ARMOR: Protecting Software Against Hardware Tracing Techniques

Tai Yue, Fengwei Zhang, *Senior Member, IEEE*, Zhenyu Ning, Pengfei Wang, Xu Zhou, Kai Lu, and Lei Zhou

***Abstract*— Many modern processors have embedded hardware tracing techniques (e.g., Intel Processor Trace or ARM CoreSight). While these techniques are widely used due to their transparency and low overhead, they also bring serious security threats. Attackers can utilize hardware tracing to trace the trusted applications from a non-secure application. Existing protection techniques fail to effectively protect the runtime information when hardware tracing is employed. To counter these threats, in this paper, we propose a novel direction called anti-hardware tracing. Our key idea is to exploit the limitations of hardware tracing: trace buffer overflow can cause trace data loss. We build a model to analyse the overflow and outline three principles for efficient triggering overflows and achieving anti-hardware tracing: numerous branches in the program, high-speed execution of the program, and the high-water mark of the trace buffer. We develop a framework called ARMOR on ARM Juno R2 to realize our approach. ARMOR protects software against the trace unit Embedded Trace Macrocell (ETM) in CoreSight by instrumenting protection and loop functions. The protection function detects runtime environments, efficiently fills the trace buffer, and employs various protection strategies like PID (process identifier) replacement and PIE+STRIP+ASLR. Meanwhile, the loop function triggers overflows efficiently based on context-based calculations and anti-ETM loop. Our evaluation demonstrates that the overhead of ARMOR is 77.31% lower than that of OLLVM on SPEC2006. ARMOR effectively hides 54.51% of basic blocks across 16 real-world applications, triggering 113× more overflows. Moreover, we showcase two practical applications of ARMOR. Firstly, we conduct a cryptographic and cross-world attack on GnuPG 1.4.13 RSA private keys using ETM, which can steal entire keys from a program in the Secure world with a single run. ARMOR successfully reduces leaked bits by 84.5%. Secondly, ARMOR impedes hardware-assisted fuzzing by reducing throughput by 89.71% and branch coverage by 47.99%.**

***Index Terms*— ARM CoreSight, hardware tracing, software protection.**

Manuscript received 1 September 2023; revised 20 December 2023 and 26 January 2024; accepted 19 February 2024. Date of publication 4 March 2024; date of current version 6 May 2024. This work was supported in part by the National Natural Science Foundation China under Grant 62372218, Grant 62002151, Grant 62272472, and Grant 62306328; in part by the Shenzhen Science and Technology Program under Grant SGDX20201103095408029; and in part by the Natural Science Foundation of Hunan Province of China under Grant 2023RC3021. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Edgar Weippl. *(Corresponding authors: Fengwei Zhang; Xu Zhou.)*

Tai Yue is with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: yuetai17@nudt.edu.cn).

Fengwei Zhang is with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: zhangfw@sustech.edu.cn).

Zhenyu Ning is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China (e-mail: zning@hnu.edu.cn).

Pengfei Wang, Xu Zhou, Kai Lu, and Lei Zhou are with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: pfwang@nudt.edu.cn; zhouxu@nudt.edu.cn; kailu@nudt.edu.cn; zhoulcs@nudt.edu.cn).

Digital Object Identifier 10.1109/TIFS.2024.3372816

I. INTRODUCTION

MODERN CPUs are equipped with hardware tracing techniques, such as Intel Processor Trace (PT) [2] and ARM CoreSight [3]. These techniques can transparently record the instructions executed by CPU and encode the runtime information into trace packets with negligible overhead (2)-5%) [4]. Then, the users can retrieve the packets from the memory buffer and decode them to analyse the information. By examining the information with the disassembly code of binary, users can recover its control flow. These techniques have been employed in various fields, including fuzzing [5], [6], [7], [8], [9], [10], malware analysis [11], [12], [13], software debugging [14], control flow integrity [15], [16], [17], [18], [19], [20], and others [21], [22], [23], [24], [25].

While hardware tracing brings convenience to program analysis, it also poses some security risks. Firstly, with the upgrade of defense mechanisms, the Trusted Execution Environment (TEE) technologies, such as Intel Software Guard Extensions (SGX) [26] and ARM TrustZone [27], have been widely deployed in commercial processors to protect the security-critical applications [28]. Researchers also extend the confidential computing to user space such as user-level (Normal world) isolated environments [29]. Even if an attacker gains the root or kernel privileges, the attack may not directly access the memory within these secure environments. However, many defense mechanisms, such as ARM TrustZone [27] or Shelter [29], did not consider the hardware tracing or provide protection against it. As a post-root attack, hardware tracing can enable cross-privilege tracing on specific devices [30], posing significant challenges to existing protection mechanisms [30], [31], [32]. For example, Ning and Zhang explored the flaws in authentication signals of ARM CoreSight and proposed the nailgun attack [30]. This attack employs ARM ETM, the trace unit in ARM CoreSight, to non-invasively trace the Trusted Applications (TA) and steal secure data (e.g., AES encryption key) in ARM TrustZone from a non-secure application. Furthermore, adversaries can combine Intel PT or ARM CoreSight with greybox fuzzing to accelerate detecting the vulnerabilities in software [5], [6], [7], [8], [9], [10]. Particularly, inspired by the nailgun attack [30], some greybox fuzzers have been built on ARM CoreSight to fuzz the TA in ARM TrustZone [33], [34], [35]. Therefore,

protecting the software against attackers who can exploit hardware tracing is crucial in specific scenarios.

However, **some existing protection techniques (e.g., anti-debugging, anti-tracing, and code obfuscation) may not effectively achieve this goal**. Traditional anti-debugging or anti-tracing techniques usually focus on identifying external debuggers or attackers and preventing them [36], [37], [38], such as measuring the time of executed instructions and examining the process of debugger. However, hardware tracing can bypass these detections due to its transparency and negligible runtime overhead [11], [14]. Some tools even leveraged hardware tracing to bypass the anti-debugging techniques [14], [31], [32]. A possible way to detect and disable the hardware tracing techniques is accessing and configuring their registers, while these registers are only accessible for at last kernel-privilege process rather than the software running at the user level without the root-privilege ability (even in the Secure world). Code obfuscation techniques primarily aim to protect software against malicious modifications or reverse-engineering, rather than hardware tracing [1], [39], [40], [41], [42], [43], [44]. Though the obfuscated binaries challenge attackers in understanding program semantics, we argue that code obfuscation cannot prevent attackers from obtaining information about the executed instructions and incurs heavy overhead (we will prove this in our evaluation).

To address software security concerns posed by hardware tracing, in this paper, we propose a new direction of software protection, named **anti-hardware tracing**. We aim to hide some runtime information of the programs under hardware tracing, such as the control-flow information of a secure applications in TEE. We leverage the limitations of modern tracing techniques: **the trace buffer overflow issue**. Specifically, modern CPUs have higher bandwidth compared to memory, which poses a challenge in directly writing trace data to memory without loss. To address this, embedded hardware tracing units like Intel PT and ARM ETM in CoreSight utilize on-chip trace buffers, such as the 64 KB SRAM on ARM Juno R2, to temporarily store trace data with low latency. Then the trace buffer exports the trace data to memory at a constant speed. However, if the trace units generate an excessive amount of trace data at a high frequency, the trace buffer might overflow and raise a signal to stop trace units for a while until it recovers from the overflow. During this time, the trace units are lockout while CPUs are executing the instructions, resulting in the loss of trace data. We define this as **escaping from hardware tracing**. Additionally, certain runtime information, such as conditional branches and destinations of indirect branches, can only be determined by decoding tracing packets rather than through static analysis. We define these instructions as **implicit-semantics points** (return instruction is regarded as the indirect branch in this paper). *When executing these instructions during trace buffer overflow, the tracing units may miss this information.* Therefore, hiding specific runtime information under hardware tracing becomes feasible by frequently triggering trace buffer overflow before reaching these implicit-semantics points.

However, efficiently triggering trace buffer overflows poses a challenge due to the careful design of trace units and buffers by vendors. Inducing overflows in many programs under normal execution can be difficult. To address this challenge, we analyse the workflow of trace buffer through model building and experiments in Section III. Then we point out **three principles** for efficiently triggering trace buffer overflow: **1) Numerous branches in program.** Techniques like Intel PT and ETMv4 conduct branch tracing, particularly the destinations of indirect branches. *Numerous executed branches, particularly indirect branches, can make trace units generate numerous trace data.* **2) High-speed execution.** *The program needs to run at high speed to generate trace packets faster than the bandwidth at which the trace buffer outputs them.* This ensures that the trace buffer can potentially overflow. **3) High-water mark of the trace buffer.** *Compared to an empty trace buffer, the buffer with a high-water mark of trace data triggers the overflow more easily.*

In this paper, we take ARM CoreSight as an example to develop a compiler-level anti-hardware tracing framework named **ARMOR**, as powerful attacks have been implemented through CoreSight [30]. Developers can compile and reinforce the software with ARMOR. Specifically, for satisfying the proposed principles, ARMOR instruments two functions in software: **protection function** and **loop function**. The protection function is inserted in the program entry to measure the execution speed of instructions for ensuring that the program is running at a high speed (principle (2)), and fill the empty trace buffer efficiently with our elaborate anti-ETM loop (principle (3)). It also conducts two protection strategies including *PID replacement* and *PIE+STRIP+ASLR*. Then ARMOR instruments the loop functions before the implicit-semantics points (and the user-specific points) in the program to hide crucial runtime information. The loop function also contains the anti-ETM loop (principle (1)), which is designed according to the features of ETM for efficiently generating trace data. The function utilizes a context-based mechanism to calculate the times of loops for triggering the overflow and avoiding heavy overhead.

We conduct comprehensive experiments to evaluate ARMOR. Compared to OLLVM [1], a typical code obfuscation tool, ARMOR introduces 77.31% lower overhead on SPEC2006 and is more effective in hiding the runtime information under ETM on 16 real-world applications. To prove the practicability of ARMOR, we employ it in resisting cryptographic attacks and impeding hardware-assisted fuzzing. Using ETM, we demonstrate an attack to extract the RSA private keys from a TA running GnuPG 1.4.13 in the Secure world. Compared to some side-channel attacks [45], attackers only need to run the program once to capture all the bits of a 2,048-bit key by ETM, which proves the security threats brought by hardware tracing. Fortunately, ARMOR can effectively resist this attack. We also utilize ARMored-CoreSight [8] to evaluate ARMOR in anti-fuzzing. The results shows that ARMOR can reduce the efficiency of fuzzer significantly. Finally, we discuss the limitations of ARMOR and compare ARMOR with other software protection techniques.

In summary, this paper makes the following contributions.

- We propose anti-hardware tracing technique. Unlike existing protection techniques which are useless under transparent hardware tracing, our technique utilizes the

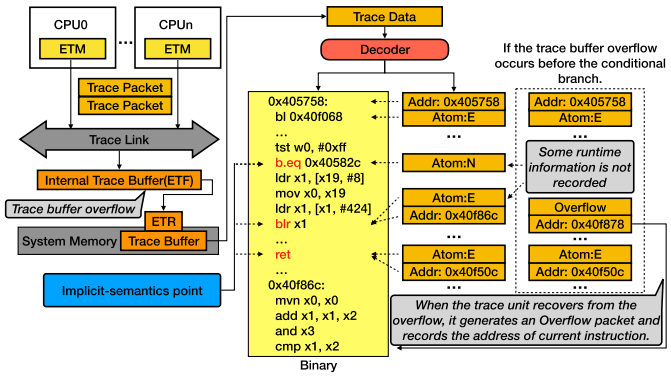

Fig. 1. The typical architecture of ARM CoreSight.

trace buffer overflow to effectively bypass the transparency and protect the runtime information.

- We analyse the buffer overflow and point out three principles in efficient anti-hardware tracing: Numerous branches in the program to generate trace data, high-speed execution of the program to accelerate the generation, and high-water mark of trace buffer for efficient triggering overflow.
- We design and implement a framework ARMOR on ARM Juno R2. ARMOR instruments the protection and loop functions to meet the three principles for triggering the overflow before the implicit-semantics points.
- We conduct comprehensive evaluations on ARMOR. ARMOR is effective in concealing the runtime information by hiding 54.51% basic blocks and 50.91% addresses of indirect branches on 16 real-world applications.
- We conduct a cross-world attack on the RSA keys of a TA by ETM and resist this attack by ARMOR. ARMOR reduces 84.5% bits of keys stolen by attackers. ARMOR also impedes the hardware-assisted fuzzer by reducing 47.99% branches.
- We open the source of ARMOR at https://github.com/MoonLight-SteinsGate/Armor.

The remainder of this paper is structured as follows. In Section II, we introduce the background of the hardware tracing technique and illustrate our threat model. In Section III, we build a model of trace buffer overflow and conduct some experiments to prove that. In Section IV, we present the design of ARMOR. In Section V, we present experimental results, including a cryptographic attack based on ETM and the corresponding protection of ARMOR. In Section VI, we take some discussion about ARMOR and anti-hardware tracing. We present related work and conclude in Sections VII and VIII, respectively.

II. Background and Threat Model

A. Hardware Tracing Technique

For capturing the executed instructions inside the processor, modern multi-core processor architecture and system have been integrated with hardware tracing modules, such as Intel PT [2] and ARM CoreSight [3]. Fig. 1 shows a typical architecture of ARM CoreSight implemented in ARM Juno R2 [46]. Generally, the trace units (i.e., ETM in ARM CoreSight) capture the runtime information by monitoring the corresponding cores and compress the trace data into trace packets to reduce the amount of trace data. Then, the units output the packets to an internal trace buffer, such as *Embedded Trace FIFO* (ETF), for addressing the significant bandwidth problem. The packets are fed into the *Embedded Trace Router* (ETR), which routes the trace data to the user-configurable buffer in system memory [47]. Finally, the users decode the trace packets to retrieve the accurate control flow with the help of additional binary disassembly [4].

Particularly, there are three crucial points in modern hardware tracing techniques: 1) **Branch tracing**. State-of-the-art techniques such as ETMv4 and Intel PT usually support branch tracing, which records the instructions that change the control flow of programs, including branches and exceptions [4]. Taking the binary in Fig. 1 as an example, starting from an entry (0x405758), by recording whether the conditional branch is taken or not taken (e.g., Atom packets in ETM) and the destinations of branches (e.g., Address packets in ETM), the analyzer can accurately recover the control flow with the disassembly code. Since the addresses of direct branches can be determined in the instructions (e.g., instruction at 0x405758), popular techniques usually record the destinations of indirect branches. Therefore, we define the conditional branches and indirect branches as *implicit-semantics points*, where the absence of tracing information in these points may lead to the unrecoverable control flow. For example, in Fig. 1, an overflow occurs before executing a conditional branch. When ETM recovers from this overflow, the program has executed several instructions and is executing the instruction at 0x40f878. As a result, information on the executed branches during the overflow is absent.

2) **Trace buffer overflow**. Existing techniques support capturing and transporting trace packets to the memory of host devices. However, due to the powerful processors in modern chips, trace units can generate data in the range of hundreds to thousands of MB/s (depending on trace filters, trace packets, and packet generation frequency) [4], which may exceed the bandwidth of memory writing. Therefore, existing techniques usually utilize the on-chip internal buffer to temporarily store trace data with low latency, which can flatten the bursts for bandwidth requirements in memory. However, limited by the capacity of this dedicated trace buffer (e.g., 64 KB of ETF in ARM Juno R2), there would be an overflow in the buffer if trace units generate lots of trace data in a short time, causing the loss of trace data. Therefore, we consider protecting the programs from hardware tracing by frequently triggering the trace buffer overflow. It is worth noting that this buffer refers to the internal trace buffer, of which the size is fixed, rather than the buffer configured by the user in the system memory to fetch the trace data.

3) **Kernel-privilege user**. Generally, hardware tracing modules can be employed by configuring their registers, which requires at least kernel privileges (or root privileges on specific Linux systems). Taking ARMv8-A architecture as an example, a core has four exception levels (EL0-EL3), which are used by applications, kernels, hypervisors, and secure monitor, respectively [48]. In addition, ARM introduces two CPU states by ARM TrustZone: the Normal world (also named Non-

TABLE I
TRACE ELEMENTS AND TRACE PACKETS OF ETM

Trace Packet	Trace Element	Size(bytes)
Exact Match Address	1 Address	1
Short Address	1 Address	2,3
Long Address	1 Address	5,9
Address with Context	1 Address	6-18
Exception	1 Address	4-22
Atom Format 1	1 Atom	1
Atom Format 2	2 Atom	1
Atom Format 3	3 Atom	1
Atom Format 4	4 Atom	1
Atom Format 5	5 Atom	1
Atom Format 6	4-24 Atom	1

secure world) and the Secure world [48]. No matter whether a non-root process is running in Secure EL0 or Normal EL0, it cannot access the registers of CoreSight.

B. Embedded Trace Macrocell

ETM is a standard trace unit in ARM CoreSight. The trace elements generated by ETM can be classified into several categories, such as synchronization and basic program flow. The basic program flow elements are related to control flow, including Atom, Exception, Address, and Context. The Address element denotes the destination of a branch, and the Atom element represents whether a branch is taken or not taken (E is taken and N is not taken). Table I lists the trace packets related to control flow.

Specifically, ETM takes some compression techniques to generate the packets related to Address and Atom [49]. First, it stores up to three recent addresses in a queue, denoted as address_reg[i] (i is 0, 1, 2). Each time a new Address element is generated, ETM updates the three addresses in the queue. Before generating an Address packet, ETM compares the new address value with the three addresses in the queue. If the new address exactly matches any of them, ETM will output an Exact Match Address packet instead of other Address packets, which costs only one byte. Moreover, the Short Address and Long Address packets only contain the least significant bits that have changed from the most recently traced address stored in address_reg[0]. If the number of changed bits is no more than 17, ETM only generates a Short Address packet, which costs no more than three bytes. Furthermore, ETM always uses one byte to generate the Atom packets, which can mostly denote 24 Atom elements. By these techniques, ETM minimizes the amount of trace data to avoid trace buffer overflow as much as possible.

According to the manual of ETM [49], ETM also supports some features, including context ID tracing, global timestamping, and **branch broadcasting**. Generally, ETM or Intel PT only records the destination addresses of indirect branches in the default mode. Unlike PT, ETM can record the destinations of direct branches under the branch broadcasting mode [49]. Furthermore, users can configure the context ID comparators and address range comparators of ETM to trace the assigned process within the specified address range [49]. Notably, in addition to instruction tracing, ETM also supports data tracing. However, this mode may generate an excessive amount of trace data. Many SoCs, such as ARM Juno R2, do not implement this function.

Particularly, there are four authentication signals in ARM CoreSight to manage the debugging or tracing privileges [49]. Among them, the signals NIDEN and SPNIDEN determine whether ETM can non-invasively trace the code in the Normal world and Secure world, respectively. Generally, the signal NIDEN is enabled by default to support users to trace and analyse their applications in the Normal world. However, the signal SPNIDEN may be ignored and enabled by manufacturers on many devices, leaving enough wiggle room for an attacker to launch attacks to the Secure world [30].

C. Threat Model and Assumptions

In this paper, we focus on protecting programs from malicious attackers who abuse the ETM. We trust the secure monitor in EL3, hypervisors in EL2, and the components in Secure world (e.g., TA and TEE OS). We also trust the hardware provided by the manufacturer. We consider the adversaries as the kernel-privilege attackers in the Normal world (i.e., Normal EL1). They have full control of the untrusted OS. Their goal is to leak the secret data of secure software, such as the control-flow information of a cryptography library. The software is a TA running in Secure EL0 or a user-privilege (i.e., Normal EL0) secure program running in the Normal world isolated environments (e.g., Shelter [29]). Under these secure environments, even the kernel-privilege attackers cannot directly access the memory of the protected process [27], [29]. However, they can maliciously utilize the ARM CoreSight to conduct malicious attacks. We also assume that the signals SPNIDEN and NIDEN in ETM are enabled by the SoC manufacturers. In fact, according to the investigations in previous works [30], [33], many devices adhere to this assumption (i.e., enabling these signals). Therefore, the sophisticated attacker can follow the nailgun attack [30] to mount a kernel module and employ ETM to trace the protected software non-invasively.

Furthermore, this paper focuses on mitigating the attacks based on hardware tracing. We do not consider physical attacks (e.g., bus snooping attacks [50]) and invasive debugging (e.g., utilizing JTAG). Side channel attacks are also out of our scope in this paper [51], [52], [53], [54], [55]. Particularly, for the interrupt-based attacks [53], [54], [55], we assume that the interrupts raised by kernel-privilege attackers (i.e., non-secure interrupts) to the core running the TA are blocked by the interrupt controller, which is feasible by conducting specific configuration in the Secure world [56], [57], [58]. For example, according to the manual of Trusted Firmware-A (TF-A) [58], we can block the non-secure interrupts by configuring the exception mask bits I and F in the PSTATE register of a core when entering the Secure world. This can make the processes in the Secure world monopolize the core, and even the kernel-privilege attackers cannot modify the register or interrupt the TA. We will discuss more details about the interrupt-based attacks and how to mask the non-secure interrupts in Section VI.

Moreover, we argue that the anti-hardware tracing technique should not prevent legitimate users from using ETM to trace

and analyse applications in the Normal world. Particularly, since the protected software is running as an EL0 process in the Secure world or isolated environment, it cannot perceive or disable the ETM through reading or writing the registers of CoreSight, which is required for at least EL1 privileges. Moreover, enforcing the access control to the tracing memory and preventing malicious programs from directly accessing it may be unfeasible as it is challenging to determine whether the access is from the attacker. Furthermore, the above countermeasures could potentially impact the normal usage of ETM by legitimate users and can still be bypassed by sophisticated attackers. For example, if the TEE OS or isolated environments disabled the ETM when the protected software traps in EL1-EL3, attackers can bypass this by restarting ETM.

III. Escaping From Hardware Tracing

A. Workflow of Trace Buffer

We first build a model to analyse the critical conditions of trace buffer overflow. For the program P running on one core C with the configurations proposed in Section II-C, the amount of trace data generated by ETM in time t can be denoted as $G(t, P, C)$ (briefly as $G(t)$). For the trace buffer, we represent its size as L_{buffer} and its constant bandwidth of exporting trace streams to the memory as V. The amount of trace data in trace buffer in time t is $D(t)$, deduced as:

$$D(t) = \int_0^t G(s)ds - Vt \tag{1}$$

Therefore, the critical conditions of the first trace buffer overflow occurring in time t_1 can be denoted as:

$$D(t_1) = \int_0^{t_1} G(s)ds - Vt_1 = L_{buffer} \tag{2}$$

When the trace buffer overflow occurs, the trace units will hang up until the trace buffer drains some trace data and recovers from the overflow. It does not mean that all of the trace data in the buffer will be drained, where we denote the amount of drained data as Δn. We utilize t_2 to represent the time that trace units recover from the previous overflow in time t_1 and Δt to denote this time interval:

$$\Delta t = t_2 - t_1, \Delta n \leq V\Delta t \leq L_{buffer} \tag{3}$$

Based on Equation (2) and (3), $D(t_2)$ can be deduced as:

$$D(t_2) = \int_0^{t_1} G(s)ds - Vt_1 - \Delta n = L_{buffer} - \Delta n \tag{4}$$

From time t_2, the trace units recover and start to trace the program until the trace buffer overflow occurs again in time t_3. We can calculate $D(t_3)$ as:

$$D(t_3) = D(t_2) + \int_{t_2}^{t_3} G(s)ds - V(t_3 - t_2) = L_{buffer} \tag{5}$$

Based on Equation (4) and (5), we can further deduce that:

$$\int_{t_2}^{t_3} (G(s) - V)ds = \Delta n \leq L_{buffer} = \int_0^{t_1} (G(s) - V)ds \tag{6}$$

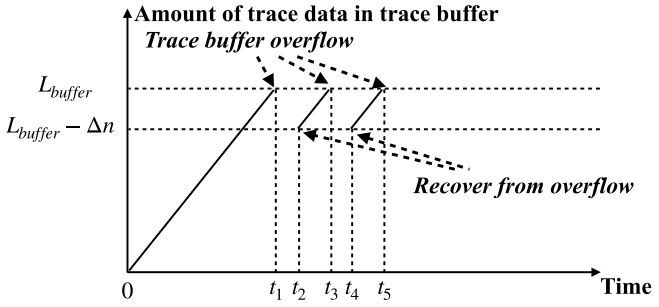

Fig. 2. The model of trace buffer overflow.

Assuming that the trace units recover from the k overflow in time t_{2k} and the (k+1) overflow occurs in time t_{2k+1}, we can extend Equation (6) as:

$$\int_{t_{2k}}^{t_{2k+1}} (G(s) - V)ds = \Delta n \leq L_{buffer} = \int_0^{t_1} (G(s) - V)ds \tag{7}$$

From starting tracing to time t_{2k+1}, the trace buffer overflow occurs $(k+1)$ times. Then we can calculate the percentage of the time that trace units hang up (denoted as p) as:

$$\begin{aligned} p &= \frac{k\Delta t}{t_{2k+1}} \\ &= \frac{k\Delta t}{t_{2k+1} - t_{2k} + t_{2k} - t_{2k-1} + \cdots + (t_2 - t_1) + t_1} \\ &= \frac{k\Delta t}{\sum_{i=1}^{k}(t_{2i+1} - t_{2i}) + t_1 + k\Delta t} \end{aligned} \tag{8}$$

From Equation (7) and (8), considering that V, Δn, and Δt are constant, the larger $G(t)$ is, the closer t_{2k+1} and t_{2k} are to. Then the p are closer to 100%, meaning more trace data loss. In contrast, if the execution speed of instructions in the program is too slow or the program contains too few branch instructions, the bandwidth of trace data generated by trace units will be less than the bandwidth of trace buffer (i.e., $G(t) < V$), never causing the overflow.

Moreover, from Equation (7), since the drained trace data during the overflow is no more than the size of the buffer, the time interval to trigger the first trace buffer overflow is no shorter than that of subsequent triggering. Particularly, we can presume $G(t)$ is close to a constant V_g. Then the model of trace buffer can be described as the curve in Fig. 2.

B. Trace Buffer Overflow on ARM Juno R2

We apply our model to measure the three crucial parameters on ARM Juno R2, including the bandwidth V, the recover time Δt, and the amount of drained data Δn, which are the basis of implementing ARMOR on Juno R2.

We utilize ETM to trace a binary containing a simple loop for 1, 000, 000 times by enabling the branch broadcasting on ARM Juno R2, which is listed in Fig. 3. From Fig. 3, almost in each loop, ETM generates 2 bytes trace data: an Atom Format 1 packet with the Atom element E and an Exact Match Address packet with the Address element 0x40062c. Moreover, since the number of loops is large enough, we can consider the execution time of each loop to be constant. The workflow of ETF will be approximate to the model in Fig. 2.

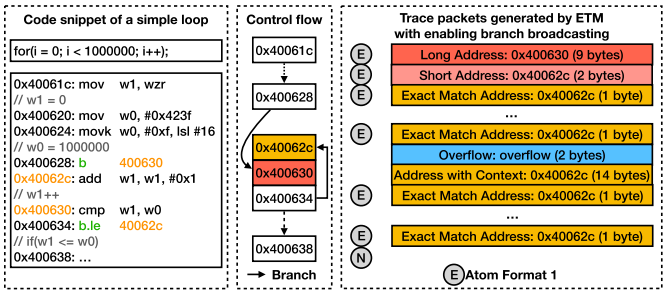

Fig. 3. The code snippet of the loop and the generated packets of ETM.

For measuring the parameters, we record the execution time of $1,000,000$ loops and the amount of trace data, denoted as T and M, respectively. We can approximate the execution time of each loop as $\frac{T}{1000000}$. We also count the number of loops and overflows traced by ETM, represented as N_{loop} and N_{ow}, respectively. Then the number of loops executed when the trace units hang up can be calculated as $\frac{1000000-N_{loop}}{N_{ow}}$. We can utilize it to measure the recovery time Δt. We count the number of loops and the amount of trace data between two consecutive overflows occurring in t_{2k-1} and t_{2k+1}, denoted as N_k and D_k, respectively. Based on our model, we calculate V, Δt, and Δn as:

$$\begin{cases} V \approx \dfrac{M}{T} \\ \Delta t \approx \dfrac{1000000 - N_{loop}}{N_{ow}} * \dfrac{T}{1000000} \\ \Delta n \approx \dfrac{\sum_{k=1}^{N_{ow}} (D_k - N_k * \frac{T}{1000000} * V)}{N_{ow}} \end{cases} \tag{9}$$

Then we execute the binary on two different cores. One is Cortex-A72 with 1.2 GHz and 2 MB of L2 cache, while the other is Cortex-A53 with 0.95 GHz and 1 MB of L2 cache. We repeat the experiments for 1000 times to reduce the randomness introduced by environments, decode the trace data by ptm2human [59], and analyse the trace packets.

Table II shows the average results. From Table II, the bandwidth of ETF V is about 503.8 MB/s. It takes ETF about 172.4 us to recover from one overflow by draining about 83 bytes of trace data, during which the program can escape from hardware tracing. Furthermore, ETM traces $330,278$ loops on Cortex-A72, less than those on Cortex-A53 (about $408,353$). The reason is that the core with higher performance executes more instructions during the recovery time, leading to more trace data loss. Therefore, when executing the same instructions, the higher execution speed may bring more protection to the program against hardware tracing.

In summary, by building the model and conducting experiments on trace buffer overflow, we prove that it is viable to hide the runtime information and achieve anti-hardware tracing by triggering lots of overflows, where the following **two principles** should be satisfied: *1) there are numerous branches, particularly indirect branches, in the programs. 2) programs are not allowed to run at a low speed (e.g., under debugging or on a processor with poor performance).* And an **additional principle** to efficiently produce overflows is also proposed: *3) running the program in the trace buffer with a high-water mark is easier to trigger the overflow.*

TABLE II
THE AVERAGE RESULTS OF 1,000 TRAILS ON CORTEX-A72 AND A53

Core	T(ns)	M(B)	N_{loop}	N_{ow}	N_k	D_k(B)	V(MB/s)	Δn(B)	ΔT(ns)
A72	1,680,948	894,274	330,278	6,448	43.5	122.9	507.4	84.1	174.7
A53	2,126,923	1,115,509	408,353	7,395	46.7	133.5	500.2	81.4	170.1

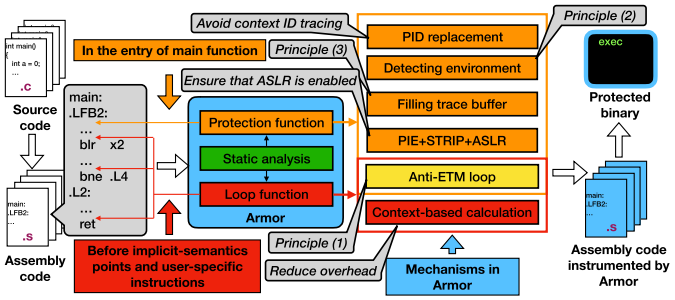

Fig. 4. Overview of ARMOR.

IV. DESIGN AND IMPLEMENTATION

A. Overview

Inspired by our model of trace buffer overflow and the experiments on ARM Juno R2, we design and implement our anti-hardware tracing framework, named ARMOR. Fig. 4 shows the overview of ARMOR. For conducting some protection strategies and efficiently triggering lots of overflows, ARMOR analyses the assembly code and instruments two crucial functions in the program to satisfy the three principles, including **protection function** and **loop function**.

For satisfying principles (2) and (3), ARMOR inserts the protection functions in the entry of main function, where the trace buffer may be empty. The protection function executes some loops and measures the execution speed of loops to *detect environment*, decides whether to stop the process, and executes the *anti-ETM loop* to quickly *fill trace buffer* and trigger the overflow. Moreover, since ETM supports tracing a specific process within an address range, we propose two countermeasures to thwart attackers from exploiting these features: 1) *PID replacement*. This function changes the PID by forking a child process that replaces the parent process, which can hide runtime information under context ID tracing. 2) *PIE+STRIP+ASLR*. ARMOR compiles the program as position-independent executable (PIE) code and strips the symbolic information. Then the protection function checks the status of Address Space Layout Randomization (ASLR) and ensures that the program runs only when ASLR is enabled. This brings challenges for attackers to pre-specify the traced address range and analyse the runtime information.

Since the protection function has filled the trace buffer in a high-water mark, according to principle (3), the program only needs to execute a few branches to trigger the subsequent overflow. For protecting some crucial control-flow information, ARMOR inserts the callers of loop functions before the implicit-semantics points and some user-specific instructions to trigger the overflow and stall the ETM when executing these branches. The loop function contains the *anti-ETM loop*, which is designed according to the mechanism of ETM and can generate significantly more trace data than the common loops, satisfying the principle (1). Moreover, the times of loops

are calculated by the *context-based calculation* mechanism to reduce the overhead introduced by loop functions.

Finally, the software compiled by ARMOR can hide some runtime information and withstand the attackers who employ ARM CoreSight to trace it. The details of these two functions in ARMOR are introduced as follows.

B. Protection Function

1) Detecting Environment: Algorithm 1 shows the logic of the protection function. Specifically, the protection function executes the anti-ETM loop, which will be introduced in Section IV-C, for MEASURE_LOOP times and measures the execution time use_time (Line 2-4) by reading the cntvct_el0 and cntfrq_el0 registers. Then it calculates the amount of trace data outputed by ETF in one loop by utilizing the bandwidth of ETF (V) (Line 5), denoted as consumed_bytes. ARMOR detects the environment and determines whether the program is running at a high speed by comparing the consumed_bytes with the pre-configured threshold CONSUME_MAX. If the trace data consumed by ETF is more than CONSUME_MAX in one loop, which means that the program spends much time on executing the loop. The protection function will terminate the program (Line 6-8) to prevent the attackers who plan to avoid trace buffer overflow by slowing down the program (e.g., running the program in a poor-performance core).

Algorithm 1 Protection Function

```
 1: pid_replacement()
 2: cur_time = get_cur_time()
 3: anti_ETM_loop(MEASURE_LOOP)
 4: use_time = get_cur_time() - cur_time
 5: consumed_bytes = use_time * V / MEASURE_LOOP
 6: if consumed_bytes > CONSUME_MAX then
 7:     exit(-123)
 8: end if
 9: aslr_check()
10: fill_speed = LOOP_BYTES - consumed_bytes
11: loop_times = BUFFER_LENGTH / fill_speed
12: anti_ETM_loop(loop_times)
13: last_time = get_cur_time()
Output: fill_speed, last_time
```

2) Filling Trace Buffer: If the function passes the check, this denotes that the execution speed of the program can satisfy principle (2) to achieve anti-hardware tracing. According to principle (3), it is easier to trigger overflows when the trace buffer is at a high-water mark than that is empty. Therefore, the protection function will execute enough indirect branches to quickly fill the empty trace buffer for triggering overflows. Specifically, the function accurately calculates the speed of the anti-ETM loop to fill the buffer (i.e., fill_speed) based on the amount of trace data generated by ETM and exported by ETF in one loop (i.e., LOOP_BYTES and consumed_bytes). Then it calculates the minimal times loop_times of the anti-ETM loop to trigger the overflow from an empty buffer by utilizing the size of the buffer and this speed (Line 10-11), and executes the loops (Line 12). Finally, the function saves the speed of filling trace buffer and current timestamp in a global array to calculate the amount of outputted trace data in next loop function.

3) PID Replacement: In addition, to withstand the attackers who configure the context ID to accurately trace the program, we add the pid_replacement() function to create a child process for replacing the parent process (Line 1). This PID replacement mechanism can hide almost all control flow under the context ID tracing of ETM. It should be noted that this strategy may not be applicable to some TAs as many TEE OSs may not support the fork mechanism. However, this may be achieved by enhancing the TEE OSs further.

4) PIE+STRIP+ASLR: To conduct this strategy, the protection function also checks the status of ASLR, such as reading /proc/sys/kernel/randomize_va_space (Line 9) in Linux. The TA may require some specific handling. If the ASLR is disabled, the function will terminate the program. Moreover, to resist the attackers who want to detect and disarm this strategy by reverse engineering, we take some tricks in this strategy, such as transferring the string /proc/sys/kernel/randomize_va_space as implicit data flows.

5) Instrumentation Position: We instrument the protection function in the entry of main function, which brings two benefits. First, if the environment is unable to support anti-hardware tracing, protection function inserted in the beginning of main function can terminate the program immediately, preventing executing the functional code to protect almost all of the useful runtime information. The PID replacement can also derive benefits from this. Second, the trace buffer may be empty when the program runs in this position, where the protection function can fill the buffer to a high-water mark to satisfy principle (3).

By the protection function, ARMOR conducts several protection strategies, ensures the program runs at a high speed, and guarantees the buffer in a high-water mark (i.e., principles (2) and (3)), providing the environment for efficiently triggering overflow in loop functions.

C. Loop Function

To generate enough trace data for triggering overflows and avoid the heavy overhead, we design the context-based calculation mechanism to calculate the loop times of the anti-ETM loop in the loop function.

1) Context-Based Calculation: Algorithm 2 shows the logic of the loop function (i.e., armor_loop in Fig. 5). Specifically, based on the timestamp, bandwidth of ETF, and amount of trace data drained in one overflow, the loop function firstly calculates the amount of trace data outputted by ETF (i.e., output_bytes) from the last anti-ETM loop to this (Line 1-2). Then it estimates the times of loops to trigger the overflow according to the outputted trace data and speed of filling trace buffer, and executes the anti-ETM loop (Line 3-4). Finally, the function updates the timestamp (Line 5).

2) Anti-ETM Loop: For efficiently generating the trace data in one loop, we elaborately design the anti-ETM loop based on the mechanism of ETM, which is shown in Fig. 5. In detail, we utilize three general registers to store the addresses of three bouncer functions before entering the loop, which contains only one `ret` instruction. During each loop, the program indirectly calls the three functions and returns from them, executing six indirect branches. Notably, we increase the offset

Algorithm 2 Loop Function

Input: fill_speed, last_time
1: cur_time = get_cur_time()
2: output_bytes = MIN((cur_time - last_time) * V + DRAINED_BYTES, BUFFER_LENGTH)
3: loop_times = output_bytes / fill_speed
4: anti_ETM_loop(loop_times)
5: last_time = get_cur_time()
Output: fill_speed, last_time

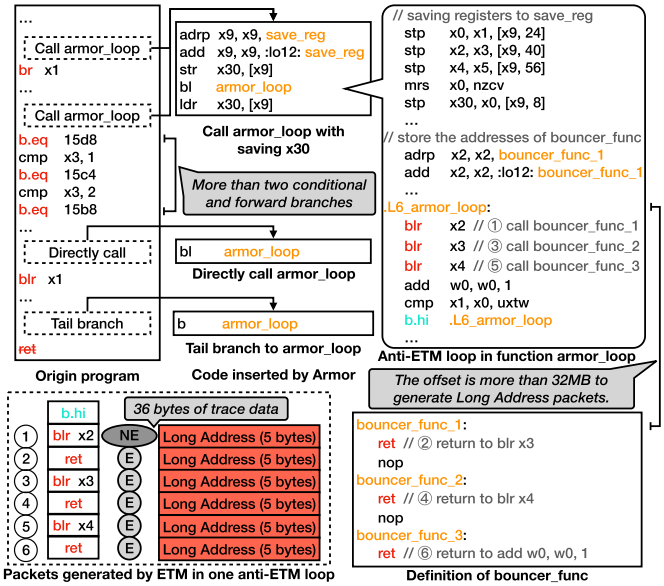

Fig. 5. The details of loop functions.

between the indirect call instructions in the loop function and the definition of bouncer functions to more than 32 MB for generating the Long Address packet in each branch, as shown in the right part of Fig. 5. Besides, to avoid ETM producing the Exact Match Address packet, we insert three bouncer functions to ensure that the new address is not in the queue of ETM. By this design, ETM generates at least 36 bytes of data in one loop, while the protected program only executes nine instructions, which ensures generating lots of trace data in acceptable overhead.

3) Instrumentation Position: By producing overflows in the loop function, ARMOR provides the program with precious intervals to escape from hardware tracing, which is about 172us in ARM Juno R2. To utilize the intervals, we call the loop function before: 1) The implicit-semantics points, including indirect branches and conditional branches. 2) Some user-specific instructions. However, some instrumentations may incur heavy overhead (e.g., calling loop function in a recursive function). For reducing the overhead, ARMOR conducts a simple but effective static analysis in assembly code to determine the instrumented points. Specifically, ARMOR does not instrument the loop functions in the recursive functions. In other functions, ARMOR inserts the loop functions before all of the indirect branches. However, there are usually more conditional branches than indirect branches in programs and many conditional branches are backward branches, which may generate loops and introduce heavy overhead if we instrument the loop functions before them. Hence, we only instrument the conditional branches which satisfy the following conditions: 1) This instruction is followed by more than two consecutive conditional branches. 2) There are no more than two instructions between the adjacent conditional branches. 3) This instruction and its followed conditional branches are not backward branches. The left part of Fig. 5 has shown some examples of instrumentation. Benefited from this, when the program executes these instructions, ETM may be in the stalling due to the overflow of ETF caused by the loop function, which protects the crucial control flow.

TABLE III
PARAMETERS UTILIZED IN ARMOR

Parameters	Value	Description
MEASURE_LOOP	4,096	Number of loops to measure the execution speed
V	512 MB/s	Bandwidth of ETF
CONSUME_MAX	26 bytes	Threshold of consumed trace data in one anti-ETM loop
LOOP_BYTES	36 bytes	Trace data generated by ETM in one anti-ETM loop
BUFFER_LENGTH	64 KB	Size of buffer in ETF
DRAINED_BYTES	83 bytes	Trace data drained in one overflow

By inserting the loop functions (i.e., principle (1)), ARMOR executes the anti-ETM loops (i.e., principle (1)) based on a context-based calculation mechanism, which can trigger overflow frequently with introducing little overhead.

D. Implementation

We implement ARMOR as a compiler-wrapper of gcc. For some parameters utilized in the protection and loop function, we define them according to the corresponding parameters on ARM Juno R2, listed in Table III.

To maintain the semantics of the program unchanged after our instrumentation, we define a global array save_reg to save and restore these registers which may be changed in protection function and loop function (e.g., x30, x0, and nzcv). We reserves one general register x9 by using the -ffixed-x9 argument in gcc to store or load the address of this array. The assembly code to call the loop function inserted by ARMOR is shown in the left part of Fig. 5 (calling the protection function is similar). We save the value of x30 in save_reg[0] and reload this value after returning from our function, keeping it unchanged after our instrumentation. Particularly, in the protection and loop function, we employ the array to maintain the context by storing the values of used registers at the beginning of the function and loading these values at the end, which is different from the traditional mode that utilizes stack frames to save the context. This can prevent ARMOR from destroying the origin data in the stack. Moreover, before blr instruction, we directly call the loop function without saving x30. For the ret instruction, we transfer it as a tail branch to the loop function. These tricks can lightly reduce the size of code.

V. EVALUATION

Research Questions: We conduct comprehensive evaluations to answer the following research questions:

RQ1:How about the overhead introduced by ARMOR?
RQ2:How about the effectiveness of ARMOR in hiding the runtime information of the program under ETM?
RQ3:Can ARMOR resist some attacks based on ETM?
RQ4:Can ARMOR impede the hardware-assisted fuzzing?

TABLE IV
THE RUNTIME OVERHEAD OF GCC, ARMOR-P, ARMOR, AND OLLVM AND THE NUMBER OF INSTRUMENTED LOOP FUNCTIONS BY ARMOR ON SPEC2006

Program	Execution Time(s)				Ins.
	Base	ARMOR-P	ARMOR	OLLVM	ARMOR
400.perlbench	20.1	20.2(+0.50%)	67.0(+233.33%)	540.0(+2586.57%)	3,562
401.bzip2	40.3	40.5(+0.50%)	51.3(+27.30%)	400.0(+892.56%)	192
429.mcf	14.9	14.9(+0.00%)	15.5(+4.03%)	48.8(+227.52%)	51
445.gobmk	85.9	86.6(+0.81%)	177.0(+106.05%)	1580.0(+1739.35%)	3,814
456.hmmer	34.8	35.1(+0.86%)	42.0(+20.69%)	559.0(+1506.32%)	1,023
458.sjeng	107.0	107.0(+0.00%)	309.0(+188.79%)	1820.0(+1600.93%)	274
462.libquantum	1.02	1.02(+0.00%)	2.71(+165.69%)	22.0(+2056.86%)	155
464.h264ref	64.2	62.8(-2.18%)	264.0(+311.21%)	610.0(+850.16%)	1,429
471.omnetpp	41.8	41.9(+0.24%)	181.0(+333.01%)	400.0(+856.94%)	3,326
473.astar	75.6	76.2(+0.79%)	84.4(+11.64%)	464.0(+513.76%)	157
483.xalancbmk	50.0	50.6(+1.20%)	339.0(+578.00%)	871.0(+1642.00%)	40,029
433.milc	11.7	11.8(+0.85%)	15.5(+32.48%)	34.4(+194.02%)	352
444.namd	12.8	12.8(+0.00%)	14.2(+10.94%)	49.6(+287.50%)	332
447.dealII	21.7	21.7(+0.00%)	55.9(+157.60%)	289.0(+1231.80%)	13,184
450.soplex	4.81	4.82(+0.21%)	7.07(+46.99%)	28.9(+500.83%)	2,310
453.povray	5.34	5.32(-0.37%)	16.5(+208.99%)	53.5(+901.87%)	2,643
470.lbm	25.3	25.3(+0.00%)	25.4(+0.40%)	48.6(+92.09%)	24
482.sphinx3	8.49	8.48(-0.12%)	11.2(+31.92%)	53.0(+524.26%)	648
Mean	21.15	21.19(+0.18%)	42.26(+99.84%)	186.17(+780.31%)	815

A. Performance Evaluation

1) Experimental Setup: We employ the widely-used SPEC2006 test suites, along with *training* input, to assess the overhead of ARMOR. In our design, the overhead of ARMOR is attributable to two components: the instrumented protection function and the loop function. The protection function is executed only once at the entry point of the main function, while the loop function is executed multiple times before numerous implicit-semantic points in the program. To explore the overhead introduced by these functions, we compile all of the C and C++ benchmarks from the INT2006 and FP2006 with gcc-7.3, ARMOR-P (only instrumenting the protection function), and ARMOR (ARMOR fails to compile 403.gcc). Additionally, we also measure the performance overhead of the state-of-the-art obfuscation technique, OLLVM [1], with all strategies enabled (-mllvm -sub -mllvm -fla -mllvm -bcf).

Considering the variability of the environment, we repeated each test five times. All experiments were performed on the Taishan server, which is equipped with a 64-core Kunpeng 920 processor (2.6GHz) and 190GB RAM. The server was running openEuler-20.03 with Linux kernel version 4.19.

2) Runtime Overhead: Table IV lists the runtime overhead of ARMOR-P, ARMOR, and OLLVM. ARMOR-P introduces negligible overhead on these benchmarks (about 0.18% on average). Therefore, the overhead of ARMOR is mainly brought by the execution of loop functions. Moreover, due to the varying number of inserted loop functions in ARMOR across different programs, the overhead of ARMOR varies significantly across these benchmarks, from 0.4% to 578%. Particularly, ARMOR introduces no more than 50% overhead on nine benchmarks. We count the number of instrumented loop functions in these benchmarks and list the results in the right column of Table IV. Generally, the more loop functions are instrumented and executed, the higher the overhead introduced by ARMOR. On some benchmarks (e.g., 429.mcf and 470.lbm), ARMOR introduces no more than 10% overhead with instrumenting few loop functions. Since ARMOR supports user-specific instrumented points, the overhead of ARMOR will decrease if the user choose to conduct fewer instrumentation. Compared to OLLVM, the geometric mean execution time of ARMOR on the tested benckmarks is 42.26s, 77.31% lower than that of OLLVM (about 186.17s).

RQ1: The overhead introduced by ARMOR mainly depends on the number of instrumented and executed loop functions, which varies significantly across different programs. The protection function introduces average 0.18% overhead on SPEC2006, while the overhead of loop function ranges from 0.4% to 578% (averagely 99.84%).

B. Security Evaluation

In this section, we qualitatively analyse the security of the PID replacement and PIE+STRIP+ASLR strategies and conduct experiments to evaluate the effectiveness of ARMOR in triggering overflows and hiding information.

1) PIE+STRIP+ASLR: In practical, users usually pre-configure the address range of ETM to reduce the unnecessary trace packets. Since recovering the integral control flow requires matching the trace information with the disassembly code, we implement the PIE+STRIP+ASLR strategy to impede attackers in these situations. By forcing the protected binaries to run under the ASLR, the Address elements generated by ETM vary in different executions, which brings a great challenge for attackers to pre-configure the address range. Though the kernel-privilege attackers can bypass the ASLR checking when tracing the programs in Normal EL0, the stripped PIE and numerous overflow introduced by ARMOR bring great trouble to attackers for recovering control flow. And the attackers are unable to disable the ASLR implemented by the TEE OS when tracing the TA. Therefore, we recommend the TEE OS vendor to provide support for ASLR to reinforce the protection. Fortunately, popular TEE OSs such as OP-TEE [60] have supported this feature.

2) PID Replacement: To accurately trace a process, users usually employ the context ID tracing. For example, some hardware-assisted fuzzers (e.g., ARMored-CoreSight [8]) have taken this way. Our PID replacement strategy can change the context ID to hide almost all of the runtime information in these scenarios. To bypass this strategy, attackers may have to track all processes instead of context ID tracing. This may introduce some noisy packets generated by other processes and bring some trouble to attacker in analysing trace data.

3) Effectiveness in Triggering Overflows: To evaluate the effectiveness of ARMOR in leading to trace buffer overflow and hiding runtime information, we select 16 real-world applications (listed in Table V) which are different in functionality and commonly used in the real world or other works [6], [61], [62]. To get enough inputs, we employ AFL [63] to test these programs on the TaiShan server for 24 hours and collect all the seeds as the testcases. Then we compile these selected applications by gcc, OLLVM, and ARMOR, respectively, and run them to reproduce these collected testcases on ARM Juno R2 development board, which runs Linux-5.3 and gcc-9.2.0 with 8 GB

TABLE V
THE CONFIGURATIONS OF THE PROGRAMS AND AVERAGE RESULTS OF TRACING INFORMATION ON THE PROGRAMS COMPILED BY GCC, ARMOR, AND OLLVM

Program	Testcases	Trace Data(bytes)			Overflows			Basic Block			Address		
		Origin	OLLVM	ARMOR	Origin	OLLVM	ARMOR	Origin	OLLVM	ARMOR	Origin	OLLVM	ARMOR
objdump-d @@	8,795	432,493	2,165,373	3,802,274	151	0	22,969	257,806	6,056,942	138,640	28,777	31,656	20,371
readelf -a @@	7,312	60,121	128,882	1,225,001	0	0	7,062	18,626	244,111	14,713	4,688	4,898	4,402
nm-new -C @@	6,189	158,912	486,429	1,069,089	0	0	6,915	79,937	1,081,006	26,630	17,175	20,344	6,063
size @@	2,003	43,518	114,155	856,187	0	0	5,068	19,250	235,445	11,343	4,717	5,363	2,695
jhead @@	327	28,971	898,046	478,273	0	0	2,916	16,017	4,013,157	16,165	1,521	1,515	1,509
nasm -f elf -o sample @@	7,173	626,553	2,748,745	5,501,063	1	0	29,907	397,891	7,015,509	207,264	53,935	78,291	33,832
pdfimages @@ /dev/null	1,653	1,661,008	4,736,564	9,722,451	35	0	63,109	1,077,526	9,743,984	398,437	164,305	213,334	70,313
pdfinfo @@	1,616	1,567,773	4,460,748	9,048,963	30	0	56,705	1,017,015	9,237,600	393,086	150,527	196,353	67,415
pdftops @@ /dev/null	1,577	1,482,603	4,184,368	8,869,894	15	0	53,209	940,364	8,462,393	402,029	133,504	176,313	66,765
pdffonts @@	1,591	1,501,648	4,318,143	8,876,360	21	0	54,263	972,692	8,943,804	406,915	137,728	181,891	69,112
tiff2bw @@ /dev/null	2,272	191,721	537,831	902,748	307	0	5,700	67,829	1,106,036	57,479	7,306	9,561	2,948
tiffinfo @@	3,377	880,111	1,429,451	2,959,830	1,258	0	19,805	137,671	1,929,800	87,983	33,569	34,629	11,591
tiff2pdf @@ /dev/null	2,157	142,396	253,507	765,638	166	0	4,879	22,443	308,262	13,034	5,825	7,211	2,240
xmllint @@	3,826	266,315	-	3,753,017	0	-	19,684	193,766	-	149,231	20,349	-	16,239
cert-basic @@	647	189,008	441,381	1,019,749	0	0	6,468	68,232	899,152	26,084	13,588	14,456	6,602
bison @@	3,192	1,602,684	-	8,535,077	1,619	-	50,279	732,008	-	389,215	113,151	-	55,122
Mean	-	677,240	1,921,687	4,211,601	225	0	25,559	376,192	4,234,086	171,140	55,667	69,701	27,326
%-Chg	-	-	+183.75%	+521.88%	-	+0.00%	+11249.93%	-	+1025.51%	-54.51%	-	+25.21%	-50.91%

*OLLVM failed to compile the xmllint and bison.

RAM on a Cortex-A72 processor cluster (two cores with 0.6-1.2 GHz CPU frequency) and a Cortex-A53 cluster (four cores with 0.45-0.95 GHz CPU frequency) [46]. For each testcase, we utilize one Cortex-A72 core with 1.2 GHz to run the program and enable the corresponding ETM. We disable the PID replacement and PIE+STRIP+ASLR strategies in this and subsequent evaluation to evaluate the effectiveness of ARMOR in triggering overflows. Specifically, we use objdump to analyse the text sections of binary and configure ETM to accurately trace these sections under the default mode by assigning the PID of the process. Then we collect the trace data and decode it with ptm2human [59].

a) Evaluation metric: We use the size of trace data and the number of overflows to evaluate the efficiency and effectiveness of ARMOR in producing trace data and triggering overflows. Moreover, we count the executed basic blocks as well as recorded addresses in the text sections of binaries to evaluate ARMOR in hiding control flow. Specifically, since ETM generates an Atom element in each branch instruction, which can differentiate the basic blocks, we calculate the number of executed basic blocks by counting the number of Atom elements behind an Address element in text sections. We do not count the meaningless blocks and addresses in the protection and loop function of ARMOR. For each binary, we calculate the arithmetic mean of these metrics.

b) Results on real-world applications: Table V lists the detailed results of these applications. In the term of effectiveness in anti-hardware tracing, ETM produces 677, 240, 1, 921, 687, and 4, 211, 601 bytes of trace data on the binaries compiled by gcc, OLLVM, and ARMOR, respectively, with recording 225, 0, and 25, 559 overflows. Compared to gcc, ARMOR increases the trace data and overflows for 521.88% and 11249.93%, respectively, proving that the binaries compiled by ARMOR bring much more workload to ETM than the original binaries. In contrast, though OLLVM increases the trace data, it fails in triggering overflows. The reason is that OLLVM introduces heavy runtime overhead (as shown in our evaluation on SPEC2006), which slows down the bandwidth of ETM in generating trace data. Therefore, compared to ARMOR, existing code obfuscation techniques may be inefficient in leading to the trace buffer overflow and achieving anti-hardware tracing.

Benefiting from the numerous overflows, ARMOR effectively reduces the leaked runtime information. Specifically, on average, ETM captures 171, 140 valid basic blocks and 27, 326 addresses of indirect branches on the ARMOR-instrumenting binaries, while 376, 192 blocks and 55, 667 addresses on the original binaries. ARMOR reduces the number of traced blocks and addresses by 54.51% and 50.91%, proving its effectiveness in hiding the runtime information. However, on the programs compiled by OLLVM, ETM captures all the basic blocks and addresses. Although OLLVM presents challenges to the analyzer in understanding program semantics, it is not designed for anti-hardware tracing and fails to conceal the runtime information under ETM. Attackers can still reconstruct the control flow completely.

Moreover, we deeply analyse the successful rate of triggering overflows in the loop function. On average, the tested programs call the loop function for 44, 139 times, of which only 8.84% calls (about 3, 904) do not incur any overflows. 91.16% of the loop function executions is under overflow or triggering overflow, proving the efficiency of our context-based mechanism and anti-ETM loop.

RQ2: ARMOR is effective in hiding the runtime information with reducing 54.51% basic blocks and 50.91% addresses of indirect branches on 16 applications.

C. Use Case: Resisting the Cryptographic Attack

The nailgun attack leverages data tracing of ETM to obtain the memory addresses of the ASE table entries on NXP i.MX53 Quick Start Board [30]. However, ARM Juno R2 does not support data tracing. Hence, we implement a cryptographic attack based on ETM to extract the private key of RSA in GnuPG 1.4.13 via the control-flow information. Then we employ ARMOR to resist this attack. Moreover, the

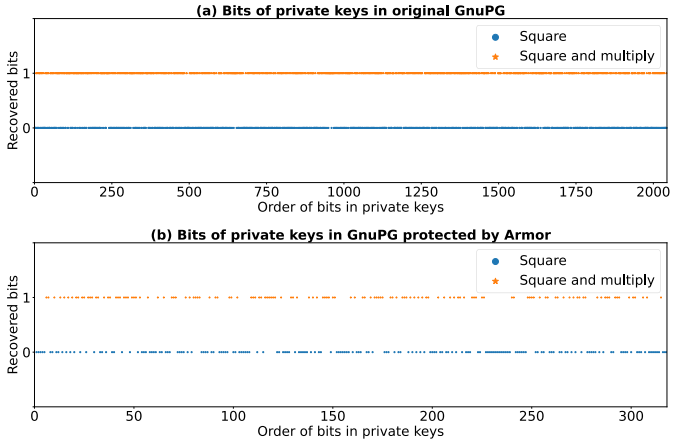

Fig. 6. The bits of private keys recovered in GnuPG without and with the protection of ARMOR, respectively.

signal SPNIDEN is enabled on the ARM Juno R2 board. Following the nailgun attack [30], we port GnuPG as a TA running in TrustZone and conduct the attack from the Normal world to leak the private key in the Secure world. Though previous works demonstrated similar attacks based on the side channel, such as FLUSH+RELOAD [45], we propose a new perspective from hardware tracing.

Our attack utilizes the defect of square-and-multiply exponentiation algorithm [64] implemented in GnuPG 1.4.13. Specifically, RSA [65] randomly selects two prime numbers p and q and a public exponent e and calculates a private key d. For optimizing the decryption function, GnuPG 1.4.13 employs CRT-RSA. It splits the private key d into d_p and d_q, which calculated as $d \bmod (p-1)$ and $d \bmod (q-1)$, respectively. Then it decrypts the message by $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$. To accelerate the process of integer exponentiations, GnuPG implements the square-and-multiply exponentiation algorithm [64] to scan each bit of the exponent and determine whether to conduct the multiply operation. In each bit, the algorithm performs the square operation (i.e., mpih_sqr_n). If the bit is 1, it will additionally perform the multiply operation (i.e., mpihelp_mul_karatsuba_case). Therefore, by tracing the entry addresses of these functions, we can deduce each bit of the exponent (i.e., d_p and d_q in CRT-RSA). As pointed out in [45], it is sufficient to attack the CRT-RSA with d_p and d_q.

1) Attack: Since the ARM Juno R2 has supported the OP-TEE as the TEE OS, we follow the guidance from OP-TEE to port the GnuPG 1.4.13 as a TA running on it [66]. Then, we use the rsa_decreypt function in GnuPG to decrypt a pre-encrypted string with a 2,048-bit key. We suppose that attackers have the offsets of the entry points of the square and multiply functions. As the program directly calls these functions, we enable the branch broadcasting mode of ETM to trace these addresses. Moreover, we run the TA on a Cortex-A53 core to avoid trace buffer overflow. After one trial, we decode the trace data to analyse the addresses of square and multiply functions to recover the 1,024-bit keys d_p and d_q.

Fig. 6 (a) shows the bits of private keys recovered from the trace data on the original program. During this attack, ETM generates 3,037 Address elements of the square and multiply functions, with 2,044 square and 993 multiply operations, while avoids to trigger any overflows. Then we try to recover the keys from the sequence by converting the square operation to 0 and the square-multiply operation to 1. Finally, we get the 2,044 bits of the private keys, which contains the 1,023 bits of d_p and 1,021 bits of d_q. Since the CRT-RSA filters the zero-bits to the first non-zero bits in the exponent, we recover all the bits in d_p and d_q based on this and compare them with the ground truth. The result shows that our attack can be carried out from the Normal world and precisely extract the private keys in a secure software across processes in one run, bringing significant threats to users. We also conduct this attack on the powerful Cortex-A72 core with 1.2GHz to evaluate its efficiency under the maximum frequency. During the 28,888 us running of the rsa_decreypt function, ETM generates 2,022 and 993 Address elements of the square and multiply functions, respectively, with triggering 2,755 overflows. Attackers can also effectively recover 98.7% (2,022) bits of the keys even under the maximum CPU frequency.

2) Protection: Since the attack is based on capturing the entry addresses of the square and multiply functions, we add those instructions calling these functions as instrumented points in addition to the default instrumentation positions, recompile the TA by ARMOR, and reconduct the attack. Due to the detection environment mechanism in ARMOR, the program is only allowed to be executed on the Cortex-A72 core with the 1.2GHz. Fig. 6 (b) shows result of the protected TA. Under our protection, ETM triggers 115,506 overflows and only traces 532 Address elements of these two functions, with 318 square and 214 multiply functions. And the rsa_decreypt function runs for 39,511us, where ARMOR introduces 36.77% runtime overhead compared to the attacks with the same configuration. Then we can only recover 318 bits (about 15.5%) from the sequence, which poses a significant challenge for attackers to recover the keys from these bits. The results show that ARMOR can effectively resist this attack and protecting some crucial information.

RQ3: Attackers can utilize ETM to obtain the entire 2048-bit key of a secure software in one run. ARMOR can protect 84.5% bits of keys with 36.77% runtime overhead.

D. Use Case: Anti-Fuzzing

1) Experimental Setup: To evaluate ARMOR in impeding the hardware-assisted fuzzing. we port ARMored-CoreSight (Edge Cov.) [8], a fuzzer based on ARM CoreSight, in AFL on our platform as AFL-CoreSight. Due to the limitation of computational resources, we only test the first two programs in Table V. We compile them with gcc and ARMOR, test them by AFL-CoreSight for 24 hours with the initial seed provided by AFL in one Cortex-A72 1.2 GHz core, and repeat five times to reduce the randomness introduced by fuzzing. We follow the configuration of ETM in Section V-B.

2) Metrics: We do not consider the crashes, as it is difficult for the fuzzer to trigger them under limited computational resources. Instead, we utilize the throughput, path coverage, and branch coverage as metrics and rerun all the seeds by AFL to uniformly measure the coverage. Moreover, to analyse the workloads brought by ARMOR to ETM, in addition to the size of trace data, we record the time of executing a testcase and rebuilding the coverage, respectively.

TABLE VI
THE DETAILED RESULTS OF AFL-CORESIGHT ON ORIGINAL BINARY AND ARMOR-INSTRUMENTING BINARY

Prog	Path Coverage		Branch Coverage		Execution(M)		Trace Data(KB)		Execution Time(us)		Rebuilding Time(us)	
	Origin	ARMOR	Origin	ARMOR	Origin	ARMOR	Origin	ARMOR	Origin	ARMOR	Origin	ARMOR
objdump	1,636	276	7,784	3,234	19.06	2.34	13.5	402.0	944	1,262	2,096	34,078
readelf	2,848	896	6,899	4,402	37.88	3.52	7.1	271.2	679	900	826	22,280
Mean	2,242	586	7,341	3,818	28.47	2.93	10.3	336.6	811	1,081	1,461	28,179
%-Chg	-	-73.86%	-	-47.99%	-	-89.71%	-	+3163.29%	-	+33.29%	-	+1828.75%

3) Result: Table VI lists the arithmetic mean of the metrics during five trials. Averagely, AFL-CoreSight only covers 586 paths and 3,818 branches under ARMOR, with 73.86% and 47.99% decreasement of the native programs. This is mainly due to the significant decrease in throughput.

Compared to the original binaries, though ARMOR introduces some runtime overhead with increasing the average execution time from 881us to 1,081us, the decrease in throughput is mainly due to the heavy overhead in rebuilding coverage. Specifically, ARMOR increases 3163.29% of the trace data of the original programs. As a result, AFL-CoreSight spends 1828.75% more time on rebuilding coverage. The throughput of AFL-CoreSight on the protected binaries is only 2.93M, 89.71% less than on the original binaries.

RQ4: By generating 32.6× trace data, ARMOR brings heavy workloads to the hardware-assisted fuzzer in rebuilding coverage, significantly impeding AFL-CoreSight by reducing 89.71% throughput and 47.99% branches.

VI. DISCUSSION AND FUTURE WORK

A. Possible Ways of Anti-Hardware Tracing

Detecting and disabling hardware tracing can be challenging due to its transparency, especially for user-privilege programs. This paper proposes a novel approach to bypass the transparency and reach anti-hardware tracing by exploiting the bandwidth issue inherent in hardware tracing instead of explicitly detecting its presence. It is indeed a legitimate concern to explore alternative methods apart from ARMOR to counter hardware tracing. There are some straightforward countermeasures which can be provided by TEE OS or the isolated environment, such as disabling the ETM or protecting the trace memory. However, in Section II-C, we have pointed that these methods can be easily circumvented by attackers and may negatively impact the normal usage of the ETM. Notice that, though the buffer overflows brought by ARMOR may impact the work of ETM, it is only occurring when the protected software is traced by ETM, which we can regard as malicious behavior. That means, the users who want to analyse programs by hardware tracing will not be effected by ARMOR if they do not maliciously trace the protected software. A possible solution is utilizing self-modifying code (SMC), which can modify the instructions at runtime to obfuscate the codes. Nevertheless, only utilizing the SMC may not lead to as significant trace data loss as ARMOR. The attackers can still obtain the runtime information. Moreover, SMC should be employed carefully due to the potential for false positives and bugs introduced by runtime modifications [67]. In addition, many compiler-level techniques, such as code obfuscation or time obfuscation, have been proposed to reinforce the programs against reverse-engineering or side channel attacks [1], [68], [69], [70]. However, they are not designed initially for anti-hardware tracing. ETM can still trace these programs and obtain the control-flow information.

B. Overhead of ARMOR.

A potential concern may be the noticeable performance overhead of ARMOR observed during the evaluation on SPEC2006. However, this overhead primarily stems from executing loop functions, which can vary significantly among different programs. It is worth noting that ARMOR exhibited low overhead on some programs. Additionally, since ARMOR allows users to configure instrumentation points, we can mitigate the performance impact by reducing the amount of instrumentation. It should be noted that this approach may potentially weaken ARMOR's ability to trigger overflow. Some overhead is necessary and unavoidable due to the requirement of executing many branches to generate trace data and trigger overflow quickly.

C. Cryptographic Attack

In Section V-C, we assumed that the adversaries only have the offsets of the certain function entries. Even when using ARMOR for protection, partial control flows may still be leaked, potentially allowing attackers to recover additional bits in the private keys by reverse-engineer. However, this process is labor-intensive, providing some degree of slowdown for attackers. It is worth noting that skilled attackers may attempt to circumvent ARMOR by configuring the ETM only to trace the addresses of the square and multiply functions. Fortunately, as explained in Section V-B1, the PIE+STRIP+ASLR strategy can hinder these attacks, as the addresses are determined dynamically during execution and cannot be readily obtained.

D. Limitation

There are still some limitations in the current implementation of ARMOR: *1) Not* 100% *protection.* While we have implemented a context-based calculation mechanism for triggering the overflow before executing some instructions, our evaluation indicate that ARMOR cannot consistently guarantee that the ETF is under overflow during the execution of these instructions. Actually, the amount of data generated by the ETM is influenced by the configuration, executed instructions, and the runtime environment. Our method is limited to estimating the amount of output data rather than accurately calculating it. This means that ARMOR hides specific information (e.g., control flow at the implicit-semantics points) with a high, but not 100%, probability.

2) The high-speed execution of programs. A crucial principle of anti-hardware tracing by triggering trace buffer

overflows is keeping the high-speed execution of the program. Though we utilize a detecting environment mechanism to guarantee this principle, adversaries may bypass this mechanism and slow down the execution to avoid overflows. For example, turning down the CPU frequency after passing the detection [53], [71]. A possible way to reinforce the protection is combining ARMOR with other anti-debugging techniques or inserting more lightweight detection to guarantee high-speed execution, which remains part of our future work.

3) The anti-ETM loops traced by ETM. The numerous trace data is generated when ETM traces the anti-ETM loops instrumented by ARMOR. Adversaries may bypass this by excluding the addresses of anti-ETM loops from the traced address range. However, precise address configuration requires attackers to obtain the memory layout of secure code in advance, which is impeded by our PIE+STRIP+ASLR strategy. During a single run, attackers may fetch and analyse part of the trace data on the fly to speculate the addresses of anti-ETM loops and then reconfigure and restart ETM to resume tracing. However, this process is uneasy and time-consuming. Attackers may belatedly obtain the addresses after the execution of some instructions in the software, where partial secret information is still undisclosed. Another possible way to exclude these addresses is disabling the ETM during the anti-ETM loops and enabling it after the loops. However, it may be challenging to conduct this attack even for timing-aware attackers. Moreover, they need to repeatedly perform the program to determine the window period of the loops. However, the context-based calculation mechanism in ARMOR calculates the times of anti-ETM loops at runtime. This implies that the timing of entering and exiting the loop is non-deterministic for each run (like some time obfuscation methods [68], [69]). It may be hard for attackers to capture the window period precisely. Inserting some lightweight random code to improve ARMOR in resisting the time-aware attackers is also feasible.

Particularly, the offensive and defensive sides are constantly engaged in a game of strategy and evolution. *Despite the possibility of* ARMOR *being bypassed, we believe that implementing anti-hardware tracing techniques from the perspective of trace buffer overflow can provide valuable insights into security research.* Moreover, ARMOR is a software-level solution for thwarting ETM-based attacks without modifying the hardware. This advantage makes ARMOR facile to deploy on real-world devices, particularly these SPNIDEN-enabled devices. It also makes ARMOR easily combinable with specific techniques, such as time obfuscation [68], [69], to reinforce the protection.

E. Interrupt-Capable Attacker

In our threat model, we assume the attacker cannot interrupt the protected software running in the Secure world or isolated environments. This assumption can be achieved by carefully managing the priorities of the interrupts or masking specific interrupts [28], [56], [57], [58]. However, if the attackers can raise the non-secure interrupts to stall the protected software, they may bring some troubles to ARMOR. Specifically, to break the high-speed principle in ARMOR and avoid the trace buffer overflow, they can frequently interrupt and slow down the software after passing our detection environment mechanism [71]. To exclude the anti-ETM loops from the traced addresses, interrupt-capable attackers only need to interrupt and jump out of the protected process once after passing our detection. Then, they can analyse the junk addresses, reconfigure the ETM, and resume the process. Moreover, the attackers even can conduct interrupt-based side channel attacks (e.g., load-step [53]) rather than utilizing ETM to steal the secret data.

Fortunately, the interrupt-based attacks [53], [55], [72], [73] on TEE have received significant attention from researchers, resulting in the development of many solutions [28], [74], [75], [76], [77]. ARM TrustZone also supports blocking the non-secure interrupts while the TA is running [56], [57], which effectively limits the capability of the interrupt-based attacks on the ARM platform [76]. Notably, different ARM architectures and devices may have different methods to mask the non-secure interrupts. On TF-M, we can set the AIRCR.PRIS during TF-M core initialization and the PRIMASK_NS in the entry of TA to achieve this [57]. On TF-A (e.g., ARMv8-A), we can use the bits I and F in PSTATE to mask the Interrupt Request (IRQ) and Fast Interrupt Request (FIQ), respectively [48], [58]. For instance, on ARM Generic Interrupt Controller (GIC) v2 mode, the non-secure and secure interrupts are mapped as the IRQ and FIQ, respectively. By setting the I bit as 1 and F as 0 when entering the Secure world, we can mask the non-secure interrupts during the execution in the Secure world [58]. In conclusion, the key idea is to conduct specific configurations in the Secure world to block foreign interrupts and prevent kernel-privilege attackers from modifying the configurations. This may require support from the secure monitor in EL3 or TEE OS in EL1. Since blocking the malicious interrupts has become an essential principle in designing and implementing the TEE firmware and TEE OS [28], [57], [74], [75], [76], it is feasible and reasonable to assume that malicious interrupts can be blocked on the ARM platform.

F. ARMOR *on Other Platforms*

Similar to ARM, Intel also proposed the tracing technique Intel PT, which is popular and employed in many researches [5], [10], [17], [24], [25]. To the best of our knowledge, trace buffer overflow also exists on Intel PT [2], [24]. Therefore, it may be possible to port ARMOR on Intel PT according to our models and principles of anti-hardware tracing. However, Though most of the features in Intel PT are similar to those in ARM CoreSight, there is some difference between them. For example, our cryptographic attack in Section V-C relies on the addresses of specific functions that are directly called in native binaries. Intel PT may be unable to capture these addresses as it only records the addresses of indirect branches, while ARM CoreSight can achieve this as the branch broadcasting mode of ETM. Moreover, ETM can non-invasively trace the processes in the Secure world on some SPNIDEN-enabled devices [30]. *In contrast, to our knowledge, Intel PT may be only able to trace the debug enclaves in Intel SGX* [26], [78]. Due to the more powerful tracing ability than Intel PT, ARM CoreSight may bring more serious security threats, which is one of the reasons that we conducted ARMOR on ARM platforms.

VII. Related Work

A. Hardware Tracing

Benefiting from the low overhead and powerful runtime information tracing capabilities, hardware tracing techniques have been widely used in various areas [5], [6], [7], [8], [9], [10], [11], [12], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. Remarkably, some works utilize hardware tracing for conducting attacks, such as extracting private keys of the AES algorithm in TA [30] or detecting cryptographic algorithms [31], [32]. However, most works focus on leveraging hardware tracing rather than resisting it. While some defense methods for nailgun are proposed to manage debug authentication signals and prevent tracing the TA [30], they do not specifically address the challenge of anti-hardware tracing. To the best of our knowledge, ARMOR is the first tool specifically designed for anti-hardware tracing.

B. Software Protection

Software protection techniques have been developed over the years, such as anti-debugging [36], [37], [38], code obfuscation [1], [39], [40], [41], [42], [43], and anti-fuzzing [79], [80]. Nevertheless, to our knowledge, these works lack specific focus on anti-hardware tracing.

Anti-debugging techniques mainly focus on detecting or preventing external debuggers, such as timing checks, process checks, and self-debugging [14], [36], [37]. As stated in Section I, they may not be effective in preventing attackers who employ transparent tracing techniques [11], [14].

Code obfuscation is a popular technique used to protect software against malicious modifications or reverse-engineering [1], [39], [40], [41], [42], [43], [44]. Some common tricks in code obfuscation, such as inserting irrelevant code [1], [44], is similar to those utilized in ARMOR. Our evaluation shows that though OLLVM increases the branches and causes ETM to generate more trace data, it is not as efficient as ARMOR in triggering the overflow and concealing runtime information.

C. Attacks and Defenses on TEE

Due to the rapid development and popularizing of TEE techniques, researchers have paid many efforts on breaking the confidentiality of TEE or reinforcing the TEE [28], [51], [52], [53], [54], [55], [72], [73], [74], [75], [76], [77]. However, most of these attacks or defenses around TEE focus on the side channel attacks, such as cache or controlled side channel [28], [51], [52], [53], [54], [72], [73], [75], while the security threats brought by hardware tracing techniques to TEE are ignored for a long time [30]. In fact, as demonstrated in [30] and our evaluation, hardware tracing can conduct powerful and high-resolution attacks due to its powerful tracing ability. We notice this threat and propose the anti-hardware tracing technique to provide our insights to the community.

D. Alleviating Buffer Overflow

Various techniques have been proposed to address the trace buffer overflow issues. Hart [71] avoids the overflow in CoreSight by configuring the Performance Monitor Unit (PMU) to interrupt the process before the overflow occurs. However, this approach requires frequent interrupts and leads to a significant slowdown in the program. As mentioned in Section VI, this method may be resisted by managing the interrupts carefully. JPortal [24] introduces algorithms to precisely recover the control flow from PT traces even under some trace data loss. However, the effectiveness of the recovery algorithm diminishes as more data is lost. With the numerous overflows and trace data loss caused by ARMOR, attackers may struggle to fully recover the control flow, even with the algorithms used in JPortal. The hardware vendors also invest efforts to alleviate the buffer overflow. The ETMv4 specification [49] suggests an optional feature in the TRCSTALLCTLR register to prevent overflows by stalling the program when the trace buffer reaches a certain threshold. However, this approach can introduce significant runtime overhead, which can be detected and prevented by the detection mechanism in ARMOR. Additionally, our investigation indicates that this feature is not implemented by some platforms, such as ARM Juno R2.

VIII. Conclusion

In this paper, we introduce anti-hardware tracing as a novel protection technique against hardware tracing. Unlike existing methods, anti-hardware tracing leverages hardware limitations to trigger overflow and safeguard the runtime information. We build a model to analyse the trace buffer overflow and point out three key principles for efficient anti-hardware tracing. Then we design a framework ARMOR on ARM Juno R2. ARMOR instruments the protection and loop functions to support these principles and conduct some strategies such as PID replacement and PIE+STRIP+ASLR. Through comprehensive evaluations, we demonstrate that ARMOR effectively safeguards control flow from ETM and successfully resists cryptographic attacks like the one conducted on GnuPG 1.4.13 by us. Additionally, ARMOR significantly hampers hardware-assisted fuzzing.

Acknowledgment

This work was done while Tai Yue was a visiting student in the COMPASS Lab of Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China.

References

[1] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM—Software protection for the masses," in *Proc. IEEE/ACM 1st Int. Workshop Softw. Protection*, May 2015, pp. 3–9.

[2] I. Corporation. (2023). *Intel 64 and IA-32 Architectures Software Developer Manuals*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[3] ARM. (2016). *ARM Coresight SOC-400 Technical Reference Manual*. [Online]. Available: https://developer.arm.com/documentation/100536/latest/

[4] S. D. Sharma and M. Dagenais, "Hardware-assisted instruction profiling and latency detection," *J. Eng.*, vol. 2016, no. 10, pp. 367–376, Oct. 2016.

[5] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 167–182.

[6] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.

[7] Y. Chen et al., "PTrix: Efficient hardware-assisted fuzzing for COTS binary," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Jul. 2019, pp. 633–645.

[8] Y. S. A. Moroo. (2021). *Armored Coresight: Towards Efficient Binary-Only Fuzzing*. [Online]. Available: https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html

[9] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proc. 44th Int. Conf. Softw. Eng.*, May 2022, pp. 1–12.

[10] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 2597–2614.

[11] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on arm," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 33–49.

[12] L. Xue et al., "Happer: Unpacking Android apps via a hardware-assisted approach," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1641–1658.

[13] D. Tian, Q. Ying, X. Jia, R. Ma, C. Hu, and W. Liu, "MDCHD: A novel malware detection method in cloud using hardware trace and deep learning," *Comput. Netw.*, vol. 198, Oct. 2021, Art. no. 108394.

[14] G. Li et al., "POSTER: PT-DBG: Bypass anti-debugging with Intel processor tracing," in *Proc. 39th IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, 2018, pp. 21–23.

[15] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient CFI enforcement with Intel processor trace," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 529–540.

[16] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 173–184.

[17] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding control flows using Intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, May 2017.

[18] M. Kadar, G. Fohler, D. Kuzhiyelil, and P. Gorski, "Safety-aware integration of hardware-assisted program tracing in mixed-criticality systems for security monitoring," in *Proc. IEEE 27th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, May 2021, pp. 292–305.

[19] D. Kwon, J. Seo, S. Baek, G. Kim, S. Ahn, and Y. Paek, "VM-CFI: Control-flow integrity for virtual machine kernel using Intel PT," in *Proc. Int. Conf. Comput. Sci. Its Appl.* Cham, Switzerland: Springer, 2018, pp. 127–137.

[20] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proc. USENIX Conf. Security*, 2017, pp. 131–148.

[21] Y. David et al., "Upgradvisor: Early adopting dependency updates using hybrid program analysis and hardware tracing," in *Proc. 16th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2022, pp. 751–767.

[22] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "ARCUS: Symbolic root cause analysis of exploits in production systems," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 1989–2006.

[23] W. Cui et al., "REPT: Reverse debugging of failures in deployed software," in *Proc. 13th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2018, pp. 17–32.

[24] Z. Zuo et al., "JPortal: Precise and efficient control-flow tracing for JVM programs with Intel processor trace," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, Jun. 2021, pp. 1080–1094.

[25] Q. Ying, Y. Yu, D. Tian, X. Jia, R. Ma, and C. Hu, "CJSpector: A novel cryptojacking detection method using hardware trace and deep learning," *J. Grid Comput.*, vol. 20, no. 3, p. 31, Sep. 2022.

[26] Intel. (2023). *Intel Software Guard Extensions*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html

[27] ARM. (2009). *ARM Security Technology Building a Secure System Using TrustZone Technology*. [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/latest/

[28] Y. Zhu, P. Li, L. Zhao, D. Meng, and R. Hou, "ChaosINTC: A secure interrupt management mechanism against interrupt-based attacks on TEE," in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2023, pp. 1–6.

[29] Y. Zhang et al., "SHELTER: Extending arm CCA with isolation in user space," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 1–18.

[30] Z. Ning and F. Zhang, "Understanding the security of ARM debugging features," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 602–619.

[31] J. Park and Y. Park, "Symmetric-key cryptographic routine detection in anti-reverse engineered binaries using hardware tracing," *Electronics*, vol. 9, no. 6, p. 957, Jun. 2020.

[32] W. Yang and Y. Park, "Identifying symmetric-key algorithms using CNN in Intel processor trace," *Electronics*, vol. 10, no. 20, p. 2491, Oct. 2021.

[33] H. Shan et al., "CROWBAR: Natively fuzzing trusted applications using ARM CoreSight," *J. Hardw. Syst. Secur.*, vol. 7, nos. 2–3, pp. 44–54, Sep. 2023.

[34] Q. Wang et al., "SyzTrust: State-aware fuzzing on trusted OS designed for IoT devices," 2023, *arXiv:2309.14742*.

[35] H. Shan et al., "LightEMU: Hardware assisted fuzzing of trusted applications," 2023, *arXiv:2311.09532*.

[36] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Secur. Privacy Mag.*, vol. 5, no. 3, pp. 82–84, May 2007.

[37] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies," *Black Hat*, vol. 1, no. 2012, pp. 1–27, 2012.

[38] P. Chen, C. Huygens, L. Desmet, and W. Joosen, "Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware," in *Proc. IFIP Int. Conf. ICT Syst. Secur. Privacy Protection*. Cham, Switzerland: Springer, 2016, pp. 323–336.

[39] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, Oct. 2003, pp. 290–299.

[40] C. K. Behera and D. L. Bhaskari, "Different obfuscation techniques for code protection," *Proc. Comput. Sci.*, vol. 70, pp. 757–763, Jan. 2015.

[41] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. NDSS*, 2008, pp. 1–13.

[42] M. Schloegel et al., "Loki: Hardening code obfuscation against automated attacks," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*, 2022, pp. 3055–3073.

[43] M. F. Oberhumer. (2004). *Upx the Ultimate Packer for Executables*. [Online]. Available: http://upx.sourceforge.net/

[44] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Comput. Surv.*, vol. 49, no. 1, pp. 1–37, Mar. 2017.

[45] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.)*, 2014, pp. 719–732.

[46] ARM. (2016). *Juno R2 ARM Development Platform Soc*. [Online]. Available: https://developer.arm.com/documentation/100114/0200

[47] ARM. (2011). *Coresight Trace Memory Controller Technical Reference Manual*. [Online]. Available: https://developer.arm.com/documentation/ddi0461/b/?lang=en

[48] ARM. (2023). *Arm Architecture Reference Manual for A-Profile Architecture*. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ja/?lang=en

[49] ARM. (2021). *Embedded Trace Macrocell Architecture Specification ETMV4.0 to ETM4.6*. [Online]. Available: https://developer.arm.com/documentation/ihi0064/h/?lang=en

[50] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An off-chip attack on hardware enclaves via the memory bus," in *Proc. 29th USENIX Conf. Security Symp.*, Aug. 2020, pp. 487–504.

[51] S. K. Bukasa, R. Lashermes, H. L. Bouder, J.-L. Lanet, and A. Legay, "How TrustZone could be bypassed: Side-channel attacks on a modern system-on-chip," in *Proc. IFIP Int. Conf. Inf. Secur. Theory Pract.*, Heraklion, Greece, Sep. 2017, pp. 93–109.

[52] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "TruSense: Information leakage from TrustZone," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 1097–1105.

[53] Z. Kou, W. He, S. Sinha, and W. Zhang, "Load-step: A precise TrustZone execution control framework for exploring new side-channel attacks like Flush+Evict," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 979–984.

[54] Z. Kou, S. Sinha, W. He, and W. Zhang, "Cache side-channel attacks and defenses of the sliding window algorithm in TEEs," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Apr. 2023, pp. 1–6.

[55] Z. Kou, S. Sinha, W. He, and W. Zhang, "Attack directories on ARM big.LITTLE processors," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Oct. 2022, pp. 1–9.

[56] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–36, 2019.

[57] T. Firmware-M. (2020). *Trusted Firmware-M Documentation*. [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/v1.7-branch/tfm/docs/technical_references/tfm_non-secure_interrupt_handling.html

[58] T. Firmware-A. (2020). *Trusted Firmware—A Documentation*. [Online]. Available: https://trustedfirmware-a.readthedocs.io/en/latest/design/interrupt-framework-design.html

[59] C. Hwang. *PTM2Human*. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/hwangcc23/ptm2human

[60] TrustedFirmware.org. (2023). *About OP-TEE–OP-TEE Documentation Documentation*. [Online]. Available: https://optee.readthedocs.io/en/latest/general/about.html

[61] T. Yue et al., "EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proc. 29th USENIX Conf. Secur. Symp.*, 2020, pp. 2307–2324.

[62] C. Lyu et al., "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. USENIX Secur. Symp.*, 2019, pp. 1949–1966.

[63] M. Zalewski. (2023). *American Fuzzy Lop*. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[64] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, pp. 129–146, Apr. 1998.

[65] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[66] O.-T. Documentation. (2018). *Trusted Applications–OP-TEE Documentation Documentation*. [Online]. Available: https://optee.readthedocs.io/en/latest/building/trusted_applications.html

[67] Z. Zhou, C. Wang, and Q. Zhao, "No-fuzz: Efficient anti-fuzzing techniques," in *Security and Privacy in Communication Networks*, F. Li, K. Liang, Z. Lin, and S. K. Katsikas, Eds. Cham, Switzerland: Springer, 2023, pp. 731–751.

[68] A. Fell, H. T. Pham, and S.-K. Lam, "TAD: Time side-channel attack defense of obfuscated source code," in *Proc. 24th Asia South Pacific Design Autom. Conf.*, Jan. 2019, pp. 58–63.

[69] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 8–11.

[70] R. M. Tsoupidi, E. Troubitsyna, and P. Papadimitratos, "Thwarting code-reuse and side-channel attacks in embedded systems," 2023, *arXiv:2304.13458*.

[71] Y. Du et al., "HART: Hardware-assisted kernel module tracing on arm," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2020, pp. 316–337.

[72] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-step: A practical attack framework for precise enclave execution control," in *Proc. 2nd Workshop Syst. Softw. Trusted Execution*, Oct. 2017, pp. 1–6.

[73] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 178–195.

[74] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "SecTEE: A software-based approach to secure enclave architecture using tee," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1723–1740.

[75] S. Constable et al., "AEX-notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 4051–4068.

[76] R. Bahmani et al., "CURE: A security architecture with customizable and resilient enclaves," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 1073–1090.

[77] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[78] (2018). *Intel Processor Trace and Intel SGX*. [Online]. Available: https://community.intel.com/t5/Intel-Software-Guard-Extensions/Intel-Processor-Trace-and-Intel-SGX/m-p/1181483#M3538

[79] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 1913–1930.

[80] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "Antifuzz: Impeding fuzzing audits of binary executables," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 1931–1947.

Tai Yue received the B.S. degree from the Department of Mathematics, Nanjing University, Nanjing, in 2017, and the M.S. degree from the College of Computer, National University of Defense Technology, Changsha, in 2019, where he is currently pursuing the Ph.D. degree. His research interests include system security, software security, and hardware-assisted security.

Fengwei Zhang (Senior Member, IEEE) received the Ph.D. degree in computer science from George Mason University. He is currently an Associate Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). His research interests include systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, transportation security, and plausible deniability encryption.

Zhenyu Ning received the Ph.D. degree in computer science from Wayne State University in 2020. He is currently an Associate Professor with Hunan University. His research interests include security and privacy, including system security, mobile security, the IoT security, trusted execution environment, and hardware-assisted security.

Pengfei Wang received the B.S., M.S., and Ph.D. degrees in computer science and technology from the College of Computer, National University of Defense Technology, Changsha, in 2011, 2013, and 2018, respectively. He is currently an Associate Professor with the College of Computer, National University of Defense Technology. His research interests include operating systems and software testing.

Xu Zhou received the B.S., M.S., and Ph.D. degrees from the School of Computer Science, National University of Defense Technology, China, in 2007, 2009, and 2013, respectively. He is currently an Associate Professor with the School of Computer Science, National University of Defense Technology. His research interests include operating systems and security.

Kai Lu received the B.S. and Ph.D. degrees in computer science and technology from the College of Computer, National University of Defense Technology, Changsha, in 1995 and 1999, respectively. He is currently a Professor with the College of Computer, National University of Defense Technology. His research interests include operating systems, parallel computing, and security.

Lei Zhou received the Ph.D. degree in computer science from Central South University. He is currently a Research Associate with the College of Computer, National University of Defense Technology. His research interests include systems security, including trustworthy execution, hardware-assisted security, and memory forensics.