown in Listing 2. The attack goal is to call the `shellcode` function that executes security-critical operations. The vulnerability is the absence of a control on the size of the target `stack_buffer` before the `memcpy` operation is executed (at Line 32). The attack targets the return address of `vuln_function` which is located in the stack frame through a *direct overflow*. The configuration of the stack frame for the `vuln_function` is shown in Figure 4 (a). The code in the example is in the style of the Wilander's suite where the malicious inputs and the attacker operations are hardcoded in the application without loss of generality. In particular, the `overflow` variable contains the distance (in bytes) between the address of the section dedicated to the function arguments `args` and the address of the `stack_buffer`, which is in the section dedicated to the function local variables. At Line 28 and 29, the attacker initializes the buffer `overflow_buffer` with a sequence of filler values (e.g. `'A'`) and the address of the `shellcode` function. For simplicity, those values are hard-coded in the example, but they can be user-supplied inputs of the vulnerable application.

```
1  #define BUFSIZE 16
2  #define OVERFLOWSIZE 256
3  u32 overflow_buffer[OVERFLOWSIZE];
4
5  void shellcode() {
6    /* Security-critical operations */
7  }
8
9  void vuln_function(u32 args) {
10   u32 stack_buffer[BUFSIZE];
11   char police_buffer[10];
12   u32 overflow;
13
14   /* Store the address of the stack-frame section
15      which is dedicated to the function arguments */
16   register u32 sf asm("x8");
17
18   /* Check the stack structure:
19      function arguments must be allocated on higher
20      addresses than function local variables */
21   if (sf > (u32)&police_buffer)) {
22
23     /* Initialize the overflow_buffer with
24        - a sequence of 'A's
25        - the address of the shellcode function
26          (as last element) */
27     overflow = (int)((long)sf - (long)&stack_buffer);
28     memset(overflow_buffer, 'A', overflow-4);
29     overflow_buffer[overflow/4-1] = (long)&shellcode;
30
31     /* Overwrite stack_buffer with overflow_buffer */
32     memcpy(stack_buffer, overflow_buffer, overflow);
33   }
34
35   /* ... */
36
37   return;
38 }
```

Listing 2. A simplified example of direct-overflow attack that targets the return address of a function [27].

At Line 32, there is the vulnerable `memcpy` operation which allows the attacker to overwrite the `stack_buffer`, the base pointer, and the return address of the function without any security check. In particular, it overwrites the return address with the pointer to the `shellcode` function (Figure 4 (b)). When the `vuln_function` terminates and returns, the unprotected core executes the instructions of the `shellcode` function.

D-RI5CY detects and stops the previously described attack through the following steps:

- in the synthetic application, the content of `overflow_buffer` (i.e., the filler values and the `shellcode` address) are assumed user-supplied inputs, therefore we mark them as spurious with the tag initialization routine (as shown in Listing 1);
- the rule of load/store-source propagation applies to the instructions at Lines 28 and 29 in Listing 2 and it marks the whole content of `overflow_buffer` as spurious;
- the same rule applies to the `memcpy` function at Line 32 and it marks as spurious the content of the `stack_buffer`, the base pointer, and the return address;
- when D-RI5CY executes the return instruction, it loads the spurious return address value into the program counter; the program-counter-check rule detects the use of a spurious value and raises a security exception.

A similar application of the security policies allows D-RI5CY to detect and stop all the other buffer-overflow attacks (both direct and indirect techniques).

### C. Format-String Attacks

We tested the format string vulnerability with two paradigmatic attacks, *QPOP 2.53/bftpd* and *wu-ftpd 2.6.0*. In both cases, the vulnerability is the use of an unchecked user input as the format string parameter in functions that perform formatting, e.g. `printf()`. An attacker can use the format tokens, to write into arbitrary locations of memory.

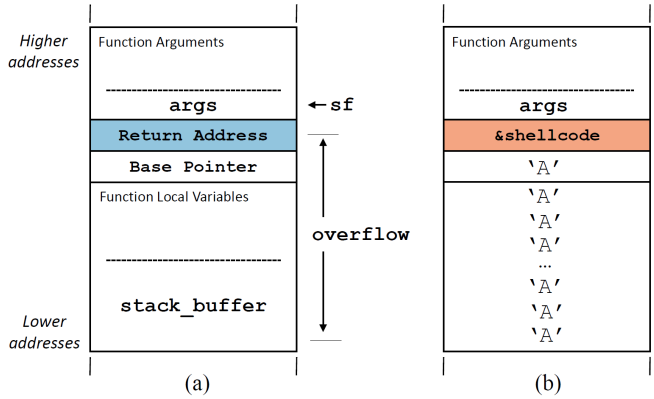


Fig. 4. The configuration of the stack for the function `vuln_function` in Listing 2.

D-RI5CY detects and stops both attacks: a combination of Load/Store-source-propagation and program-counter-check rules detect the first attack; the store destination-address check rule detects the second attack.

### D. False-Positive Analysis

We also investigated whether D-RI5CY produces false positives on well-known non-malicious benchmarks. We chose the software regression suite provided with PULPino to validate the correctness of our security mechanisms. In particular, the suite contains eight programs: 2D Convolution, AES, Discrete Cosine Transform, Fast Fourier Transform, Finite Impulse Response, Inflection Point Method, Matrix Multiplication, and Keccak/SHA-3.

For example, in the matrix multiplication application we marked the two input matrices as spurious. Due to our tag propagation rules, the content of the result matrix is spurious as well. However, the tag check rules are not violated, because the content of the spurious result matrix is never used as program counter, load/store source address, or load/store destination address, therefore, D-RI5CY does not rise any security exception. We similarly tested all of the other non-malicious applications. We marked the inputs as spurious for each of them. As expected, there were no security exceptions, because none of the generated (intermediate) results were used in an unsafe manner.

## VI. Concluding Remarks

We presented D-RI5CY, an application of DIFT to a RISC-V processor for IoT. We showed that securing a RISC-V core with DIFT is feasible, does not incur in any run-time overhead, and requires negligible resources (less than 1% in area overhead). D-RI5CY detects and stops memory-corruption attacks, such as buffer overflows and format strings from well known security suites. The implemented mechanisms and the software-programmable security policy make D-RI5CY flexible and extendable. As future work, we plan to focus on the security of RISC-V cores that can run a fully-featured Linux OS. For this, we will target new sets of attacks and enhance the policy infrastructure by introducing an OS module that detects potentially malicious channels and performs further software analysis when a security exception is raised. Moreover, we plan to support multiple concurrently active policies that target both high-level and low-level vulnerabilities. Finally, we plan to investigate efficient multi-granular mechanisms for managing tag storage with reduced memory overhead.

## REFERENCES


[1] GrowthEnabler. Market pulse report, Internet of Things (IoT). https://www.growthenabler.com, 2017.

[2] G. Gupta, T. Nowatzki, V. Gangadhar, and K. Sankaralingam. Kickstarting semiconductor innovation with open source hardware. *Computer*, 50(6):50–59, June 2017.

[3] RISC-V Foundation. https://riscv.org.

[4] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. CS Division, EECS Department, University of California, Berkeley, 2017.

[5] Y.-H. Hwang. IoT security privacy: Threats and challenges. In *Proc/ of the 1st ACM Workshop on IoT Privacy, Trust, and Security*, April 2015.

[6] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[7] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, pages 393–407, October 2014.

[8] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register transfer level information flow tracking for provably secure hardware design. In *Proc. of the Conf. on Design, Automation & Test in Europe*, pages 1695–1700, March 2017.

[9] Andreas Traber, Michael Gautsch, and Pasquale Davide Schiavone. *RI5CY: User Manual, Revision 1.7*. ETH Zurich, University of Bologna, July 2017.

[10] Andreas Traber and Michael Gautsch. *PULPino: Datasheet*. ETH Zurich, University of Bologna, June 2017.

[11] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium*, pages 177–192, 2005.

[12] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[13] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[14] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. SIFT: A Low-overhead Dynamic Information Flow Tracking Architecture for SMT Processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, 2011.

[15] H. Chen, X. Wu, L. Yuan, B. Zang, P. c. Yew, and F. T. Chong. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. In *Proceedings of the International Symposium on Computer Architecture*, 2008.

[16] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks*, 2009.

[17] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[18] M. Tiwari, X. Li, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Gate-Level Information-Flow Tracking for Secure Architectures. *IEEE Micro*, 2010.

[19] J. Porquet and S. Sethumadhavan. WHISK: An uncore architecture for Dynamic Information Flow Tracking in heterogeneous embedded SoCs. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[20] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: hardware-assisted data-flow isolation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 1–17. IEEE, 2016.

[21] Krste Asanovic /nd Davbd Patterson. The case for open instruction sets. *MICROPROCESSOR Report*, August 2014.

[22] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. An agile approach to building RISC-V microprocessors. *IEEE Micro*, 36(2):8–20, 2016.

[23] RISC-V Foundation. RISC-V Foundation Members. https://riscv.org/members-at-a-glance.

[24] Andrew Waterman, Yunsup Lee, Rimas Avizienis, Henry Cook, David Patterson, and Krste Asanovic. The RISC-V instruction set. In *2013 IEEE Hot Chips 25 Symposium (HCS)*, pages 25–27. IEEE, 2013.

[25] The PULP Platform. http://www.pulp-platform.org.

[26] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.9.1*. CS Division, EECS Department, University of California, Berkeley, 2016.

[27] John Wilander and Mariam Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.

[28] Scut, Team Teso. *Exploiting format string vulnerabilities*, March 2001.

[29] RISC-V ELF psABI specification. https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md.

[30] RI5CY GNU toolchain. https://github.com/pulp-platform/ri5cy_gnu_toolchain.