View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Politecnico di Milano

TaintHLS: High-Level Synthesis For Dynamic Information Flow Tracking

Christian Pilato, *Member, IEEE,* Kaijie Wu, *Member, IEEE,* Siddharth Garg, *Member, IEEE,* Ramesh Karri, *Senior Member, IEEE,* and Francesco Regazzoni, *Member, IEEE*

***Abstract*—Dynamic Information Flow Tracking (DIFT) is a technique to track potential security vulnerabilities in software and hardware systems at run time. Untrusted data are marked with tags (tainted), which are propagated through the system and their potential for unsafe use is analyzed to prevent them. DIFT is not supported in heterogeneous systems especially hardware accelerators. Currently, DIFT is manually generated and integrated into the accelerators. This process is error-prone, potentially hurting the process of identifying security violations in heterogeneous systems.**

We present TAINTHLS, to automatically generate *a micro-architecture to support baseline operations and a shadow microarchitecture for intrinsic DIFT support* in hardware accelerators while providing variable granularity of taint tags. TaintHLS offers *a companion high-level synthesis (HLS) methodology* to automatically generate such DIFT-enabled accelerators from a high-level specification. We extended a state-of-the-art HLS tool to generate DIFT-enhanced accelerators and demonstrated the approach on numerous benchmarks. The DIFT-enabled accelerators have negligible performance and no more than 30% hardware overhead.

***Index Terms*—Dynamic Information Flow Tracking, High-Level Synthesis, Hardware Security.**

I. INTRODUCTION

THE increasing demand of high-performance systems is pushing towards heterogeneous architectures that include an increasing number of application-specific accelerators [1]. These accelerators are up to 100× energy-efficient relative to the corresponding software implementations [2], [3], [4]. Advances in high-level synthesis (HLS) and reconfigurable platforms are allowing creation of accelerators at low cost [5]. Applications are leveraging such accelerators by interleaving hardware and software execution. Mobile devices with custom hardware and FPGA-based cloud systems are running third-party applications (e.g., Apple App Store and Google Play). Such applications can leak personal information without authorization [6] or can be compromised using software attacks (e.g., code injection [7] and return-to-libc [8]). This requires methods for identification of malicious uses. *Dynamic Information Flow Tracking* (DIFT) measures the influence of potentially untrustworthy external data by understanding how the information is propagated by an application [9]. Untrusted data is *tainted* and propagated through the application to monitor and prevent its unsafe use. DIFT can monitor the control flow of an application and prohibit execution of malicious code when selected jump conditions become tainted [7].

Solutions have been proposed to implement DIFT in user applications. They range from custom virtualization environments [6] to the integration of hardware support through dedicated DIFT co-processors [10]. An attacker may exploit the weaknesses in the accelerators or the heterogeneous computation to compromise the system. BAMBU [11] and other HLS tools generate accelerators with a memory architecture that allows dynamic pointer resolution [12]. This enables the acceleration of irregular applications [13]. Tampering with the pointers passed to the accelerator can offer access to specific memory locations (either within or outside the accelerator). Accelerators without DIFT support may compromise the tag propagation and identification of anomalies. Hence, solutions have been proposed to study how the taint tags are exchanged at the system level (e.g, between the processor and the accelerators and with the memory components). WHISK allows designers to evaluate the impact of DIFT in such heterogeneous systems [14]. However, the DIFT-enhanced accelerators are not already available. DIFT support can be generated by augmenting the Boolean gates with logic for information flow tracking [15]. But this solution does not exploit high-level information to reduce the hardware overhead.

Since DIFT support is expensive, understanding how to efficiently implement DIFT inside accelerators is necessary to offer security when executing applications on heterogeneous architectures. Analyzing the flow of information must be precise regardless of where the computation is performed (the processor cores or the hardware accelerators). This paper is the first one to address this issue by exploring the automatic generation of DIFT-enhanced accelerators.

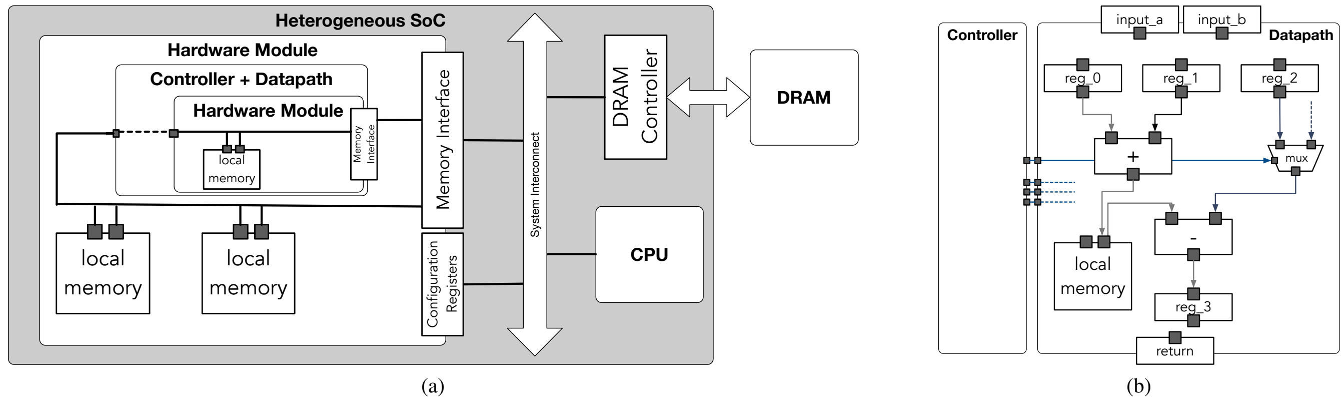
A. Contributions

TAINTHLS is a HLS-based solution enabling DIFT in hardware accelerators. TAINTHLS automatically generates DIFT-enhanced accelerators that can be integrated into heterogeneous architectures, while achieving the same DIFT as the corresponding software solutions. Automated DIFT generation can be configured to operate on taint tags at different granularity (from variables to bits) to trade off resource overhead and

Manuscript received August 1, 2017; revised November 29, 2017 and February 23, 2018; accepted April 10, 2018. This paper was recommended by Associate Editor J. Xu. R. Karri is supported in part by NSF (A#: 1526405) and CCS-AD. S. Garg is supported in part by an NSF CAREER Award (A#: 1553419). S. Garg and R. Karri are both with the NYU Center for Cybersecurity (cyber.nyu.edu) and supported in part by Boeing Corp.

C. Pilato and F. Regazzoni are with the Advanced Learning and Research Institute (ALaRI), Faculty of Informatics, Università della Svizzera italiana (USI), Lugano, Switzerland (Contact email: christian.pilato@usi.ch).

K. Wu, S. Garg, and R. Karri are with the NYU center for cybersecurity (htt://cyber.nyu.edu), New York University (NYU), New York, NY, USA.



(a) (b)

Fig. 1. Organization of a classical heterogeneous architecture with hardware accelerators (a) and microarchitecture of the each hardware module (b).

accuracy of taint analysis. After introducing the basic concepts and the motivation for this work (Section II), we present three contributions:

- an accelerator architecture that includes efficient hardware support for taint propagation (Section III);
- an HLS-based methodology to automatically generate accelerators with DIFT support starting from high-level descriptions in C (Section IV);
- a comprehensive analysis of the DIFT-enhanced accelerators in terms of performance and resource overheads (Section V).

We implemented TAINTHLS in BAMBU [11], a state-of-the-art open-source HLS tool.

II. Background

This section summarizes the background of the research fields addressed by this paper. First, we describe the *model of the target architecture*, focusing on *hardware accelerators* and the challenges encountered designing them. Then, we present the *attack model* that is considered and, finally, introduce *Dynamic Information Flow Tracking*, explaining how it secures a computation.

A. Target Architecture Model

Fig. 1(a) shows an example heterogeneous architecture with an accelerator and a processor core (CPU). Hardware accelerators offload computation-intensive kernels of the application, while the rest is executed by the CPU. The CPU prepares the data for the accelerator and configures it by writing the proper parameters into *configuration registers*. This configuration is performed with memory-mapped operations, like the ones performed by an OS device driver, on the interconnection system (e.g., a bus or a network-on-chip). The DRAM is accessed through a memory controller by both components. The memory space is thus used to share data between the processor and the accelerator. A hardware accelerator is tailored to execute a specific functionality. It improves performance (up to 10-100×) and lowers energy consumption (up to 100-1,000×) relative to the corresponding software implementation [16]. However, this comes at the cost of flexibility: the designer must determine the accelerator microarchitecture at design time to maximize performance. Similarly, executing an extra function (e.g., DIFT) requires extra logic.

Complex accelerators are organized as submodules to manage the design complexity. Each module is based on the classical Finite State Machine with Data (FSMD) model [17] and includes three components as shown in Fig. 1(b):

- a **controller**, which determines the operations to execute in each clock cycle. The control flow is represented by a Finite State Machine (FSM) that sends control signals to the datapath.
- a **datapath**, which is composed of functional units to execute the computation and the registers to hold temporary values during the computation. Multiplexers are used to drive the values based on the control flow;
- **memory elements**, such as *scratchpad memories* (SPMs) to locally store data and the *memory interface* to access external data (e.g., in DRAM).

Functional units in the datapath exchange information through registers, local SPMs, and the DRAM. Input data values are provided in the configuration registers or stored in the DRAM and accessed through the memory controller. Local SPMs are heterogeneous and distributed memories, tailored on the data structures to be stored. These memories enable multiple memory operations in parallel on different data with fixed latency, increasing hardware parallelism [5], [12], [18], [19]. These memory blocks impose constraints on the synthesis process. For example, they have a limited number of input and output ports; the operation schedule and the microarchitecture to access the data must be accordingly created [12], [18]. Memory architectures have been proposed to support a wide range of memory operations, enabling computation even on memory addresses (e.g., pointer arithmetic) [12]. Such memory architectures create a daisy chain of all memory components, including local memories and controllers for the external memory. An accelerator can automatically identify the memory location accessed by a memory request based on the value of the address given at runtime (*dynamic pointer resolution*). This allows the implementation of software code without any semantic changes, enabling the possibility of dynamically migrating the execution of a task between software and hardware.

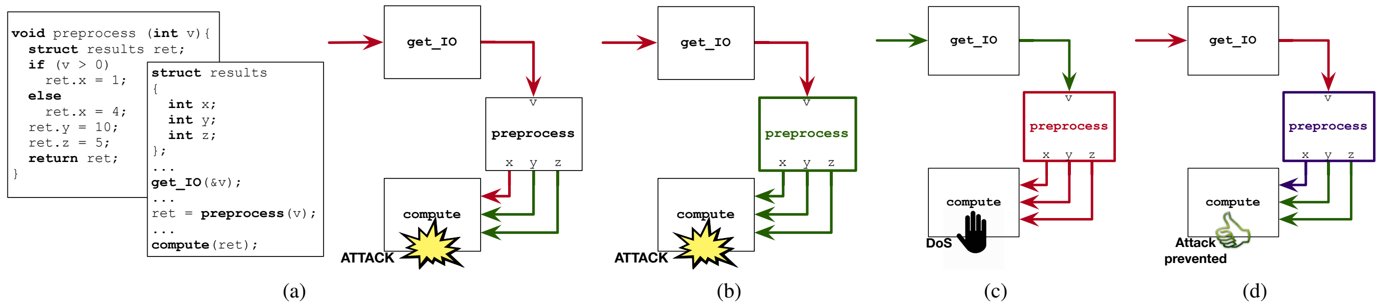


Fig. 2. Motivating example for implementing DIFT inside accelerators. Green links represent trusted data, red links represent malicious data, pink links represent tainted data. (a) Original code with unprotected execution can potentially lead to a security violation; (b) Permissive execution. Accelerators that are always considered trustable may invalidate taint tags and lead to security violations; (c) Untrusted execution. Services can be blocked when accelerators are always considered untrusted and taint all data; (d) Execution with DIFT. Correct DIFT allows the system to behave correctly.

B. Attack Model

We consider a heterogeneous system (see Fig. 1(a)), composed of a processor core (CPU) and one or more hardware accelerators. The CPU executes software applications. Part of the computation is offloaded to the accelerators. Assuming that the design tool chain (i.e., the hardware and software compilers) is trusted, the applications are vulnerable to software-based attacks. Even though accelerators implement a fixed functionality, an attacker can tamper with the system. Accelerators receive the input parameters through their configuration registers, perform the computation, and return the results in the same way as software routines. The flow of information as it transits through the accelerators is exposed to the same security problems as software applications. Knowing a vulnerability in a privileged program, attackers generate malicious inputs that allow them to exploit the vulnerability.

We consider systems with different protection mechanisms. First, we consider legacy systems (i.e., without support for non-executable memory). In this case, buffer overflow and format string bugs can be exploited for code injection by executing malicious code in the address space. We also consider systems that support non-executable memory, where the attacker can leverage existing code to compromise the system (e.g., return-to-libc attacks). In these cases, attacks exploit the flow of information (buffer overflow, format string bugs, etc.) to take control of a vulnerable program, jump to a target address, and execute a malicious routine [8], [7]. When implementing security protections for these systems, one needs to guarantee that offloading the computation to an accelerator does not compromise the security. In accelerators, the malicious inputs that corrupt the program execution can be: 1) provided by the user and simply go through the accelerators; 2) generated by the accelerator based on specific set of inputs given to the component; or 3) generated by unauthorized interactions between the system and the accelerator (e.g., unexpected interrupt requests). First, one has to guarantee that security protections for software applications can be executed on heterogeneous systems without losing security information. Second, one must prevent accelerators from tampering with the data in the shared memory (i.e., DRAM). Such data could be changed by compromised accelerator in such a way that the software routine receiving it would become a suitable spot for hijacking. Finally, one must verify the interactions between the accelerators and the system.

C. Dynamic Information Flow Tracking

Fig 2(a) is a simple example in C code, composed of a function `preprocess`, which returns three values through a `struct` construct. The value of variable `ret.x` depends on a user-input data, while the values of variables `ret.y` and `ret.z` are not. An attacker may tamper with the input data to force a malicious behavior in the function `compute`.

Dynamic Information Flow Tracking (DIFT) allows one to determine whether there is a dependency between an untrusted value and a variable that is used in a specific location of the code (e.g., in the control condition of a jump instruction). If so, the variable is *tainted*, i.e., it cannot be trusted. Specific operations on tainted variables can be forbidden to avoid exploits such as buffer overflow or unauthorized access to memory locations. DIFT includes three steps:

- **Tag Initialization**: a tag is associated with each variable to be tracked. In our example, tags are associated with each field of the structure.
- **Tag Propagation**: based on the operations on the data, the taint tags are propagated from input to output. Precise tag propagation can be obtained by instrumenting the code in software [20], by modifying the processor microarchitecture [15], or by offloading the tag propagation to an external module [10].
- **Tag Check**: taint tags are regularly analyzed based on the given security policy to determine unsafe uses.

However, if the component executing the given code has no support for DIFT, the function is a black box from a DIFT viewpoint and the user must rely on simplistic assumptions to mark the data. One possibility is to consider all data propagated by the function (e.g., output values or memory data) as untainted regardless of its input values and the performed computation. This is represented in Fig 2(b), where the return values of the function `preprocess` are considered trusted. We lose information on the taint tags and the analysis may lead to a *false negative*, i.e., a computation that is considered secured when it is not, leading to an attack exploitation. The designer can take a conservative approach, marking all data propagated by the function as untrusted, as in Fig 2(c). All

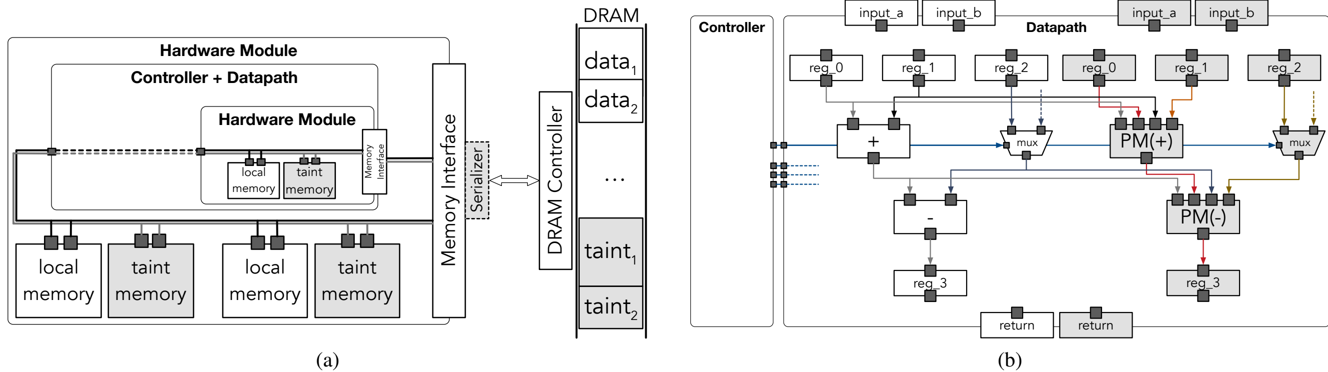


Fig. 3. (a) Microarchitecture of an accelerator with DIFT support. (b) Connections of shadow logic inside the datapath: grey elements support taint propagation, while PM modules represent the taint propagation modules associated with the respective datapath functional units.

variables within the `struct` are marked as untrusted for the function `propagate`. Over-tainting could cause *false positives*, avoiding correct execution of an application. Resources that are otherwise available may be blocked due to excessive tainting.

In both cases, simple assumptions do not capture the flow of information inside the function, limiting the legitimate use of the accelerator or opening the system to potential attacks. It is important to perform precise DIFT, marking only the variables that are untrusted and correctly propagating the taint tags through the accelerator and without tainting other variables. The use of taint propagation is shown in 2(d), where DIFT is implemented in function `preprocess` and so only the data related to `ret.x` is tainted for the function `compute`, enabling the designer to identify the attack.

Precise DIFT is critical in accelerators. Without taint propagation *inside* the accelerators, it is impossible to obtain correct results in most cases. However, intrinsic support for DIFT requires modifications to the accelerator microarchitecture and, in some cases, a wholesale re-design. HLS tools can help designers to automatically create accelerators with DIFT support so as to correctly propagate the taint tags within the accelerator and achieve the same taint propagation results as obtained in software.

III. Accelerators with DIFT Support

This section describes how we extend the baseline accelerator microarchitecture to obtain a DIFT-enhanced accelerator that can support DIFT with different taint granularity. One can assign a taint tag to each variable, to each byte, or to each bit of data. The corresponding microarchitecture is accordingly generated. We describe the solutions to limit the DIFT overheads. The different parts of an accelerator (Section II) are modified as follows:

- the **controller** requires only a few modifications, corresponding to the implementation of taint checks according to the security policy (see Section III-A);
- the **datapath** includes additional logic to generate the taint tags associated with temporary values, and to compute and propagate their values according to and concurrent with the accelerator execution (see Section III-B);
- the **memory elements** are extended to store taint tags and to support an efficient exchange of the taint tags associated with the data stored in DRAM (see Section III-C).

TaintHLS generates a DIFT-enhanced accelerator once the baseline accelerator microarchitecture is available.

A. Controller

The controller specifies the operations to be executed in each clock cycle and generates the control signals to drive the multiplexers, to enable writing into registers and to execute memory operations using the memory interface. When implementing DIFT, the controller is also responsible for detecting and managing *security exceptions*, sending proper signals to the rest of the system [7]. This is implemented with an additional output pin, connected to a dedicated interrupt line. When a security violation occurs, the accelerator is halted, a security interrupt is raised, and the corresponding security exception routine is executed by the processor. The designer can configure *security policies* to specify what is allowed (or disallowed) when executing on tainted data. While the detection of security violations is distributed across the different FSM transitions, such exceptions are managed by a central *security manager*. We consider two cases: FSM transitions based on potentially-tainted values coming from the datapath and memory operations that use potentially-tainted addresses affecting the DRAM data.

First, based on the outcome of the datapath operations, (e.g., the result of arithmetic comparisons or the flags set by arithmetic operations), the FSM can perform unsafe transitions. Inputs to the datapath may be malicious external data and so are the results of the operations. The tainted data may change the result of a condition and a branch taken. The data and the associated taint tag are sent to the controller, while the detection of violations is implemented according to the security policy. For instance, according to a security policy, a security violation is raised for all tainted transitions or only for critical ones determined by the designer.

Second, the accelerator may access external memory using a tainted address. The attacker may have tampered with the address to read from or write to an unauthorized memory location. Even though systems are protected against memory

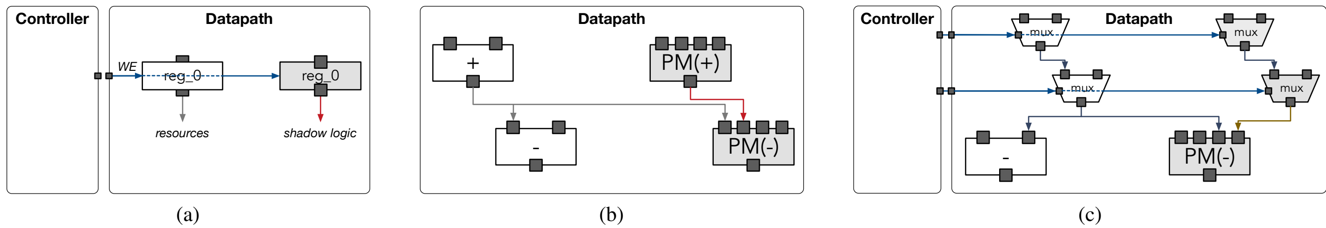


Fig. 4. Microarchitectural solutions to ensure *data flow consistency*. (a) Enabling concurrent operations on both data and tag registers. (b) Direct connections between datapath operators (i.e., *chaining*). (c) Connections of shadow logic with a multiplexer tree.

accesses outside of the memory space allocated to the accelerator, it is not guaranteed that the accelerator is always used in a safe way. The accelerator may access sensitive data within the memory space of the accelerator and leak private information. The accelerator should be configured to raise exceptions when addresses are tainted on a case-by-case basis. The accelerator can be configured either to prevent operations when any part of the address used for the operation is tainted (enforcing *strict memory protection*) or only when specific bits are altered. For example, the controller can be configured to notify a security violation only when the most significant bits are tainted, specifying that the accelerator is stopped only when the memory location is significantly changed (*permissive memory protection*). This configuration is based on the application developer input on the pointers and its knowledge of the application code. One can mark specific pointers as critical or some memory operations as benign even when one of the elements is tainted.

B. Datapath

As shown in Fig. 3(a), the *microarchitecture* is supplemented with a *shadow microarchitecture* [15]. The shadow microarchitecture implements DIFT and includes:

- **taint registers** to store the taint tags associated with the data values propagated by the functional units in the baseline microarchitecture;
- **propagation modules** to combine the taint tags and the data to compute the taint tags associated with the output of any operations.

We discuss these extensions of the shadow microarchitecture and their interconnection.

Taint Registers. We associate one taint tag for each variable that goes through the accelerator, including temporary values generated during the computation. These variables are stored in datapath registers and the corresponding tags represent the level of security/trust associated with the data. Data variables and taint tags have identical lifetimes during the accelerator execution. Hence, register allocation on the taint tags yields the same results as that for the corresponding data variables. Hence, we create a taint register for each register of the datapath. Then, when two variables share the same datapath register, we assign the corresponding taint tags to share the same tag register. When one variable is written into a register, the associated taint tag is written into the corresponding taint register. We use the same write-enable signal to control these two registers, as shown in Fig. 4(a) so that the values and the taint tags are written at the same time.

The size of each taint register depends on the selected granularity. We support three levels of granularity: bit, byte and variable. In bit-level taint propagation, a taint register has as many bits as the corresponding variable [15]. For byte-level taint propagation, we have a taint bit for every byte of the variable [10]. For variable-level taint propagation, we have one taint bit per variable [6].

Taint Propagation Modules and Logic Interconnection. We add extra ports to the interface of each hardware component to propagate the taint tags associated with the input parameters and the return value with the code executing on the CPU. Taint propagation modules determine a taint tag associated with the output of a functional unit, i.e., how much the input values are tainting the output. For example, when adding two values, the associated propagation modules determines the taint tag corresponding to the result of the addition. To this end, we add a taint propagation module for each functional unit.

Given an operation to be performed, the corresponding taint propagation module combines the input values and the associated taint tags (see Fig. 3(b)) to determine the taint tag associated with its result. We associate a different *taint propagation rule* with each operation type. Each propagation module depends on: 1) the operation to perform; and 2) the granularity of the taint propagation. For example, in bit-level taint propagation, every change at the input of a module must be precisely tracked at the output [15]. Precise analysis requires more logic than taint propagation performed on variable-level tags, where simple Boolean operations (e.g., OR gates) suffice to propagate the taint values.

We create the shadow microarchitecture so that it has the same topology as the baseline microarchitecture, as shown in Figs. 4(b) and 4(c). For example, when two functional units are directly connected by chaining them, the two propagation modules are also chained in the same way (Fig. 4(b)). Similarly, when a multiplexer tree is used to connect some registers to a functional unit, the respective taint registers are connected with a similar multiplexer tree controlled by the same select signals (Fig. 4(c)). In this way, TAINTHLS ensures *data flow consistency* between the baseline and the shadow microarchitectures, allowing the concurrent propagation of the data values and the associated taint tags.

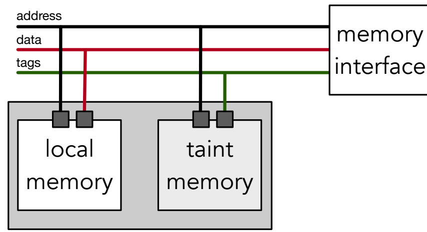


Fig. 5. Operations on taint memories are performed in parallel to the ones on the data by the additional bus carrying the taint tags.

C. Memory Elements

Hardware accelerators store part of the data locally in heterogeneous and distributed memories (*scratchpad memories* [21], [22] or *private local memories* [18]), while the rest of the data is stored in DRAM and accessed via a memory interface [12], [22], [18]. Since local memories can be connected in a daisy chain for dynamic address resolution [12], tampering with the memory address may result in memory accesses to unauthorized memory locations. So, one must keep track of memory operations with taint tags associated with both the addresses and the data values. TAINTHLS extends the daisy-chain memory architecture as follows:

- Addresses are handled like other variables stored in registers. So, the corresponding taint tag is generated and propagated when needed.
- Each local memory is supplemented with an additional *taint memory* to hold the taint tags. An additional *taint bus* carries the taint tags associated with the data values so that the memory accesses to the data and the tags can be performed using the same memory address. This is shown in Fig. 5, where operations on the internal memories are performed with the same latency assumed for execution without DIFT.
- Accesses to the external memory may be constrained by the architecture. When it is not possible to introduce a dedicated taint bus to the external interface, TAINTHLS serializes the memory accesses to data and associated tags.

This shadow memory architecture has no impact on the scheduling of the memory operations. Operations on the external memory require a latency-insensitive protocol so that the computation is resumed when the transfers are complete. As a result, the shadow logic is created with no information about the external architecture and modifications are required only to the top module to serialize the external accesses. This is shown in Fig. 3(b) as the *Serializer* component. The layout of the memory tags, especially in DRAM, is important to understand how to translate the memory addresses provided to the memory controller. For example, when the data and tags are interleaved, the memory address is shifted by one bit and the less significant bit is used to address the data or the tags. Memory readdressing is performed by simple logic that is connected to the input ports of the memory elements[1].

[1]It is also possible to use a dedicated memory space for the tags, eventually with specific optimizations to reduce the memory footprint.

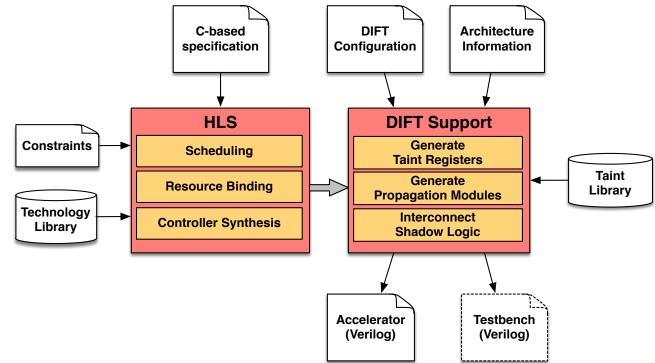


Fig. 6. Overview of TAINTHLS methodology.

IV. DESIGN METHODOLOGY

This section presents the TAINTHLS methodology to automatically generate hardware accelerators with intrinsic DIFT support outlined in Section III. As shown in Fig. 6, TAINTHLS uses high-level synthesis (HLS) and receives as input a C specification, along with synthesis constraints (e.g., the clock period) and the target technology (e.g., the description and the timing of the functional units). This information is used to generate the controller, the datapath, the local memories and the memory interface [5]. To generate the shadow microarchitecture, one must provide a *Taint Library*, containing the description of the taint propagation modules associated with each functional unit (e.g., adder, multiplier, or shifter) In this way, one can evaluate different taint propagation rules and alternative shadow microarchitectures with no changes to the datapath functionality. One can also evaluate the effects on the resources and the critical path. Possible optimizations (e.g., merging functionally equivalent nodes) will be performed during logic synthesis. As output, TAINTHLS produces a Verilog RTL description of the DIFT-enhanced accelerator, along with a testbench for simulation.

Besides the traditional HLS steps (e.g., scheduling, resource binding, and controller synthesis), TAINTHLS includes three additional steps to synthesize DIFT support: generation of the taint values, generation of the taint propagation modules, and interconnection of the shadow microarchitecture components.

A. High-Level Synthesis

A traditional HLS flow interfaces with compilers (e.g., GCC or LLVM) to parse the input C code, apply compiler optimizations, and extract the intermediate representation [5]. Most modern HLS tools exploit the Single Static Assignment (SSA) form, which is composed of simple instructions that can be easily manipulated and translated into hardware [23]. The SSA form generates a unique identifier for each assignment to the same variable, increasing the number of temporary values but simplifying the subsequent HLS steps. After the assignment of resources and memories (called *allocation*), scheduling is performed to determine the operations to be executed in each clock cycle. Operations scheduled in different clock cycles can potentially reuse the same resources. Temporary values crossing the clock boundary are assigned to registers to be

stored. Different algorithms can be used to solve the *module binding* and *register binding* problems. For example, liveness analysis followed by register coloring guarantees the minimum number of registers with no conflicts [24]. The penultimate step in the accelerator creation is the *interconnection binding*, where the different resources (e.g., functional units and registers) are interconnected with multiplexers. Finally, the controller is generated (*controller synthesis*), where the control flow of the accelerator is implemented with an FSM. The FSM generates, for each clock cycle, the proper signals that control the datapath resources and drive the data values based on the operations to be executed and the datapath microarchitecture. The synthesis of each accelerator is performed hierarchically, starting from the innermost C functions. In this way, when generating a module, all submodules have been already generated and they can be interconnected as any other datapath resource.

B. Generation of Taint Values

In the first step, we generate all components that are necessary to store the taint tags (either taint registers or taint memories). As discussed in Section III, we associate a taint register with each datapath register, whose bitwidth depends on the chosen granularity. This information is specified in the *DIFT Configuration* file. We then generate an additional tag for each input parameter (to be specified with additional memory-mapped write operations on the input interface) and one for the return value, if any (to be specified with one additional memory-mapped read operation). These operations are performed only once when the accelerator is configured (for the input parameters) or during the interrupt routine (for the return value). The performance overhead is thus negligible compared to the execution time of the accelerator. In addition, such registers are initialized to 1 after resetting the circuit in order to identify the values as *tainted* by default. In this way, any misuse of the accelerator will assume that the provided data cannot be trusted.

Next, we generate the taint memories that will be used to store the taint tags associated with the data stored in the local SPMs. Given a data structure, which has N words of M bits, we use a dual-port embedded memory to store the data [12], [18]. If the HLS scheduling guarantees that there is only one memory operation per clock cycle and the size of the data structure plus the size of the corresponding taint tags can fit into the same embedded memory, we can use the second port to perform the taint-related operation in parallel. Additional logic is generated to convert the memory address on the second port to properly address the taint area. In this way, we can store the data and the corresponding tags in the same embedded memory, reducing the memory occupation. Otherwise, we must instantiate an additional embedded memory for storing the taint tags, where the same address is used to address both memories (see Fig. 5)

C. Generation of Propagation Modules

Based on the results of module binding, we add the modules to propagate the taint tags. First, we select the proper propagation modules based on the functional units, the selected granularity, and the input *Taint Library*. In our *Taint Library*, we support modules designed with any existing techniques, like gate-level information flow tracking [15] and precise approaches [25]. We assume that the modules are available in the library since the definition of the propagation rules and the optimization of the corresponding propagation modules is out of the scope of this work. Our approach can be used to evaluate the trade-off between complexity and precision of such modules [26]. Different from gate-level tracking that creates shadow microarchitecture for all hardware resources, we exploit additional HLS information to reduce the resource overhead. For example, the accelerator is considered in a *correct* state if it has passed all security checks imposed on the FSM transitions (see Section III-A). In this case, the operations executed in the clock cycle are legitimate and the corresponding control signals are generated to propagate the data values and the taint tags through the datapath resources and the propagation modules, respectively. So, the interconnection multiplexers do not require to be implemented with DIFT support. So, for each functional unit, we determine the available propagation modules in the taint library (e.g., the ones associated with the same operation) and we select the instance corresponding to the level of granularity selected for the accelerator. Given the finite set of operations that may be generated by the HLS frontend, the definition of the corresponding complete *Taint Library* guarantees that any C function is correctly synthesized with TAINTHLS.

When the complexity of these modules is simpler than the corresponding datapath operators (e.g., in case of variable- and byte-level tracking), the critical path is not affected. In all other cases, the HLS engine can be configured to have a larger slack in each clock cycle after the scheduling to reduce *operation chaining* [5]. Operation chaining may increase the latency of the circuit, but it guarantees to generate enough margin for the shadow logic. As a result, the shadow microarchitecture can be safely created after all steps for generating the datapath have been completed. This makes the TAINTHLS methodology modular and easy to implement.

D. Interconnections in the Shadow Microarchitecture

Finally, we interconnect all these components, along with the memory interfaces, to create the shadow microarchitecture. In particular, the propagation modules are connected not only to the taint registers, but also to the functional units and registers providing the actual values of the computation to generate the final description of each hardware module. For system-level integration, we provide the characteristics of the target architecture to determine how to interface the generated accelerator with the system (e.g., if the accelerator is tightly or loosely coupled with the processor). In the architectural configuration, we specify if we can instantiate the additional taint bus to carry the taint tags to the DRAM. If not, we introduce the *Serializer* (see Fig. 3(b)) between the memory interface and the bus connection. We provide information on the memory layout to determine where to read/write the memory tags. Such information is provided to the methodology and used to configure and customize the proper components (memory

TABLE I
CHARACTERISTICS OF THE BENCHMARKS USED TO EVALUATE TAINTHLS. LOC IS THE LINES OF C CODE.

BENCHMARK	LOC	FUNCTIONS	ARRAYS	EXTERNAL MEMORY
ICRC	76	1	1	✓
AES	326	7	3	✓
BFS	221	1	5	✓
Viterbi	146	1	5	✓

TABLE II
IMPLEMENTATION CHARACTERISTICS OF THE *Baseline* ACCELERATORS (XILINX VIRTEX-7 FPGA VX485, 100 MHZ). LUT: LOOK-UP TABLES; FF: FLIP-FLOPS; DSP: DIGITAL SIGNAL PROCESSOR CORES; BRAM: BLOCK RANDOM ACCESS MEMORIES.

BENCHMARK	LUT	FF	DSP	BRAM	CLOCK CYCLES
ICRC	3,459	323	0	2	1,957
AES	10,641	4,571	0	2	2,266
BFS	650	570	0	2	23,265
Viterbi	1,087	974	2	0	1,594,464

interfaces and *Serializer*) as described in Section III-C. Finally, additional registers are added in the top-level interface to store the taint tags for input/output ports.

V. EXPERIMENTAL EVALUATION

This sections evaluates the DIFT-enhanced accelerators generated by TAINTHLS.

A. Experimental Setup

We extended BAMBU ver. 0.9.4 [27], a modular and open-source framework for research in high-level synthesis [5], [11]. BAMBU receives as input C-based specifications to be synthesized for FPGA targets, leveraging the GCC compiler to perform many compiler-related optimizations (e.g., loop transformations or alias analysis). We implemented the TAINTHLS methodology presented in Section IV as an additional pass on the top of the HLS results generated by BAMBU. To evaluate the overhead introduced by DIFT support, we targeted a Xilinx Virtex-7 FPGA (xc7vx485t) at the frequency of 100 MHz and local memories implemented with BRAMs. We performed logic synthesis using Xilinx Vivado 2016.3 and RTL simulation with Mentor ModelSim SE 10.3 for area and performance results. We evaluated the hardware resources necessary to implement the shadow microarchitecture (Section V-B) and the overhead in terms of clock cycles to perform the tag operations (Section V-C). We validated the DIFT-enhanced accelerators against full-software execution (Section V-D).

We applied TAINTHLS to generate accelerators for four benchmarks:

- Invariant Cyclic Redundancy Check (ICRC) is a popular error detection code [28];
- Advanced Encryption Standard (AES) is an iterated block cipher (encryption algorithm) [29];
- Breadth-First Search (BFS) is a queue-based algorithm for graph traversal [29];
- Viterbi is a dynamic programming method to compute the most likely hidden Markov chain [29].

These kernels are likely implemented as accelerators and used in different applications. The kernels are in C code and their characteristics are reported in Table I. In this table, we include the size of the code, the number of functions, the number of local memories, and the presence of memory interface. Table II reports the the characteristics of the baseline accelerators obtained using BAMBU without DIFT.

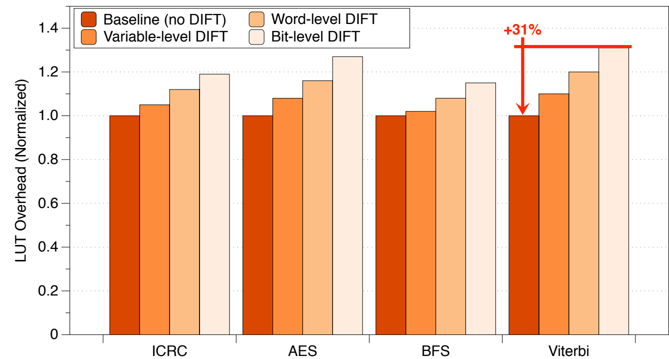


Fig. 7. Look-up table overhead in terms of LUT for DIFT support in accelerators.

B. Resource Overhead

We evaluate the overhead of the propagation modules, which have been implemented as *precise modules* [25], described in Verilog, and provided as input to BAMBU in the *Taint Library*. We compare the resources for implementing the accelerators without and with DIFT support at different levels of granularity. Fig. 7 shows the overhead in terms of LUTs when introducing taint propagation in accelerators. As expected, the number of LUTs increases with the granularity of the taint propagation. We need complex rules to achieve the same results obtained when the DIFT is performed in software (see Section V-D). BFS benchmark has the minimal overhead (up to 15%) since its logic is simple and we need simple modules for the operations of this application. On the contrary, Viterbi benchmark requires up to 31% of additional logic. However, this overhead is small compared to gate-level tracking [15]. The low overhead is because multiplexers do not require additional select logic and the memory interfaces are minimally affected as they only forward one additional memory operation. Additional flip-flops are used to store the taint tags and their number is proportional to the number of variables and depends on the chosen granularity. In the worst case, for bit-level taint propagation, we simply double the number of flip-flops used by the accelerator. Similarly, BRAMs are doubled in all cases since we do not use any compression on the taint tags. So, TAINTHLS doubles the memory footprint of the accelerator. Similar overhead is obtained for the taint tags stored in the external memory. On one hand, the memory footprint of the entire application doubles. On the other, this simplifies the memory interfaces. DSPs are never affected by

DIFT support since the propagation rules are simple and do not require such components.

When increasing the granularity to the bit-level, propagation modules become complex. For complex arithmetic operations (e.g., multiplication and integer division), the delay starts becoming significant (∼30% longer than the critical path of the corresponding functional units). However, none of the shadow microarchitectures violated the clock period of 10 ns, allowing the accelerators to effectively run at 100 MHz.

Even though BAMBU does not fully support ASIC technologies, we performed tests on micro-benchmarks to evaluate the impact for these targets. These micro-benchmarks are simple functions with no memory accesses. We created the baseline and DIFT-enhanced versions of these micro-benchmarks, targeting ASIC (Synopsys 32 nm SAED library at 500 MHz) and FPGA (Xilinx Virtex-7 xc7vx485t) and compared the overheads of the taint logic. In ASIC implementations, the shadow microarchitecture introduces an area overhead that is slightly larger (∼6%) than the one introduced when targeting FPGAs. FPGA synthesis tools jointly optimize the DIFT logic and the functional units better, using a lower number of LUTs.

C. Performance Overhead

The *Serializer* in Fig. 3(b) is the only element that can introduce performance overhead. All other taint operations are performed in parallel to the corresponding baseline microarchitecture counterparts. We simulated the accelerators with and without the *Serializer*, exploring the cases with a single shared data bus or with an extra taint bus to the external memory. ICRC benchmark requires 1,957 clock cycles for the baseline case and when taint propagation uses a dedicated bus. In this architecture all memory operations on data and tags are performed in parallel. When the *Serializer* is introduced, every memory operation doubles its latency and the ICRC now takes 2,039 cycles (+4.20%). Since we do not optimize when exchanging the taint tags with the external memory, there is no difference when using different taint granularity. Other benchmarks have a lower overhead (less than 1%). So, one can achieve the same performance results even without the dedicated bus.

D. Validation of DIFT Results

To verify the correctness of the DIFT logic generated by TAINTHLS, we compared the results of the taint propagation against those for the corresponding software obtained by instrumenting the initial C code using a taint propagation library [20]. We generated 100 combinations of data and taint values as input and compared the taint tags after the execution of the accelerator (i.e., the returned tag and taint tags in the external memory) with the ones obtained in software. We carried out these experiments for all levels of granularity and, in all cases, the results matched (i.e., the same values for the tags and the same number of false positive/negative executions). Propagation rules were designed to achieve the same results. This demonstrates that TAINTHLS generates DIFT-enhanced accelerators with the same level of security as the corresponding DIFT-enhanced software.

VI. RELATED WORK

Information flow tracking has been studied for processors to identify malicious uses. It can be performed statically (by analyzing the code) or dynamically (by tracking the data during the execution). Static methods allow for a thorough analysis of the application, but the set of problems that can be identified is limited [30]. Dynamic tracking allows detection and prevention, but may incur large overheads [7]. DIFT solutions can be implemented by modifying the software [6], [20] or the underlying hardware [10], [15]. `libdft` is a software library that can be used with unmodified binaries to perform information tracking [20]. `TaintDroid` is a modified virtualization environment to perform information flow tracking in Android-based applications [6]. A dedicated DIFT coprocessor that operates on 4-bit tags on the execution trace of the processor is presented in [10]. This solution is general and partially executes in parallel to the processor, but it is not efficient for fixed-function components. Gate-level information flow tracking (GLIFT) has been reported in [15]. Processors built on the top of these GLIFT-enabled gates can track information. However, this solution does not apply to already-fabricated processors. GLIFT methods have been extended to trade off precision and cost. However, the overhead of GLIFT may be large since no information on the specialized microarchitecture is included in the DIFT logic generation [25], [26].

None of these solutions support tracking information in heterogeneous architectures. WHISK discusses the requirements for information flow tracking in heterogeneous architectures [14]. WHISK is complementary to this paper; it focuses on how the taint tags should be exchanged in the system but does not discuss generation of the shadow logic and taint propagation inside the accelerators. Raksha implements DIFT in the processor pipeline but does not support tightly-coupled accelerators [8]. DIFT support at the RTL level was proposed in [25], but it has not been fully integrated into a high-level design methodology for complex accelerators with memory accesses. However, heterogeneous architectures are popular in high-performance, energy-efficient systems [1]. They feature a large number of accelerators that are tightly [21], [31] or loosely [3], [22] coupled with the processors. TAINTHLS can generate DIFT for both types of accelerators.

HLS is used to generate accelerators [5]. On one hand, security aspects like information flow tracking have not been considered during accelerator generation. On the other hand, HLS reduces the effort required in designing security-enhanced accelerators [32].

VII. CONCLUDING REMARKS

TAINTHLS is a methodology based on high-level synthesis to automate the design of accelerators with intrinsic support for dynamic information flow tracking. TAINTHLS adopts a modular microarchitecture shadowing the baseline microarchitecture to ensure consistency of the data-flow of the accelerator data and the taint tags. One can configure the tags with distinct rules for propagation through each module. We test the approach using a state-of-the-art HLS tool to

confirm that it produces the same results as software-based DIFT. While variable-level taint propagation yields a low resource overhead (less than 10%), bit-level taint propagation requires up to 30% of additional logic to implement more complex propagation rules. Researchers can use the framework to secure heterogeneous SoCs in terms of tag propagation rules, memory architectures, and system-level integration. HLS for security is an important research field and TAINTHLS should be complemented with other HLS-based methods for trustworthy system design.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 365–376.

[2] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich CMPs," in *Proceedings of the Design Automation Conference (DAC)*, Jun. 2012, pp. 843–849.

[3] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *Proceedings of the Design Automation Conference (DAC)*, Jun. 2015, pp. 202:1–202:6.

[4] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The Accelerator Store: A shared memory framework for accelerator-based systems," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, pp. 48:1–48:22, Jan. 2012.

[5] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA High-Level Synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[6] W. Enck, P. Gilbert, 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," in *Proceedings of the Conference on Operating Systems Design and Implementation (OSDI)*, Oct. 2010, pp. 393–407.

[7] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004, pp. 85–96.

[8] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2007, pp. 482–493.

[9] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[10] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *Proceedings of the International Conference on Dependable Systems & Networks (DSN)*, Jun. 2009, pp. 105–114.

[11] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2013, pp. 1–4.

[12] C. Pilato, F. Ferrandi, and D. Sciuto, "A design methodology to implement memory accesses in high-level synthesis," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2011, pp. 49–58.

[13] M. Minutoli, V. G. Castellana, A. Tumeo, M. Lattuada, and F. Ferrandi, "Enabling the high level synthesis of data analytics accelerators," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2016, pp. 1–15.

[14] 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)*, Sep. 2013, pp. 1–9.

[15] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1128–1140, Aug. 2011.

[16] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *ISSCC Digest of Technical Papers*, Feb. 2014, pp. 10–14.

[17] J. Zhu and D. D. Gajski, "A unified formal model of ISA and FSMD," in *Proceedings of the International Workshop on Hardware/Software Codesign (CODES)*, Mar. 1999, pp. 121–125.

[18] C. Pilato, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "System-level optimization of accelerator local memory for heterogeneous Systems-on-Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 435–448, Mar. 2017.

[19] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 2014, pp. 199–208.

[20] V. P. Kemerlis *et al.*, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the Conference on Virtual Execution Environments (VEE)*, Mar. 2012, pp. 121–132.

[21] F. Conti, A. Marongiu, and L. Benini, "Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2013, pp. 1–10.

[22] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *Proceedings of the Design Automation Conference (DAC)*, Jun. 2014, pp. 1–6.

[23] D. Novillo, "Design and implementation of Tree SSA," in *GCC Developers' Summit*, 2004, pp. 119–130.

[24] L. Stok, "Data path synthesis," *Integration, the VLSI Journal*, vol. 18, no. 1, pp. 1–71, Dec. 1994.

[25] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*, Mar. 2017, pp. 1691–1696.

[26] A. Becker, W. Hu, Y. Tai, P. Brisk, R. Kastner, and P. Ienne, "Arbitrary precision and complexity tradeoffs for gate-level information flow tracking," in *Proceedings of the Design Automation Conference (DAC)*, Jun. 2017, pp. 1–6.

[27] Politecnico di Milano, "Bambu 0.9.4," Available at http://panda.dei.polimi.it, 2016.

[28] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power MCORE architecture," in *Proceedings of the Power Driven Microarchitecture Workshop*, Jul. 1998, pp. 145–150.

[29] B. Reagen, R. Adolf, Y. S. Shao, G. Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings on the International Symposium on Workload Characterization (IISWC)*, Oct. 2014, pp. 110–119.

[30] Y. Liu and A. Milanova, "Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, Mar. 2010, pp. 146–155.

[31] N. Goulding-Hotta, J. Sampson, Q. Zheng, V. Bhatt, J. Auricchio, S. Swanson, and M. B. Taylor, "GreenDroid: An architecture for the Dark Silicon Age," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2012, pp. 100–105.

[32] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni, "Securing hardware accelerators: a new challenge for High-Level Synthesis," *IEEE Embedded Systems Letters*, pp. 1–4, First online: Nov. 2017.

Christian Pilato received the Laurea degree in Computer Engineering and the Ph.D. degree in Information Technology from the Politecnico di Milano, Italy, in 2007 and 2011, respectively. From 2013 to 2016, he was a Post-doc Research Scientist with the Department of Computer Science, Columbia University, New York, NY, USA. He is currently a Post-doc at the University of Lugano, Switzerland. His research interests include high-level synthesis, reconfigurable systems and system-on-chip architectures, with emphasis on memory aspects. In 2014 Dr. Pilato served as program chair of the Embedded and Ubiquitous Conference (EUC) and he is currently involved in the program committees of many conferences on embedded systems, CAD, and reconfigurable architectures (e.g., FPL, DATE, CASES). He is a member of the Association for Computing Machinery.

Kaijie Wu received the BE degree from Xidian University, Xi'an, China, in 1996, the MS degree from the University of Science and Technology of China, Hefei, China, in 1999, and the Ph.D. degree in electrical engineering from Polytechnic University (Now Polytechnic Institute of New York University), Brooklyn , New York, in 2004. He then joined University of Illinois, Chicago, USA as an Assistant Professor. From 2014 to 2016, he was a professor at the College of Computer Science, Chongqing University, China. His research is on the big area of trustworthy computing with special interest on dependable computing and hardware security. He is the recipient of the 2004 EDAA Outstanding Dissertation Award for "new directions in circuit and system test.", and the "Most Significant Paper" award from the International Test Conference, 2014. He is a member of the IEEE.

Siddharth Garg received his Ph.D. degree in Electrical and Computer Engineering from Carnegie Mellon University in 2009, and a B.Tech. degree in Electrical Engineering from the Indian Institute of Technology Madras. He joined NYU in Fall 2014 as an Assistant Professor, and prior to that, was an Assistant Professor at the University of Waterloo from 2010-2014. His general research interests are in computer engineering, and more particularly in secure, reliable and energy-efficient computing. In 2016, Siddharth was listed in Popular Science Magazine's annual list of "Brilliant 10" researchers. Siddharth has received the NSF CAREER Award (2015), and paper awards at the IEEE Symposium on Security and Privacy (S&P) 2016, USENIX Security Symposium 2013, at the Semiconductor Research Consortium TECHCON in 2010, and the International Symposium on Quality in Electronic Design (ISQED) in 2009. Siddharth also received the Angel G. Jordan Award from ECE department of Carnegie Mellon University for outstanding thesis contributions and service to the community. He serves on the technical program committee of several top conferences in the area of computer engineering and computer hardware, and has served as a reviewer for several IEEE and ACM journals.

Ramesh Karri is a Professor of ECE at New York University. He co-directs the NYU Center for Cyber Security (http://cyber.nyu.edu). He also leads the Cyber Security thrust of the NY State Center for Advanced Telecommunications Technologies at NYU. He co-founded the Trust-Hub (http://trust-hub.org) and organizes the Embedded Systems Challenge (https://csaw.engineering.nyu.edu/esc), the annual red team blue team event. Ramesh Karri has a Ph.D. in Computer Science and Engineering, from the UC San Diego and a B.E in ECE from Andhra University. His research and education activities in hardware cybersecurity include trustworthy ICs; processors and cyber-physical systems; security-aware computer-aided design, test, verification, validation, and reliability; nano meets security; hardware security competitions, benchmarks and metrics; biochip security; additive manufacturing security. He has published over 240 articles in leading journals and conference proceedings. Karri's work on hardware cybersecurity received best paper nominations (ICCD 2015 and DFTS 2015) and awards (ACM TODAES 2018, ITC 2014, CCS 2013, DFTS 2013 and VLSI Design 2012). He received the Humboldt Fellowship and the National Science Foundation CAREER Award. He serves on the editorial boards of several IEEE and ACM Transactions (TIFS, TCAD, TODAES, ESL, D&T, JETC). He served as an IEEE Computer Society Distinguished Visitor (2013-2015). He served on the Executive Committee of the IEEE/ACM Design Automation Conference leading the SecurityDAC initiative (2014-2017). He delivers invited keynotes, talks, and tutorials on Hardware Security and Trust Trust (ESRF, DAC, DATE, VTS, ITC, ICCD, NATW, LATW, CROSSING, etc.). He co-founded the IEEE/ACM NANOARCH Symposium and served as program/general chair of conferences (IEEE ICCD, IEEE HOST, IEEE DFTS, NANOARCH, RFIDSEC and WISEC). He serves on several program committees (DAC, ICCAD, HOST, ITC, VTS, ETS, ICCD, DTIS, WIFS).

Francesco Regazzoni is a senior researcher at the ALaRI Institute of University of Lugano (Lugano, Switzerland). He received his Master of Science degree from Politecnico di Milano and his PhD degree at the ALaRI Institute of University of Lugano. He has been assistant researcher at the Université Catholique de Louvain and at Technical University of Delft, and visiting researcher at several institutions, including NEC Labs America, Ruhr University of Bochum, EPFL, and NTU. His research interests are mainly focused on cyber-physical and embedded systems security, covering in particular side channel attacks, cryptographic hardware, and electronic design automation for security.