AFRL-RY-WP-TR-2019-0206

TRAILS: EFFICIENT DATA-FLOW TRACKING THROUGH HW-ASSISTED PARALLELIZATION

Georgios Portokalidis

Stevens Institute of Technology

SEPTEMBER 2019
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2019-0206 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

TOD J. REINHART
Program Manager
Resilient and Agile Avionics Branch
Spectrum Warfare Division

//Signature//

DAVID G. HAGSTROM, Chief
Resilient and Agile Avionics Branch
Spectrum Warfare Division

//Signature//

NEERAJ PUJARA, Acting Chief
Spectrum Warfare Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2019	Final	28 September 2016 – 30 June 2019

4. TITLE AND SUBTITLE
TRAILS: EFFICIENT DATA-FLOW TRACKING THROUGH HW-ASSISTED PARALLELIZATION

5a. CONTRACT NUMBER
FA8650-16-C-7662

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)
Georgios Portokalidis

5d. PROJECT NUMBER
1000

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
Y1KJ

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Stevens Institute of Technology
1 Castle Point on Hudson
Hoboken, NJ 07030

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2019-0206

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. Report contains color.

14. ABSTRACT
The goal of this effort is to make currently opaque computing systems transparent by providing high-fidelity visibility into component interactions during system operation at the hardware architecture layer, while imposing minimal performance overhead. The scope of this effort is to develop a novel architecture for efficiently performing dynamic data-flow tracking through hardware-assisted parallelization. The goal includes the creation of an execution environment that provides low-overhead, fine-grained, data-flow tracking (DFT) as a utility for security, privacy, and other applications. Trails aims to exploit the parallelism available in modern architectures to improve DFT performance without the demand for more resources than the ones already required to apply it in line with the application.

15. SUBJECT TERMS
hw-assisted parallelization, dynamic data-flow tracking

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Tod Reinhart
Unclassified	Unclassified	Unclassified	SAR	30	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Table of Contents

Section **Page**

List of Figures

Figure **Page**

List of Tables

Table **Page**

1.0 SUMMARY

Project Goal We propose to develop Trails, a novel architecture for efficiently performing dynamic data-flow tracking through hardware-assisted parallelization. Our goal is to create an execution environment that provides low-overhead, fine-grained, dynamic data-flow tracking (DDFT) as a utility for security, privacy, and other applications. We envision an architecture where spare processor cores or specialized application processors transparently track data as they are processed by executing applications. Developers will be able to use the architecture to build security applications that employ DDFT as a primitive. For example, by exploiting the data provenance information offered by DDFT, we can prevent software exploitation and information leakage, we can also improve forensic analysis of compromised or malfunctioning systems, and generally shed light to the origin of data and how systems use them. Trails will provide a modular design that can be applied to existing processors with tracing facilities for debugging. It will also enable new systems on chip (SoC) with DDFT capabilites that do not require changes to the core design of modern processors, such as ARM, but utilize their debugging interfaces. Our proposal is within the thrust area that aims to "Guarantee trustworthy computing and information".

Current Approaches and Limitations A large group of previous approaches relied on recording execution and applying DFT during the replay. These solutions are not appropriate for preventive applications and require significant resources to reconstruct execution with DFT. Other approaches use speculative execution to run application code instrumented with DFT operations in multiple threads running in parallel. These solutions do not actually decouple DFT, but concurrently explore multiple execution paths. Hence, they also sacrifice significant resources, since many threads need to be discarded during execution. Current approaches that actually decouple and run DFT in parallel are more efficient, but in practice they have not been able to deliver the expected performance gains, due to the cost of collecting and communicating run-time information to the threads or processes performing DFT. In the past, we have also seen many proposals of new hardware designs that include DFT. However, these solutions demand sometimes extensive changes to current processors and as a result have not seen adoption by vendors. In contrast, our approach will employ existing hardware capabilities to reduce the overhead of decoupling DFT.

Project Contributions Towards achieving our goal, we have investigated how execution tracing technologies can be utilized for decoupling DDFT from application code and running it in parallel. Such technologies can be found in Intel and ARM processors, and they are commonly used for assisting in debugging. We have built a system, named JITrace, that employs Intel's Processor Trace technology to reconstruct the control-flow of an application running on a virtual machine employing just-in-time translation. JITrace is the basis for a decoupled DDFT system, like ShadowReplica that was previously developed with the help of the PI for the STONESOUP project. JITrace has been developed for Linux and the Pin virtual machine. The performance of the prototype is currently comparable to that of a software-only implementation, however, we are confident that, if our efforts are continued, its performance can significantly improve, through the use of optimized algorithms and programmable hardware (i.e., an FPGA). We have also conducted a preliminary investigation in using the complete execution traces provided by

some ARM processors towards the same goal. Our investigation revealed that significant more effort would be required to develop such a solution, due to limitations of the hardware. Finally, we have begun developing a framework, VEX2TRACK, that will automatically generate optimized code that implements DDFT logic for binary applications, independently of their architecture (i.e., instruction-set of the target architecture). This framework would allow us to completely replace the previously used system, ShadowReplica, that is limited to 32-bit systems and custom optimizations.

Impact Our work has shown that hardware-based tracing on Intel processors has similar overhead to a software-only approach, which is attributed to the large volume of data produced by the hardware that requires decoding. Additional research is required to develop more optimized algorithms and to utilize programmable hardware for resolving the decoding bottleneck. Additional work is also required to develop prototypes for ARM processors and to achieve our long term goal of a system-on-chip design that performs DDFT with negligible overhead.

2.0 INTRODUCTION

2.1 Motivation

Dynamic data flow tracking (DFT) is being used extensively in security research for protecting software [9, 24], analyzing malware [19, 35], discovering bugs [7, 23], reverse engineering [30], information flow control [37], etc. However, dynamically applying DFT tends to significantly slow down the target application, specially when a virtualization framework [5, 21] is used to apply it on binary-only software. Overheads can range from a significant percentage over native execution to several orders of magnitude, depending on the framework used and particular traits of the implementation [18]. When performance is not an issue, the overhead can still be problematic: (a) if it changes the behavior of the application (e.g., when network connections timeout or the analysis is no longer transparent), or (b) when computational cycles are scarce or CPU energy consumption needs to be kept to a minimum, like in mobile devices.

2.2 Goals

The goal of this project is to develop and experimentally evaluate technologies that will enable low-overhead, fine-grained, data-flow tracking (DFT) capabilities for the development of security, privacy, and other applications. To achieve our goal, we developed technologies that take advantage of debugging features found in modern CPUs to *efficiently* decouple dynamic data-flow tracking (DFT) from application code and run it in parallel.
Our approach for efficiently decoupling DFT involves: (i) performing static and dynamic analysis to extract DFT semantics from binaries, (ii) generating code that implements those semantics, and (iii) instrumenting the application to communicate any run-time information required by (ii). DFT can be executed in parallel with the running process, either in software, or hardware. In current approaches, the major challenge in terms of performance is step (iii). To tackle this challenge, we propose using branch and execution tracing facilities in modern Intel and ARM CPUs to obtain the run-time information required by the now decoupled DFT, thus greatly reducing the overhead imposed on applications. We plan to leverage two debugging facilities: branch tracing, available on Intel CPUs, and branch and data tracing available on ARM CPUs. Our first task, is to use branch tracing to reduce the software instrumentation on the process, leveraging prior work that focuses on decoupling DFT in software only [16, 17]. Second, we will use branch and data tracing on ARM CPUs to rely entirely on hardware for providing the necessary information.

2.3 Summary of Contributions

During this project, we have accomplished the following:

- We have built a new system, coined JITrace, that utilizes Intel PT data to reconstruct the control flow of an application running on top of a VM that employs just-in-time translation (JIT), such as Pin. JITrace can reduce the dependency on instrumentation and has the potential to accelerate DDFT systems.
- We have begun investigating utilizing execution tracing, as implemented by certain ARM processors, for completely removing the need for a middle layer like Pin.

- We have begun developing a new framework, coined VEX2TRACK, for generating code for decoupled, parallelized DDFT systems independently from the instruction-set of the targeted platform.

2.4 Background

2.4.1. Decoupling DDFT from Execution

Decoupling analysis from execution to run it in parallel is by no means a novel concept [6, 8, 13, 25, 28, 33, 36]. Previous approaches can be classified into three categories. The first is based on recording execution and replaying while applying DFT on a remote host, or simply a different CPU [6, 8, 28]. These are geared toward offline DFT-based applications and can greatly reduce the overhead they impose on the application. However, the speed of DFT itself is *not* improved, since execution needs to be replayed and instrumented with inline DFT code. These solutions essentially hide the overhead from the application, by sacrificing computational resources at the replica. Due to their design, they are not a good fit for applying preventive security measures, even though they can be used for post-fact identification of an intrusion.

The second category uses speculative execution to run application code including inlined DFT in multiple threads running in parallel [25, 33]. While strictly speaking the analysis is not decoupled, it is parallelized. These approaches sacrifice *significant* processing power to achieve speed up, as at least two additional threads need to be used for any performance gain, and the results of some of the threads may be discarded. Furthermore, handling multi-threaded applications without hardware support remains a challenge.

The third category aims at offloading the DFT to another execution thread [13, 36]. These instrument the application to collect all the information required to run the analysis independently, and communicate the information to a thread running the analysis logic alone. In principle, these approaches are more efficient, since the application code only runs once. However, in practice, they have not been able to deliver the expected performance gains, due to inefficiently collecting information from the application and the high overhead of communicating it to the analysis thread.

Our approach is inspired by the third category of systems, which means that it exploits the parallelism available in modern architectures to improve DFT performance without the demand for more resources than the ones required to apply it inline with the application. The reason for this is conceptually shown in Fig. 1. DFT involves accurately tracking selected data of interest as they propagate during program execution. Applying it dynamically on binaries usually involves the use of dynamic binary instrumentation (DBI) frameworks or virtual machine monitors (VMM) that transparently extend the program being analyzed. Such frameworks retrofit DFT in binaries by interleaving framework and DFT code with application code, as shown in Fig.1 (b). The total overhead is then a compound of the overhead imposed by the instrumentation framework and the DFT logic injected in the application. In past work, we demonstrated that we can greatly accelerate DFT by instrumenting in the application to extract the run-time information required to independently apply DFT and running it in parallel. As shown in Fig. 1 (c), this provides a significant impact in performance. Our experiments showed that the total run time is reduced to half

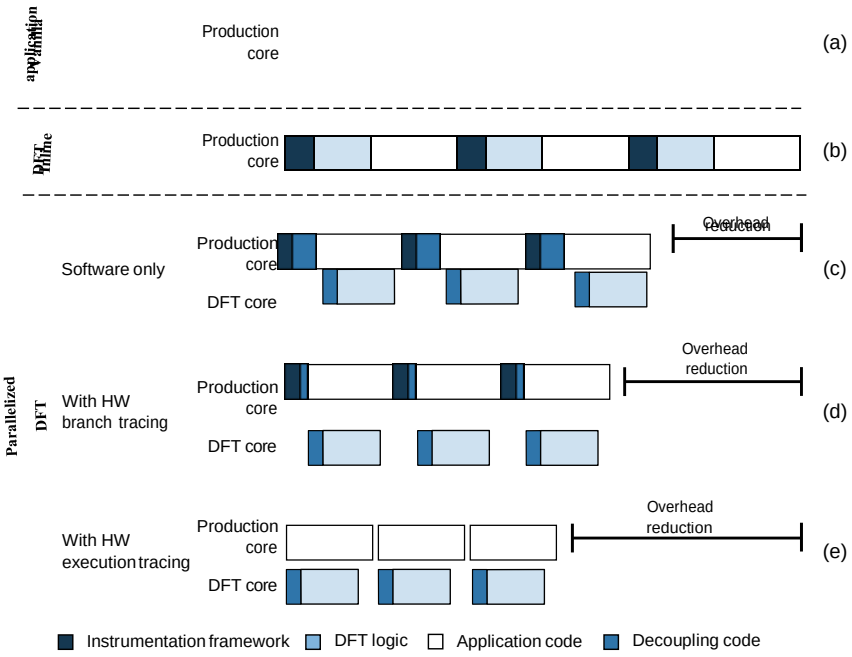


Figure 1: Benefits of Trails and Decoupling DFT from Application

compared with inlining DFT, while concurrently using less CPU cycles. Trails will further reduce overhead by using hardware to obtain the data required by the DFT logic. Performance benefits will be gained first by using hardware to obtain run-time control-flow information (Fig. 1 (d)) and, second, by using hardware to obtain all the run-time information (Fig. 1 (e)), hence, requiring zero or very little instrumentation of the application. Compared with previous proposals for implementing DFT in hardware [10, 11, 12, 27, 31, 32], Trails is actually building on facilities available on modern, popular processors.

2.4.2. Intel's Pin Dynamic Binary Instrumentation (DBI) Tool

Pin [21] enables the development of tools that can augment, modify, or simply monitor a binary's execution at the instruction level. It provides a rich API that can be used by developers of tools (Pintools) to install callbacks to inspect a program's instructions and routines, as well as intercept system calls and signals. In Pin's terms, it allows the *instrumentation* of the application. Additionally, instrumentation routines can modify original code by removing instructions or by more frequently adding new code, referred to as *analysis* code. The instrumented application executes on top of Pin's virtual machine (VM) runtime, which essentially consists of a just-in-time (JIT) compiler that combines the original and analysis instructions, and places the produced code blocks into a *code cache*, where the application executes from.

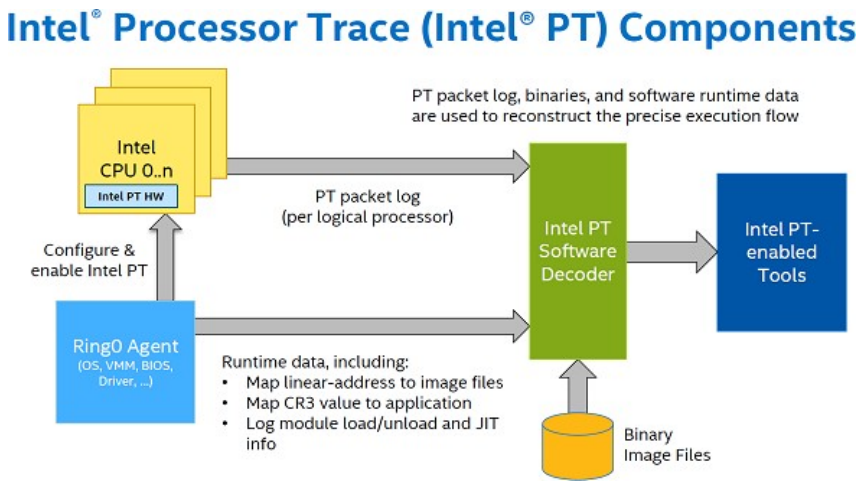


Figure 2: Intel Processor Trace Components

(This picture is property of Intel corporation.)

2.4.3. Intel Processor Trace

Intel Processor Trace [15] (PT), shown in Fig. 2 enables the recording of branch outcomes with low overhead. The processor can be configured to generate a stream of data packets that include information such as the type of branch instruction executed, whether it was taken for conditional branches, its target address, and even includes packets for asynchronous control-flow events (e.g., signals and exceptions). The data are stored directly to physical memory using multiple buffers, which can be mapped into the application, and the process raises an interrupt when the buffer it was using becomes full and continues by using the next buffer. The data stream produced by the processor is also highly compressed. For instance, only a single bit is used to indicate the direction of a conditional branch, while the target of a return instruction is omitted, when the information was previously recorded in the stream due to a call instruction. Moreover, higher bytes of addresses are omitted, when they are the same as previously logged ones.

PT Packets There are a lot of different Intel PT packets, each one associated with a different control flow tracing attribute. The main types of packets are the following:

- **PGE** or Packet Generation Enable packets provide the IP at which the tracing of the program begins.
- **PGD** or Packet Generation Disable packets mark the end of a program's tracing.
- **TIP** or Target IP packets are the packets that are created when the program's execution reaches an indirect call, a return, an exception or an interrupt. They record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These packets can contain the IP, although that IP value may be compressed by eliminating upper bytes that match the last IP.

- **TNT** packets track the direction of up to 6 conditional branches, containing information about whether that conditional branch was taken or not taken.
- **FUP** or Flow Update packets: provide the source address for asynchronous events (interrupts and exceptions).
- **PSB** or Packet Stream Boundary packets: are unique patterns in the output log to serve as sync points for software decoders.
- **OVF** or Overflow packets: are sent when the processor experiences an internal buffer overflow, resulting in packets being dropped. This packet notifies the decoder of the loss and can help the decoder to respond to this situation.
- **TSC** or Time-Stamp Counter packets: aid in tracking wall-clock time, and contain some portion of the software-visible time-stamp counter.

Filtering PT includes various facilities for filtering packets, before they are logged. The most interesting ones include filtering based on the special register `CR3`, which essentially configures PT to only log packets for a specific virtual address space, i.e., a specific process. Moreover, it is possible to set two virtual address ranges of interest, which cause PT to automatically activate or deactivate, upon entry and exit of execution into those ranges, respectively.

2.4.4. ARM CoreSight

The ARM CoreSight architecture [4], depicted in Fig. 3, is an infrastructure for facilitating real-time debugging and tracing on ARM processors. A variety of modules, referred to as macrocells, are available providing different capabilities. Processor vendors can select which modules to incorporate in their processors based on their requirements. Macrocells are can be configured and activated programmatically from a program running on-chip by setting its coprocessor registers using instructions `MRC` and `MCR`. In recent version, these registers can be also memory mapped, while more conventional external interfaces, like JTAG, can also be used. Here, we present some of the CoreSight components that we plan to make use of.

Program Trace Macrocell (PTM) ARM's PTM is similar in many ways to Intel's PT. It enables control-flow tracing by generating a stream of packets that encode the result of control-flow instructions. We can use this stream in a similar way to reconstruct control-flow and perform DDFT in-parallel with an application. The module also provides filtering capabilities to only activate program tracing for a range of instruction addresses.

Embedded Trace Macrocell (ETM) ARM's ETM is a more advanced debug module that provides execution and data tracing. Similarly to the PTM it produces a highly-compressed stream of packets that contain information like the instruction addresses executing and the addresses of data being read/written. Address and data comparators may also be available to filter the instructions that will produce tracing data based on the address of the instruction, the value of the data being accessed, etc. Instruction tracing is actually similar to PTM, since it involves tracing

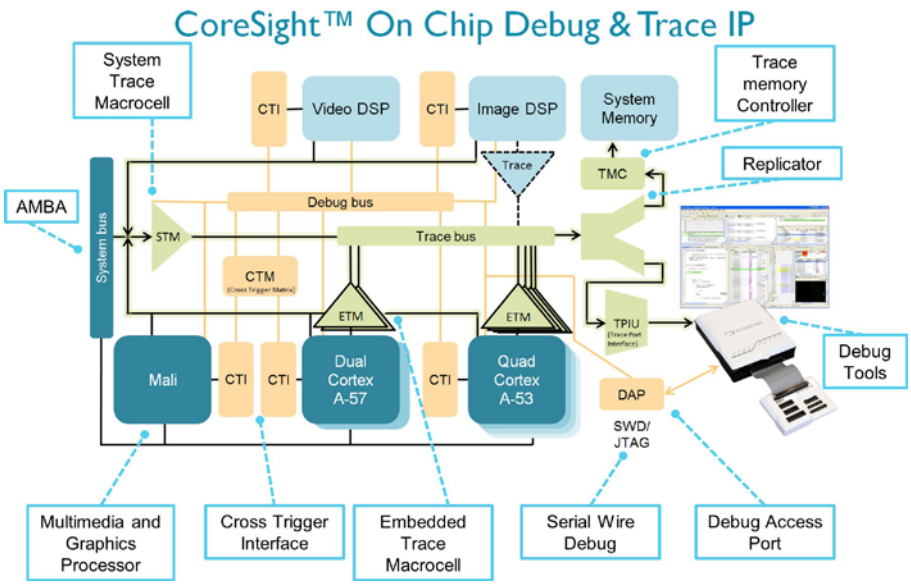


Figure 3: ARM CoreSight Architecture Intellectual Property (IP)

(This picture is property of ARM ltd.)

branch instructions whose outcome cannot be statically determined (i.e., indirect and conditional control-flow instructions). It also involves recording context changes in the executing process and tracking the execution of individual Java instructions, as ARM processors are able to directly execute certain Java bytecodes. Data tracing involves recording the memory address involved in the executing instruction. If for some reason the ETM cannot immediately record addresses, it can also record it out-of-band, while it also generates the appropriate packet to notify the stream reader of that event. If an instruction is actually using multiple memory addresses, only the first address is recorded.

System Trace Macrocell (STM) The STM [3] is designed for enabling instrumentation of an application. Applications developers can define templates of trace statements that contain placeholders for run-time data variables. At run time, the STM will use these templates to produce tracing packets based on the template and the values observed, attaching cycle accurate timestamps at the same time. The STM can be stacked with other debugging modules, such as the PTM and ETM, to timestamp their trace data.

Capturing Trace Data through the TPIU The above components are connected on the AMBA AXI interconnect bus, which provides fast connectivity between cores and coprocessors on ARM chips. The trace data are written on the advanced trace bus (ATB), which provides various options for capturing them. Older ARM chips typically route the trace data to a debug access port (DAP), which can be captured by an external debugger through an interface like JTAG. They can be also stored in a small on-chip buffer, the embedded trace buffer (ETB), which can also be

accessed only though the JTAG interface. However, ARM CoreSight also provides alternatives; trace data can be routed to a trace port using the trace port interface unit (TPIU). A trace port can lead to off-chip pins where an external trace port analyzer (TPA) can capture them, or can be routed on on-chip ports leading to a coprocessor hosted on the same chip. Such coprocessors need to be designed-in by the vendor. Prototyping such system-on-chip designs is also possible using development boards that combine an ARM processor with a field-programmable gate array (FPGA) [2, 20, 22, 34].

Capturing Trace Data through the TMC The trace memory controller (TMC) [3], included in designs with an STM, is a successor to the CoreSight (ETB). It extends the capture of trace streams to system memory, through the embedded trace router (ETR), or to a dedicated embedded trace FIFO (ETF) on chip. As a result, processors with STM and TMC can capture data from PTM and ETM on system memory, where they can be decoded and analyzed by another core. TMC essentially enables the use of trace data without slow off-chip peripherals or custom on-chip coprocessors. Moreover, it is also transparently stalls the processor, if new data are going to overwrite existing data, i.e., when data are produced much faster than they are consumed.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Decoupling Dynamic DFT from Application Execution with ShadowReplica

DDFT techniques track the flow of data throughout the execution of a program by propagating tags, also referred to as labels, that have been associated with particular data. Decoupling DDFT from the core executing the application and running it on another core, in parallel, requires access to the instructions being executed, as well as certain run-time information, like computed memory addresses used in read and write operations. For example, the semantics of the x86 instruction `mov eax, ebx` mandate that a tag should be propagated between two registers, when it executes, while no run-time information is required. However, when an instruction like `mov eax, [ecx+ebp*4]` executes, DDFT must propagate a tag between a memory location and a register, requiring the address computed by `ecx+ebp*4` at run time.
The major challenge in decoupling DDFT is efficiently obtaining this kind of run-time information from the application. The data required are: (i) tag propagation semantics based on program instructions, (ii) addresses used in memory access instructions that are generated at run-time and cannot be inferred, and (iii) control-flow decisions like conditional branches and indirect control-flow transfers. With this information, we can track data independently of the DFT "flavor" we are interested in. For example, we can track data at different granularity, track only explicit or also implicit dependencies, apply a variety of security policies, etc.

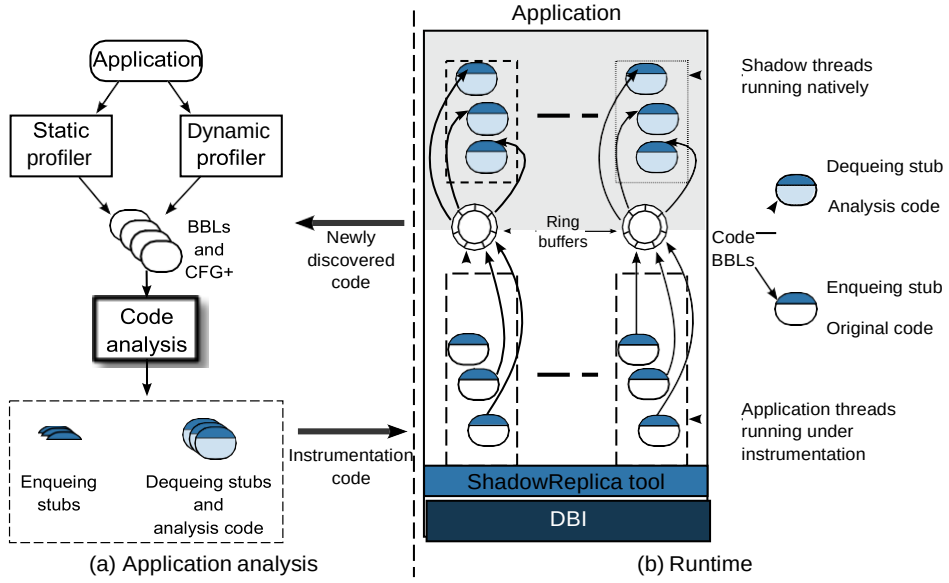
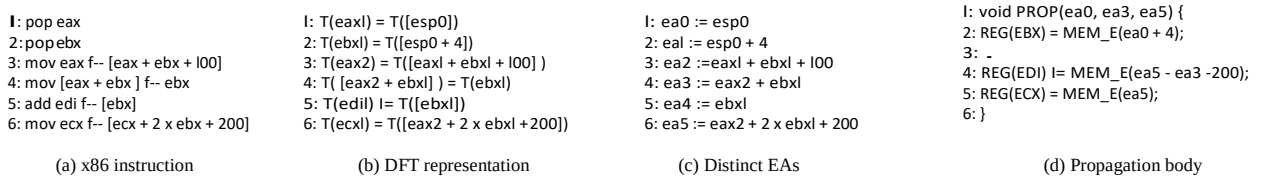


Figure 4: ShadowReplica Architecture

In previous work [16], the PI developed an architecture based on Pin to decouple DDFT and run it in parallel to the application, which this project enhances. An overview of ShadowReplica is depicted in Fig. 4(b). For each thread of an application, a shadow thread is spawned to perform the necessary DDFT computations. Pin is used to instrument the target application with logic



(a) x86 instruction (b) DFT representation (c) Distinct EAs (d) Propagation body

Figure 5: BBL Transformation Example During Code Analysis

that extracts and transmits the addresses used in memory access instructions and control-flow decisions to the shadow thread, so that it can propagate tags and apply policy decisions.
To operate efficiently ShadowReplica builds on information extracted during an offline analysis stage, which is sketched in Fig. 4(a). This phase begins by disassembling a binary application to extract its basic blocks (BBL), that is, blocks of instructions terminated by a control-flow transition, as well as a partial control-flow graph (CFG) that shows how some of these basic blocks are connected. To extract this information we can utilize tools static tools, like the IDA pro disassembler [14], and dynamic tools, like Intel's Pin framework [21]. We analyze this information to generate optimized code for the DFT logic (i.e., for propagating tags) in the shadow process and for enqueueing and dequeuing the data necessary. Note, however, that we do not require that this analysis is perfect, as we can also analyze the application just-in-time.

3.1.1. Control Flow Recording

The most straightforward approach to replicate control flow involves enqueueing a unique basic block id (BBID) for every BBL being executed. Unfortunately, simply doing this is *too* costly. Control-flow transitions in x86 architectures can be classified in three categories: (a) direct jumps, (b) direct branches, and (c) indirect jumps. For direct jumps, BBIDs for successor BBLs can be excluded from logging, since there is only a single, fixed exit once execution enters into BBL0. Direct branches can have two outcomes. They are either taken, or fall through where execution continues at the next instruction. We exploit this property to only enqueue a BBID, when the least frequent outcome, according to our dynamic profiling, occurs. We use the *absence* of BBL3's id to signify that BBL4 followed as expected. Note that if a BBL has two predecessors and it is the most frequent path for only one of them, we log its BBID. Last, for indirect jumps we always record the BBID following them, since they are hard to predict. Fortunately, the number of such jumps are less compared to direct transfers.

3.1.2. Recording Effective Addresses

A naive approach to obtain the memory addresses accessed by the application in the shadow process involves recording all of them. However, as such operations are very common, we can exploit locality in memory accesses to minimize the amount of information that needs to be exchanged. For this, we intend to build on prior work [16, 17].
We begin by extracting data dependency semantics from basic blocks during offline analysis, which allow us to identify the memory locations of interest. In Fig. 5, we show an example, where we extract the data-tracking semantics (b) from a block of x86 instructions (a). Each reg-

ister is treated like a versioned variable and any update operation causes the creation of a new version, e.g., the `POP` instruction generates `eax1` in line 1. The function `T()` corresponds to the tag for a particular register or memory location, which, for simplicity, we assume to be a binary tag in this example. For example, in lines 1 to 4 and 6 of Fig. 5 (b) a tag is copied, while in line 5 two tags are combined using the OR ( ) operator. When looking at the effective addresses (EA) used in this representation, Fig. 5 (c), we see that initially we would have to transfer six of them to the analysis thread. We can perform the following optimizations to reduce this number.

Intra-block Optimization Within a basic block, we search for effective addresses that correlate with each other to identify the minimum set required that would correctly restore all of them in the analysis thread. For instance, we only need to enqueue one of `[esp0]` or `[esp0 + 4]` from Fig. 5 (b), as one can be derived from the other by adding/subtracting a constant offset.

DFT Optimization This optimization identifies instructions, and consequently memory operands, which are not going to be used by the DFT logic in the shadow process, by applying compiler optimizations, such as dead-code elimination and data-flow analysis, against our DFT-specific representation of code. For instance, in Fig. 5 (b) we determine that the propagation in line 1 is redundant, as its destination operand (`eax1`) is overwritten later in line 3, before being referred by any other instruction. This allows us to ignore its memory operand `[esp0]`. In our example, this reduces the number of memory addresses from six to three, see Fig. 5 (d).

Inter-block Optimization We extend the scope of the intra-block optimization to cover multiple blocks connected by control transfers. This implements backward data-flow analysis [1] with the partial CFG gathered during profiling. We begin by defining the input and output variables for each basic block. We then produce a list of input and output memory operands which are live before entering and when leaving a basic block. Using our representation, input memory operands are the ones with all of its constituent register variables in version 0, and output memory operands are the ones that have all of its constituent register variables at their maximum version. In our example in Fig. 5, the inputs list consists of `[esp0]`, and the outputs list includes `[esp0]`, `[ebx1]`, `[eax2 + ebx1]`, and `[eax2 + 2 × ebx1 + 200]`. If all the predecessor basic blocks in the CFG contain `[esp0]` in their outputs list, the block can harmlessly exclude it from logging because its previous value is still valid. The optimization has greater effect as more inputs are found on the outputs lists of a block's predecessors.
Since we only have a partial CFG of the application, it is possible that at runtime we identify new execution paths that invalidate the backwards data-flow analysis. To tackle this issue we make inter-block optimization more conservative using two heuristics. First, if we find any indirect jumps to a basic block, we assume that others may also exist and exclude it from optimization. Second, we assume that function entry point, may be reachable by other, unknown indirect calls, and we also exclude them. We consider these measures to be enough to cover most legitimate executions, but they are not formally complete, and as such may not cover applications that are buggy or malicious, in which case this optimization can be disabled. Note that with the latter, we are not referring to vulnerable applications that may be compromised, an event that can be prevented by DTA, but malicious software that one may wish to analyze.

3.1.3. Ring Buffer

ShadowReplica uses an N-way buffering scheme [36], where the application and shadow process share the buffers, to transfer data efficiently. In this scheme, a ring buffer is divided into N separate sub-buffers, and the application and shadow processes work on different sub-buffers at any point of time, maintaining a distance of least as much as the size of a single sub-buffer. Also, as part of enqueueing in the ring buffer, we need to check whether space is available. We eliminate some of these checks to further reduce overhead. During the static analysis of an application we identify chains of basic blocks, where we can deterministically establish their length, to collapse multiple checks to a single one, performed when entering a chain. For this approach to be effective we also need to identify and avoid over-optimizing loops, while, we can also allocate a write-protected memory page at the end of the ring buffer that will generate a page fault, which can be intercepted and handled, to safeguard from errors.

3.1.4. Recording System Calls

System calls are used to read and write data from the operating system. As such, they play an important role in DFT, as they signify points where data may need to be tagged and policies enforced. Based on the analysis being applied, we can enqueue system call arguments and return values along with memory addresses. For instance, opening and reading from a sensitive file needs to be communicated to tag the correct number of bytes with the appropriate tag.

3.1.5. DFT Logic

During offline analysis, we generate tag propagation code blocks for each of application's basic blocks. Each of these, consumes the effective address produced by the instrumented application to propagate tags. The DFT logic will be generated based on the methodology introduced in one of the PI's previous works [17]. Briefly, this involves extracting tag propagation semantics and representing them in a DFT-specific form, which is susceptible to multiple compiler-inspired optimizations that aim at removing propagation instructions that have no practical effect or cancel out each other, as well as reducing the number of instructions required for propagation by grouping them together. At the end of each block and after filtering Pin-related control-flow transfers, we use the Intel PT data stream to decide which block to execute next.

Generic Block Handler Application code that was not identified offline will be processed by a generic handler. In this case, we disassemble the newly discovered code and analyze it on-the-fly. The newly found code blocks are cached, so this process needs to be run only once.

3.1.6. Multithreaded Applications

For multithreaded applications, the shadow process will spawn one thread for each corresponding application thread. Similarly to control-flow data, we can use a separate data structure for each thread to record memory-address data, so no demultiplexing is necessary. To correctly handle critical sections, where only one thread can be active at a given time, we rely on existing locking facilities used by threads. For example, a POSIX thread can use a mutex to ensure exclusive access to a critical section. We record the order by which threads obtain this lock and enter the

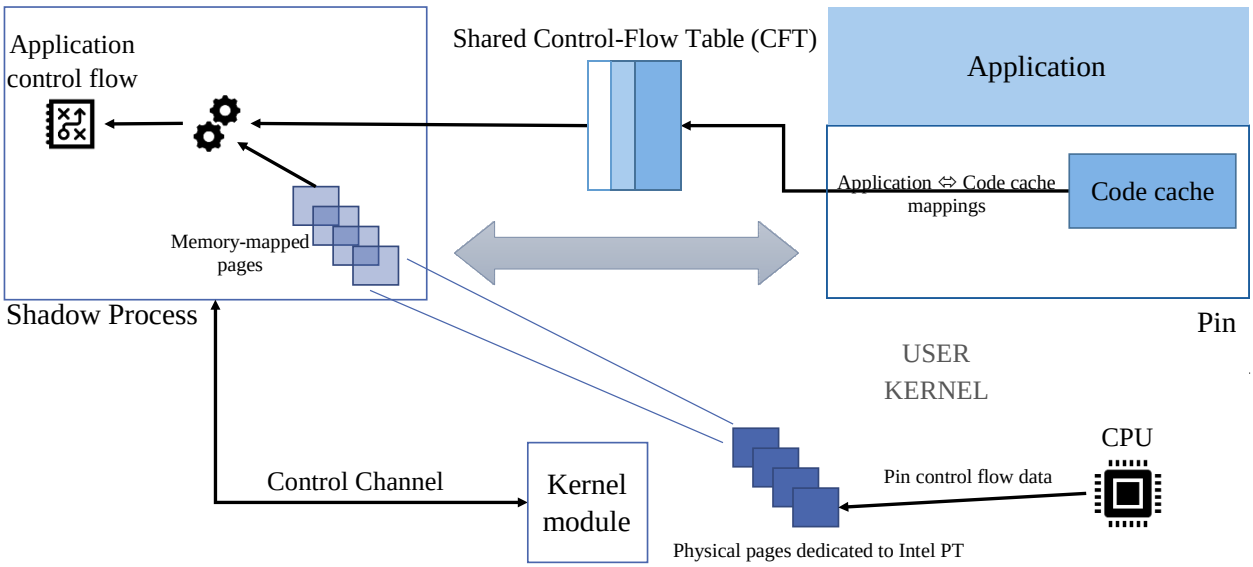


Figure 6: JITrace Architecture

critical region, by wrapping lock-handling functions, so that we can replicate the locking order in the shadow process. This strategy enables us to maintain the correctness of tag propagation, assuming that the tracked application correctly uses locks to access shared data.

3.2 DDFT Acceleration using Intel PT

We developed a new system, coined JITrace, that utilizes Intel's Processor Trace (PT) technology to reduce the overhead of DDFT architectures, such as ShadowReplica, described above. Applications running on top of Pin execute as just-in-time (JIT) generated code, cached within a *code cache*. JITrace uses the data produced by Intel PT, which correspond to Pin's and code-cache code execution, to reconstruct the original control-flow of the application, as if Pin was not present. By offloading control-flow recording to the hardware, JITrace reduces the amount of data that need to be transferred through Pin-based instrumentation to the shadow process for performing DDFT.

Figure 6 illustrates the architecture of JITrace. A kernel module is employed to manage Intel PT, which is a privileged hardware feature, and to efficiently expose PT-generated data to user space. The kernel module is controlled by a shadow process, which also collects PT data and reconstructs application control flow. This process is launched along side the Pin-hosted application and is responsible for initializing the PT-based tracing of the Pin process. The CPU writes PT data directly to a set of buffers, which are also mapped into the shadow process. After initialization, the shadow process receives notifications from the kernel module, whenever a buffer has been filled by the CPU, and proceeds to decode the data contained within.

The data obtained from PT does not describe the execution of the application, but that of Pin and its code cache. JITrace maps the execution trace described by PT data to the actual application running on top of Pin. We utilize Pin's code cache APIs to analyze the code generated by Pin and

map code blocks to that of the applications. We encode these mappings into a control-flow table (CFT) that is shared between shadow process and Pin. Using the CFT and decoded PT packets, the shadow process is able to successfully reconstruct the execution of the application. Finally, an auxiliary channel between the shadow and Pin processes is used to transfer low-frequency data, such as notifications that the execution has entered/exited the code cache. Below we describe the various components of JITrace in more detail.

3.2.1. Mapping Pin's Code Cache to Application Code

To create these mappings, we need to examine all of the code entered in the code cache by Pin, before it is executed. Fortunately, Pin provides various APIs for monitoring the code cache.[1] In particular, we use the `CODECACHE_AddTraceInsertedFunction()` API to be notified whenever a new trace is inserted in the cache. The trace can be disassembled using another Pin API, `INS_Disassemble()`. Note that the traces being inserted in the cache, include both application code, as well as any other instrumentation that has been injected using Pin. For example, the instrumentation required by ShadowReplica to send memory addresses to the shadow process. This fact, along with the certain Pin limitations and optimizations, complicate the mapping process, which we describe below.

Every trace inserted in the code cache is composed of one or more BBLs. In the simplest case, we can rely on Pin APIs to resolve if a code cache basic block BBL_{CC} corresponds to an application basic block BBL_{APP}. For example, `CODECACHE_OriginalAddress()` returns the original address of a code-cache instruction. With this information, we can match BBL_{CC} and BBL_{APP}, since it will fail for instructions that were added by Pin (e.g., due to instrumentation). Each BBL_{CC} can correspond to only one BBL_{APP}, as there can be no control-flow within a BBL by definition. However, the opposite is possible, one BBL_{APP} can be split in two BBL_{CC}, because of instrumentation. We easily handled this case by ignoring the second BBL_{CC} match.

BBL_{CC} that contain a single instruction, which by definition will be a conditional branch, require special handling, as they do not contain any original application instruction. This is because Pin replaces the target of the branch with an appropriate value pointing in the code cache. To identify the BBL_{APP} for these blocks, we also disassemble and analyze all application code, as it is discovered by Pin. We use `TRACE_AddInstrumentFunction()`, which installs a callback that *saves* each trace of the original application $TRACE_{APP}$, before it is inserted in the code cache. We match single-instruction BBL_{CC} by looking at the opcode used in the conditional branch and its relative position in the trace and compare them to unmatched BBL_{APP} in the trace. We also use the saved traces to confirm that the mappings we calculated are correct, by checking that no block was left unmapped.

BBL_{APP} that include instructions with the prefix `REP` also require special handling. This prefix executes an instruction multiple times, based on the value of a register. Pin expands such instructions into a loop, implemented by multiple BBL_{CC} in *two* traces. First, the code of the BBL_{APP} itself is split in two blocks, the first (*block1*) containing all the instructions before the `REP`-prefixed instruction, and the second (*block2*) containing the `REP`-prefixed instruction (without its prefix)

[1] The API was deprecated after Pin 3.2.

CODE CACHE ADDRESS	ORIGINAL BBL ADDRESS	CONDITIONAL	TARGET ENTRY

Figure 7: Control-flow Table Format

that ends the BBL. The loop is actually implemented in a different trace, while *block2* seems to be an optimization for loops that will execute only once. When a REP-prefixed instruction is the only instruction in the BBL, a mapping is created for both BBL_{CC} pointing to the same BBL_{APP}. Pin also splits others instructions, like CPUID and POPF, into multiple BBL_{CC}.

3.2.2. Populating the Control-flow Table

The control-flow table (CFT) needs to contain sufficient information for: (a) reconstructing the control-flow of Pin's code cache, and (b) translating that to the control-flow of the application run over Pin. The first requirement mandates that we populate the CFT with information about everything that executes in the code cache, not just the BBLs mapped to application BBLs. Hence, we insert an entry in the table for every BBL in the code cache. To then map code-cache to application execution, for each BBL_{CC} that has been mapped to a BBL_{APP}, we additionally include the latter's information (e.g., its address).
Figure 7 shows the format of a CFT entry. The following information is stored in each entry:

- CODE CACHE ADDRESS This field holds the address of the branch instruction that ends the BBL_{CC}.
- ORIGINAL BBL ADDRESS If this BBLs has been mapped to a BBL_{APP}, this field contains its address, or -1 otherwise.
- CONDITIONAL This flag indicates of the BBL ends in an conditional branch (TRUE) or an a direct/indirect branch.
- TARGET ENTRY This fields points to a target entry in the CFT. If the exiting branch of the BBL was a conditional, this points to the entry where execution is transfer if the branch is taken. The entry below corresponds to the BBL that will execute if the branch is not taken. For BBLs that are end in a direct branch, this field points to the BBL where control will be transferred. The field is -1 for blocks that end with an indirect-transfer.

Entries for Pin Trampolines/Glue Code Pin owes its performance to *hot trace linking*, an optimization that targets frequent transitions between traces, usually when indirect transfers (e.g., through the RET instruction) are involved. When a specific address is repeatedly targeted, Pin updates the code cache with a check and direct jump to the target trace, instead of actually exiting the code cache, verifying that the target code has been already translated, etc. Linking introduces new code in the code cache, which is treated specially by Pin and is opaque to the developer.

We analyzed this glue code and identified that it comprises small trampolines pieces of code, which are both used for trace linking and to efficiently instrument the application. To handle these trampolines for each BBL_{CC} that ends with a conditional or direct branch, we attempt to disassemble the target address, if it is within the memory ranges that Pin uses for the code cache. If we find valid code that ends in a direct `JMP` or `RET` instruction, we generate a CFT entry for the (previously unknown) block.
Oftentimes, Pin enters the code cache directly in such trampolines or glue code. We can only discover such code by monitoring where the code cache is entered and looking for unknown code chunks. We use Pin's `CODECACHE_AddCodeCacheEnteredFunction` API to monitor entries to the code cache. If BBL_{CC} had not been previously inserted at that address, we jump in and disassemble the code found there before allowing the code cache to execute.

Pin Backpatching/Trace Linking In order to reduce the number of accesses to the code cache, Pin backpatches traces that are connected. In particular, links are created to and from other traces, mainly based on their relationship in the original code. For example, traces are linked when they are one after another in the program. The links are created by overwriting the exit instruction of the source trace with a conditional or direct jump to the destination trace. This kind of linking is available to the developer, through the `CODECACHE_AddTraceLinkedFunction()` API. We initially handled these kind of links by storing them in queue and applying them updates on the CFT in batch to avoid using stale entries during the reconstruction. However, now we are experimenting with a new design, where updates are made directly on the CFT and we take advantage of the characteristics of traces before linking (e.g., they lead to an exit from the code cache) to avoid concurrency issues.

3.2.3. Setting up and Using Intel PT

The shadow process is responsible for setting up PT, reading, and processing PT data, always through the kernel module. Initialization, involves starting PT and configuring it, so it only tracks the Pin process and to filter out any data that do not correspond to the memory area that can host code-cache blocks. After setup, the shadow process issues an `mmap()` system call to the kernel module to map the pages that will contain PT data, as they are produced by the CPU. Data decoding occurs in a loop, where an `ioctl()` system call is used to signal that the process is ready to process data. Whenever one of the pages assigned to PT are filled, the module receives an interrupt, which it uses to "return" the given page, through the `ioctl()`, to the shadow process for decoding. The kernel module ensures that the processor never overwrites data that have not been processed by user space. It keeps track of the page being currently processed and where the processor is writing, and ensures that a few pages are always available between the two. If not, it stops the Pin process by issuing a `SIGSTOP` signal. Pin is resumed, when more pages have finished processing and are made available.

3.2.4. Control-flow Reconstruction

The shadow process models Pin's execution state as *IN* or *OUT* of the code cache. Pin enters the code cache and begins executing using and indirect control-flow transfer. We have observed

that on 64-bit systems this occurs through an indirect jump from Pin into the trace to be executed, while on 32-bit systems the indirect jump is to a trampoline in the code cache that directly jumps to the targeted trace. This trampoline is updated at code cache entry. In both cases, this causes the generation of *target IP (TIP)* packet. TIP packets are generated when an indirect control-flow transfer occurs and we use, when the state is *OUT* , them to identify an entry in the code cache, switch to *IN* state, and begin reconstruction. Exits from the code cache result in a *Packet Generation Disable (PGD)* packet, because of the way we configured Intel PT. We use this packet to toggle execution state.

Whenever we enter the code cache, or when we have to deal with an indirect or asynchronous (e.g., signal) transfer, we use the targeted address to lookup the corresponding entry in the CFT. That entry corresponds to the BBL_{CC} that executed on the Pin process. To perform the lookup, we utilize a hash table for mapping code cache traces to CFT entries. Each time a new trace is identified an entry is created in this hash table, along with all the other entries created in the CFT. To keep track of the current CFT entry corresponding to the Pin's execution in the code cache, we maintain a cursor, CUR_{CFT} , to the active entry. Once the code cache is entered, we follow the meta data stored within to determine our next action. For entries that correspond to block ending with direct transfers ($CONDITIONAL = 0 \wedge TARGET\ ENTRY \neq -1$), we just update the CUR_{CFT} to point to *TARGET ENTRY* . For blocks ending with a conditional branch ($CONDITIONAL = 1$), we use PT data, part of the a TNT packet, to determine the next CFT entry, which is other the next entry or *TARGET ENTRY* . For indirect transfers ($CONDITIONAL = 0 \wedge TARGET\ ENTRY = -1$, we perform a lookup, as stated earlier.

As we traverse the CFT with the help of PT data, for every entry touched that corresponds to a BBL_{APP} ($ORIGINAL\ BBL \neq -1$), we discover an application BBL that executed, essentially, reconstructing the original control flow.

3.2.5. Support for Multi-threaded Applications

JITrace includes preliminary support for threads. In particular, we have implementing an extension to the kernel module, which hooks the context-switching routines to be notified whenever a thread is scheduled to run on a core. The module then ensures to activate and deactivate PT, whenever a traced thread is scheduled to run. It also ensures that each core uses a separate set of pages for writing PT packets. This process involves saving and restoring the Machine Specific Registers (MSR) on the CPU, across context switches. The principle is the same as saving and restoring general purpose registers across context switches. Further work is required to expose and use this functionality in the shadow process.

3.3 DDFT Acceleration using ARM ETM

Hardware-enabled execution tracing, offered by components such as ARM's ETM, can help reduce the overhead of DDFT in two ways by offloading both control flow and memory access reconstruction to the hardware. We have performed an initial investigation using the i.MX51 evaluation toolkit [26], which features an ARM A8 core, towards achieving this goal. A8 processors contain an embedded trace macrocell (ETM) provides a stream that includes the compressed

addresses of executed instructions, so it is similar in this fashion to Intel PT. Additionally, it can trace memory accesses by logging the address of the data accessed by instructions (e.g., the address read or written). Hence, the ETM enables us to obtain the effective address of memory operations from the shadow process with close to zero overhead.

Execution Trace Storage Unlike Intel PT, A8 processors only support storing trace data in an onboard buffer of 4KB. To make things worse, when the buffers is full, the processor simply wraps around and overwrites previously stored data, while there is not mechanism to filter the data, that are going to be stored. That is, all processor instructions are logged, not just that ones of an application of interest. We are investigating an approach for mitigating these limitations. In particular, we are looking into using performance counters to trigger an interrupt after a certain number of instructions execute. The goal is to use this interrupt to check the status of the trace buffer and copy its contents to memory, before they are overwritten. Additional research in this direction is required.

3.4 Generating DFT Logic

We have made some preliminary work on a framework that will automatically generate DFT logic in an instruction-set independent manner. The framework, named *VEX2TRACK*, defines data-flow tracking semantics for basic blocks expressed in VEX intermediate representation (IR). Existing tools [29] can translate a variety of instruction sets, including x86, x86-64, and ARM, to VEX IR allowing us to easily port our approach to other architectures.
VEX2TRACK uses defines data-flow tracking semantics per VEX instruction and generates LLVM IR instructions that implement the tag propagation. These instructions are compiled using LLVM, applying common compiler optimization to improve tag-propagation performance. We are currently representing registers and memory tags (shadow memory) as fixed size arrays. To apply generated code in decoupled manner, as in ShadowReplica, we also need to calculate the addresses used in memory access operations. The approach we are currently exploring is transferring register values whenever they are loaded from memory, and generating instructions for propagating their values. We rely on compiler optimizations to eliminate some of the latter, when the values generated are not used as memory addresses. Further research is required in this direction.

4.0 RESULTS AND DISCUSSIONS

4.1 JITrace

We measured the performance of JITrace using the 32-bit SPEC CINT2006 benchmark suite. Experiments were run on x86-64 Linux v4.15.0, on an Intel(R) Xeon(R) CPU E3-1225 v5 3.30GHz CPU and 32GB of RAM. Our goal is to establish, if using Intel PT to obtain the control flow of the application is faster, than doing it through Pin (using instrumentation). Table 1 shows the overhead of JITrace, normalized using two baselines: Pin without any tool and Pin with a tool reconstructing the control flow of the application. The numbers shown are the average after running each benchmarks twice. We find that on average, JITrace is 1.68 times slower, than using instrumentation to obtain the control flow of the application. While in some benchmarks, like 401.bzip, JITrace is faster, it is almost three times slower in in 464.h264ref. We have been investigating the cause of the overhead, and we have identified that the enormous number of PT packets generated and the overhead associated with traversing the CFT are the biggest contributors. Additional research is required in exploiting common patterns in PT data and producing a highly optimized decoder and CFT. Another promising direction is using an FPGA to decode the packets.

Correctness We established the correctness of the traces produced by JITrace, by comparing them with the ones produced using a Pin tool. We found that for all benchmarks, we are able to correctly reconstruct the control flow. However, we have detected cases where JITrace occasionally reports one extra BBL when running 400.perlbench. We are still investigating this bug.

Table 1: Normalized Overhead of JITrace with SPEC CINT2006 Benchmark

Benchmark	Baseline	
	Pin no tool	Pin-based reconstruction
400.perlbench	3.90	1.15
401.bzip	2.09	0.79
403.gcc	8.23	2.75
429.mcf	2.09	1.07
445.gobmk	5.42	1.85
456.hmmer	2.12	0.97
458.sjeng	8.91	2.39
462.libquantum	5.33	2.37
464.h264ref	5.37	2.87
471.omnetpp	3.57	1.57
473.astar	2.96	1.38
483.xalancbmk	3.61	1.03
Average	4.47	1.68

5.0 CONCLUSIONS

In this project we developed new technologies that will enable low-overhead, fine-grained, dynamic data-flow tracking (DFT) capabilities for the development of security, privacy, and other applications. Our focus was on taking advantage of debugging features found in modern CPUs to improve the efficiency of decoupling dynamic data-flow tracking (DFT) from application code and running it in parallel.

We have developed a new system, called JITrace, that utilizes HW data to reconstruct the control flow of applications running on Pin, a DBI framework employing JIT. JITrace can work together with systems, like ShadowReplica, which were developed in previous projects, to transfer part of the control flow reconstruction functionality from software to hardware. Our findings indicate that additional work is required for developing systems that can efficiently consume the massive amounts of data produced by the hardware, by using novel designs and programmable hardware, such as FPGAs. Both Intel and ARM have released processors with on-board FPGAs, which can be used to prototype such designs, that could lead to new system-on-chip designs further in the future.

We have also conducted research in using more extensive hardware tracking, like the execution tracing offered by some ARM processors. While this feature is powerful, these processors are more prevalent in embedded systems and are, hence, more lightweight and with less resources. Additional research is required to investigate efficient ways to overcome the resource limitations in such architectures.

Finally, we have begun developing a framework that can generate the necessary logic for implementing DDFT applications quickly. Our approach, takes advantage of tools that translate architecture-specific code into an intermediate representation and works with them, so applications can be ported with little effort. Moreover, the generated code is friendly to compiler-based optimizations, which greatly reduce the number of instructions required for the DDFT applications. Additional research is required in smoothly integrating the application into frameworks like JITrace to facilitate deployment.

6.0 REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] Altera. Arria 10 ARM-Based SoCs. `https://www.altera.com/products/soc/portfolio/arria-10-soc/overview.html`, 2016.

[3] ARM. Better trace for better software, 2010. Introducing the new ARM CoreSight System Trace Macrocell and Trace Memory Controller.

[4] ARM. CoreSight Debug and Trace, 2015. `http://www.arm.com/products/system-ip/debug-trace/`.

[5] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proc. of VEE*, 2012.

[6] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceeding os Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[8] J. Chow, T. Garfinkel, and P. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Procedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[10] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 221–232, 2004.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 482–493, 2007.

[12] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 137–148, 2010.

[13] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[14] Hex-Rays. The IDA Pro Disassembler and Debugger, cited Aug. 2013. `http://www.hex-rays.com/products/ida/`.

[15] James Reinders (Intel). Processor tracing, Sep 2013. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`.

[16] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2013.

[17] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, February 2012.

[18] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 121–132, March 2012.

[19] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[20] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek. Towards a practical solution to detect code reuse attacks on ARM mobile devices. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Jun 2015.

[21] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of PLDI*, 2005.

[22] MICRO/SYS). SBC1652. `http://www.embeddedsys.com/subpages/products/sbc1652.shtml`, 2014.

[23] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2007.

[24] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[25] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[26] NXP. MCIMX51EVKJ: i.MX51 Evaluation Kit, 2019. `https://www.nxp.com/design/development-boards/i.mx-evaluation-and-development-boards/i.mx51-evaluation-kit:MCIMX51EVKJ`.

[27] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri. SIFT: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF)*, pages 37:1–37:11, 2011.

[28] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *ACSAC*, 2010.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive

Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, pages 138–157, 2016.

[30] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[31] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, 2004.

[32] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 109–120, 2009.

[33] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2007.

[34] Xilinx). Zynq-7000 all programmable SoC. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`, 2016.

[35] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[36] Q. Zhao, I. Cutcutache, and W. Wong. PiPA: pipelined profiling and analysis on multi-core systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008.

[37] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. In *SIGOPS Oper. Syst. Rev.*, 2011.

LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS

Acronyms and Abbreviations

IP – Intellectual Property
DFT – Data Flow Tracking
DDFT – Dynamic Data Flow Tracking
PT – Processor Trace
DBI – Dynamic Binary Instrumentation
VM – Virtual Machine
VMM – Virtual Machine Monitor
AP – Application Processor
JIT – Just-in-Time
PTM – Program Trace Macrocell
ETM – Embedded Trace Macrocell
STM – System Trace Macrocell
ATB – Advanced Trace Bus
ETB – Embedded Trace Buffer
DAP – Debug Access Port
TPIU – Trace Port Interface Unit
TPA – Trace Port Analyzer
FPGA – Field-Programmable Gate Array
TMC – Trace Memory Controller
ETR – Embedded Trace Router
ETF – Embedded Trace FIFO
CFT – Control-Flow Table
BBL – Basic Block
BBID – Basic Block ID
CC – Code Cache
TIP – Target IP
PGD – Packet Generation Disable
PGE – Packet Generation Enable
TNT – Taken-not-Taken
SoC – System on Chip
IR – Intermediate Representation
EA – Effective Address

Symbols

BBL_{APP} – Application Basic Block
BBL_{CC} – Code cache Basic Block