Implementing an Application Specific Instruction-set Processor for System Level Dynamic Program Analysis Engines

Ingoo Heo, Seoul National University
Minsu Kim, Korea Advanced Institute of Science and Technology
Yongje Lee, Samsung Electronics Co., Ltd and Seoul National University
Changho Choi, Korea Advanced Institute of Science and Technology
Jinyong Lee, Seoul National University
Brent Byunghoon Kang, Korea Advanced Institute of Science and Technology
Yunheung Paek, Seoul National University

In recent years, dynamic program analysis (DPA) has been widely used in various fields such as profiling, finding bugs and security. However, existing solutions have their own weaknesses. Software solutions provide flexibility in DPA but they suffer from tremendous performance overhead. In contrast, core-level hardware engines rely on specialized integrated logics and attain extremely fast computation, but they have a limited functional extensibility because the logics are tightly coupled with the host processor. To mend this, a prior system level approach utilizes an existing channel to integrate their hardware without necessitating the host architecture modification and introduced great potential in performance. Nevertheless, the prior work did not address the detailed design and implementation of the engine, which is quite essential to leverage the deployment on real systems. To address this, in this paper, we propose an implementation of programmable DPA hardware engine, called program analysis unit (PAU). PAU is an application specific instruction-set processor (ASIP) whose instruction-set is customized to reflect common features of various DPA methods. With the specialized architecture and programmability of software, our PAU aims at fast computation and sufficient flexibility. In our case studies on several DPA techniques, we show that our ASIP approach can be successfully applicable to complex DPA schemes while providing hardware-backed power in performance and software-based flexibility in analysis. Recent experiments on our FPGA prototype revealed that the performance of PAU is 4.7-13.6 times faster than pure software DPA, and the power/area consumption is also acceptably small compared to today's mobile processors.

Categories and Subject Descriptors: C.3 [**Computer Systems Organiztion**]: Special-Purpose and Application-Based Systems

General Terms: Design, Performance, Algorithm

This work was partly supported by the IT R&D program of MSIP/KEIT [K10047212, Development of homomorphic encryption supporting arithmetics on ciphertexts of size less than 1kB and its applications], the Brain Korea 21 Plus Project in 2015, IDEC and the National Research Foun- dation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2014R1A2A1A10051792). Also, this research was supported by the MSIP(Ministry of Science, ICT & Future Planning), Korea, under the BrainScoutingProgram(H7106-14-1011) supervised by the IITP(Institute for Information & Communication Promotion), and Agency for Defense Development (ADD) under Grant No. UD140002ED.
Author's addresses : I.Heo, Y.Lee, J.Lee and Y.Paek, Department of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC), Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul, 151-742, South Korea; email:igheo@sor.snu.ac.kr, yjlee@sor.snu.ac.kr, jylee@sor.snu.ac.kr,ypaek@snu.ac.kr; M.Kim(co-primary author), C.Choi and B.B.Kang, Graduate School of Information Security, Korea Advanced Institute of Science and Technology, 291 Daehak-ro, Yuseong-gu, Daejeon, 305-701, South Korea; email:pshskms@kaist.ac.kr, zpzigi@hgu.edu, brentkang@kaist.ac.kr
Corresponding authors : Y.Paek and B.B.Kang

DOI : http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2746238

Additional Key Words and Phrases: Dynamic Program Analysis (DPA), System level analysis hardware, Application Specific Instruction-Set Processor (ASIP), Dynamic Information Flow Tracking (DIFT)

ACM Reference Format:
Ingoo Heo, Minsu Kim, Yongje Lee, Changho Choi, Jinyong Lee, Brent Byunghoon Kang and Yunheung Paek, 2015. Implementing an Application Specific Instruction-set Processor for System Level Dynamic Program Analysis Engines *ACM Trans. Des. Autom. Electron. Syst.* 9, 4, Article 39 (March 2015), 33 pages.
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2746238

1. INTRODUCTION

Dynamic program analysis (DPA) is to analyze software code as it executes on a processor. In recent years, it has been widely used in profiling system performance, finding software bugs for reliability and runtime monitoring for system security. As an example, Memcheck [Seward and Nethercote 2005] is a DPA tool implemented in the Valgrind binary instrumentation framework [Nethercote and Seward 2007] and uses dataflow tracking to observe the memory usage behaviors of the target applications to detect unintended misuses of memory. Dynamic information flow tracking (DIFT) is also a representative DPA technique which tracks and restricts the use of designated data by managing metadata called *tag*. In many studies, DIFT has been used to effectively resolve their various problems such as runtime monitoring [Newsome and Song 2005; Dalton et al. 2007] or malware analysis [Bayer et al. 2009]. Likewise, DPA has been applied to enable many other types of techniques [Deng and Suh 2012] such as memory protection [Witchel et al. 2002], array bound checking [Devietti et al. 2008], software debugging support [Nethercote and Seward 2007] and garbage collection [Joao et al. 2009]. Consequently, with the ever-increasing importance, the use of DPA is being expanded to a wide range of security and reliability problems.

To achieve their goals of DPA, many researchers rely on dynamic binary instrumentation (DBI) frameworks such as Valgrind [Nethercote and Seward 2007], Pin [Luk et al. 2005], and DynamicRIO [Bruening 2004]. While dynamic analysis through software DBI provides complete analysis environment with the extreme flexibility, the amount of analysis at either test-time or runtime is bounded by the performance impact that can be tolerated [Tiwari et al. 2009]. The performance overhead is especially crucial in complex DPA techniques which require amount of computation as the target program executes. For example, LIFT [Qin et al. 2006], a DIFT solution with DBI tool, slows down the program execution by around 4 times at runtime even with aggressive optimizations. Although several approaches have been proposed to utilize multiprocessors [Chen et al. 2008; Nagarajan et al. 2008; Nightingale et al. 2008] that are readily available in modern multicore architectures where each core is a general-purpose processor (GPP), they could also not achieve sufficient performance improvement mainly because the original architectures were not optimized for DPA in the first place [Tiwari et al. 2009].

To address the shortcoming of software-based analysis, several *core-level* hardware supports for DPA have been proposed [Dalton et al. 2007; Venkataramani et al. 2008; Suh et al. 2004; Deng et al. 2010; Chen et al. 2008; Deng and Suh 2012], where extra hardware logic customized for DPA operations is integrated into a processor core. Even though they could bring the overhead down to a few percents, they require invasive modifications to the core internal (e.g., registers and pipeline data paths). In fact, microprocessor development may take several years and hundreds of engineers from an initial design to production [Kannan et al. 2009]. Therefore, the substantial costs of development to integrate the logic would hamper processor vendors to adopt new hardware unless its generality and versatility are clearly proven. For this reason, some proposed a flexible core-level accelerator integrated in a processor that can support a set of diverse DPA functions by reconfiguring the accelerator [Deng and Suh

2012]. However, they still have a limited functional extensibility in that a new DPA function cannot be supported by hardware unless it was considered in the initial hardware design.

As an alternative direction to avoid invasive core-level modification, a *system-level* DPA acceleration engine was proposed, which is integrated into a system by being connected to the processor through existing channels such as peripheral interfaces. In [Tiwari et al. 2009], they built a working prototype, called *Hardgrind*, where the engine is implemented as an external device and connected to the host system via a PCI bus. In the experiment, they demonstrated that even without internal changes to an existing CPU, heavy-weight DPA tools [Seward and Nethercote 2005] benefit from the acceleration strategy, and the speedup can be great, being up to 4.4 times faster than pure software techniques stated above. These results reveal a potential advantage of a system-level DPA engine that it may offer a more affordable solution to extend the engine for new DPA functions than the core-level ones because the extension could be made separately from the host system without necessitating the overall host architecture modification. Such an advantage would be particularly beneficial to recent mobile SoC platforms where the system-level integration provides a better extendibility by enabling the *platform-based design* which is a de facto standard methodology to develop complex SoCs including commercial products like smartphone application processors (APs). Because the platform-based design tends to foster systematic reuse of already-implemented modules [Bailey et al. 2005], it is important to preserve the other components intact when a functionality like DPA is additionally supported. In the system-level approach, all special logics customized for DPA are fully integrated into an independent module so that the other modules can be reused thereby lowering development risks, costs and time to market. Although the existing system-level approach [Tiwari et al. 2009] has evinced not only a great potential in performance but also the extendibility to support a variety of DPA methods, they have not described the detailed architecture of their hardware engine or its implementation. Instead, by assuming it as simple core logics for analysis methods [Tiwari et al. 2008; Zhou et al. 2007], they have just tried to quantify the potential of their approach. Therefore, in order to leverage the deployment of the system-level engine in real machines, it is mandatory to consider the realistic design issues as the engine being implemented for the component in an existing SoC.

For this purpose, in this paper, we propose a DPA hardware engine, called the *program analysis unit* (PAU), which has been fully implemented and integrated as a system level component in an existing computing platform. The novelty of our engine is that it is software programmable in order to attain not only the high performance but also the great expandability of our DPA solutions. For this, we have implemented PAU in the form of an application specific instruction-set processor (ASIP) whose instruction-set is customized to reflect common features of various DPA methods. First, by enabling the user to decouple DPA operations from the host code and accelerate them on PAU, we have substantially reduced the performance overhead of DPA. Furthermore, in practice, PAU can execute any designated DPA as software codes running on the processor, offering a great deal of flexibility and extensibility for a wide range of DPA functions.

To examine the effectiveness of our approach, we chose three exemplary DPA techniques for case studies: DIFT, Uninitialized Memory Checking (UMC) and Bound Checking (BC). We implemented the DPA schemes with the software code for PAU and enabled it to carry out the DPA computations off-loaded from the host CPU. Also, we built an in-house instrument tool to insert data gathering code segments for the CPU and generate actual analysis codes for PAU automatically. By mapping those codes to both processors, we have parallelized DPA computations between the CPU and PAU

thereby improving the analysis performance. In addition, our DPA engine was able to adopt the optimization techniques suggested in the software-based approaches [Zhu et al. 2009; Qin et al. 2006; Venkataramani et al. 2007; Clause et al. 2007] simply by programming them on PAU. The case studies show that our approach can be applied to various time-consuming DPA techniques by providing hardware-backed power in performance as well as software-based flexibility in analysis.

In order to show our experimental results on a working prototype SoC, we implemented our proposed design in RTL and the full system is prototyped on a Xilinx Virtex-5 FPGA board. Recent experiments have demonstrated that our proposed design can enhance the performance for the three implementation examples substantially as compared when the DPA schemes are conducted by pure software-based approaches. Furthermore, our PAU is far more energy/area efficient than general purpose commodity cores.

In this paper, we make the following contributions:

— We proposed PAU, which is a system-level hardware DPA engine that does not require the modification of the host core. We designed PAU as an ASIP to achieve both the programmability and the acceleration of hardware.
— We implemented our PAU with Verilog HDL and integrated it into a SoC prototype to build a full-system. We, then, measured the overheads of PAU in terms of performance, area and power by running mibench [Guthaus et al. 2001] benchmark to empirically show the efficacy of our approach.
— To show the programmability of PAU, we chose three well-known DPA techniques (i.e., DIFT, UMC and BC) and implemented them on our PAU.

The paper is organized as follows. Section 2 explains the background of tag-based DPA techniques and how a system-level hardware helps this DPA execution. Section 3 gives an architectural/functional overview of our ASIP, and Section 4 describes the programmable processing core of the hardware engine. After our case studies are introduced in Section 5, Section 6 will discuss how software optimizations for DIFT can be adopted into our PAU with the help of its flexibility. Then, Section 7 reports the experimental results and Section 8 relates our work with others. Finally, in Section 9, we will conclude this paper.

2. BACKGROUNDS

To enable our PAU to cover the broad class of DPA, we should look into several widely-used DPA techniques and figure out the characteristics that are commonly inherent in those. For this reason, in this section, we will first introduce the generalized DPA model which has been proposed in previous literature [Deng and Suh 2012; Chen et al. 2008], in order to understand the commonalities of DPA. Then, we will explain the execution flow of DPA with a system-level hardware engine.

2.1. Understanding Tag-based DPA Techniques

To understand the core features of various DPA techniques, it is noteworthy that previous studies [Deng and Suh 2012; Chen et al. 2006; Chen et al. 2008] have already analyzed a number of the techniques whose characteristics [Devietti et al. 2008; Newsome and Song 2005; Venkataramani et al. 2007; Clause et al. 2007; Zhou et al. 2007] are summarized in Table I. Note in the table that many techniques commonly maintain and check the meta-data information, called *tag*, to describe the status of the host program [Deng and Suh 2012] despite the differences in their types or granularities. For example, DIFT maintains a 1-bit tag to indicate whether a word/byte is from a potentially malicious sources [Newsome and Song 2005]. On the other hand, a tag may be associated with a location such as a memory address instead of a value to keep

information on the properties of storage itself, as in memory bound checking [Devietti et al. 2008]. Also, some of DPA schemes keep coarse-grained tags for composite program objects such as records and arrays [Joao et al. 2009].

With the tags, DPA generally conducts mainly two types of tag operations to achieve its purposes; tag updating and tag checking. Throughout program execution, DPA maintains the tags by updating the tags at specific events. Some tags are occasionally updated at certain events such as library calls while the others might be updated at nearly every monitored instruction. Tag checking is to test if an invariant of the program is violated. In DIFT, for instance, an alarm is triggered when any of the data from untrusted sources involve in potentially illegal activities by checking the tag. With these two types of tag operations, DPA can find bugs, profile system performance or detect various attacks on monitored programs. In this paper, we have designed our PAU based on the *tag-based DPA model*.

Table I. Tag types and operations of several DPA schemes

DPA Technique	Tag Type	Description	Tag Operations
DIFT	1-bit tag to indicate taintness	Prevents common malwares from leaking the critical information by tracking and limiting uses of untrusted I/O inputs.	tag update tag checking
Uninitialized Memory Checking	1-bit tag to present whether the memory location has been initialized	Checks whether a certain memory location is initialized before reading it.	tag update tag checking
Bound Checking	multi-bit tag to match the data and its location	Both tags for memory location and tags for pointers are set. On each memory access instruction, the tag of the pointer that is used to access memory is compared to the tag of the accessed memory location.	tag update tag checking
Reference Counter	multi-bit tag to store referenc e count for the data location	Performs reference counting to aid garbage collection mechanism. On an instruction that creates a new pointer, the tag is incremented. On an instruction that destroys an existing pointer, the tag is decremented.	tag update tag checking
LockSET	1) multi-bit tag for the current set of locks held by the thread 2) multi-bit tag to maintain a candidate set of locks	Performs race detection among multithreaded applications.	tag update tag checking

2.2. DPA Execution on a System-Level Hardware Engine

This subsection describes the DPA execution flow in the system-level hardware approach, where the system mainly consists of a host CPU and an off-core hardware engine, as depicted in Figure 1. In the system, the host is regarded as a producer that gathers the data required for analysis, called the *execution traces*, then sends them to the analysis hardware engine. Conversely, the off-core engine is regarded as a consumer that receives the traces and performs the actual analysis task. For example, in case of Memcheck [Seward and Nethercote 2005], the host captures memory access behaviors of the monitored program, such as accessed memory addresses and a set of data written to the memory. Then, the captured information is transferred to the analysis engine which performs the analysis task that tracks dataflow and detects unintended memory uses in the program by updating and checking the tags. As presented in much literature [Deng and Suh 2012; Kannan et al. 2009; Deng et al. 2010; Chen et al. 2008; Tiwari et al. 2009], these approaches with separate engines have shown to be efficient because it relieves the burden of the host CPU by reducing the competition for resources (i.e., cycles, registers and caches) between the original program and DPA.

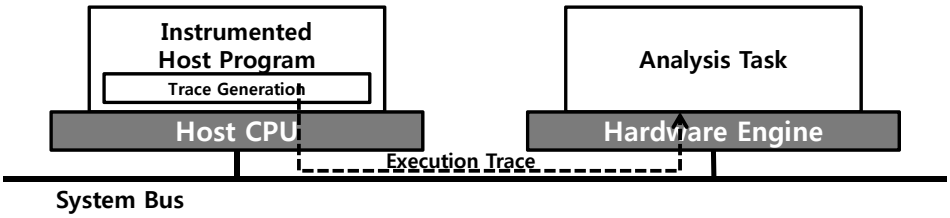


Fig. 1. Execution model of system level hardware engine

As explained, in this execution model, the execution traces should be created and then transferred to the analysis hardware. In the core-level approaches [Kannan et al. 2009; Chen et al. 2006; Chen et al. 2008; Deng and Suh 2012], the traces are transparently gathered with a dedicated hardware that observes the instructions executed by the monitored program and creates the corresponding execution trace. On the other hand, in the system-level approach [Tiwari et al. 2009], the host program is augmented with the code for trace generation so that a stream of traces is created by the code on the host. Meanwhile, the analysis task on the hardware engine can be implemented in the form of either hardware or software. In Hardgrind, the analysis tasks of MemCheck [Seward and Nethercote 2005] and Helgrind [Savage et al. 1997] are implemented with specialized hardware modules like Range Cache [Tiwari et al. 2008]. With the help of ASIC-style design, Hardgrind could achieve the speedups from 29% to 440% in the two DPA techniques.

3. SYSTEM-LEVEL PROGRAMMABLE DPA ENGINE FOR EXTENDIBILITY

In this section, we will give an architectural overview including our hardware engine and discuss how DPA is performed on the proposed system. Also, we will discuss the efficient communication strategy between the host and our engine.

3.1. Overall System Design with PAU

Based on the execution model of the system-level approach introduced in the previous section, we designed our overall system which mainly consists of a host CPU and PAU as depicted in Figure 2 where PAU is connected via a general system bus to the host CPU along with other modules including special purpose processors.In this work, our SoC employs an AMBA-compliant system bus [Limited 1999] which is a shared bus architecture conforming to the AMBA protocol, a de-facto standard for master-slave communication in modern SoC design. Hence, as long as our design obeys the AMBA protocol, it can be used in every SoC based on AMBA protocol without any hardware change.

The key components of PAU are the *tag processing core* (TPC) and the *main controller*. TPC is a processor core of PAU which executes software codes. Its main task is to perform tag operations along the program execution flow running on the host and analyze the monitored program. We will postpone the discussion of this task to Section 4. The main controller manages all transactions related to the DPA computation. It contains *configuration registers* whose values can be changed to specify various types of transactions. Thus, the host can control the action of PAU directly by setting these registers to certain values. To facilitate this control from the host, the configuration registers are memory mapped.

The central role of PAU is the management of all the tags used for DPA. During DPA computation, all the tags being accessed are located in either PAU or the main memory. For every host processor register, TPC has a corresponding 32-bit tag, all of

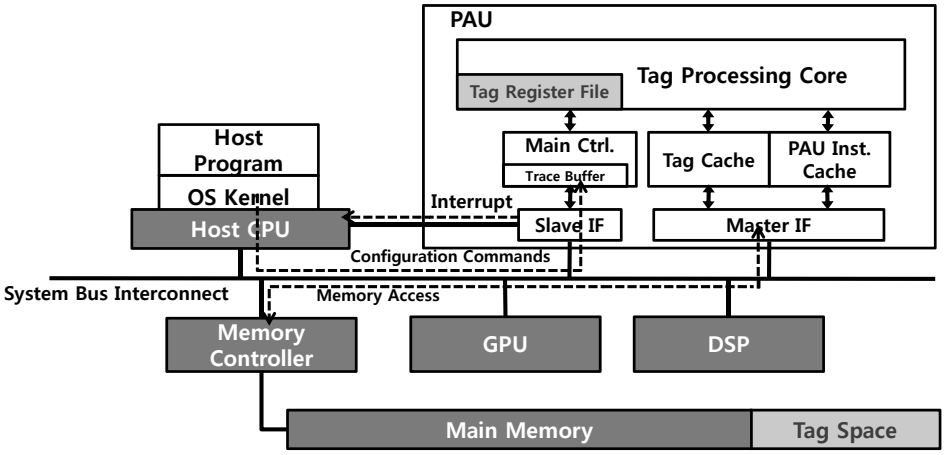


Fig. 2. The overall system design with PAU

which together form a single register file in TPC, called the *tag register file* (TRF). Since our host processor has 32 general registers, the TRF also consists of 32 entries. Since many DPA schemes augment tags to the registers, it is efficient to employ the TRF to support the tag-based DPA. Likewise, we allocate a space in the main memory, called *tag space* [Venkataramani et al. 2008], to manage various types of tags in the memory. This space is maintained by TPC throughout program execution to support various types of tags which cannot be allocated in the TRF. Although such structure of memory tags might be a good way to support diverse tag types using the existing memory architecture, it should be too slow if tags are frequently accessed from the tag space in the main memory. Therefore, to reduce the access latency, our PAU has an internal SRAM, called *tag cache* [Kannan et al. 2009; Deng and Suh 2012], for caching frequently referenced tags from the memory. In consequence, we would like to emphasize that our design for tag management with TRF and tag cache intends to empower PAU supporting fast tag lookups.

Since our system is implemented as a SoC, we have integrated our PAU to the multiprocessor SoC platform, strictly following a platform-based design methodology. There are two design criteria that we have endeavored to satisfy when following the methodology for the development of our SoC hardware. First, we have tried to reuse as many existing modules as possible. They include various commodity IP cores, DDR memory and shared interconnects through which every module in the system is attached. Second, we have forced newly added hardware modules to comply with all the specifications required by our target SoC platform. For instance, ARM regulates that any IP module added to their platform obey the AMBA protocol. Therefore, in our implementation based on the AMBA platform, the interface to our PAU conforms completely to the AMBA protocol so that it can be connected to the host processor via the AMBA bus. In this sense, our solution differs from previous core-level approaches where their acceleration modules are added and connected to processors via custom lines or interconnects [Suh et al. 2004; Dalton et al. 2007; Kannan et al. 2009; Nagarajan et al. 2008; Chen et al. 2008]. Also, all special logics customized for DPA are fully integrated into our PAU. This confirms our assertion that most hardware modules except the newly added PAU have been reused for our SoC implementation.

3.2. Execution Trace Communication

Now, we will discuss how the execution traces are transferred from the host to PAU through the system bus. To buffer the difference between the times for handling the assigned tasks on the two processors, we have implemented a dedicated queue, called the *trace buffer*, in PAU. In fact, other solutions based on separate processing units usually also need queues [Chen et al. 2008; Nagarajan et al. 2008; Kannan et al. 2009; Deng and Suh 2012] for similar purposes. However, others are rather core-level approaches thus demanding a change to the structure of their host CPU core or internal caches to some degree. On the contrary, by placing the buffer outside the host CPU and connecting it via a system bus, we preserve the original processor core architecture intact.

Figure 3 presents an example of the instrumented host code for a DPA method which analyzes the accessed memory addresses. It also displays the overall flow of trace transactions via the trace buffer between the host code and TPC in PAU. In the example, note that two additional instructions, being marked with boldface, have been inserted to the original code after instrumentation. They are added to generate two execution traces for memory addresses used by load/store instructions (traces #1 and #2). Suppose that the code is running on the host and reaches the code segment. Since *ld* instruction (1) is executed, the corresponding memory address stored in register %i0 should be gathered for DPA. As can be seen in the example, register %g4 is memory-mapped to the physical address of the trace buffer so that it provides a direct way to store the trace in %i0 using *st* instruction (2). In a similar manner, a trace for memory address used by instruction (6) is also pushed into the trace buffer with the instruction (7). Finally, the stored traces are consumed by TPC for actual analysis task.

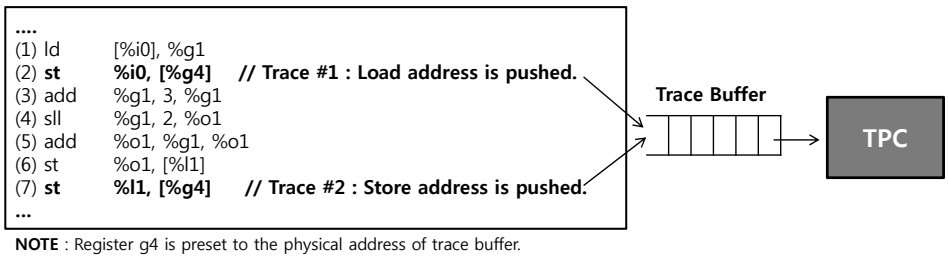


Fig. 3. Execution trace communication

3.3. Synchronization and Multi-threading Support

In general, the approaches with separate hardware engine for DPA including ours should be able to handle the synchronization between the host CPU and the hardware engine. In our approach, the trace buffer is used to minimize the overhead of synchronizing the data transactions among these processing units by buffering the traces generated from the host, which helps the host continue its execution without being halted for the synchronization with PAU. However, such a basic synchronization mechanism based on the trace buffer may create a potential loophole in security for some DPA techniques that are to detect malicious attacks. For instance in DIFT, even if the host has just generated an important trace that is linked to a malicious activity, PAU may not recognize the activity until the trace is extracted from the buffer for the analysis in PAU. If the buffer is already filled with many preceding traces, the adversary may succeed in the attack long before PAU reaches the trace. To eradicate this loophole,

it would be necessarily required that the host CPU and PAU should be synchronized at every instruction [Venkataramani et al. 2008]. However, as discussed in [Kannan et al. 2009; Garfinkel et al. 2004; Rajagopalan et al. 2006], such fine-grained synchronization may cause tremendous performance degradation in most cases. Thus in our implementation, the two computing units are synchronized at a more coarser granularity (i.e., at every system call), following the strategies of previous approaches [Kannan et al. 2009]. The rationale for this decision is due to the fact that many compromised applications usually exploit system calls. For example, when an attacker wants to leak some sensitive data outside, the system call to open the network should be invoked. In this case, the data leak can be prohibited by checking the tag of data before sending the information through the network when DIFT keeps track of data flow during the runtime. Thus, we also utilize the system calls as an optimal granularity for synchronization in our architecture, to detect most malicious behaviors [Kannan et al. 2009], and yet to substantially lower the performance overhead.

In our prototype implementation, every time a system call is invoked on the host, the OS kernel informs PAU of the event by sending a configuration command to PAU, and stops the execution of the monitored program. For synchronization on each system call, the host sets the *sync_syscall* register in the main controller. Once it is set, PAU consumes all traces left in the trace buffer and then reports its status to the host by sending an interrupt signal. Then, the host resumes its task after clearing the sync_syscall register.

Another important synchronization point we should consider is the context switch between applications. On the host CPU, many applications with different contexts are concurrently loaded. Thus, PAU should be notified of which process or thread is currently executed on the host CPU. In our work, these critical events are also announced to PAU by the OS kernel. Every time the OS scheduler is activated and a context switch occurs, the host notifies PAU this event by writing the current process ID and the thread ID to the *current_PID* and the *current_TID* registers in the main controller, respectively. By doing so, PAU can identify the current process and thread ID on the host.

4. TAG PROCESSING CORE

In this section, we will explain the detailed design of TPU, the key component of our PAU, which is a processor core that enables us to write the software code for the DPA task in our approach. We will first describe the ISA of TPC whose mission is to support a wide range of tag-based DPA techniques. Then, the microarchitecture of TPC will be discussed.

4.1. TPC Instruction-Set Architecture

Basically, the TPC ISA is extended from a simple RISC ISA so that the general structure of software can be constructed with the ISA. Then, several types of instructions are added to the ISA, which perform the specialized analysis operations that are commonly inherent in the tag-based DPAs listed earlier. We will explain the data types handled by TPC and the types of instructions.

4.1.1. Data Types. In the TPC ISA, three types of data are supported to construct analysis task software; (1) tag, (2) general and (3) execution trace. Many DPA techniques typically associate a tag (that is meta-data) with each piece of state in the monitored program [Deng and Suh 2012]. Thus, it is very natural to support the tag data type in our TPC design. Many details for the tag type were, in fact, discussed in Section 2. With the TRF and tag space in memory, TPC carries out a variety of tag operations to update and check the tags.

The general data type is necessary to construct a program structure for supporting tag operations. For example, to organize a loop structure that iterates the execution of a code segment for processing tags in an analysis code, there needs a set of data to control loop iterations such as loop indices and temporary variables. To express this type of supportive operations (e.g., loop iteration control) in the DPA algorithms, we provide the general data type for our TCU ISA. In the analysis code, the operands of the general type are distinct from those of the other two types in a sense that they do not contain any analysis specific information like tags or execution traces. Therefore, they are stored in a separate register file, called the *general register file* (GRF). During code execution, they must be loaded from memory to the GRF before being processed.

Lastly, the execution trace type is for the traces delivered from the host program. As stated earlier, the execution traces which contain the runtime information of the host are delivered to the analysis hardware engines such as our PAU. During the execution, TPC in PAU consumes the traces in order to follow the program execution flow and receive runtime traces which are not determined at instrumentation time. We assign the traces a different data type in order to distinguish the operations on them in the code from those on the other types of data (i.e., general and tag). For this reason, a trace in the trace buffer in PAU is regarded as a data of the execution trace type, which is accessed by a specific set of instructions. For example, in DIFT, the results of branches and the memory addresses accessed by load/store instructions are delivered to the trace buffer as the execution traces. In TPC, the traces are regarded as the data of execution trace type and processed by the special instructions to recognize the behaviors of the host. The further details will be given in the next subsection.

4.1.2. Instruction Types. To support the analysis tasks for DPA, we have designed four types of instructions in our TPC ISA; (1) general, (2) tag ALU, (3) tag load/store and (4) trace handling. The first set of instructions corresponds to those in a RISC-style instruction set to organize general program structure. In fact, it is directly matched to the general data type and gives our PAU the general programmability to construct analysis task software. The general group includes general ALU operations, load/store, data movement and branches, as shown in Table II. They usually make use of GRF as the operands for computation and access memory space to load/store data. In addition to them, there are several instructions newly added for data movement between GRF and TRF. For example, *mov.tg* instruction moves the data in TRF to GRF. Then, the tag value can be manipulated by general instructions for the purpose of analysis and written back to TRF with *mov.gt* instruction. These move instructions widen the way to deal with the tags so that the degree of programmability in TPC ISA can also be improved.

Many DPA schemes need to operate on the tags associated with processor registers. Thus, for these types of operations, TPC ISA includes tag ALU instructions that perform the operations among the register tags, as shown in Table II. These instructions are functionally similar to the ALU instructions for the ordinary data except that their operands come from TRF. This type of instructions might be most frequently used in analysis software because tag updating and checking are the kernel parts of most DPAs.

On the other hand, in case of the tags located in tag space, they should be loaded to the registers for computation. To access the tag space, two types of tag load/store instructions are supported in TPC ISA. As given in Table II, the GRF and TRF load/store instructions take operands from GRF and TRF, respectively. In most cases, analysis software performs TRF loads/stores to propagate tags between registers and memory locations. However, as in reference counting, DPA should update their tags located in the tag space without interacting with TRF. In these cases, a GRF load or store is

Table II. Overview of TPC instruction-set

Instruction Type	Sub-Type	Instructions	Example	Action
General	ALU/Data Movement	add,sub,mov, …	add R2,R3,R1	R1 = R2 + R3
	Load/Store	load	load [R2],#4,R1	R1 = Mem[R2+4]
		store	store R2,[R1],#4	Mem[R1+4] = R2
	Branch	beq,bneq,jump, …	beq #imm	PC = PC+#imm
	Data Movement to TRF	mov.gt	mov.gt R1,T1	T1 = R1
	Data Movement to GRF	mov.tg	mov.tg T2, R2	R2 = T2
Tag ALU	Register-Register	add.t,sub.t,and.t,xor.t, …	xor T2,T3,T1	T1 = T2 xor T3
	Register-Immediate	addi.t,subi.t,andi.t,xori.t, ...	addi.t T1,#1,T1	T1 = T1 + 1
	Tag Check	cmp.g	cmp.g T1,R1	compare T1 with R1
		cmp.t	cmp.t T1,T2	compare T1 with T2
Tag Load/Store	GRF Load/Store	load.g, store.g	load.g [R2],R1	Mem[TLB(R2)] = R1
	TRF Load/Store	load.t, store.t	load.t [T2],T1	Mem[TLB(T2)] = T1
Trace Handling	Trace Movement	mov.bg	mov.bg trace,R1	R1 = trace
		mov.bt	mov.bt trace,T1	T1 = trace
	Trace Compound ALU	add.tc, sub.tc, or.tc, …	add.tc trace,T2, T1	T1 = T2 + trace
	Trace Compound ALU/Load	add.tcl, sub.tcl, or.tcl, ..	or.tcl T2, [trace],T1	T1 = T2 | Mem[trace]
	Trace Compound ALU/Store	add.tcs, sub.tcs, or.tcs, …	or.tcs T1,T2,[trace]	Mem[trace] = T1 | T2

useful because it does not pollute the status of TRF which contains the meta-data for processor registers. It is noteworthy that the tag load/store instructions are different from the ordinary load/store ones in that they use the *tag TLB*. In order to efficiently manage tags in memory, PAU employs the tag TLB proposed in Harmoni [Deng and Suh 2012] that translates a data address to a tag address. By utilizing the specialized logic, the tag load/store instructions can be performed with the low address translation overhead.

Lastly, the execution traces from the host should be handled in PAU to follow the program execution flow. For this purpose, the trace handling instructions are provided. Recall that the traces are located in the trace buffer and accessed sequentially in order. In the trace handling instructions, the buffer is regarded as a register. To move a trace from the buffer to GRF/TRF, PAU supports two types of move instructions; *mov.bg* and *mov.bt*. The move instructions can be used when the trace should be further manipulated or interpreted to extract the required information. Also, there are a set of instructions, called the *compound instructions*, which take a trace as an operand. They substantially reduce the number of instructions to be executed when a trace from the buffer is used as an operand in tag updating. For example, in DIFT, memory addresses of store instructions are transferred to PAU as execution traces. In this case, the trace would be used as a destination operand in tag computations. If we would not have the compound instructions, the computations should require five TPC instructions as follows.

— instruction executed by the host : st [%g1], %g2
— execution trace : address value in register %g1
— tag propagation rule : Tag[Mem_addr[%g1]] = Tag[%g1] | Tag[%g2]
— DIFT analysis code in PAU :
  (1) mov.bg R7 : trace movement to GRF (address in %g1)
  (2) mov.tg T1, R1 : tag movement from the TRF to GRF
  (3) mov.tg T2, R2 : tag movement from the TRF to GRF
  (4) or R3, R1, R2 : tag propagation to a temporary register
  (5) store.g [R7], R3 : update memory tags

A compound instruction, *or.tcs*, can substitute for the set of instructions. When it is executed, the address in the trace is used as a store address for tag space, and the tags in the two registers (T1 and T2) are propagated to the memory tag, while it increases

the analysis performance. Because many analysis schemes tend to directly associate the traces to their tags during execution, the instruction set is a good way to support them.

4.2. TPC Microarchitecture

In this subsection, we will show the microarchitecture of TPC for the ISA design described in the previous subsection. Figure 4 represents the internal block diagram of TPC. In order for TPC to launch tag computation, two preliminary conditions must be met. First, the analysis code associated with the instrumented host code has been created and located in the main memory. Second, the host CPU has begun the host code execution and deposited execution traces into the trace buffer, as demonstrated in Figure 3. As soon as a trace arrives, a notification signal is sent to TPC. Upon receiving the signal, TPC reads trace entries one-by-one from the trace buffer.

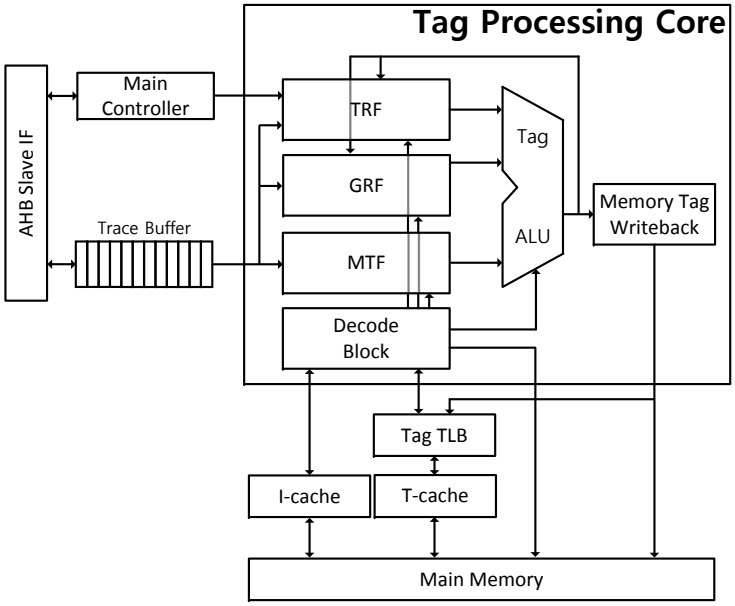


Fig. 4. TPC microarchitecture

In order to function as a programmable processor, TPC has many general components of RISC architecture with a three-stage pipeline: (1) fetch-decode, (2) execution, and (3) write-back. At the first stage, TPC code is loaded from the memory and decoded by the *decode block*. Since main memory is normally implemented with external DRAM devices, off-chip memory access latency can be a serious performance bottleneck. To alleviate this problem, we have implemented the *instruction cache* between TPC and main memory.

At the next stage, TPC fetches operands from the two register files, and accesses the tag space with the tag TLB and the tag cache. At the same time, TPC schedules the *memory tag fetcher* (MTF) unit to fetch the tags in memory according to the memory addresses in the trace buffer, in order to support the trace compound instructions. An operand of the general type is also loaded from the main memory at this stage. In our current prototype, there is no dedicated cache for the data of general type mainly

because the operations on general data are relatively few as being compared to the other types of operations. However, if a developer wants to cache a set of general data, it is also possible to map them to the tag space so that they can be cached in the tag cache. After all operands are ready, they are then forwarded to the tag ALU that takes these tags as the operands to conduct tag computations or other general ones.

At the last stage, TPC updates the result back to either the register files or the memory space, depending on the executed TPC instruction. Just in case the tag cache is updated with new data, we have implemented a write-through cache scheme in order to keep the consistency of data stored both in this tag cache and the tag space inside main memory. Once TPC has completed the execution of all instructions fetched and there is no trace from the host, it will be idle waiting for new traces filled into the buffer by the host. If not, it reiterates the normal execution procedure as described so far.

5. CASE STUDIES

The programmability of the TPC ISA offers developers a good capability of implementing a variety of their DPA schemes with flexibility in software on our PAU. In this section, as case studies, we will discuss how several well-known DPA techniques can be realized on our prototype as software codes that are composed of the TPC instructions. As examples, we picked three techniques; DIFT [Newsome and Song 2005], uninitialized memory checking (UMC) [Venkataramani et al. 2007] and bound checking (BC) [Clause et al. 2007]. After briefly introducing the idea of each DPA scheme, we will discuss our DPA implementation on PAU.

5.1. Case Study 1 : DIFT for Data Leak Prevention

5.1.1. Background of DIFT. To protect the confidential data inside computing devices, an approach called data leak prevention (DLP) has been proposed. In the approach, security policies defining critical information and the corresponding actions (that is, deny/permit) on the specific output channels are forced to prevent any critical information from flowing into the outside of devices. A common way to realize DLP has been to use DIFT [Yin et al. 2007; Enck et al. 2010], one of the widely used DPA techniques. This analysis scheme sets up rules to tag (or taint) internal data of interest and keeps track of the taintness of their tags throughout the system [Kannan et al. 2009]. At run time, every data derived from the one with tainted tag has its tag tainted. An alarm will be triggered as soon as any of the tainted data involves in potentially illegal activities, such as pointing inside the code or being included in a data stream on the output channels [Enck et al. 2010].

When DIFT is employed for DLP, the first step is to tag or taint as sensitive the input data from sensitive sources like confidential files. Then, through code execution, the data tags are then also propagated by tagging as sensitive the data derived from those with tainted tags, following the tag propagation rule of DIFT. If the code makes an unauthorized attempt to leak any of tainted data [Qin et al. 2006], a security exception will be raised to announce the existence of data leak.

5.1.2. Tag Initialization and Check Procedures. As introduced, DIFT uses the tags to indicate the taintness of the data. To support the tag-based analysis in this case study, we assign a 1-bit tag for every host processor register and store it into our TRF in TPC. Each 1-bit assigned to a register in TRF represents whether or not the corresponding processor register currently holds sensitive data. Likewise, one bit is assigned for each word in memory, and these bits are all arranged in the tag space, in a similar way to that suggested in [Venkataramani et al. 2008].

In general, we can divide the tasks of DIFT for DLP into the three stages: tag initialization, propagation and check. In this study, the tag initialization and check stages are executed by the OS kernel in the host processor. Depending on whether data originates from a sensitive source, the kernel initializes its tag by setting the bit on or off. Just before data is transferred to an output channel, the kernel checks its tag to decide if the data transfer is safe. We will discuss the tag propagation stage later in order to first focus our discussion on these two stages which require a close interaction between the kernel and PAU.

On a computing device, sensitive sources include GPS, files with confidential contents and SIM cards with private information. In specific, in this study, we focus on the confidential files on the system as our sensitive sources and the kernel maintain the list of them. To monitor every access of an application to any file in the system, we modified `open` system calls in our Linux prototype system. When one of applications opens a file by invoking the system call, the kernel determines if the file is in the list. If so, the file pointer will be tainted by setting its bit on. For this tag initialization, the kernel function *tag_init* is invoked. In our system on the host processor, the function is implemented as a device driver interacting with our PAU. Its task is reporting to PAU the location (i.e., register number or memory address) of the data that must be tainted. Depending on its type, the location is written to either *source_taint_reg* or *source_taint_addr*, both of which can be configured via changing the values of memory-mapped configuration registers in the main controller of PAU. Then PAU responds the report from the kernel by tainting the tag for the location.

For the tag check stage, we also have embedded a new function *tag_checking* into the system calls involved in network packet generation. When data is about to be transferred outside as a network packet through an output channel, this kernel function checks the data tag with the assistance of PAU. As the first step of this check, the function writes the data location into either *sink_taint_reg* or *sink_taint_address* in the configuration registers, similarly to the tag initialization stage. Then it sends to PAU the inquiry of the current tag value at this location. Upon receiving the inquiry, PAU retrieves the value from either TRF or tag cache, and interrupts the host to notify the result back to the kernel. Now the kernel knows whether or not the data of interest is from sensitive sources.

5.1.3. Tag Propagation. As explained in Section 2, in our approach, the time-consuming part of DPA is delegated to PAU to relieve the burden of the host processor and it corresponds to the tag propagation computations in DIFT for DLP. To carry out the propagation task on our PAU, TPC code includes propagations rules and operands extracted from the original program run on the host processor. When our in-house instrument tool generates the code, most required information like register operands can be statically extracted and embedded in the generated code. But some dynamic information that can only be resolved during code execution is still missing in the generated code. In our DIFT implementation, such information includes (1) an execution path of the original program and (2) memory addresses of load/store instruction. Currently, this missing information is supplemented and sent as execution traces at run time by the host processor to PAU, hence helping PAU have the enough information to track tag propagation at any circumstance.

Figure 5 shows a segment of the original program in (a), its associated pseudo propagation code in (b) and the realized TPC code in (c). As can be seen from (b), a pseudo instruction is comprised of a propagation rule and operands. They are to specify the semantics of tag propagation by the matching instruction in (a). For instance, the third DIFT instruction in (b), which is parallel to the *sll* instruction in (a), has "%o1 and %g1" as operands and "copy from the right tag to the left" as a rule. When the original

instruction is executed, so does the pseudo instruction and thus if the tag of %g1 is tainted, %o1's tag will also be tainted because of the 'copy' tag propagation rule.

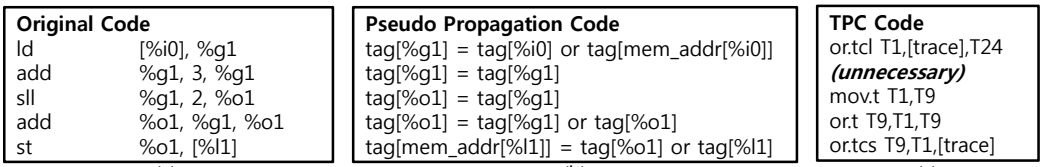


Fig. 5. A TPC code example for DIFT computation

In the analysis task of PAU, the propagation rules are expressed by the TPC instructions as given in Figure 5 (c). For two load/store instructions in the host code, the propagations are processed by the compound instructions since they make use of the trace in the trace buffer to figure out the location of tags. As seen in this example, our compound instruction can reduce the number of instructions required for the trace handling. For the other ALU operations in the host, the tag ALU instructions are used to propagate the tags between the tag registers. However, for the second instruction *add*, the corresponding TPC instruction is omitted because it does not change the tag status of PAU. In our software implementation, we have made efforts to remove these unnecessary propagation operations, being empowered by the programmability of TPC. The TPC codes are generated by our in-house instrument tool and allocated to TPC code memory region before the execution.

During the host program execution, PAU expects execution traces from the host processor to compensate for the missing dynamic information that is indispensable for correct operation. The load/store addresses can be easily gathered into a trace since they are readily computable from the host program at run time. As for the execution path, we may express it with a set of basic blocks and edges that connect them. Thus in our implementation, we assign every basic block a *unique identification (ID)* number, and during code execution, let the host processor deliver the ID of a block to PAU so as to pinpoint the exact block that the host execution path currently comes to.

As seen in Figure 1, the original program installed on the host has to be instrumented to enable communication with PAU for orchestrating analysis operations in our solution. Figure 6 presents an example of the instrumented host code generated from the original one in Figure 5. It also displays the overall flow of trace transactions for DIFT via the trace buffer between the host code and TPC. In the example, note that four additional instructions, being marked with boldface, have been inserted to the original code after instrumentation. They are added to generate three traces, one for the current basic block information (trace #0), and two for memory addresses used by load/store instructions in the same block (traces #1 and #2). Suppose that the code is running on the host and the execution path comes to this block **LL5**. Then, *mov* instruction (1) is first executed to initialize register %g3 with the basic block ID. As can be seen in the example, register %g4 is memory-mapped to the physical address of the trace buffer, thereby providing a direct way to store the trace in %g3 using *st* instruction (2). In a similar manner, traces for memory addresses in the basic block are also pushed into the trace buffer with the instructions (4) and (9). The stored traces are consumed by TPC for tag propagation.

In order to conduct the analysis task for DIFT with the basic block ID, we decomposed the TPC code into multiple regions, each containing a single basic block of TPC instructions, as Figure 7 depicts. Each region includes a header for its basic block.

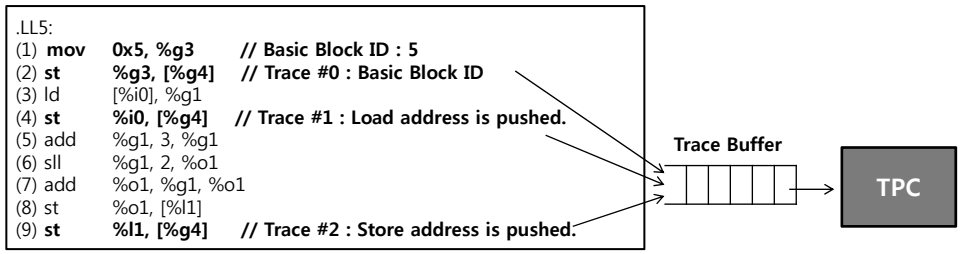


Fig. 6. Execution trace communication for DIFT

The basic block header holds the useful information for PAU (e.g., the number of TPC instructions and the number of load/store). At the beginning of the TPC code region, there is a lookup table, called the *basic block jump table*, which is used to access every basic block in the TPC code. During execution, when TPC receives a trace indicating a basic block ID, it accesses the basic block jump table. Then, it jumps to the address and finds the TPC instructions within this basic block. After finishing the block, TPC will find another trace in the buffer and continues the analysis if the buffer has one.

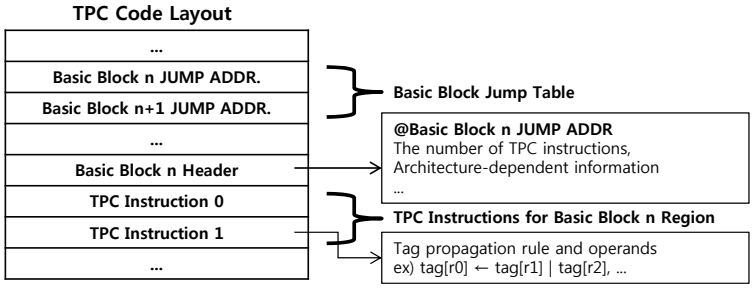


Fig. 7. TPC code layout

5.2. Case Study 2 : Uninitialized Memory Checking

5.2.1. Background. As the second implementation example of our case study, we chose UMC which was firstly proposed in [Venkataramani et al. 2007]. The objective of this DPA technique is to detect a read access to the memory region where initialization is not performed yet. Since the read event to an uninitialized location causes a memory error which is often exploited by attackers as a security hole, it is vital to detect and remove such cases in program execution for security purposes [Venkataramani et al. 2007].

In Figure 8, we depict the state transition diagram of the UMC scheme to explain the principle of the DPA. As explained in Table I, UMC augments an 1-bit tag for every word in memory to indicate whether the corresponding location is initialized or not. When the system is reset, every memory location is assumed to be uninitialized. If a value is stored to a certain memory location, the state of the location is transited from *Uninitialized* to *Initialized* as shown in Figure 8. Once a memory location enters the *Initialized* state, both load and store operations from/to the location are permitted . However, any load access to a location with *Uninitialized* will be called an error. Now

let us explain how this memory checking model can be mapped to the tag-based DPA using PAU.

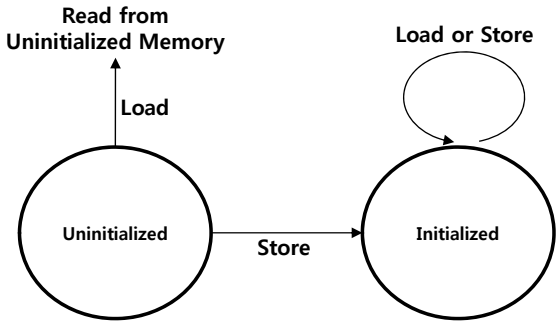


Fig. 8. State transition diagram for UMC

5.2.2. UMC Implementation. To carry out UMC on PAU, we assign a 1-bit tag for every word in the application memory region and store the set of tags into the tag space in the main memory. As explained above, the tasks of UMC are mainly divided into two parts; tag initialization and check. In our UMC implementation, both the operations are composed of TPC instructions with the execution traces delivered from the host as input. For every memory write on the host, the write address is transferred to TPC from the host. Then, the tag corresponding to the address is set to '1' in order to indicate that the location is initialized. This is the tag initialization process. Likewise, when the host reads a memory location, the address is also delivered to TPC. At this moment, TPC checks the value of the corresponding tag and if it is not '1' (i.e., *Uninitialized*), an exception is raised to inform the host of an unallowable memory access.

Figure 9 illustrates a segment of the original example program in (a), its associated pseudo UMC code in (b) and the implemented TPC code in (c). When the original code is executed on the host, TPC code also runs in parallel to check whether the memory access rule enforced by UMC is violated or not.

For each load instruction in the host code, the tag check is performed by three TPC instructions. At first, a compound instruction, "mov.tcl" is used to read a trace in the trace buffer which contains the accessed address and load the memory tag corresponding to it. Then, a comparison between the memory tag and the register R1 is performed by the "cmp.g" instruction. To mark the *Initialized* state, the register R1 is set to '1'. Thus, if the comparison between the tag and R1 produces the "not equal (ne)" condition, it implies that the load instruction attempts to read an uninitialized memory location. For these cases, according to the rule of UMC, TPC jumps to an exception routine to trigger an alarm by sending an interrupt to the host. In this example, the label for the routine is named as "trigger_alarm". On the other hand, for each store instruction in the host code, the tag initialization is performed by the mov.tcs instruction. In our example, the register T2 is also preset to '1' to indicate the *Initialized* state. By writing the value (i.e., 1) to the memory tag of the delivered address, TPC carries out the tag initialization process.

5.3. Case Study 3 : Bound Checking

5.3.1. Background. As the last implementation example, we chose BC, which is a DPA technique proposed in [Clause et al. 2007]. The objective of this scheme is to check whether or not each memory operation with a pointer accesses the location within the

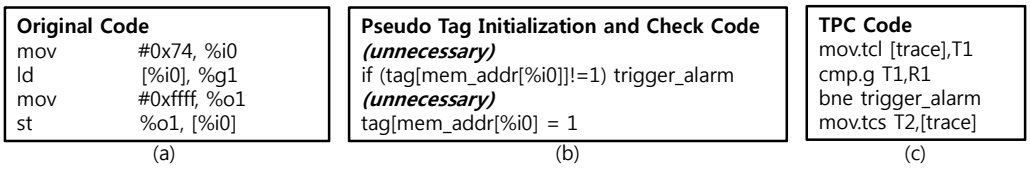


Fig. 9. A TPC code example for UMC computation

legitimate range allocated by the program for the pointer. If it ever makes an out-of-bound access, BC reports the access as a memory error since it can be exploited as a security vulnerability.

The tasks of BC can be divided into three stages; tag initialization, propagation and check. To perform the procedure, BC augments the tags for both pointers and corresponding memory locations [Deng and Suh 2012]. Whenever a memory region is allocated at runtime, BC initializes the tags for both the memory locations and the pointer to the starting address. In a high-level language like C/C++, special functions are provided for memory allocation, such as *malloc*. They usually take the size of the requested memory as input and return the starting address of the allocated region. Every time the functions perform their task, BC identifies the range of the allocated memory in the form of "[p, p+*size*)", where p is the starting address returned and *size* is the size of the allocated region [Clause et al. 2007]. Then, BC assigns the same tag value to both the tags of the allocated memory locations and the tag of the register which contains the returned pointer.

During the execution, the pointer tag is propagated to the other storage location in accordance with the propagation rule of BC [Clause et al. 2007]. On each memory instruction such as load or store, the register tag for the pointer is compared with the tag of the accessed memory region. Obviously, the two tags must be identical for in-bound accesses but different for out-of-bound accesses [Clause et al. 2007]. Therefore, only when the tags are identical, the memory access will be granted. Otherwise, BC reports the memory error.

5.3.2. BC Implementation. To implement BC in this study, we assign 4-bit tags for memory locations and pointers as in [Deng and Suh 2012]. In Figure 10, we depict the host code in assembly level in (a), the pseudo tag initialization/check code in (b), and the pseudo tag propagation code in (c). In (a), a memory-allocation function, `malloc`, is invoked at line 6. The parameter of the function is the size of the requested memory and set at line 5 (in register %o0). After the `malloc` allocates the memory region, the starting address for the region is written to the register %o0 according to the calling convention. At this time, the host transfers two traces to the TPC; the size of the region and the value of the pointer. Then, TPC performs the tag initialization which assigns the same tag values to the tags for the allocated memory region and the tag for the corresponding register %o0 as shown in (b).

After that, during execution, the pointer tag is propagated to other registers or memory locations as shown in (c). The tag propagation rule of BC is almost the same to that of DIFT. Then, when the host attempts to access the allocated memory at line 17, with the delivered trace which contains the accessed address, TPC checks whether or not the tag of the pointer (the tag for %g1) is matched to the tag of the accessed memory location. If the both tags do not match, an exception is raised and the interrupt to the host is triggered.

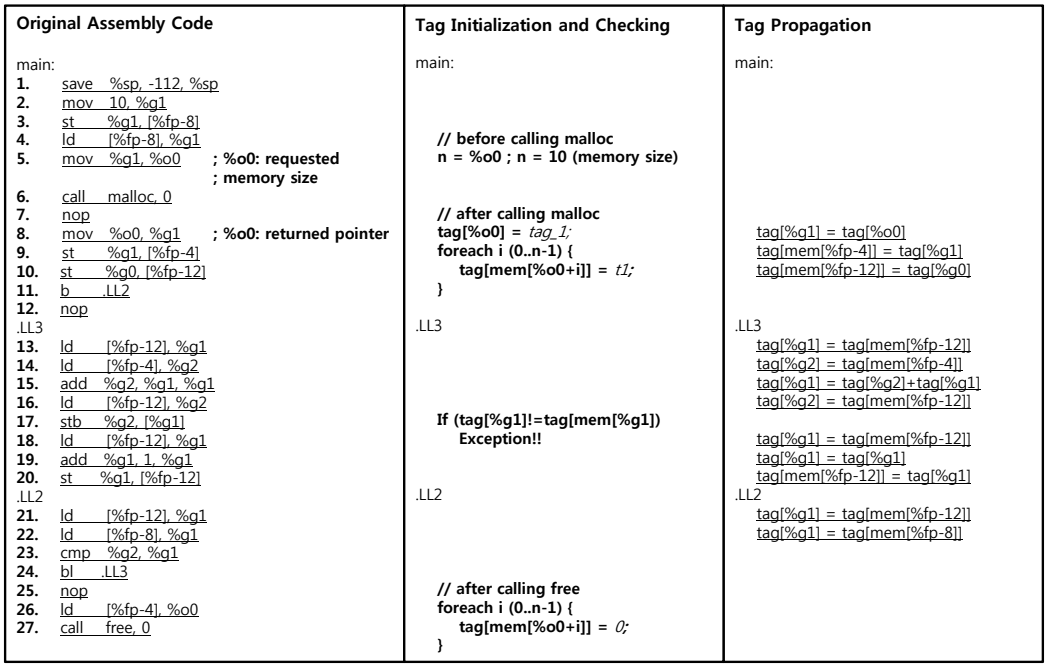


Fig. 10. A pseudo code example for BC

Finally, when the host executes the deallocation-function `free` at line 27, the tags associated with the deallocated memory area are cleared. As discussed in [Clause et al. 2007], the pointers that were tainted for the deallocated region might not be cleared. This is because, with the uncleared tag, BC can detect the memory accesses to the deallocated region by checking if the both tags have the same value. For our BC implementation, we do not describe the detailed TPC instructions that correspond to the pseudo code in Figure 10 because the most parts of the implementation are the same to our other DPA examples. For the tag initialization and the tag check procedures, our BC implementation is almost the same to the UMC implementation. On the other hand, for the tag propagation process, the most TPC codes used for DIFT were re-used for BC.

6. IMPLEMENTING OPTIMIZATIONS FOR DIFT WITH TPC

In our case studies, we have clarified how different DPA techniques can be realized on PAU simply by programming the algorithms. In this section, we will present another practical example where the programmability of TPC can be well exploited. Discussing the DIFT implementation in Section 5, we explained the basic instruction-level tag propagation for DIFT. However, since the instruction-level tracking incurs too much overhead, several previous studies on DIFT have centered their efforts on the overhead reduction by adaptively choosing coarser granularities (i.e., basic blocks or functions) [Zhu et al. 2009; Qin et al. 2006]. In this section, we will show that these optimizations can be adopted into our DIFT implementation with the programmability of TPC. By doing so, once an application is chosen to be monitored, DIFT algorithm running on TPC can find optimal code granularities of tracking operations for each different part of the application. In the followings, we will discuss how code analysis

information is applied to help our DIFT implementation to adaptively choose optimal granularities for tag propagation within individual basic blocks or functions such that data leaks can be prevented while computation overheads are minimized. We will also describe how our PAU supports multi-level tag propagation efficiently in hardware.

In an attempt to choose optimal granularities for tag propagation within an application, we first divide application code into functions of three categories as follows, referring to the prior researches on DIFT [Qin et al. 2006; Zhu et al. 2009]:

1. Their output tags are independent of input tags.
2. Their tag propagation behaviors are known a priori and so summarized in a well-defined form.
3. None of the above.

For categories 1 and 2, tag propagation can be optimized by either skipping the computation completely or doing efficient *function-level* computation with only a few TPC instructions and execution traces, thus relieving the computation loads from PAU. For the last category, exhaustive finer-grained computations are inevitable since intensive monitoring is mandatory due to the nature of these functions. Fortunately in our DIFT implementation, we can still hinge the optimization of these heavy computations on our PAU which helps us not only to accelerate *instruction-level* computation but also to enjoy faster *block-level* computation for some parts of the functions in this category. In the followings, we will discuss our optimization strategies according to these categories.

6.1. Function Level Tag Propagation Optimization

Given a function of category 1 or 2, the whole tag propagation can be virtually turned off even though a small number of TPC instructions along with traces still need to be executed to fulfill complete tag propagation for those in category 2. Since huge performance gain can be obtained via function-level tag propagation, we try to maximize it by classifying as many functions as possible into categories 1 and 2 during our offline binary translation. This classification can be done by adopting traditional static analysis [Zhu et al. 2009; Saxena et al. 2008]. Another way to achieve it might be collecting a list of highly utilized functions encountered in applications such as library functions whose semantics are also well known and defined. For this purpose, we profiled a set of real programs in order to choose such functions that consume most time in them. When a function is found to be of category 1 or 2, we construct a function summary which is composed of the function name, TPC instructions and the code for execution traces that will be added to the original binary for the function. These summaries are created into the *function summary table* (FST). During binary translation with the original application, every function name in the code is brought to see if any function summary in FST has the name. If so, our instrument tool uses the information in the summary to produce the optimized TPC code as well as the host code that is instrumented to generate execution traces.

We present a code example in Figure 11 to explain in more detail how our adaptive multi-level DIFT is applied. Figure 11 (a) shows the original application code, where the invocation to the `malloc` function at line (3) takes the size as an input and returns a pointer to the allocated memory space as the output. A simple analysis on this function may easily reveal that the output tags cannot be derived from the input one because the resulting pointer and memory locations are not data dependent on the input size. As a consequence, the function should belong to category 1 by definition, and so its name has to be found in FST. As shown in the example, we see that our instrument tool produces neither code for traces nor for DIFT to save our computing resources, according to our optimization policy.

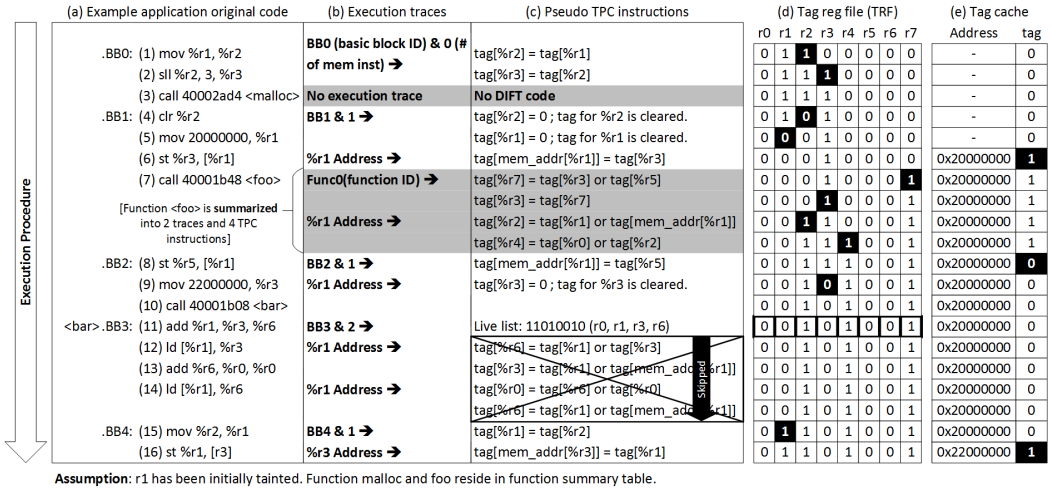


Fig. 11. An example for adaptive multi-level tracking

We assume that the function `foo` at line (7) is of category 2. Then, we can apply function-level DIFT to the function, as we explained. Therefore, small numbers of execution traces and TPC instructions are enough for complete tag propagation within `foo`. The figure shows that our instrument tool generates only two traces and four TPC instructions referring to FST. Figure 11 (b) and (c) respectively depict execution traces and TPC instructions generated after the multi-level tracking optimizations. Now notice that, in (b) for `foo`, the *function ID* is assigned in the first entry of the trace buffer. Similarly to basic block IDs in Section 5, a function ID is used to point PAU at the position where its TPC code starts to execute.

The shaded regions in Figure 11 represent the function-level tag propagation for the functions of categories 1 and 2. This clearly assures our argument that function-level optimizations improve the performance of both the host processor and PAU by drastically reducing or eliminating the computation loads due to execution traces and TPC instructions.

6.2. Block Level Tag Propagation Optimization

Although function-level tag propagation has a great affirmative impact on performance, all functions cannot take such benefits. Not surprisingly in real applications, a majority of functions fall into category 3. In principle, these functions necessitate instruction-level propagation, which will slow down the processing speed. To mitigate the overhead and further improve the DIFT performance, we exercise a coarser-grained tag propagation on some basic blocks dynamically during code execution. The optimization technique on block level was proposed in LIFT [Qin et al. 2006]. The basic idea is that the whole tag propagation in a basic block can be safely precluded if all the live-in/out rags of registers and memory locations into/from the block are untainted (i.e., the tag bits are all set off) at the boundary of the basic block. In LIFT, the decision is made just before the host CPU enters the entry of each block at run time.

In our work, we also implement and apply the same optimization scheme on our DIFT. The main difference between ours and LIFT is that the decision for every basic block is performed by PAU in our work. Consequently, this makes the host CPU to be liberated from the decision task, which otherwise slow down the host performance due to the computation overhead required for the task (e.g., instructions for managing and

checking the relevant tags, context switches for preserving the host program's states). That is, whether or not a basic block satisfies the above conditions, the host proceeds with its normal execution of the instrumented binary for this block, as described in Section 5. At the same time, TPC would extract from the trace buffer the execution traces that were issued from the host at the beginning of the current block. Recall that whenever TPC enters a new basic block, it reads the block ID from the first trace to execute the TPC instructions in the block. At this moment, it will examine all live data tags crossing the block boundary. If none is tainted, TPC just skips the execution of this block and be ready to extract new execution traces for the next block as directed by the host.

To enable this block-level optimization, we need to collect a summary about live tags around each block. For this, we have augmented our instrument tool to support live range analysis that identifies the live register tags coming into/out of every basic block, and puts them into the live list attached to each block. Assuming that n_r is the total number of registers, the live list is a bitmask of the size n_r bits. If a bit is set to 1, this represents that the corresponding register tag is alive at the entry of the basic block. By simply reading this list, TPC can determine the liveness of all register tags with ease. Contrary to the case of register tags, we do not apply static analysis to identify the live tags of memory locations obviously because exact memory addresses referenced in the code cannot be statically known in most cases. Therefore in our system, TPC collaborates with the host to dynamically figure out the liveness of memory tags at run time. To attain this objective, it accepts from the host all memory references in a basic block through the trace buffer.

Figure 12 displays a small decision logic that is vital to TPC's taint check of each live register or memory tag, which in turn collectively leads to the final decision on the applicability of block-level optimization to the current block. This logic determines whether live register tags are tainted or not by performing bitwise AND operation between the live list from the TPC code and the contents of TRF which is also an n_r-bit bitstream of tags of registers. If the result of this operation is zero, all the live register tags are untainted. Live memory tags should also be considered by accessing the tag cache with the addresses stored in the trace buffer. As shown in Figure 11, the number of memory addresses to be handled in the block is delivered as the execution trace (basic block ID and the number of memory addresses are encoded to a word). Thus, TPC can know how many memory tags should be considered for the decision. Since the tag cache has a single read port, it may take multiple cycles to load all memory tags depending on the number of memory addresses in the trace buffer. Finally, if all the live tags are declared untainted, TPC bypasses time-consuming instruction-by-instruction tag propagation inside the block, just waiting for the next direction from the host.

In Figure 11, we can see an example of block-level tag propagation within the function `bar`. Let us assume that `bar` is of category 3. Then, this function would not be found in FST as defined earlier. Instead, the instrument tool generates the TPC code in which the live list for each basic block in `bar` is attached to its entry. The tool also produces an instrumented code that will run on the host. Suppose that the host is about to execute the basic block **BB3** at line (11). At that time, TPC is ordered to execute the TPC code at the same line. As the first step, TPC has to extract the live list from the code, which is a bitmask 11010010 in this example. Then, it will compare it with the contents of TRF shown in Figure 11 (d). Simultaneously, the memory tag at address 0x20000000 is also retrieved from the tag cache shown in Figure 11 (e). We can successfully verify that all the tags are untainted in this example. This means that the entire tag propagation in the basic block can be safely omitted. Figure 11 exhibits that all TPC instructions for the block **BB3** are crossed out to indicate that the entire code execution on TPC is bypassed. This example shows that our block-level optimiza-

tion can enhance the DIFT performance with the cooperation of software analysis and hardware support.

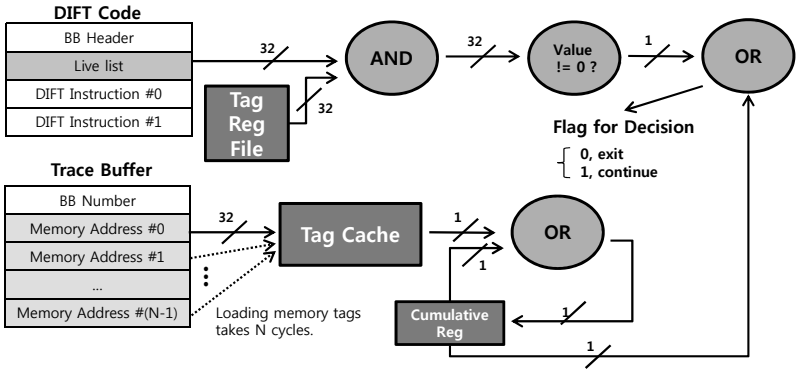


Fig. 12. The decision logic for block-level optimization

7. EXPERIMENT

7.1. Prototype System

To evaluate our approach, we have developed a full-system FPGA prototype, where the host processor is the SPARC V8 processor, a 32-bit synthesizable core [Manual 2004] which uses a single-issue, in-order, 7-stage pipeline. It has separate 16K-byte 2-way set associative instruction and data caches. The trace buffer has been implemented to accommodate at most 64 execution traces (i.e., 64x32bit) passed from the host. The architecture of our PAU follows the description in Section 3 and 4. It has the tag cache which is a 512-byte, 2-way set-associative cache with 8-byte cache lines, and the instruction cache which is a 4K-byte, 2-way set-associative cache with 32-byte cache lines. The bus compliant with AMBA2 AHB protocol [Limited 1999] is used to interconnect the all modules in our prototype system. Linux 2.6.21.1 is used as our OS kernel and a small portion of it has been modified to provide supports for our hardware engine as described before. Based on the parameters for the prototype as described above, we synthesized our DPA engine and verified it on a FPGA prototyping board with a Xilinx XC5VLX330 FPGA and 64MB external SDRAM.

7.2. Synthesis Results

When our hardware engine is employed in the systems that have severe resource constraints such as mobile devices, the area and power budgets of PAU are also strictly limited. Thus, in the systems, the area/power efficiency of the hardware engine is the foremost priority. In order to assess the area efficiency of our PAU, we quantified the resources necessary for PAU including the tag cache, instruction cache and trace buffer, in terms of gate counts using Synopsys Design Compiler [Guide 2009] with a commercial 45 nm process library. In Table III, the number of gates required for each component of PAU is described and compared to those of the baseline system including the host processor. The total area overhead for PAU is about 14.47%, as compared to the baseline system. Considering that our host processor is a very small SPARC RISC processor, we assure that the area overhead for PAU is not critical to be deployed in the commercial platforms.

As shown in the table, PAU can be divided into four parts; the components for tag computation which might correspond to the DPA engines in the hardware only approach [Deng and Suh 2012], the components added for the programmability, the trace buffer and the remaining parts such as AHB interface and interrupt generator. To support a wide range of DPA schemes with the programmability, PAU requires additional resources as described in the table (especially for the I-Cache). Although the amount of resources seems to be substantial, the total area overhead of PAU is not so huge as stated above.

To estimate the power consumption of PAU, we simulated our framework on Modelsim [Graphics 2007] and run the power estimation tools in Synopsys Design Compiler [Guide 2009] using the simulation result as an input vector. As a result of experiment using a commercial 45 nm process library, the power consumption of PAU is estimated to be 224.2 mW at 1 GHz operating clock frequency. Since it is acceptably small when compared to the power consumption of SPARC host processor (940 mW at 1 GHz), our PAU can be deployed in the commercial SoC platforms which have the limited power budget.

Table III. Synthesis result

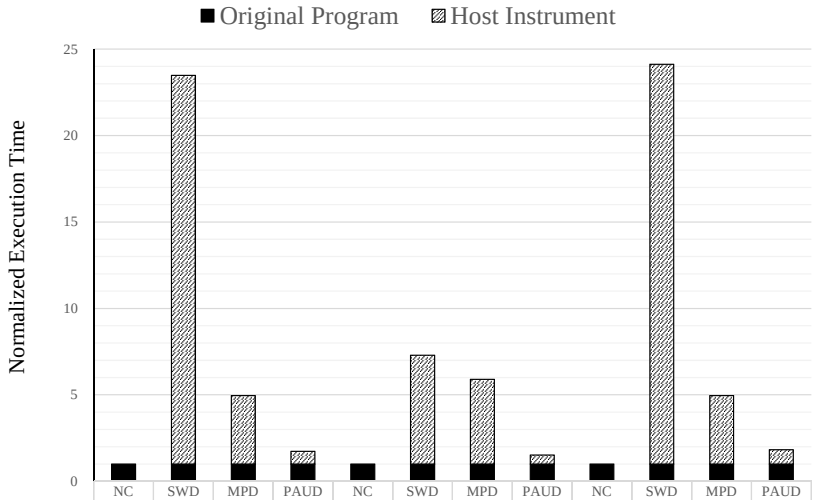
Category	Component	Gate Counts
Baseline System	SPARC V8 Core (Host Processor)	1761079.777
	Bus components (AHB Buses + AHB/APB bridges)	2137.61
	Memory Controller	3812.52
	Peripherals (TIMER, UART, and etc.)	3304.15
	Total Baseline System	**1770334.057**
PAU	Tag Register File (TRF)	957.4992
	Decoder	2305.6902
	Tag Cache	13253.2245
	Tag ALU	4932.274
	Tag TLB	10266.833
	Total Resources for Tag Compuation	31715.5209
	General Register File (GRF)	950.1425
	Main Controller	1339.0524
	Memory Tag Fetcher (MTF)	13253.2245
	I-Cache	203811.3338
	Total Resources for Programmability	219353.7532
	Trace Buffer (16 x 32-bit)	3425.1589
	ETC (including AHB Slave Interface)	1683.0324
	Total Resources for PAU	**256177.4654**
	% PAU over Baseline System	**14.47%**

7.3. Performance Evaluation

We have measured the performance improvement of our hardware engine over the previous approaches by choosing applications from the mibench benchmark suite [Guthaus et al. 2001] and comparing in performance with the four configurations. In this experiment, the configuration NC stands for native code which executes the original codes on the host CPU with DPA disabled. This is used as baseline, and all the other configurations are set with DPA enabled. For SWD, not only the code of original program but also the code for DPA are executed on the host core. Thus, whenever the DPA procedure is needed at a certain point of the program, the host invokes

the function which performs the tag computations. For MPD, to improve the performance, the tag computations for DPA are offloaded to another general purpose core which is dedicated for the analysis, called *analysis core*. However, the handling codes for sending the execution traces still needs to be invoked on the host since we assume that the multiprocessor approach in our experiment does not have the dedicated hardware queue such as the trace buffer. The register file of the analysis core acts as the shadow register file to store the tags of the host registers, as proposed in [Nagarajan et al. 2008]. For this reason, the analysis core should preserve and restore the states of the registers when the DPA needs to use the registers for other general computations, such as trace handling. Lastly, for the configuration PAUD, the tag computations are offloaded onto our PAU. With the help of the dual register file architecture (that is, GRF and TRF), PAU can remove the overhead of the context switches which is paid in MPD, because it uses the registers of the GRF for general computations. Also, the host can reduce the overhead for transferring execution traces since the trace buffer can be accessed by simply storing the traces to the predefined memory address, instead of executing the codes for trace communication.

In Figure 13, we depict the performance comparison among the four configurations. We have measured the average of host execution time for eight applications in mibench (i.e., dijkstra, bitcnt, rijndael, sha, blowfish, strsearch, patricia and qsort) and they are normalized to that of NC. SWD runs on average 7.3-24.1 times slower than NC because the additionally instrumented codes for DPA are performed by the host. In MPD, an additional general purpose core is dedicated to perform DPA computations but the slowdown of the multiprocessor approach reaches up to 4.9-5.9 times of NC due to the inefficient structure of general cores. To the contrary, PAUD substantially cuts the overhead down to 52.0-82.8% of NC for the three DPAs through the acceleration with PAU. It is 4.7-13.6 times faster than SWD. Even from MPD, PAUD enhances the analysis performance up to 2.7-3.8 times. The results clearly shows that our approach can be effective in leveraging DPA performance.



	NC	SWD	MPD	PAUD	NC	SWD	MPD	PAUD	NC	SWD	MPD	PAUD
	DIFT				UMC				BC			
Host Instrument	0.00	22.48	3.97	0.73	0.00	6.29	4.90	0.52	0.00	23.12	3.96	0.83
Original Program	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Fig. 13. Comparison of execution time (normalized to native)

So far we have assumed that PAU and the host processor operates at the same clock speed, but this assumption might not apply to recent systems in which the host operates far faster than other modules [Samsung Electronics co. 2012; ARM co. 2012]. In this environment, our host would swiftly produce execution traces at a rate more than PAU could consume. This may cause the host to stall frequently, thereby slowing down the overall program execution. To remedy this problem, PAU should either operate at a higher frequency or have a bigger trace buffer that may accumulate more traces. However, these remedies might be unacceptable since they raise hardware costs. Therefore, our PAU should be able to tolerate the performance gap between the two processing cores.

To certify that our PAU can circumvent this very problem, we conducted an experiment with the configuration PAUD, under the condition that the host processor runs 2 to 8 times faster than PAU as done in [Kannan et al. 2009]. Figure 14 depicts the execution times of PAUD normalized to NC when the performance gap is increased. As shown in the figure, the execution time of PAUD is affected by the three component; the execution time of original program, the overhead incurred by the instrumentation on the host code and the synchronization overhead due to the performance gap between the host and PAU. For the three DPA techniques, when PAU and the host operate at the same frequency (see 1X in Figure 14), the performance of PAUD is affected only by the host instrument. That is, PAU can keep up with the processing speed of the host at this condition. However, as the performance gap increases, the synchronization overhead is also increased since the relative computation power of PAU decreases. Nevertheless, the amounts of increased overhead are less than 30% for the three DPAs even when the performance gap reaches up to eight times. The results imply that our PAU is applicable to a broad range of platforms even when the performance ratio between the host processor and PAU becomes increased, with the help of the specialized architecture of PAU.

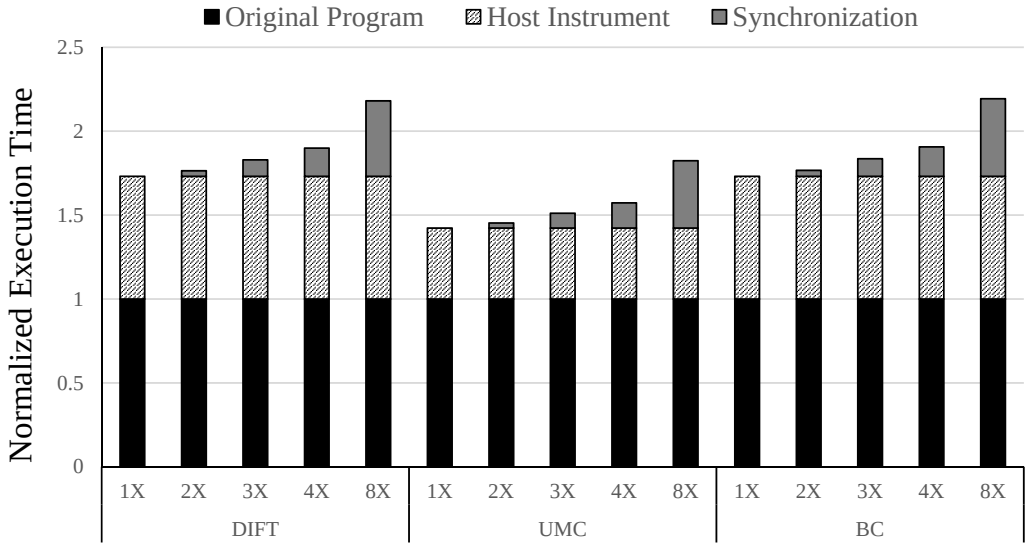


Fig. 14. Execution time of PAUD when PAU is paired with higher frequency host processor(normalized to native)

In Section 6, as a practical example where the programmability of TPC can be utilized, we introduced the multi-level tracking optimizations for DIFT. In Figure 15, the performance improvement achieved by the optimizations is shown, for the eight applications of mibench. The configuration PAUD_multi is the optimized DIFT implementation explained in Section 6. For the fair comparison, we also apply the optimizations to other approaches. In the two configurations, SWD_multi and MPD_multi, the same optimizations are added to SWD and MPD respectively. The performance improvement of the block-level optimization is affected by the taintness of the input [Qin et al. 2006]. To maximize the performance improvement, we assume that the input files accessed by the applications are not the confidential ones so that the tag for the input is not set. Thus, virtually all basic blocks are skipped by the block-level optimization although the computations for the decision are still required.

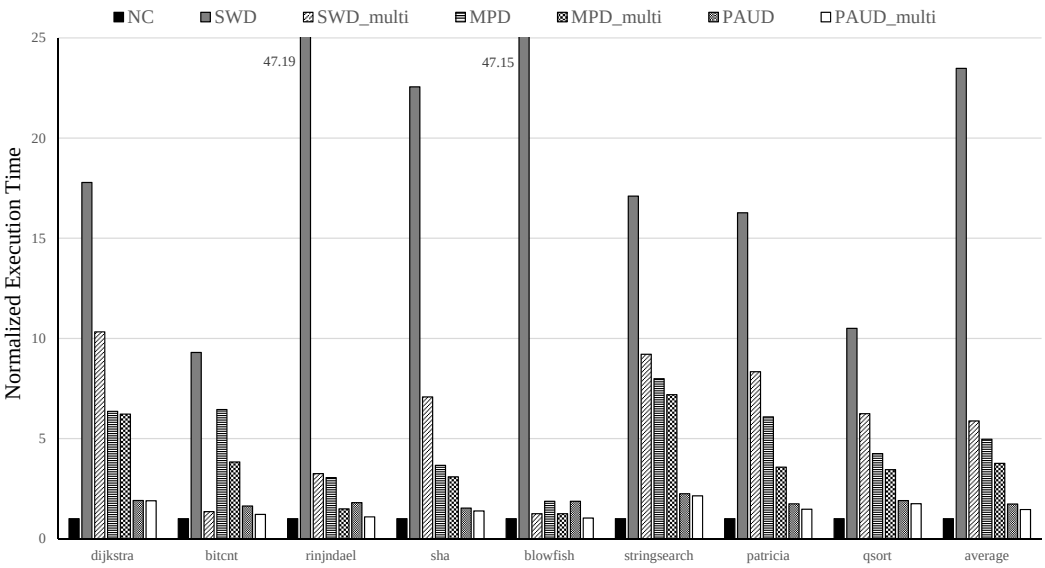


Fig. 15. Comparison of execution time for DIFT implementations (normalized to native)

As depicted in Figure 15, the adoption of the optimizations improves the performance in all the approaches compared in our experiment. SWD_multi is about 4 times faster than SWD, and MPD_multi improves the performance of MPD by 31.9%. In our approach with PAU, the performance of PAUD is also enhanced by 18.8% with the optimizations and it consequently reduces the DIFT overhead to only 45.7% in comparison with NC. In Figure 16 shows the execution time of PAUD_multi for various performance gap ratio between the host and PAU. As shown in the figure, the optimizations applied to our DIFT implementation can reduce the DIFT overhead substantially. Even when the performance ratio is 8x, the increased execution time is about 33.1%. The results show that the programmability of PAU can help our DIFT implementation to improve the performance by taking the advantage of software's flexibility and to be more tolerable to the performance ratio.

8. RELATED WORKS

Most software DPA approaches [Devietti et al. 2008; Savage et al. 1997; Seward and Nethercote 2005; Newsome and Song 2005; Cheng et al. 2006; Yin et al. 2007; Qin

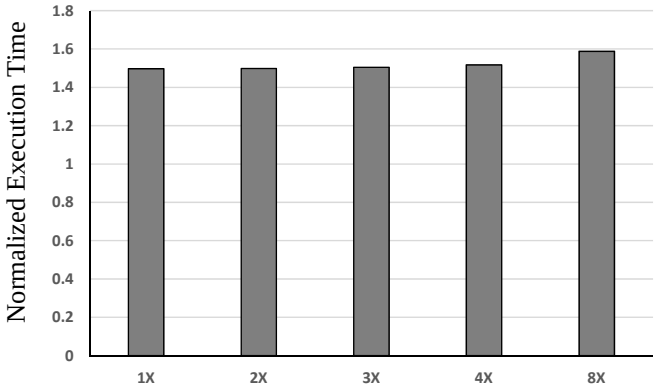


Fig. 16. Performance overhead of PAUD_multi when PAU is paired with higher frequency host processor(normalized to native)

et al. 2006; Zhu et al. 2009] have relied on binary-code instrumentation to augment the codes for DPA to carry out their analysis schemes. However, even in simple analysis such as array bound checking, it requires substantial analysis computations [Tiwari et al. 2009]. For example, Memcheck [Seward and Nethercote 2005] uses dataflow tracking to detect a wide range of memory errors in programs as they run. Under the analysis, the monitored program typically run 20-30 times slower than normal. Although it is paid at test-time, the performance overhead sometimes limits the amount of analysis due to restricted time for the software development, thereby making it difficult to remove all errors in the program. In case of DIFT, the performance overhead of software-based solutions [Newsome and Song 2005; Cheng et al. 2006; Yin et al. 2007] reaches up to 37 times the original code execution [Newsome and Song 2005]. Several efforts were made to curtail the overhead with optimization techniques [Qin et al. 2006; Zhu et al. 2009], but it yet remains one or two orders of magnitude higher than the execution time of the original program. Considering that DIFT is usually used for runtime monitoring, the analysis performance is not acceptable level to be deployed in real applications.

In order to improve the analysis performance, there are several software-based approaches to utilize multiprocessors [Chen et al. 2008; Nagarajan et al. 2008; Nightingale et al. 2008] that are readily available in modern multicore architecture, such as Intel i7, where each core is a GPP. The key idea here is to devote GPP cores to run *helper threads* whose missions are actual analysis for the host program running concurrently on another GPP core. For example, Speck [Nightingale et al. 2008] offers up to 7.5X speedup with 8 cores for light-weight analyses like scanning the address space for sensitive data. However, although they can lessen the performance overhead with existing architectures, the achieved performance is not sufficient for more powerful analyses, mainly because the original GPP architecture is not optimized for program analysis in the first place [Deng and Suh 2012]. For instance, in [Chen et al. 2006; Chen et al. 2008; Nagarajan et al. 2008], the program execution times get 3-7 times slower when the analyses being enabled so that it is too slow to be used for runtime monitoring. Moreover, even in test-time analyses, there is also a demand for more complex analysis tools that incur overheads from 100 to 300 X [Tiwari et al. 2009; Seward 2014; Vogt et al. 2007; Mysore et al. 2008].

To mitigate the performance overhead, in several multiprocessor approaches [Chen et al. 2008; Nagarajan et al. 2008], they modified the host CPU's internal architecture and integrated the specialized hardware modules like our trace buffer. By doing

so, they were able to reduce the overhead to acceptable levels, which is around 50% or less [Nagarajan et al. 2008] in DIFT problem. Nevertheless, such modifications in these approaches also impose the same problem of the core-level approach that mandates the alteration of existing commodity processors.

To address the shortcoming of software-based analysis, several core-level hardware engines have been proposed [Dalton et al. 2007; Venkataramani et al. 2008; Deng et al. 2010; Chen et al. 2008; Deng and Suh 2012; Witchel et al. 2002; Devietti et al. 2008; Clause et al. 2007; Joao et al. 2009; Zhou et al. 2004; Meixner et al. 2007]. In those approaches, extra hardware logics customized for analysis operations are integrated into a processor. A number of DPA schemes are supported in the core-level engine [Deng and Suh 2012] such as fine-grained memory protection [Witchel et al. 2002], array bound checking [Devietti et al. 2008], software debugging support [Zhou et al. 2004], managed language support like garbage collection [Joao et al. 2009]. The main advantage of the core-level approaches is that they do not need to instrument the host code since they can extract the necessary information from the processor's pipeline transparently. Thus, they could bring the overhead down to under 5%. However, they have a disadvantage in that invasive modifications to the processor internal (e.g., registers and pipeline data paths) are required. In fact, modern microprocessor development may take several years and hundreds of engineers from an initial design to production [Deng and Suh 2012; Kannan et al. 2009]. Therefore, the substantial costs of development to integrate the customized logic would hamper processor vendors to adopt them, unless the necessity is clearly established.

Several previous works [Deng et al. 2010; Deng and Suh 2012; Dhawan et al. 2015] have been proposed to leverage the flexibility to support various DPA schemes, generalizing from the core-level engines. FlexCore [Deng et al. 2010] is a hybrid architecture that combines a general core with a decoupled on-chip FPGA fabric. Although the FPGA logic can be reconfigured to conduct a set of DPA schemes in hardware, the low throughput of FPGA can cause high performance overheads [Deng and Suh 2012]. To mend this problem, in Harmoni [Deng and Suh 2012], they proposed a high performance and reconfigurable co-processor for a wide range of DPAs. With the considerations on the tag-based DPA model, Harmoni has the specialized pipeline architecture which can achieve very high performance, while it also has sufficient flexibility thanks to the configurable tables in the engine. Nevertheless, as discussed in Section 1, it still has the extendibility issues when a new analysis method is suggested.

In recent years, for DPA techniques or malware detection, the system-level hardware engines like our PAU have been proposed, which do not require any modification on the host core [Petroni Jr et al. 2004; Tiwari et al. 2009; Moon et al. 2012; Lee et al. 2013]. Among them, Hardgrind [Tiwari et al. 2009] is the previous work closest to ours where the computation of an instrumented program is paralleled between the host and the accelerator. However, since they only quantified the potential of this approach by considering how the execution traces are delivered to the engine, the detailed structure of the accelerator was not sufficiently addressed. On the contrary, in this paper, we designed the detailed architecture of our PAU that supports various DPA schemes as well as enhances the analysis performance, in order to leverage the system-level approach.

Another difference between Hardgrind and ours is the method for the trace communication. In Hardgrind [Tiwari et al. 2009], they allocate a buffer space in the host memory and store the traces into the region. Once the buffer is full, DMA transfer is triggered so that the traces are delivered to the analysis engine in a bulk. As compared to our approach with the trace buffer, when this transfer method is used, the buffer access latency can be reduced because the buffer that can be cached by the host CPU's cache is much faster than the trace buffer located in the off-core module. On the

contrary, the DMA-based transfer mode increases the number of the added instructions to manage the buffer in the memory. That is, there exists a trade-off between the two overhead sources; the number of instructions and the access latency to the buffer. In our work, we chose to use the trace buffer because the access latency of the trace buffer located on the system bus is relatively short and it is more efficient than the DMA-based transfer mode in our prototype. However, in desktop platforms where PAU is implemented as a PCI card, the DMA transfer mode might be more efficient way to communicate, as presented in Hardgrind [Tiwari et al. 2009].

As one of complementary works for our PAU, it is noteworthy that Log-Based Architectures (LBA) [Chen et al. 2006; Chen et al. 2008] proposed a decoupling execution strategy in the context of the multiprocessor approaches while it also have core-level logics to compress, deliver and decompress the traces of host programs. By employing the LBA architecture, the host can deliver the traces to our engine without instrumenting the host code thereby improving the analysis performance. Although this architecture is not yet available in the commodity market, we hope that it will be implemented and sold as a commercial product in the near future.

In several hardware engines [Shankar and Lysecky 2009; Kannan et al. 2009; Majzik 1996], they have proposed to utilize special channels for acquiring runtime information, without the modification on the host CPU's internal pipeline. Although they had to slightly modify the hardware design of host CPU to provide such channels in their works, it is noteworthy that this problem can be resolved by incorporating the trace interfaces available in recent commodity cores. For example, the recent ARM processors, such as Cortex-A9 or A15, include the CoreSight architecture [ARM co. 2013] to support efficient and convenient tracing. It can provide the analysis engines with various runtime information such as branch results, context switches and exceptions, without incurring performance overhead for trace communication. If the interface can be combined with the hardware engines, they can achieve high performance in DPA computations while the system-level integration is still viable. In this context, it is noteworthy that in Extrax [Lee et al. 2015] proposed by J.Lee et. al, the core debug interface available in many CPU architectures is employed for efficient kernel integrity monitoring. Since the interface can provide many informative signals to retrieve the context of runtime execution without performance loss, Extrax can detect any malicious attempt to compromise the kernel with negligible overhead. Although they only focused on the kernel integrity, we believe that the use of the core debug interface can be exploited in DPA. Motivated from this, we also have a plan to incorporate the interfaces to PAU in our future work, in order to achieve both the programmability and performance improvement.

9. CONCLUSION

This paper presented a system-level hardware engine, called PAU, which is an application-specific programmable processor to support a wide range of DPA techniques with the enhanced analysis performance. PAU can speed up the analysis performance with the help of specialized architecture based on the tag-based model, which otherwise would be substantially slow as in computations on GPP cores. In addition, with its programmability, it can support a wide range of DPA techniques and enable flexible computations for evolutionary analysis strategies. In our case studies, we demonstrated the effectiveness of our approach by realizing several DPA techniques on our PAU and successfully adopting the software-assisted optimizations for DIFT. Moreover, following the system-level approach in Hardgrind, our PAU has been designed as a system-level component without any modifications in the host processor internal and it is integrated with an existing platform. Therefore, our approach can be easily implanted to a commercial mobile platforms or desktop ones.

Our experiments on FPGA prototype revealed that our solution can reduce the DPA performance overhead substantially compared to the previous solutions. While multiprocessor approaches slow down the execution of a program by more than a factor of 4, our PAU incurs overwhelmingly low overhead, that is only 45.7% for a group of mibench applications in our DIFT implementation. Even when our PAU is several times slower than the host processor, the DIFT overhead increases only slightly about 33.1% for the same applications. The experiments also revealed that the power consumption and area overhead of PAU are acceptably small compared to today's mobile processors. All in all, we hope that our proposed ASIP approach would become an attractive DPA solution to production-quality commodity platforms.

REFERENCES

LTD ARM co. 2012. Mali-400 MP. (2012). http://www.arm.com/

LTD ARM co. 2013. ARM CoreSight Architecture Specification v2.0. (2013). http://infocenter.arm.com/help/topic/com.arm.doc.ihi0029d/IHI0029D_coresight_architecture_spec_v2_0.pdf

Briant Bailey, Grant Martin, Martin Grant, and Thomas Anderson. 2005. *Taxonomies for the Development and Verification of digital systems*. Springer.

Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering.. In *NDSS*, Vol. 9. Citeseer, 8–11.

Derek Lane Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Shimin Chen, Babak Falsafi, Phillip B Gibbons, Michael Kozuch, Todd C Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R Ganger, Bin Lin, and others. 2006. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 63–65.

Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible hardware acceleration for instruction-grain program monitoring. In *ACM SIGARCH Computer Architecture News*, Vol. 36. IEEE Computer Society, 377–388.

W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Computers and Communications, 2006. ISCC '06. Proceedings. 11th IEEE Symposium on*. 749–754. DOI:http://dx.doi.org/10.1109/ISCC.2006.158

James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. 2007. Effective memory protection using dynamic tainting. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 284–292.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: a flexible information flow architecture for software security. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 482–493.

D.Y. Deng and G.E. Suh. 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. 1–12. DOI:http://dx.doi.org/10.1109/DSN.2012.6263925

Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. 2010. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 137–148. DOI:http://dx.doi.org/10.1109/MICRO.2010.17

Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 103–114. DOI:http://dx.doi.org/10.1145/1346281.1346295

Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. (2015).

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–6. http://dl.acm.org/citation.cfm?id=1924943.1924971

Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition.. In *NDSS*.

Mentor Graphics. 2007. ModelSim. (2007).

Design Compiler User Guide. 2009. Version C-2009.06. *Synopsys.(a)(b)(c)* (2009).

Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 3–14.

José A Joao, Onur Mutlu, and Yale N Patt. 2009. Flexible reference-counting-based hardware acceleration for garbage collection. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 418–428.

H. Kannan, M. Dalton, and C. Kozyrakis. 2009. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. 105–114. DOI:http://dx.doi.org/10.1109/DSN.2009.5270347

Hojoon Lee, HyunGon Moon, DaeHee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. 2013. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *Presented as part of the 22nd USENIX Security Symposium*. USENIX, Washington, D.C., 511–526. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/lee

Jinyong Lee, Yongje Lee, Hyungon Moon, Ingoo Heo, and Yunheung Paek. 2015. EXTRAX: Security Extention To Extract Cache Resident Information For Snoop-based External Monitors. In *Design Automation and Test in Europe Conference and Exhibition (DATE)*.

ARM Limited. 1999. AMBA Specication. (1999).

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. DOI:http://dx.doi.org/10.1145/1065010.1065034

István Majzik. 1996. Concurrent error detection using watchdog processors. In *Fault tolerant computing systems*, Vol. 283. Kluwer Academic.

LEON3 Processor User's Manual. 2004. Gaisler Research. (2004).

Albert Meixner, Michael E Bauer, and Daniel J Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE, 210–222.

Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. 2012. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 28–37. DOI:http://dx.doi.org/10.1145/2382196.2382202

Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. 2008. Understanding and Visualizing Full Systems with Data Flow Tomography. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 211–221. DOI:http://dx.doi.org/10.1145/1346281.1346308

Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. 2008. Dynamic information flow tracking on multicores. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices* 42, 6 (2007), 89–100.

James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. (2005).

Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. 2008. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 308–318. DOI:http://dx.doi.org/10.1145/1346281.1346321

Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. 2004. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor.. In *USENIX Security Symposium*. San Diego, USA, 179–194.

Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 135–148.

Mohan Rajagopalan, Matti A Hiltunen, Trevor Jim, and Richard D Schlichting. 2006. System call monitoring using authenticated system calls. *Dependable and Secure Computing, IEEE Transactions on* 3, 3 (2006), 216–229.

LTD Samsung Electronics co. 2012. Exynos 5 Dual. (2012). http://www.samsung.com/global/business/semiconductor/

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 74–83.

J Seward. 2014. Origin tracking tool, valgrind release-3.4. 0. Pre-release at svn co svn. (2014). svn://svn.valgrind.org/valgrind/trunk

Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision.. In *USENIX Annual Technical Conference, General Track*. 17–30.

Karthik Shankar and Roman Lysecky. 2009. Non-intrusive dynamic application profiling for multitasked applications. In *Proceedings of the 46th Annual Design Automation Conference*. ACM, 130–135.

G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *ACM SIGOPS Operating Systems Review*, Vol. 38. ACM, 85–96.

Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. 2008. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 94–105. DOI:http://dx.doi.org/10.1109/MICRO.2008.4771782

Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. 2009. Quantifying the potential of program analysis peripherals. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 53–63.

Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 173–184.

Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. 2007. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 273–284.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis.. In *NDSS*.

Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 304–316. DOI:http://dx.doi.org/10.1145/605397.605429

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 116–127.

Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. 2004. iWatcher: Efficient architectural support for software debugging. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. IEEE, 224–235.

Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-assisted lockset-based race detection. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 121–132.

Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2009. *Privacy Scope: A precise information flow tracking system for finding application leaks*. Ph.D. Dissertation. University of California, Berkeley.

Received February 2014; revised ; accepted