WIP: InSight - A CoreSight Trace Interpreter for Dynamic Information Flow Tracking

Quentin Ducasse[1 [0000−0001−9927−675X]], Guillaume Hiet[1 [0000−0002−7176−9760]], Volker Stolz[2 [0000−0002−1031−6936]], and Pierre Wilke[1 [0000−0001−9681−644X]]

[1] CentraleSupélec, Inria, CNRS, IRISA, Av. de la Boulaie, Cesson-Sévigné, France
{name}.{surname}@centralesupelec.fr
[2] Western Norway University of Applied Sciences, Inndalsveien 28, Bergen, Norway
volker.stolz@hvl.no

Abstract. Dynamic Information Flow Tracking (DIFT) associates metadata with memory and registers and enforces propagation rules at run time to detect violations such as control-flow hijacking, use of uninitialized data, and data-only attacks. This work-in-progress presents a DIFT framework for AArch64 built on ARM CoreSight debug components. It combines LLVM-based compiler instrumentation with InSight, an offline interpreter for CoreSight traces and metadata annotations. The system supports tag propagation and rule enforcement over instrumented binaries. Preliminary results demonstrate its ability to detect control-flow violations. Future work includes kernel-level tag initialization via Linux Security Modules and the development of a hardware co-processor for online DIFT enforcement.

1 Introduction

Memory safety vulnerabilities are among the most common vulnerabilities found in binaries, arising from out-of-bound accesses, use of uninitialized resources, or dereferencing of dangling pointers. These result in memory errors and invalid pointers, and provide attackers with tools to modify the running program state. The effects range from information leakage to privilege escalation and arbitrary code execution in the worst cases. Although Szekeres et al. [20], in their seminal paper *"Eternal War on Memory"*, present the main types of exploit and corresponding countermeasures, deploying a complete set of defenses to mitigate real-world threats remains complex and incurs unacceptable overheads in production.

Dynamic Information Flow Tracking (DIFT), also known as Dynamic Taint Analysis (DTA) [16], is a technique that associates metadata with memory and registers, propagating it during program execution. A complete implementation enables the enforcement of strong security guarantees over information flows, such as spatial memory safety or control-flow integrity [1].

This article presents a work-in-progress framework supporting a DIFT system on the AArch64 architecture, leveraging the CoreSight tracing subsystem to extract execution traces, extending prior work on the base ARM ISA [21,22].

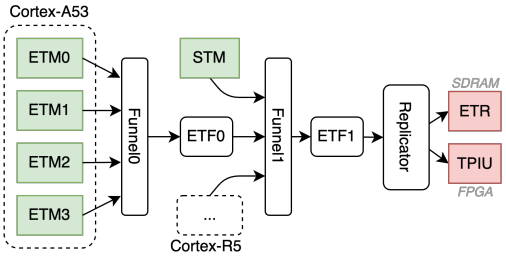
Built on this framework, we introduce InSight, a CoreSight trace interpreter that defines tagged memory, propagates tags, and checks tag rules offline. The framework aims to serve as a blueprint for a coprocessor-based DIFT system operating at runtime alongside the target program.

The paper is structured as follows: Section 2 outlines the inner workings of the ARM CoreSight tracing subsystem and the foundations of DIFT systems; Section 3 details the project setup; Section 4 presents initial results from the framework; and Section 5 discusses planned experiments and components for a complete DIFT system.

2 Background

2.1 ARM CoreSight

The ARM CoreSight subsystem [5] is a network of components that enables complete system tracing without interfering with the running application. Each *Processing Unit (PU)* (or core) is associated with a tracing unit, namely an *Embedded Trace Macrocell (ETM)* [6] (the successor to the *Program Trace Macrocell (PTM)*). Traces consist of information from program execution captured by the ETM, hardware events monitored by the *Performance Monitor Unit (PMU)*, and user-generated software events collected via the *Software Trace Monitor (STM)* [4,3] over a dedicated AXI region. These trace sources are routed through additional components responsible for delivering trace data to a "trace sink", either in RAM or a peripheral. Figure 1 (left) illustrates the CoreSight component architecture on the UltraScale+ board as an example.




```
1  Idx:8491 ID:10 I_ADDR_IS0: 0x2161EC
2  Idx:8497 ID:10 I_ATOM_F1: E
3  Idx:8498 ID:10 I_ADDR_IS0: 0x215E28
4  Idx:8501 ID:10 I_ATOM_F2:  NN
5  Idx:8502 ID:10 I_EXCEPT: Syscall
6  Idx:8504 ID:10 I_ADDR_IS0: 0x215E64
7  ...
8  Idx:8576 ID:20 M8: 0x48
9  Idx:8578 ID:20 D64M: 0x236348
10 Idx:8587 ID:20 D64M: 0x236348
11 Idx:8597 ID:20 D64M: 0x236348
12 Idx:8606 ID:20 D64M: 0x236000
```

Fig. 1: Tracing pipeline of CoreSight components on the UltraScale+ board (left). Overview of the main trace packets (right).

CoreSight traces consist of a succession of interlaced packets from all sources, as shown in Figure 1 (right). Each packet contains an index (order of arrival) and an ID (source). The main packets generated by the ETMs associated with each core are address and atom packets. These provide information on the basic blocks executed during program execution. An address packet corresponds to the

last address the program branched or jumped to, while an atom packet contains between 1-24 atom elements. Each atom element represents a basic block ending in a control-flow diversion and is marked as either taken (`E`) or not taken (`N`). The combination of address and atom packets generally allows reconstruction of the program's control flow. However, atom elements marked as taken may represent indirect jumps, whose targets are not directly available in the trace and require binary analysis. If the CoreSight subsystem supports it, a branch-broadcasting option ensures that an address packet is emitted before every indirect jump, enabling full control-flow reconstruction without requiring access to the binary. Exception packets notify system calls and behave similarly to atom `E` elements. STM packets consist of master channel selection and user data packets.

2.2 Dynamic Information Flow Tracking

In the 2000s, several a *Dynamic Information Flow Tracking (DIFT)*, as originally defined by Suh et al. [19] and also named *Dynamic Taint Analysis (DTA)*, as presented by Newsome et al. [14] emerged as techniques that leverage metadata to track information flow within a system. Their application requires three main components: (1) tag insertion, (2) tag propagation and storage, and (3) tag checking. Software-based solutions [7,24,9] support the three parts of a DIFT system by instrumenting the main binary, resulting in flexible solutions at the cost of a high performance overhead.

Other systems use in-core solutions to store metadata close to the corresponding data [8,15,12]. Although these systems require adapting the program or application to the new architectures, they offer better performance by embedding tag propagation within the processor pipeline. Bridging both approaches, hardware-software co-designed solutions [10,21,22] integrate program instrumentation with hardware-based tag propagation and checking logic, providing in additional flexibility.

Complete tracking of information flow enables enforcement of multiple guarantees on the running program. Spatial memory safety is enforced by coloring heap memory chunks, following the approach of ARM MTE. Fine-grained permissions can be implemented by checking specific tags before granting read or write access, effectively providing intra-process isolation. Data-flow isolation, especially on memory reads and writes, prevents user-supplied input from being written to sensitive memory regions.

Several challenges remain open problems in implementing DIFT. Among these, handling implicit flows and conditional instructions is critical, as the tag associated with a branch condition must be properly propagated throughout the branch body. Propagating only direct flows can result in false negatives, missing real attacks, while conservative propagation of all tags causes the *"taint explosion"* problem identified by Slowinska and Bos [17,18], where aggressive tagging complicates analysis. Although no definitive solution exists, adjusting tag granularity and size, combined with accelerated tag checking and propagation, forms the basis of efficient approaches.

3 System Design

To support information flow propagation and tracking over AArch64 binaries, we define regions of shadow memory and shadow registers designated to hold tags. A tag consists of metadata associated with registers and memory chunks. For each executed instruction, a set of *annotations* is defined to propagate tags within shadow memory. Finally, rules are established on tags to prevent security breaches at run time. For example, data tagged as insecure must not be used as a return address, preventing control-flow diversion.

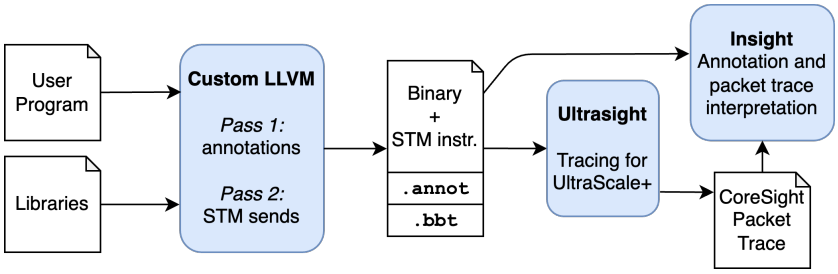


Fig. 2: Overview of the tracing and interpretation framework to support DIFT using CoreSight on an UltraScale+ board.

To this end, several components are developed and summarized in Figure 2. First, a set of LLVM backend passes is added to the compilation toolchain to (1) insert annotations into the binary—instructions for propagating tags in shadow memory—and (2) embed additional information in the trace through the STM to determine the addresses of loads and stores using base registers. The *CoreSight Access Library (CSAL)* is extended to properly configure the CoreSight subsystem for the UltraScale+. After tracing, the packet trace and annotated binary are passed to InSight, an interpreter that initializes and propagates tags offline, serving as a blueprint for a future online co-processor.

3.1 Annotations & Instrumentation

Tag propagation rules are defined following the approach initially proposed by Suh et al. [19]. Instructions are categorized into classes, presented in Table 1, and use operators $\oplus$ configurable via dedicated registers, based on prior work [8,15,12]. The table provides an introductory overview of annotations and is not exhaustive. These operators process source metadata to propagate tags to the destination, either a register or memory. Instructions that operate on tags are defined as *annotations*. The size of tags and the propagation registers, along with these operators, are critical in determining the DIFT system's scope, balancing under-tainting and over-tainting.

Class	**Instructions**	**Example**	**Annotations**
Arithmetic	`ADD`, `SUB`, `MULH`, `SDIV`, ...	`add Rd, Rn, Rm`	$\underline{\texttt{Rd}} \leftarrow \underline{\texttt{Rn}} \underset{\texttt{ari}}{\oplus} \underline{\texttt{Rm}}$
Logical	`AND`, `ORR`, `BIC`, `EOR`, ...	`and Rd, Rn, Rm`	$\underline{\texttt{Rd}} \leftarrow \underline{\texttt{Rn}} \underset{\texttt{log}}{\oplus} \underline{\texttt{Rm}}$
Bit manip.	`CLS`, `REV`, `SXTW SBFM`, ...	`cls Rd, Rn`	$\underline{\texttt{Rd}} \leftarrow \underline{\texttt{Rn}}$
Branches	`B`, `BR`, `B.cond`	`br Rn`	$\underline{\texttt{PC}} \leftarrow \underline{\texttt{Rn}}$
Calls	`BL`, `BLR`	`blr Rn`	$\underline{\texttt{LR}} \leftarrow \underline{\texttt{PC}}$ $\underline{\texttt{PC}} \leftarrow \underline{\texttt{Rn}}$
Loads	`LDR`, `LDP`, `LDUR`, ...	`ldr Rt, [Rn]`	$\underline{\texttt{Rt}} \leftarrow \underline{\texttt{[<Rn>]}} \underset{\texttt{loa}}{\oplus} \underline{\texttt{Rn}}$
Stores	`STR`, `STP`, ...	`str Rt, [Rn]`	$\underline{\texttt{[<Rn>]}} \leftarrow \underline{\texttt{Rt}} \underset{\texttt{sto}}{\oplus} \underline{\texttt{Rn}}$
Cond. operations	`CCMP`, `CSEL`, `CSINC`, ...	`csel Rd, Rn, Rm, cond`	$\underline{\texttt{Rd}} \leftarrow \underline{\texttt{Rn}} \underset{\texttt{cse}}{\oplus} \underline{\texttt{Rm}} \underset{\texttt{cse}}{\oplus} \underline{\texttt{cond}}$
Cond. branch.	`CBNZ`, `CNZ`, `TBNZ`, `TNZ`, ...	`cbnz Rn, <label>`	$\underline{\texttt{NZCV}} \leftarrow \underline{\texttt{Rn}} \underset{\texttt{cbr}}{\oplus} \underline{\texttt{Z}}$ $\underline{\texttt{PC}} \leftarrow \underline{\texttt{[<PC>+rel]}} \underset{\texttt{cbr}}{\oplus} \underline{\texttt{Rn}}$

Table 1: Classification of `AArch64` instructions and their tag propagation rules. `Rd` is a destination register, `Rn`, `Rm` and `Rt` are arguments. $\underline{\texttt{Rn}}$ corresponds to the tag associated with the register. Similarly, $\underline{\texttt{[<Rn>]}}$ is the tag in memory at the address contained in `Rn`. The $\leftarrow$ operator propagates tags from sources (right) to destination (left). `NZCV` and `cond` refer to flag registers, and `rel` to a relocable label address.

Annotations are added in a dedicated binary section (`.annot`) and linked to their corresponding basic blocks via entries stored in the *basic block table* section (`.bbt`). These two sections are designed to be loaded and used by the co-processor alongside the CoreSight trace. An address packet followed by an executed (`E`) atom packet triggers propagation of annotations associated with that basic block. The LLVM instances for basic blocks (`MachineBasicBlock`) may contain multiple branch or jump instructions within the same `MachineBasicBlock`. CoreSight atoms are delimited by control-flow changes, and to keep a one-to-one mapping, we force all control-flow changes emit a new label in the binary. Two LLVM passes, applied last in the `AArch64` backend, implement this: the first inserts annotations in text form into the `.annot` section; the second inserts relocatable labels linking basic blocks to their annotations in the `.bbt` section.

Actual register values are unnecessary for tag propagation during arithmetic or logical instruction execution. However, propagating tags in memory requires the values of base and potentially offset registers. These register values are embedded in the trace via the STM interface. Once configured, any writes to the STM's AXI region are translated into CoreSight STM packets. Practically, a store of the base register is inserted before each memory access using it, increasing binary size by a factor two at worst. As an optimization, for accesses using both base and offset registers, the `stp` (store pair) instruction is employed. To avoid instrumenting common stack accesses, annotations provide sufficient offset information (extracted at compile time) allowing the tag propagator to maintain the stack pointer value internally, requiring only its initial value at program start.

3.2 UltraSight: A Program Tracer on the UltraScale+

The CSAL library is used to configure the CoreSight subsystem on our architecture, providing an interface to access and connect components within the tracing pipeline. Building on prior work using CSAL to accelerate fuzzing [13] and define a hardware CoreSight trace decoder [23], we present UltraSight. It manages the complete tracing pipeline configuration on the UltraScale+ board as shown in Figure 1. Trace sources include all four Cortex-A53 cores via their ETMs. Their trace outputs route to a funnel and are stored in a FIFO before merging with other sources, STM packets and the unused Cortex-R5 core tracing. The combined trace set passes to a replicator that copies the output to designated sinks, either RAM or the board's programmable logic interface.

The software component manages the traced program's start and end using a parent process and `ptrace`. The STM region for embedding user data into the trace is initialized in two ways: either via a dedicated library preloaded for unannotated dynamically linked binaries, or by using an annotated `musl-libc` version to enable DIFT across the entire C standard library and program.

This setup already presents an out-of-the box framework to communicate with external monitors in the programming logic. To demonstrate this, in the EU Horizon project "COEMS", Ahishakiye et al.[2] use control-flow tracing through CoreSight together with instrumentation to check selected memory accesses for data races. The combined control flow and data trace-events were not directly verified on the on-SoC FPGA, but rather the FPGA communicates with an externally attached further UltraScale-based appliance of sufficient capacity to implement generic stream-based processing (without re-synthesization) via the TeSSLa specification language by Leucker et al. [11]. We have successfully ported their approach for data-race checking to the framework we use here, where we record control-flow events, most importantly starting new threads via `pthread_create`, and use instrumentation to generate data-events with memory addresses in the STM. We did not implement any verification in hardware, instead for this proof-of-concept, the decoded trace-data from the FPGA is processed offline with the data race-specification in the TeSSLa-interpreter.

3.3 InSight: A CoreSight Parser and Annotation Interpreter

Before developing a run-time hardware co-processor using the programmable logic part of the board, we implement an offline software trace and annotation interpreter named InSight. Its primary purpose is to model the hardware counterpart and explore the state space of tag sizes, granularities, propagation rules, and annotation formats. It defines several parsers for the new binary sections and generated CoreSight packets.

The annotations section is segmented according to the basic block table. A map links each basic block to its parsed annotations. The decoded CoreSight trace is split into packets, retaining only the main four types: address packets indicating the landing address after a branch or jump; atom packets signaling executed instruction blocks and whether the final control-flow instruction is taken;

exception packets marking system call execution; and data packets containing user-sent data embedded in the trace.

The shadow memory handles metadata reads and writes to registers and memory, updated after each interpreted annotation. Tag check rules are defined as functions executed after each annotation. The stack pointer is maintained by the interpreter itself, initialized from the first STM packet and updated through annotations tied to functions prologues and epilogues. For tag initialization, the intended goal is a kernel-level mechanism via Linux Security Modules (LSM). Currently, a system call capture mechanism is used, identifying the syscall by name and capturing its arguments.

Annotation interpretation proceeds as follows. All STM data packets are first executed to populate the STM queue with the register values required for memory accesses. Upon encountering an address packet, the embedded address is extracted. When an atom packet is found, this address is used to retrieve the corresponding annotations via the basic block table. Atom elements are then processed: if the element is `E` (branch taken), interpretation stops and waits for the next address packet; if it is `N` (branch not taken), interpretation continues by falling through to the next entry in the basic block table and its associated annotations. Exception packets also use the captured address to interpret the associated annotations, ending with the system call instruction `svc`.

4 First Evaluation

4.1 Experimental Setup

The complete framework is deployed on a Xilinx Zynq UltraScale+ MPSoC evaluation board (`ZCU104`). The Linux kernel `6.1.5-xilinx-v2023.1` runs on the Cortex-A53 cores using `petalinux-2023.1`, with CoreSight-related configuration options enabled. Additional backend passes are implemented in LLVM version 19.1.6. The CSAL library is extended from version 3.1 to configure the tracing pipeline and activate branch broadcasting. Trace decoding uses OpenCSD version 1.5.6. The standard library is musl-libc version 1.2.5, combined with LLVM runtime libraries (compiler-rt, libc++/libc++abi, and libunwind). The musl-libc entry point is modified to invoke an assembly stub that configures the CoreSight STM, maps its memory into the user process, and transmits the initial stack pointer. The entire musl-libc is instrumented using the custom toolchain, with manually inserted annotations for raw assembly files.

4.2 Security Evaluation

Threat model: We assume there are memory safety vulnerabilities (e.g., out-of-bounds access, use-after-free, uninitialized memory, or type confusion) in victim user programs and aim to prevent control-flow hijacking, data-only attacks, and information leakage. We assume the main core is trusted and free from side-channel attacks.

```c
// Should NOT be called
void secure_function() {
  exit(0);
}

// Forces return address
// spillage on the stack
__attribute__((noinline))
void force_frame() {
    __asm__ __volatile__("");
}

// Overflow of buf
void vulnerable() {
  char buf[8];
  force_frame();
  read(0, buf, 32);
}

// Passing the payload with the
// address of secure_function
int main() {
  vulnerable();
  return 1;
}
```

```
0x2167ac <vulnerable>:
  sub  sp, sp, #0x20
  stp  x29, x30, [sp, #16]
  mov  w0, wzr
  add  x1, sp, #0x8
  mov  x2, #0x20
  bl  216800 <read>
...
0x2167cc <vulnerable_0>:
  ldp  x29, x30, [sp, #16]
  add  sp, sp, #0x20
  ret
...
0x216800 <read>:
  ...
  svc
```

```
0x2167cc (vulnerable_0)
  FP <- [<SP> + 8*Imm(2)]_64 loa SP loa Imm(2)
  LR <- [<SP> + 8*Imm(2) + 64]_64 loa SP loa Imm(2)
  SP <- SP add Imm(32)
  PC <- PC ret LR
  END
```

Listing 1: An example of a control-flow hijack detection using InSight: **(a)** C program with stack overflow (left), **(b)** raw instructions of the program (top right) and **(c)** annotations associated with block `vulnerable_0` (bottom right).

As a first experiment, InSight is configured to use single-bit tags at word-level granularity. All tag propagation operations are implemented as logical ORs, which, in practice, lead to overtainting [17,18]. Two enforcement rules are defined to prevent control-flow hijacking: the program counter (`PC`) and the link register (`LR` or `X30`) must remain untagged at all times.

A simple stack overflow `read()`-based program is implemented and instrumented alongside the `musl-libc`, as shown in Listing 1. The attack performs a classic stack overflow by reading user input *(**a** - line 17)* into the local variable `buf` *(**a** - line 15)*, overwriting the return address stored on the stack. The destination buffer address used by `read`, located in `x1`, is visible in the instruction dump (**b** - line 5). InSight captures this value when it detects the execution of the system call instruction `svc` *(**b** - line 16)*. Annotations tied to the `vulnerable_0` basic block theb propagate the tag associated with the input. The first two annotations *(**c** - lines 2-3)* correspond to the `ldp` (load pair) instruction *(**b** - line 10)*. During interpretation, the tag set for `x1` gets propagated to the stack and within the link register, breaking the tag rules, and halting interpretation.

5 Conclusion & Future Works

This work introduces a practical DIFT framework combining compiler instrumentation, runtime tracing, and offline annotation interpretation. InSight, the software interpreter, enables systematic exploration of tag propagation parameters and validates the feasibility of hardware co-design.

Current limitations include offline tag initialization and incomplete runtime support, both of which are targeted in upcoming work. A full performance evaluation of instrumented binaries will follow the extension of instrumentation to the remaining runtime libraries. Tag initialization will migrate to the kernel through Linux Security Modules (LSM), extending the task structure with tag metadata and propagating tags during system calls. Entry points and propagations include file operations (`file_open`), socket reads (`socket_recvmsg`), memory mappings (`file_mmap`), and program execution (`bprm_check`).

On the hardware side, implementing the co-processor requires a CoreSight trace decoder [23], a DIFT propagation engine, shadow memory, and an annotation processor. The viability of the system depends on the annotation format and its efficient hardware execution, especially in comparison to in-core or fully tagged architectures [12,8,15].

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity: Principles, implementations, and applications. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. pp. 340–353. CCS'05, ACM (Nov 2005). https://doi.org/10.1145/1102120.1102165
2. Ahishakiye, F., Jarabo, J.I.R., Pun, V.K.I., Stolz, V.: Hardware-assisted online data race detection. In: Bartocci, E., Falcone, Y., Leucker, M. (eds.) Formal Methods in Outer Space. Lecture Notes in Computer Science, vol. 13065, pp. 108–126. Springer (2021). https://doi.org/10.1007/978-3-030-87348-6_6
3. ARM: ARM Coresight STM-500 r0p1 - TRM. Tech. rep. (2013)
4. ARM: ARM STM - Programmer's model version 1.1. Tech. rep. (2013)
5. ARM: ARM CoreSight Architecture Specification v3.0. Tech. rep. (2017)
6. ARM: ARM Embedded Trace Macrocell Specification (v4.0-v4.6). Tech. rep. (2023)
7. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles. pp. 133–147. SOSP'05, ACM (2005). https://doi.org/10.1145/1095810.1095824
8. Dalton, M., Kannan, H., Kozyrakis, C.: Raksha: A flexible information flow architecture for software security. In: Proceedings of the 34th Annual International Symposium on Computer Architecture. pp. 482–493. ISCA'07, ACM (2007). https://doi.org/10.1145/1250662.1250722
9. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: Libdft: Practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments. pp. 121–132. VEE'12, ACM (2012). https://doi.org/10.1145/2151024.2151042
10. Lee, J., Heo, I., Lee, Y., Paek, Y.: Efficient security monitoring with the core debug interface in an embedded processor. ACM Transactions on Design Automation of Electronic Systems **22**(1), 1–29 (2016). https://doi.org/10.1145/2907611
11. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: Runtime verification of non-synchronized real-time streams. In: Proceedings of the 33rd ACM Symposium on Applied Computing. pp. 1925–1933. SAC'18, ACM (2018). https://doi.org/10.1145/3167132.31673
12. Liu, Z., Rong, Y., Li, C., Tan, W., Li, Y., Han, X., Yang, S., Zhang, C.: CC-TAG: Configurable and combinable tagged architecture. In: Proceedings of the

32nd Network and Distributed System Security Symposium. NDSS'25, Internet Society (2025). https://doi.org/10.14722/ndss.2025.240862
13. Moroo, A., Sugiyama, Y.: ARMored CoreSight: Towards efficient binary-only fuzzing (Nov 2021), https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html
14. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12th Network and Distributed System Security Symposium. NDSS'05, Internet Society (2005)
15. Palmiero, C., Di Guglielmo, G., Lavagno, L., Carloni, L.P.: Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for IoT applications. In: Proceedings of the 7th IEEE High Performance Extreme Computing Conference. pp. 1–7. HPEC'18 (Sep 2018). https://doi.org/10.1109/HPEC.2018.8547578
16. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 21st IEEE Symposium on Security and Privacy. pp. 317–331. SP'10 (May 2010). https://doi.org/10.1109/SP.2010.26
17. Slowinska, A., Bos, H.: Pointless tainting?: Evaluating the practicality of pointer tainting. In: Proceedings of the 4th ACM European Conference on Computer Systems. pp. 61–74. EuroSys'09, ACM (Apr 2009). https://doi.org/10.1145/1519065.1519073
18. Slowinska, A., Bos, H.: Pointer tainting still pointless: (but we all see the point of tainting). ACM SIGOPS Operating Systems Review **44**(3), 88–92 (Aug 2010). https://doi.org/10.1145/1842733.1842748
19. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 85–96. ASPLOS'04, ACM (2004). https://doi.org/10.1145/1024393.1024404
20. Szekeres, L., Payer, M., Tao Wei, Song, D.: SoK: Eternal war in memory. In: Proceedings of the 34th IEEE Symposium on Security and Privacy. pp. 48–62. CCS'13, IEEE (May 2013). https://doi.org/10.1109/SP.2013.13
21. Wahab, M.A., Cotret, P., Allah, M.N., Hiet, G., Lapotre, V., Gogniat, G.: ARMHEx: A hardware extension for DIFT on ARM-based SoCs. In: Proceedings of the 27th International Conference on Field Programmable Logic and Applications. pp. 1–7. FPL'17, IEEE (Sep 2017). https://doi.org/10.23919/FPL.2017.8056767
22. Wahab, M.A., Cotret, P., Allah, M.N., Hiet, G., Biswas, A.K., Lapotre, V., Gogniat, G.: A small and adaptive coprocessor for information flow tracking in ARM SoCs. In: Proceedings of the 13th International Conference on ReConFigurable Computing and FPGAs. pp. 1–8. ReConFig'18, IEEE (Dec 2018). https://doi.org/10.1109/RECONFIG.2018.8641695
23. Weingarten, M.E., Hossle, N., Roscoe, T.: High Throughput Hardware Accelerated CoreSight Trace Decoding. In: Proceedings of the 31st Design, Automation & Test in Europe Conference & Exhibition. pp. 1–6. DATE'24, IEEE (Mar 2024). https://doi.org/10.23919/DATE58400.2024.10546666
24. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. OSDI'06, USENIX Association (Nov 2006), https://usenix.org/event/osdi06/tech/zeldovich.html