A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components

Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Vianney Lapotre, Gogniat Guy, Arnab Kumar Biswas

A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components

Muhammad Abdul Wahab [α], Pascal Cotret [β], Mounir Nasr Allah [γ], Guillaume Hiet [γ]
Vianney Lapôtre [δ], Guy Gogniat [δ] and Arnab Kumar Biswas [δ]

[α] IETR - CentraleSupélec, muhammad.abdulwahab@centralesupelec.fr
[β] Independent researcher, pascal.cotret@gmail.com
[γ] INRIA CIDRE - CentraleSupélec, firstname.lastname@centralesupelec.fr
[δ] Lab-STICC - University of South Brittany, firstname.lastname@univ-ubs.fr

(preprint)

Abstract

Most of hardware-assisted solutions for software security, program monitoring, and event-checking approaches require instrumentation of the target software, an operation which can be performed using an SBI (*Static Binary Instrumentation*) or a DBI (*Dynamic Binary Instrumentation*) framework. Hardware-assisted instrumentation can use one of these two solutions to instrument data to a memory-mapped register. Both these approaches require an in-depth knowledge of frameworks and an important amount of software modifications in order to instrument a whole application. This work proposes a novel way to instrument an application, at the source code level, taking advantage of underlying hardware debug components such as CS (*CoreSight*) components available on Xilinx Zynq SoCs. As an example, the instrumentation approach proposed in this work is used to detect a double free security attack. Furthermore, it is evaluated in terms of runtime and area overhead. Results show that the proposed solution takes 30 μs on average to instrument an instruction while the optimized version only takes 0.014 μs which is ten times better than usual memory-mapped register solutions used in existing works [1, 2].

1 Introduction

Software security is still a hot topic despite an important amount of research and development. Existing solutions are either too expensive in terms of cost, performance, power and area or either target a limited threat model. On the other side, attackers have more and more tools and vulnerabilities available in order to exploit existing systems. Therefore, it is important to provide solutions that can be easily adjusted on existing embedded systems without requiring important development effort.

One common approach for software security is performed through events monitoring (such as library calls, syscalls, specific instructions and so on). However, software-only solutions add important runtime overhead [3]: that is the reason why hardware-assisted solutions have been proposed. While they improve the performance overhead, proof-of-concepts are usually implemented on FPGAs rather than heterogeneous SoCs due to the amount of effort, time and money required to develop secure solutions on these platforms. Therefore, most of existing works cannot retrieve information required for monitoring purposes on a hardcore CPU.

Instrumentation is a common solution to implement hardware-assisted software security solutions. It is used in DIFT (*Dynamic Information Flow Tracking*) [2] in order to protect against different types of software attacks. Instrumentation can also be used for behavior monitoring [4], performance analysis [5] and software error detection [6].

As it was previously written, instrumentation can be done statically or dynamically. Static approaches modify the binary without requiring another process and cover all paths of the code while dynamic solutions require another process that instruments the binary and provides information only on the path taken by the application. Static approaches provide less information than dynamic ones but are usually faster in terms of performance because they do not require another external process.

All existing works using instrumentation do not provide a detailed description of their implementations. Some works such as [1] use custom tools without describing them while others use compilers or existing dynamic instrumentation frameworks for instrumentation without providing in-depth details. This work puts forward a novel approach for static instrumentation that can be used on ARM SoCs with CS (*CoreSight*) debug components. This work does not target some features of instrumentation frameworks allowing to place the code at a given location. Nevertheless, it presents how the code is instrumented and how the instrumented data can be recovered on a reconfigurable device such as those included in Xilinx Zynq SoCs. The main goal of this work is to propose an instrumentation solution for ARM-based SoCs which is easy to implement with minor modifications, targeting modern OS (*operating system*) such as Linux kernel. Furthermore, the approach developed in this work is able to send both user space and kernel space information.

This paper is organized as follows. Section 2 provides insights on existing instrumentation approaches. Section 3 presents the proposed architecture and provides implementation details. Section 4 provides different use cases. Section 5 details implementation results and Section 6 gives some conclusions and future perspectives.

2 Related work and assumptions

A lot of works have been done on instrumentation frameworks: these frameworks can be architecture-dependent or independent, secure, static or dynamic. Soot [7] is a static instrumentation framework for Java and Android applications. Wala [8] also provides static instrumentation library for Java bytecode. Atom [9] is another static binary instrumentation framework on the Alpha processor platform for the Tru-64 OS. PEBIL [10] is a static binary instrumentation framework for x86-64 architecture. Dyninst [11] is a static and dynamic instrumentation framework for multiple platforms (mainly x86-64 and ARM). Hijacker [12] is an open-source customizable static binary instrumentation tool. CSI (*Comprehensive Static Instrumentation*) [13] is also a static instrumentation framework for LLVM. LLVM [14], an open-source compiler, can be used to create passes and instrument code.

In [15], the author presented an original solution using the debug components and a secondary CPU core (based on the NXP CPU12X architecture) to extract instrumentation data. The main drawback of this solution is that it wastes another GPP (*General Purpose Processor*) for instrumentation: as a consequence, the power consumption of this solution is doubled.

Hardware-assisted instrumentation can use one of the above methods to instrument the application in order to recover instrumented data on the FPGA part of a heterogeneous SoC. However, using these frameworks require an in-depth knowledge of their API. In modern OS (e.g. Linux), the memory-mapped solution used in [1, 2] requires an important number of software modifications. More precisely, it requires to map the physical address of the instrumentation register to a virtual address. This modification requires changing the kernel ELF binary loader. Then, this virtual address is sent towards the register included in instrumented instructions (which can be done through relocation). Finally, the binary can use this virtual address to write instrumented data. These changes require invasive modifications of the kernel. The solution proposed in this work provides the lowest amount of software modifications.

The contributions of this work are the following:

- A novel, simple and hardware-assisted instrumentation approach that takes advantage of CS debug components.
- This work proposes to add a system call to take advantage of the context ID feature provided by the CS PTM component and use the CS TPIU component and the EMIO interface in order to send trace and instrumented data towards the FPGA part.
- An improved and the first open-source version of the PFT (*Program Flow Trace*) decoder that allows to decode trace and recover instrumented data on the FPGA part of ARM SoC such as Xilinx Zynq.

3 Proposed approach

3.1 Software modifications

Instrumenting code using CS components can be done with a specific configuration; then, by adding a syscall and using the newly added syscall with a few lines of C code.

3.1.1 Configuring Coresight

ARM CS components technical reference manual [16] explains how to program all CS components. The Linux kernel 4.9 provides a driver for CS components. However, the support for CS components on Zynq SoC was missing. The device tree was patched in order to use Linux kernel drivers of CS components. The approach proposed in this work requires the activation of a specific feature known as context ID, that has not been used in any existing work to the best of our knowledge. Enabling context ID generates specific PFT (*Program Flow Trace*) packets providing information about the context ID of an application. The context ID of an application consists of the PID (*Process ID*) and ASID (*Application Specific ID*) of the application. The kernel is responsible for writing this value to a specific context id register. By enabling context ID tracing in the CS PTM component, this value is sent into the trace. Instead of writing PID and ASID into the context ID register, the value to be written is the value to instrument.

Table 1: CS configuration in Linux kernel driver

File name	**Value**	**Description**
`mode`	0x30	Enable branch broadcast and context ID feature
`addr_idx`	1	Choose address comparator
`addr_acctype`	0	Choose comparison access type
`addr_range`	**0x106a0 0x10700**	Enable trace in address space given
`enable_source`	1	Enable PTM component

Table 1 shows the values used to program the PTM component in the Linux kernel sysfs file system. TPIU (*Trace Port Interface Unit*) is used as trace sink to recover trace on the FPGA part via EMIO interface [2].

3.1.2 Adding syscall

Listing 1 shows the kernel code to be added in order to write the value to the context ID register.

Listing 1: Custom syscall code in `kernel/sys.c` file

```
SYSCALL_DEFINE1(wctxtid, unsigned int, instrumentation_data){
  asm volatile("mcr p15, 0, %0, c13, c0, 1\n"
               ::"r"(instrumentation_data):);
  isb();
  return 0;
}
```

The `mcr` instruction is used to write the instrumented value to the context ID register located in coprocessor 15 while the `isb()` instruction triggers the write [17].

Listing 2: Custom syscall definition in `include/linux/syscalls.h` file

```
asmlinkage long sys_wctxtid(unsigned int instrumentation_data);
```

The syscall definition needs to be added into the kernel source as shown in Listing 2. Finally, a number needs to be associated with the new syscall as shown in Listing 3.

Listing 3: Associate a number to custom syscall in `arch/arm/kernel/calls.S` file

```
/* 397 */ CALL(sys_wctxtid)
```

3.1.3 Writing C code

Once the kernel code has been modified, it is compiled with the write to the context ID register option enabled in order to obtain the Linux kernel binary file (`uImage`). Once the modified kernel is booted, Listing 4 shows how the newly added syscall can be used in any C program.

Listing 4: Example instrumented code

```
#include <unistd.h>
#include <sys/syscall.h>
int main(){
  unsigned dataToSend = 0x1234abcd;
  // system call number is 397
  // (cf calls.S file change)
  syscall(397, dataToSend);
  return 0;
}
```

3.2 Hardware design

3.2.1 Overall architecture

Figure 1 shows the overall architecture of the approach proposed in this work (implemented on a Zynq SoC). After configuring CS components and running the program, trace is obtained on the FPGA part through the EMIO interface according to the PFT (*Program Flow Trace*) protocol [18]. The important module developed in this work is the PFT decoder that differs with previous existing implementation. Furthermore, this work improves the solution of Wahab et al. [2] by adding the support of the context ID register. To make this work beneficial for the community, everything will be published online at: https://bitbucket.org/hardblare/hw-static-instrumentation

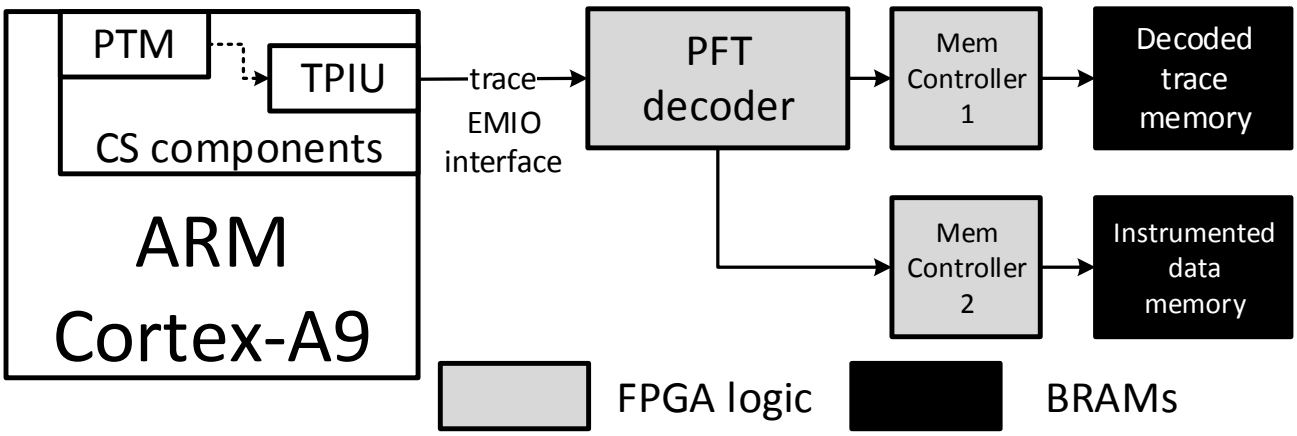

Figure 1: Overall architecture

3.2.2 PFT decoder

The TPIU export raw traces to the FPGA part via EMIO interface. Listing 5 shows an example of raw trace. Trace must be decoded in order to recover instrumented data on the FPGA part. The PFT protocol [18] presents 11 different trace packets and their corresponding headers. Figure 2 shows the overall architecture of the PFT decoder.

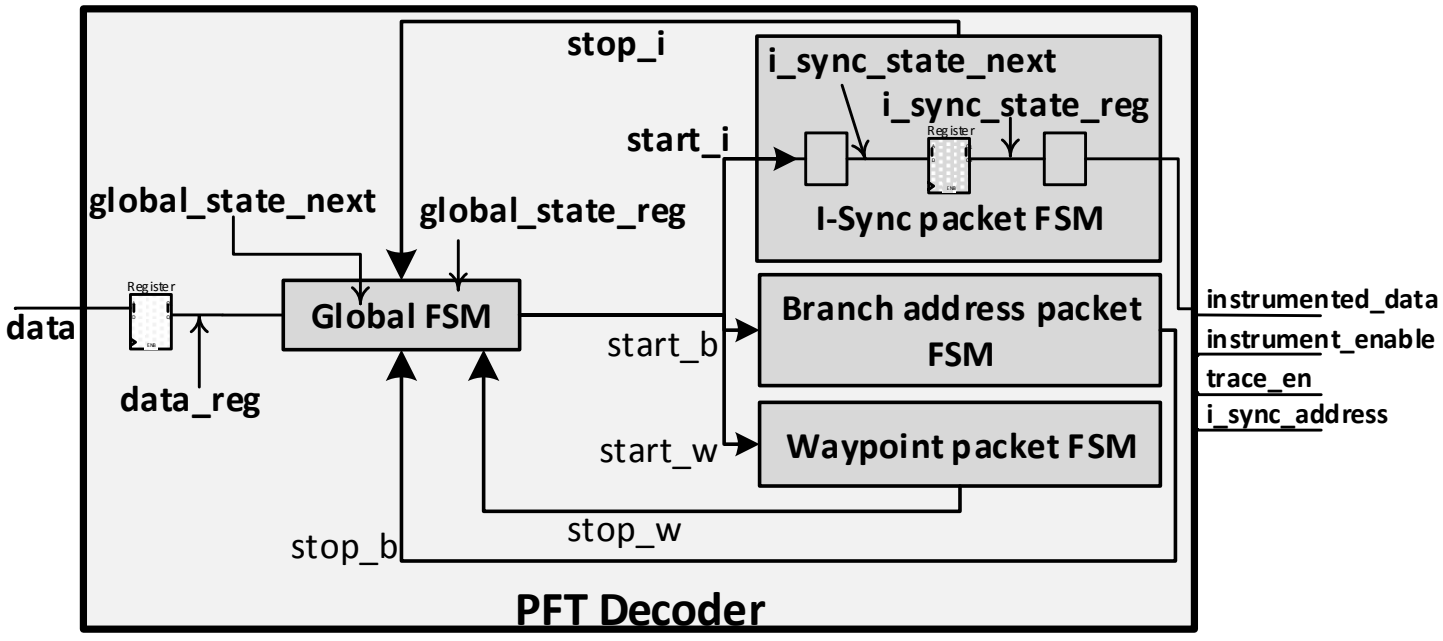

Figure 2: PFT decoder architecture

The PFT decoder consists of four FSMs (*Finite State Machines*): a global FSM that controls the other three packet FSMs (I-Sync, branch address, and waypoint). There are other PFT packets such as a-sync, exception that are also decoded by the global FSM. The most important PFT packet in this work is the I-Sync packet as it contains the instrumented data. The global state machine detects packet type and enables the corresponding packet FSM by setting start signal.

Figure 3 shows the FSM diagram of I-Sync packet. The FSM is in `wait_state` by default. When the start signal is set by the global FSM, I-Sync packet FSM goes into `i_sync` state and starts counting trace samples. The received four samples contain the current PC (*Program Counter*) value of ARM CPU. Once four samples are received, the state machine goes into `i_sync_ib` state. If context id tracing is enabled, it can send one, two or four bytes.

Depending on the generic `ctxtid` value, the context id packet is decoded. For instance, if the generic `ctxtid` is equal to "11", then the FSM starts a counter that counts the number of received trace samples. When this number is equal to 4, then the instrumented data bytes are correctly received. When the packet FSM finishes decoding packet, it goes back to `wait_state` and sends the stop signal to global FSM which then looks for the next packet type. The other packet FSMs use similar mechanism to communicate with global FSM and have similar state machines.

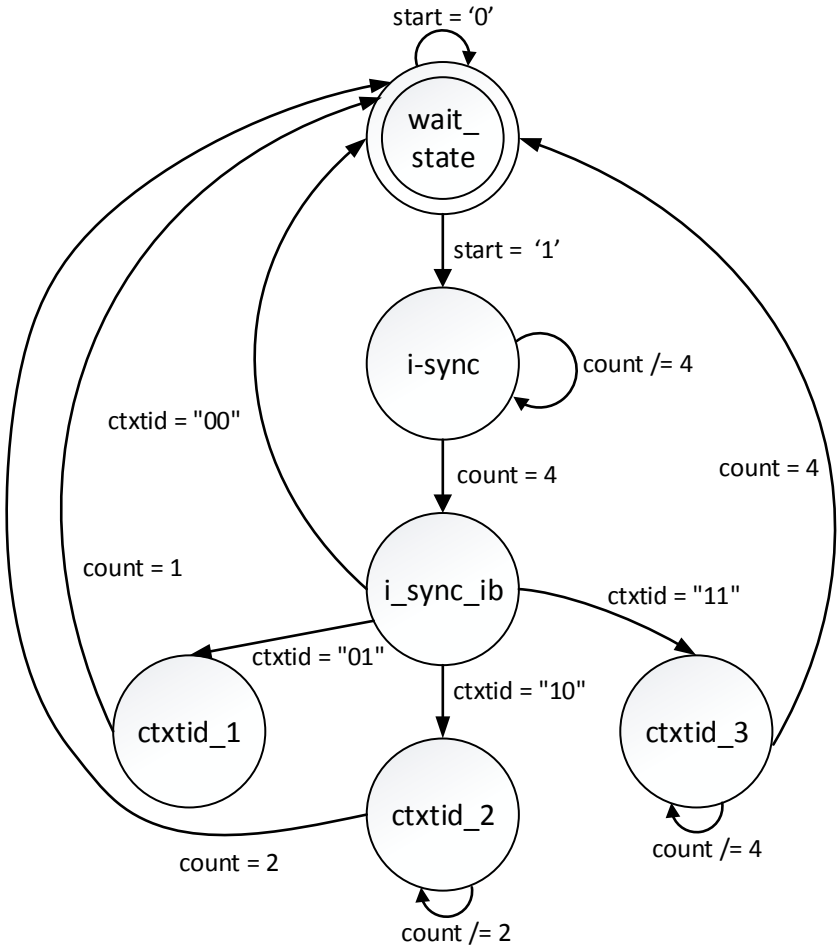

Figure 3: Finite State Machine diagram for the I-Sync packet decoder

Figure 4 is a timing diagram showing how the PFT decoder works at each clock cycle.

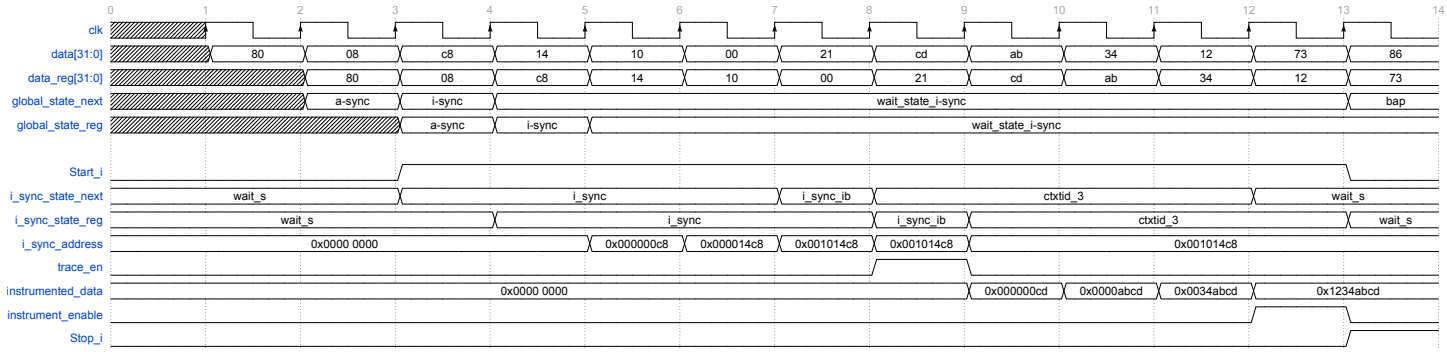

Figure 4: Timing diagram of global FSM and I-Sync FSM of PFT decoder (Figure 2)

All inputs are registered: for instance, input `data` is registered to obtain `data_reg` signal. All FSMs work with `data_reg` registered signal. When the global FSM detects an I-Sync packet, it enables a `start_i` signal (e.g. this is the case at the third clock cycle) to enable the corresponding

slave FSM. The I-Sync packet FSM then decodes the packet according to the PFT architecture as shown by `i_sync_state_reg`, `i_sync_state_next`, `i_sync_address`, `instrumented_data` and `instrument_en` signals. The `instrumented_data` signal contains the value instrumented by software. In the meantime, the global FSM waits for a stop signal which is enabled by the packet FSM when the packet is decoded. Enabling `stop_i` signal at the thirteenth clock cycle changes the global FSM state according to `data_reg` signal. This allows to decode the received trace on-the-fly. The `trace_en` signal allows to determine when valid decoded trace is available while the `instrument_enable` signal allows to determine when valid instrumented data is available. Outputs of the PFT decoder are also registered to avoid timing failures due to longer critical paths.

Listing 5: Raw trace

```
root@zedboard-hardblare:~/tests_appli#./trace_etb
00000000  00 00 00 00 00 80 08 78 04 01 00 21 f4 ee 03 00
00000010  8b 03 08 8c 04 01 00 21 f4 ee 03 00 9d 03 08 98
00000020  04 01 00 21 ff ff ff ff 9d 03 08 a8 04 01 00 21
00000030  dd dd dd dd 85 03 08 b4 04 01 00 21 dd dd dd dd
00000040  9d 03 08 c4 04 01 00 21 aa aa aa aa 9d 03 08 d4
00000050  04 01 00 21 11 11 11 11 fd bc cf db 0d 01 00 00

PFT packets a-sync packet i-sync packet branch address packet
```

Listing 5 shows the trace at the input of PFT decoder. It contains different packets detailed in a Technical Reference Manual provided by ARM [18].

Listing 6: Instrumented data from decoded raw trace

```
0003eef4 0003eef4 ffffffff
dddddddd aaaaaaaa 11111111
```

Listing 6 shows the content of the instrumented data memory. The values obtained correspond to the values inserted in the program source code.

3.3 Optimization

Rather than creating a new syscall as proposed earlier, this work proposes to modify an existing syscall called in the instrumented application. For example, if a `malloc` call is done in the instrumented program and its behavior needs to be monitored, the associated syscall , which is either `mmap` or `brk`, can be modified in order to add the proposed code inside the existing syscall. This way, a specific syscall is not required while reducing the runtime overhead.

This optimization is not possible if the program does not use any syscall which is rather rare as most programs require kernel services which are only accessible through syscalls. Another optimization could be to send multiple instrumented data values at each syscall rather than sending one value. However, this is not possible due to the fact that the PTM sends only the last instrumented value rather than all instrumented values during a syscall.

4 Case studies

4.1 Double Free

Listing 7 shows a straightforward double-free attack. There are three allocated memory areas: A, B and C. First, A and C are allocated. Then, A is freed. Later, B is allocated and A is freed again. This is the

line where the attack happens. However, when this code is compiled and executed, the Linux kernel does not detect any error. This type of attack can lead to heap overflow, attack code execution or illegitimate privilege elevation.

Listing 7: Double free vulnerability example code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define WORD_SIZE (size_t)-1>0xFFFFFFFFUL?8:4
#define SIZEOFNORMALCHUNK 0x100-WORD_SIZE
#define SIZEOFFASTCHUNK 0x60-WORD_SIZE

int main(){
  char *A, *B, *C;
  A = malloc(SIZEOFNORMALCHUNK);
  C = malloc(SIZEOFNORMALCHUNK);
  free(A);
  // same location as A
  B = malloc(SIZEOFNORMALCHUNK);
  if (B)
    // Double Free!
    free(A);
  return 0;
}
```

Listing 8 shows the instrumented code in order to detect the double free attack. Each time a `malloc` or `free` function is called, another instruction is added in order to send the information about the library function. In order to distinguish a `malloc` call from a `free` call, an example value of 12 MSB bits (`0xfff`) is added to the `malloc` call whereas `(0xffe)` is added to the `free` library call. Then, the lower 20 bits are used to specify the region allocated or freed. As an example, numbers are shown in instrumented value but static analysis can provide generic variables, containing numerical values, that can be easily inserted instead of numbers.

Listing 8: Instrumented code for double free detection

```
A = malloc(SIZEOFNORMALCHUNK);
// instrumented code
syscall(397,0xfff00001);
C = malloc(SIZEOFNORMALCHUNK);
// instrumented code
syscall(397,0xfff00002);
free(A);
// instrumented code
syscall(397,0xffe00001);
// same location as A
B = malloc(SIZEOFNORMALCHUNK);
// instrumented code
syscall(397,0xfff00003);
if (B){
  free(A);
  // instrumented code
  syscall(397,0xffe00001);
}
```

Then, the program is compiled and executed after configuring CS components as explained in subsection 3.1.1. An example output of the program execution and the content of memories is shown in Listing

9. After the configuration of CS components, the program is launched and the content of the decoded trace memory is recovered using AXI BRAM controller (not shown in Figure 1 as it is not required for the proposed design but is added for debug purposes).

The first value obtained in the decoded trace is `0x106a0` which is the address where tracing is started as configured in CS components (Table 1). Then, next values show the control flow changes in the program. In this case study, the decoded trace is not used. Then, the content of the instrumented data memory is displayed (containing the values inserted into the program). It can be noticed that the value `0xffe00001` is present twice while `0xfff00001` is present only once which means that the first region is freed twice and allocated once. This example shows that the instrumented data can be successfully recovered on the FPGA part and with a simple static analysis, the double free attack can be detected.

Listing 9: Execution output for double free example code

```
root@zedboard: /tests-appli#./trace-tpiu-topleaks

coresight-tpiu f8803000.tpiu: TPIU enabled
coresight-replicator amba:replicator: REPLICATOR enabled
coresight-funnel f8804000.funnel: FUNNEL inport 0 enabled
coresight-etm3x f889c000.ptm0: ETM tracing enabled

DECODE TRACE
00 106a0 10358 106c0 104d4 106ec b6e3ec88 1057c
10378 10598 1039c 105a0 103c0 105b8 1039c 105c0
103c0 105d8 10384 105e0 103c0 105f0 10378 10600
1039c 10608 103c0 10620 10384 10628 103c0 10638
10390 10644 10378 10654 10378 10660 1050c 1066c
10390 10678 10378 10690 b6e3ecf8 b6e3ec00 00 00

root@zedboard: /tests-appli#
./get-instrumented-data.elf
/dev/mem opened.
Memory mapped at address 0xb6fba000.
fff00001 fff00002 ffe00001 fff00003
ffe00001 00 00 00 00 00 00 00 00 00
```

4.2 Other use cases

This method allows reconstructing the CFG on the FPGA side using a decoded trace [2]. Therefore, using decoded trace, a control flow checker unit can be implemented on the FPGA to make sure that no control-flow variations occur during program execution. This information is very important as it cannot be determined statically. Moreover, it can be recovered without adding syscall by correctly configuring CS components and avoiding important runtime overhead in existing control-flow checker solutions.

There are other attacks that can be detected using this instrumentation strategy (for instance, event checking): static analysis can detect events that are sent towards the FPGA part in order to verify some properties. Therefore, this work suits the LTL (*Linear Temporal Logic*) hardware-assisted verification solutions.

5 Implementation Results

Xilinx tools 2017.1 are used on a Xilinx Zedboard with a Z-7020 SoC (dual-core Cortex-A9 running at 667 MHz and an Artix-7 FPGA) to implement the architecture shown in Figure 1. The PFT decoder and

memory controllers are working at the frequency of 250 MHz. Following subsections present an evaluation of this work in terms of runtime and area overheads.

5.1 Time overhead analysis

CS components do not add any execution time overhead as shown by Wahab et al. [2]. Even though this work uses a different configuration of CS components (context ID tracing is enabled), CS components still do not affect the execution time. The time overhead is measured by using Linux `perf` command and by launching MiBench benchmark applications with and without enabling CS components.

The overhead is only due to syscalls. On the target platform, the overhead of added syscall is 30 μs while being 0.150 μs for the memory-mapped register approach.

This work presents a method which is slower than existing memory-mapped approaches. However, with the optimization (subsection 3.3), time overhead can be reduced. If additional instructions `mcr` and `isb()` are added to an existing call, the overhead due to the syscall is removed and the global overhead is only due to the execution of `mcr` and `isb()` instructions. The `mcr()` instruction takes 1+B cycles (where B is the number of cycles spent in the coprocessor busy-wait loop) and the `isb()` instruction takes 4 cycles. Therefore, both instructions take 1+B+4 = B+5 cycles at 667MHz. If this value is equal to ten cycles which is the case if the coprocessor is availble, then the runtime overhead introduced would be 0.014 μs which is more than ten times better than the 0.15 μs required for the memory-mapped register approach.

5.2 Latency and bandwidth

If `n` is the number of packets received in a PFT packet, the PFT decoder requires (n+1) clock cycles to decode a trace. The maximum value of `n` in this work is 10. In other words, the PFT decoder requires only one additional cycle to decode a trace and recover instrumented data while the solution based on memory-mapped register requires on average 30 cycles in average to receive valid instrumented data due to timing delays introduced by the handshake between different channels. In this work, as 8 bits are sent at each clock cycle at the frequency of 250 MHz, the trace bandwidth is 250*8 = 2000 Mbits/s. The maximum bandwidth of the trace port is 250*32 = 8000 Mbits/s. The AXI interface on a Zedboard can go up to 200 MHz. Therefore, the maximum bandwidth of the AXI port is 200*32 = 6400 Mbits/s.

5.3 Area results

Table 2 shows the area overhead of proposed architecture which only requires 0.34% of slice LUTs, 0.47% of slice registers and 2.86% of BRAMs on a Zynq Z-7020 device. The memory-mapped solution requires 1684 slice LUTs (3.17%), 1523 slice registers (1.43%) and no BRAM. The memory-mapped register solution takes more area because the memory required to store data is implemented in FPGA itself rather than in BRAM as in this work. Furthermore, the memory-mapped register approach requires an AXI interconnect block which has internal FIFOs making it larger in terms of area occupation.

Table 2: Post-implementation area results of overall architecture on Xilinx Zynq Z-7020

IP Name	Slice LUTs (in %)	Slice registers (in %)	BRAM tiles
PFT Decoder	126 (0.24%)	240 (0.23%)	0
Mem controller 1	18 (0.03%)	79 (0.08%)	0
Mem controller 2	18 (0.03%)	79 (0.08%)	0
Decoded trace memory	2 (0.01%)	0	2 (1.43%)
Instrumented data memory	2 (0.01%)	0	2 (1.43%)
Miscellaneous	16 (0.03%)	97 (0.09%)	0
Total Design	**182 (0.34%)**	**495 (0.47%)**	**4 (2.86%)**
Memory-mapped	**1684 (3.17%)**	**1523 (1.43%)**	**0 (0%)**
Total Available	**53200**	**106400**	**140**

5.4 Comparison with related works

Table 3 compares this work with the usual memory-mapped register solution which is widely used in existing hardware-assisted instrumentation approaches. This work requires only few minor modifications of the kernel (addition of `mcr` and `isb()` instructions) while memory-mapped register solutions need a modified kernel ELF loader and relocation units. This work has a lower latency and a higher maximum bandwidth than the memory-mapped solution. Furthermore, the area overhead is 6 times smaller than existing works.

Table 3: Comparison with previous works

Metric	This work	Optimized solution	Existing works (memory-mapped)
Software modifications	+	+	++
Latency (clock cycles)	(n+1) $\leq$ 10	(n+1) $\leq$ 10	30
Maximum bandwidth (Mbits/s)	**8000**	**8000**	6400
Runtime overhead (μs)	30	**0.014**	0.150
Area overhead (%)	**0.47**	**0.47**	3.17

6 Conclusion

This work proposes to exploit the context ID feature of the CoreSight PTM component in order to instrument an application. The generated trace, which also contains the instrumented data, is sent to the FPGA part using the CS TPIU component and EMIO interface. Software modifications required to instrument are minimal (about 10 lines are changed in the Linux kernel). Furthermore, the hardware design required to decode traces takes less than 0.5% of the FPGA area on a Zynq SoC. This work takes

30 μs for each instruction added in the source code. The optimized version, which consists in removing context switch time overhead by modifying existing syscall used in the application rather than adding a new syscall, give a runtime overhead of 0.014 μs: it is 10 times better than 0.15 μs required for a memory-mapped register solution. In terms of perspectives, the approach presented in this work will be ported to Intel-based SoCs by taking advantage of the Intel PT (*Processor Trace*) debug component.

References

[1] I. Heo, M. Kim, Y. Lee, C. Choi, J. Lee, B. B. Kang, and Y. Paek, "Implementing an application-specific instruction-set processor for system-level dynamic program analysis engines," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, pp. 53:1–53:32, Sept. 2015.

[2] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. LapÃ´tre, and G. Gogniat, "Armhex: A hardware extension for dift on arm-based socs," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7, Sept. 2017.

[3] 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 '07, (New York, NY, USA), pp. 482–493, ACM, 2007.

[4] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *Proceedings of the First USENIX Workshop on Offensive Technologies*, WOOT '07, USENIX Association, 2007.

[5] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[6] S. So, Y. Lim, S. D. Cha, and Y. R. Kwon, "An empirical study on software error detection: voting, instrumentation and fagan inspection," in *Proceedings 1995 Asia Pacific Software Engineering Conference*, pp. 345–351, Dec. 1995.

[7] Sable Research Group of McGill University, "Soot - a framework for analyzing and transforming java and android applications."

[8] IBM T.J. Watson Research Center, "T.j. watson libraries for analysis (wala)."

[9] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," *SIGPLAN Not.*, vol. 39, pp. 528–539, Apr. 2004.

[10] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS 2010)(ISPASS)*, vol. 00, pp. 175–183, Mar. 2010.

[11] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, (New York, NY, USA), pp. 9–16, ACM, 2011.

[12] A. Pellegrini, "Hijacker: Efficient static software instrumentation with applications in high performance computing: Poster paper," in *2013 International Conference on High Performance Computing Simulation (HPCS)*, pp. 650–655, July 2013.

[13] T. B. Schardl, T. Denniston, D. Doucet, B. C. Kuszmaul, I.-T. A. Lee, and C. E. Leiserson, "The csi framework for compiler-inserted program instrumentation," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, pp. 43:1–43:25, Dec. 2017.

[14] University of Illinois, "The llvm compiler infrastructure."

[15] P. Fogarty, "Minimising the impact of software instrumentation using on-chip debug and a secondary cpu core," in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pp. 1–5, Sept. 2012.

[16] ARM, *CoreSight Components Technical Reference Manual*.

[17] ARM, *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition*.

[18] ARM, *CoreSight Program Flow Trace Architecture Specification*.