Towards Efficient Control-Flow Attestation with Software-Assisted Multi-level Execution Tracing

Dimitrios Papamartzivanos*, Sofia Anna Menesidou, Panagiotis Gouvas, Thanassis Giannetsos
Ubitech Ltd., Digital Security & Trusted Computing Group, Greece
Email: {dpapamartz, smenesidou, pgouvas, agiannetsos}@ubitech.eu

***Abstract*—In the face of an increasing attack landscape, it is necessary to cater for efficient mechanisms to verify software and device integrity for detecting run-time modifications in next-generation systems-of-systems. In this context, remote attestation is a promising defense mechanism that allows a third party, the verifier, to ensure a remote device's configuration integrity and behavioural execution correctness. However, most of the existing families of attestation solutions suffer from the lack of software-based mechanisms for the efficient extraction of rigid control-flow information. This limits their applicability to only those cyber-physical systems equipped with additional hardware support. This paper proposes a multi-level execution tracing framework capitalizing on recent software features, namely the extended Berkeley Packet Filter and Intel Processor Trace technologies, that can efficiently capture the entire platform configuration and control-flow stacks, thus, enabling wide attestation coverage capabilities that can be applied on both resource-constrained devices and cloud services. Our goal is to enhance run-time software integrity and trustworthiness with a scalable tracing solution eliminating the need for federated infrastructure trust.**

***Index Terms*—Control-flow Integrity, Remote Attestation, Intel Processor Trace , extended Berkeley Packet Filter, Tracing**

I. Introduction and Background

Attacks that try to compromise the configuration integrity of remote systems or subvert the control-flow of legitimate computer programs, are a large concern towards the vision of trustworthy "Systems-of-Systems" (SoS). The main challenge, here, is how to securely detect such attacks with minimal overhead on the target host platform. Towards this direction, program tracing mechanisms have been proposed to record run-time information and system audits about a program's execution and enable flexible and powerful offline analysis. However, most of such software analysis solutions provide an offline approach that first gathers system audit data, during the execution of monitored functionalities, and then decode this data to infer possible violations or other issues.

Unfortunately, this approach does not capture the real-time constraints of emerging attestation security enablers [1] that require detailed dynamic tracing of properties stemming from diverse levels of a system's architecture: kernel shared libraries, low-level code, etc. resulting in an in-depth investigation of the systems behaviour and execution flow towards detecting any cheating attempts or if any type of (non-previously identified) exploits are resident to the memory.

This work was supported by the European Commission, under the ASSURED and ASTRID Horizon 2020 projects with Grant Agreement No. 952697 and 786922, respectively.

To date, several remote attestation techniques have been proposed to verify the integrity of a software or the control-flow of devices. However, most of them are static and verify only the software integrity of devices and only recently some run-time Control-flow Attestation (CFA) and Control-flow Integrity (CFI) schemes, leveraging control-flow information of a program, have been proposed [2]. With CFA, sophisticated attacks that tamper with state information in the program's data memory (e.g., the stack and the heap) can be detected. CFA mechanisms can be also deployed as part of holistic runtime risk assessment frameworks and contribute towards a more concrete cyber risk quantification [3]. One of the first dynamic CFA solutions was presented in C-FLAT [4] where the focus was on measuring the valid execution paths undertaken by embedded devices. However, this approach requires instrumentation of all control-flow instructions, thus, incurring significant performance overhead on the underlying software tracing [1].

In general, while CFA provides strong security guarantees and verifiable evidence on the correctness of a device's configuration and execution, enforcing control-flow integrity in a practical way is challenging due to the inefficiency of existing tracing solutions. Despite sustained interest in CFA, current approaches try to overcome these limitations by either targeting (custom) optimized generation of CFGs (LO-FAT [5], LiteHAX [6], SCAPI [7] and BLINDTRUST [8]) or leveraging hardware features ([9], CFIMon [10], μCFI [11], PITTYPAT [12], PT-CFI [13] and C-ITS [14]). That is, current mechanisms require the physical devices to be equipped with specialised hardware components to enable tracing. These components can be either isolated co-processors within the CPU of a system or completely standalone (out-of-band) hardware parts. In both cases, they act as interposers and have access to the CPU registers on per-cycle execution and through low-level execution probes can detect register values to track specific application execution.

In all of these approaches, however, there is always the compromise between performance, security and usability. This sets the challenge ahead: *How to provide near real-time low-level code inspection and tracing, thus, capturing the requirements of CFA while striking a balance between precision of control-flow information, efficiency and transparency?*

Contributions: This paper presents a novel hybrid tracer that enables the in-depth investigation and tracing of a system's configuration and operation while simultaneously meet-

ing these three requirements. Our approach is a software-assisted (or pseudo-hardware-based, due to the prerequisite existence of an Intel processor) solution based on a two-tier granularity tracing technique that combines two of the most prominent mechanisms, namely the extended Berkeley Packet Filter (eBPF) and Intel PT [11], [13], [15], [16]. The main idea is that the installed eBPF and IntelPT probes will be programmed to intercept internal operations towards producing a run-time control-flow path. Such probes can be used to capture the execution of specific software components in both physical and virtual devices so that we can check and attest the integrity of the execution behaviour based on already defined policies from embedded devices to cloud services. Overall, this new family of tracing mechanisms is *precise* since IntelPT has all program's control-flow traces; it is *efficient* since the combination of these tracing techniques provide different levels of details of logged information and are activated only when needed by the CFA, thus, enhancing the security and trustworthiness of the integrity verification steps; finally, it is *transparent* such that it requires no binary instrumentation, and can be readily deployable on commodity systems.

II. Control-Flow Attestation Based On Multi-level Tracing

This section describes the high-level conceptual architecture of the proposed tracing mechanism [1]. The core component, as can be seen in Figure 1, is the *Runtime Tracer*, which is deployed on the target devices we wish to attest by tracing specific properties. As is the case in typical remote attestation protocols, such as in [4], we assume the existence of a *Verification Engine* which is the typical Verifier actor responsible for assessing the attestation-related data and events registered by the Prover device. More specifically, the core responsibility of the Verification Engine is to verify the integrity, the authenticity and the freshness of the captured traces by leveraging cryptographic tools, such as digital signatures. Such traces that represent the current system state (of the target devie) and can be used for assessing it's overall security posture are collected by he *Runtime Tracer*; including the eBPF-based tracing and the binaries hashing for CIV, and Intel-PT-based tracing for low-level mission-critical processes. The outcome of the tracing process is in all cases the extraction of the corresponding CFGs (in the context of CFPA) and/or binary hashes (in the context of CIV).

The motivation behind this multi-level (i.e., eBPF and Intel-PT based) approach is that the employed attestation should not target the whole stack of a system's architecture, unless there is an indication of suspicious activities. More precisely, the scope of monitoring and tracing granularity should be increased only when additional evidence and information need to be collected, on a detected incident, for the assistance in finding the province of the attack as well as in the development of new enforceable attestation policies that should be able to catch this newly identified threat. A detailed activity diagram of the flow is depicted in Fig. 1. As aforementioned, this escalated tracing leverages two of the most prominent tracing

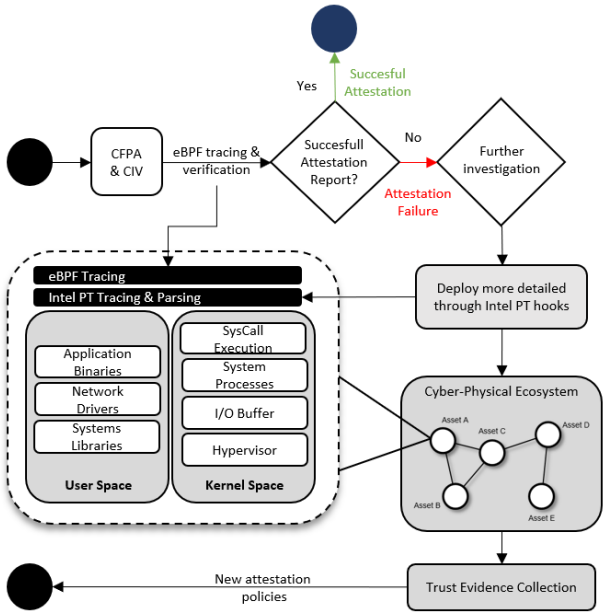

Fig. 1. Multi-level architecture activity diagram

mechanisms towards overcoming the pressing challenge of lightweight process monitoring:

- eBPF[1] execution hooks capable of providing near real-time low-level code inspection, thus, capturing the strict constraints of SoS-enabled ecosystems, and
- Intel PT[2] introspection agents capable of traversing the entire physical memory of a Cyber Physical System (CPS), via Direct Memory Access, for known execution signatures which can then be used to navigate to the relevant information to be traced. This approach allows fast tracing without affecting the normal system execution (non-intrusive).

Initially, the efficient and highly scalable eBPF technology is focused on the user space and used to monitor the configuration of a targeted binary or the sequence of executed software commands. This allows for the tracing of the configurational integrity of the target device without posing any significant overhead (Section III). These eBPF hooks are enhanced with more thorough tracing capabilities based on the use of the emerging Intel PT. eBPF technology is used to continuous monitor kernel shared libraries, system calls, shared data, memory address space etc., while Intel PT enables the in-depth investigation of the systems' behaviour for detecting possible exploits to the program and data memory. The utilization and granularity of this tracing process depends on the behavioural and attestation of the device; that is, if the initial attestation (based on the instantiation of the eBPF tracing) fails to meet some pre-specified requirements, then new policies can be enforced for automatically deploying the Intel PT tracer which will in turn gather more detailed data on how, why and which

[1]BPF Compiler Collection (BCC): https://github.com/iovisor/bcc#tools

[2]Intel PT: https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html

executable was behaving as potential point of intrusion. On the other hand, if no alarms are triggered, then the lightweight eBPF tracer will continue to monitor the device and send the gathered data through attestation reports as verifiable evidence of a device's trustworthiness. More specifically, if the attestation based on the eBPF traces fails, a more detailed investigation is performed through Intel-PT-based hooks for low-level mission-critical processes. The outcome of the tracing process is, in all cases, the extraction of the corresponding CFGs (in the context of CFA [1]) and/or binary hashes (in the context of Configuration Integrity Verification [17]).

III. Evaluation of tracing techniques

This section presents a detailed evaluation of the effectiveness on extracting a system's behavior and the performance of the aforementioned tracing technologies in order to uncover any adoption challenges and critically appraise the viability of such solutions in the CIV and CFA context for achieving systems' operational assurance.

A. On the Scientific Value of the Experimentation Setup

In what follows, we first start by documenting the assumptions made while designing the set of experiments to be performed. We focus the experimental evaluation on the computational complexity of the configuration or behavioural properties to be traced and analyze the timings of the individual tracing technologies without considering any possible constraints that might be posed by the specification of the underlying hardware (e.g., resources, memory structure, etc.). Trust is evaluated by (securely) measuring the state (and configuration behavior) of a device at any given point in time, and then comparing the measured state with reference (expected) states. This is captured through the following assumptions:

- **AS1 - Availability of Resources:** Depending on the complexity of the codebase to be traced, a fluctuating amount of allocated resources may be required on the underlined system. In fact, this is supported by the analysis of Section III. Thus, for demonstration purposes of this work, we assume that all the resources needed for the tracing are available, and the tracing is agnostic of the hardware specifications.
- **AS2 - Immutability:** The measurement process must be immutable, such that the tracer always returns the correct (actual) measurements. To enable secure tracing we assume that the tracer is part of the Trusted Computing Base (TCB) of the monitored system. It is out of the scope of this work to investigate on isolated processing environments. In fact, this is a common assumption in software-assisted CFA protocols.
- **AS3 - Functional Tracing Properties:** In order to deploy the necessary hooks and tracing configurations, the underlined system must be equipped with the respective enablers. For instance, in our case Intel-PT requires the existence of an Intel processor.

B. eBPF Tracer Evaluation

As aforementioned, the designed eBPF tracer aims to provide an efficient solution both in the context of CIV by tracing data of interest, such as the configuration of the binaries included in the whitelist to be loaded in a host, but also in the context of CFA for the tracing sequence of commands of targeted resources. In this section, we offer an evaluation in the context of the Trust Computing realm, by tuning the tracer to intercept and decode the sequence of Trusted Platform Module (TPM) commands exchanged in a virtualised TPM instantiation, where a dedicated TPM is triggered by TCG Software Stack (TSS)-enabled cloud-based applications. The designed tracer is built on top of BPF Compiler Collection (BCC) and the deployed hook intercepts the TCP/IP packets exchanged between the TPM and a TSS.

More specifically, we triggered various TPM commands to measure the actual overhead of eBPF tracing. When a TPM command is executed, the TSS takes on the task of marshalling the data. That is when the eBPF tracer starts its execution by intercepting the commands at the network layer, it handles the packets so as to extract the proper payload in order to unmask the traced events and acquire structured information of the TPM commands sequence.

In this case, the properties of interest are the actual timing of a function to be executed, and the time required by the tracer to monitor the executed function. By comparing these two timings we can define the tracing overhead. The timing results of basic TPM commands can be found in TABLE I. Stage 1 refers to TPM command preparation and data marshalling. Stage 2 is the time required from the data marshalling to the interception of the outbound write packet (from the TSS to the TPM). Stage 3 is the elapsed time right after we receive the response from the TPM until we decode the parsed and traced executed functions. With these facts in mind, we see that on average, Stage 1 takes 0,006865 secs, Stage 2 takes 0,236163 secs, and the Stage 3 takes 0,014625 secs. Hence, the overhead of the eBPF tracer is given by calculating the percentage of Stage 3 w.r.t the total time required to execute a TPM command, i.e., $Stage3/(Stage1 + Stage2) \times 100$. We conclude that the eBPF solution has an average overhead of 6.02% which proves the fact that eBPFs execution hooks are very efficient and lightweight. It is important to note that this overhead includes both the interception and the unmasking of the commands from the packet payload, advocating the efficient execution of our eBPF tracer.

C. Intel PT Tracer Evaluation

The evaluation of the Intel PT tracer is based on different test cases, written as simple C programs, to provide representative codebases for covering diverse challenging aspects. All test cases were compiled with GCC's optimization level set to 0. This ensures fine-grained control over the control-flow of the test case via source code. The effectiveness and efficiency of the implemented solution was evaluated based on the tracer's ability to output the correct control-flow profile for the target test case in Section III-C1. The performance

TABLE I
EBPF TRACING SEQUENCE OF TPM COMMANDS TIMINGS (SECONDS)

TPM commands	Stage 1	Stage 2	Stage 3
TPM_CC_Startup	0,007531	0,256136	0,000118
TPM_CC_CreatePrimary	0,007523	0,246996	0,137073
TPM_CC_Create	0,008129	0,246298	0,003777
TPM_CC_Create	0,008110	0,240702	0,006160
TPM_CC_Load	0,003019	0,245251	0,004703
TPM_CC_EncryptDecrypt	0,007668	0,229365	0,001704
TPM_CC_PCR_Extend	0,008806	0,226297	0,003735
TPM_CC_EncryptDecrypt	0,007701	0,238259	0,000154
TPM_CC_Quote	0,002381	0,219403	0,002293
TPM_CC_VerifySignature	0,008388	0,233758	0,001026
TPM_CC_Shutdown	0,006260	0,215326	0,000140
Average (in seconds)	0.006865	0.236163	0.014625

was measured against two scaling profiles: a high- and a low-complexity one. As will be analysed in Section III-C2, complexity refers here to the target test cases' branch densities. The maximum complexity, in this context, would be a codebase consisting exclusively of branching instructions, while the minimum would contain no branching instructions aside from the final return. In our evaluation we utilise two scaling factors to instrument the number of executed instructions, and thus the complexity, in the target program. Due to space limitations, the codebases used for the following cases are given in this GitHub repository[3]. The interested reader can refer to this repository for more examples, source codes and performance graphs.

*1) **Effectiveness of Control-flow Extraction**:* As has been reported in the literature [4], [15], one of the major challenges of extracting the control flow of a program lies on the fact that codebases include conditional branches and loops that can lead to an explosion of possible control flow paths. That is, we opt to evaluate the effectiveness of the designed IntelPT tracer under a test case which considers a loop containing a conditional branching instruction and a function call. Thus, this case tests the combination of all of the branch types.

This test case (under the name "t6" in our repository) was used in order to evaluate the ability of the IntePT tracer to capture the execution flow of a complex, but still auditable codebase. Due to space limitations, we opt to analyse the most complex scenario; however, more examples can be found in our repository. As can be seen in Fig. 2, the assembly code reveals the execution flow of the codebase of interest. The CFG of Fig. 2 illustrates the flow of each iteration of the loop, with the addition of both the single conditional branch and the function call and subsequent context re-entry. We have to note that for simplicity reasons our analysis was not expanded outside the context of the target function, i.e., the main function, while the condition evaluated is the logical inverse of the if-condition in the source code.

More specifically, the loop starts with an unconditional jump instruction to the set of instructions that evaluate the loop condition and jumps to the top of the loop's contents. Consequently, the flow meets the *if* statement and the respective conditional jump that leads to the function call. The code re-

[3]https://github.com/ubitech/Multi-level-Execution-tracing-evaluation-Intel-PT

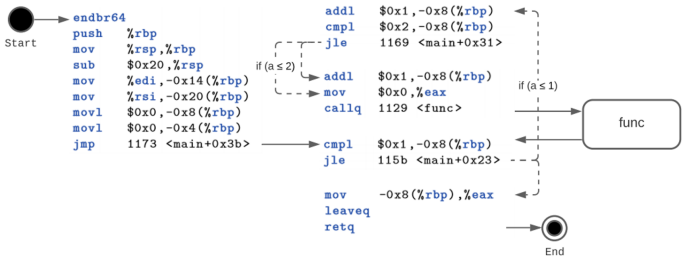

Fig. 2. Test case ontrol-flow graph

```
$ bin/tracer 1 "$(pwd)/bin/t6" main
>ENTER @ 0x55ae44e7c138                          # int main(...)
|JUMP  @ 0x55ae44e7c177 target: 0x55ae44e7c15b  # while (a < 2)[0]
|JUMP  @ 0x55ae44e7c163 target: 0x55ae44e7c169  # if (a > 2)
>ENTER @ 0x55ae44e7c173                          # return from func()
|JUMP  @ 0x55ae44e7c177 target: 0x55ae44e7c15b  # while (a < 2)[1]
|JUMP  @ 0x55ae44e7c163 target: 0x55ae44e7c169  # if (a > 2)
<CALL  @ 0x55ae44e7c16e target: 0x55ae44e7c129  # func()
>ENTER @ 0x55ae44e7c173                          # return from func()
<RET   @ 0x55ae44e7c17d                          # return
```

Fig. 3. Test case trace output

enters the main context and takes a conditional jump at the *while* and the inner *if* statement; enters the function context; and, upon return, terminates its execution. As can be seen in Fig. 3, the aforementioned assembly-based flow is reflected to the IntelPT tracing output, which indicates the combination of these control-flow structures. The decoder outputs a jump to the top of the loop contents, followed by the conditional jump of the *if* statement and the context switch to the function. Then, the tracer captures again the second loop execution with the respective conditions and the function call.

As also depicted in Fig. 3, the tracer outputs the specific values of the memory addresses ($0xFFFFFFF$) that represent the various states in the overall control-flow sequence comprising multiple jump, call and ret instructions; instructions that will change the program counter of the system's execution to point to a different state (memory address). More specifically, these values point to the addresses where the program counter register will point for executing the next instruction in the normal execution flow. This feature is very important as it enables the in-depth investigation of the system's operation in case of an identified deviation from the normal CFG.

Given the tracing output, it becomes clear that IntelPT is a solution that can cope with the documented challenges and barriers of control-flow extraction when dealing with loops, conditions, and function invocations [4], [15]. In addition, IntelPT enables control-flow extraction with adjustable tracing granularity levels depending on the selected configurations. Thus, based on the tracing requirements, one can configure the process to limit the scope to functions and libraries of interest or to treat other codebases as black boxes, if by nature cannot be exploited by attackers to modify their execution flow. More details on the performance of Intel PT under various granularity tracing scenarios are given in Section III-C3.

*2) **Performance Evaluation**:* After documenting the viability of the IntePT tracer to offer an accurate CFG of a codebase,

TABLE II
INTELPT TRACING PERFORMANCE EVALUATION CASES

Case ID	High-Complexity test case				
	Prim. Factor (PF)	*Secon. Factor (SF)*	*Without trace (ms) min/max*	*With trace (ms) min /max*	*Overhead*
HC.1	10^3 to 10^6	1	0,2 / 4,14	5,76 / 5.397,76	26,42x to 2.184,38x
HC.2		10^1	0,22 / 15,8	5,97 / 5.480,87	26,12x to 409,13x
HC.3		10^2	0,34 / 148,07	6,22 / 5.337,76	17,29x to 38,51x
HC.4		10^3	1,5 / 1.342,46	6,03 / 5.449,92	2,69x to 3,24x
HC.5		10^4	14,2 / 13.002,38	18,07 / 17.060	1,09x to 35,54x
	Low-Complexity test case				
LC.1	10^3 to 10^6	1	0,2 / 3,78	2,52 / 1.722,46	11,61x to 1.189,63x
LC.2		10^1	0,23 / 14,75	2,36 / 1.748,74	9,25x to 130,1x
LC.3		10^2	0,37 / 142,84	2,67 / 1.757,86	3,2x to 12,35x
LC.4		10^3	1,71 / 1.413,17	5,07 / 2.761,13	11,76x to 196,62x
LC.5		10^4	14,44 / 13.895,4	30,65 / 34.583	24,63x to 175,59x

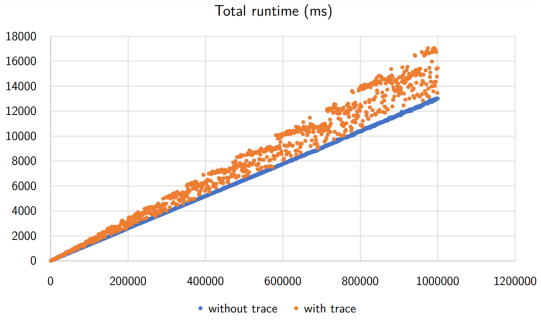

Fig. 4. HC.5 test case performance timings

in what follows, we stress test our tracer under different conditions in order to evaluate the additional overhead incurred in both cases of high- and low-complexity tests.

a) High-complexity test case: This test case designed to facilitate two scaling factors, as it contains two *for* loops, one in the main function and another inside a function outside the trace context, which is called in every iteration of the main loop. The number of iterations for the outer and inner loops are called the primary and secondary scaling factors, PF and SF, respectively. The PF ranges from 10^3 to 10^6 in increments of 1000 and we opt five exponential increments: 1, 10^1, 10^2, 10^3 and 10^4 for the SF. The source code of the target program is given in our repository under the name "t7".

A small SF leads to the worst-case-scenario with maximum control-flow complexity. In the HC.1 case of TABLE II, where SF=1, each iteration of the main loop causes a jump and a call which returns after a single iteration of the inner loop, making for an extremely branch-dense execution. The results show a deviation from 0,2 to 4,14 ms for the target program without tracing, and 5,76 to 5.397,76 ms for the tracer. Taking a look over the results of TABLE II for the tracing case, on can note that the tracer performs in a rather stable manner. However, it becomes obvious that when compared with the program's execution timing an overhead is added, as the tracer tries to cope with the decoding of an extremely branched execution flow. For the HC.1 case, the performance overhead ranges from 26,42x to 2184,38x among the measured execution timings.

As the SF increases, so does the total execution time of the target program without tracing. This is reasonable, as the total iterations of the double-loop codebase needs more time to complete. However, the time needed for tracing shows a stable behavior regardless the increasing SF. This is because the high SF, i.e., the increased time spent by the code in the internal loop (which is out of the tracing context), gives the opportunity to the tracer to catch up the codebase execution and decode the captured execution flow without increasing the overhead. This is advocated by the timing measurements of TABLE II (With/Without trace (min/max)) for the HC.1-4 cases. Specifically, it can be observed that the execution time increases among the cases, while tracing remains rather stable, leading to steady decrease of the posed overhead.

The HC.5 case is of special interest. The time spent executing instructions outside the trace context allows the decoder to catch up with the end of the trace buffer's contents. Briefly put, when the decoder catches up with the tracer, it must note its position in the buffer, re-synchronize to the previous Packet Stream Boundary (PSB) packet and read until it reaches the same place again, at which time there will likely be more to read. A graph of the results is shown in Fig. 4. We need to note here the greatly increased variance in the tracing execution time. This is correlated with the number of bytes the decoder must re-read after re-synchronization. Target program timings now range from 14,2 to 13.002,38 ms, with tracing taking from 18,07 to 17.060 ms. The performance overhead incurred ranges from 1,09x to 35,54x.

b) Low-complexity test case: To form a clear picture on the scaling characteristics of the IntePT tracer, we created a low-complexity codebase which contains a single loop in the main function. As in the high-complexity case, the PF controls the number of iterations in the main loop. On the other hand, the SF is the number of non-branching instructions between branches inside the trace context. The source code of the target program is given in our repository under the name "t8". Our codebase includes macros to allow multiple repetitions of the ADD instruction equivalent to a++. The SF instructed 1, 10^1, 10^2, 10^3 and 10^4 executions of the ADD instruction. TABLE II summarises the execution times of both the standalone target program and the tracing process across the range of the PF and SF scaling values.

LC.1 case shows very large performance overheads, as only one non-branching instruction separates the jump instructions for each iteration of the loop. The execution time of the standalone target program ranges from 0,2 to 3,78 ms, while the tracer runtime ranges from 2,52 to 1722,46 ms, incurring performance overheads between 11,61x and 1.189,63x. Taking a look over the LC.2 and LC.3 results, we note a visible growth in the runtime of the target program, but the tracing timings remain at the same levels among LC.1-3 cases, demoting that the increment on the SF indeed eases the tracer to stay in sync with the target program execution. For the LC.2 and LC.3 we note a decrease in the performance overhead incurred by tracing w.r.t. to execution timings without trace. The LC.4 and

TABLE III
INTEL PT TRACER TIMING (IN SECONDS) FOR VARIOUS GRANULARITY SCENARIOS

Binaries	Full decoding	All branches	All main branches
Binary 1	5,2515	0,17496	0,03698
Binary 2	6,11039	0,20266	0,06549
Binary 3	72,03029	3,25014	0,04501
Binary 4	313,66655	8,32616	1,24342

LC.5 cases are of particular interest. As was the case for HC.5 case, the increased SF give the chance to the decoder to catch up with the tracer and require re-synchronization, and thus, increasing the performance overhead. This behavior justifies the incurred performance overheads of 196,62x and 175,59x for the LC.4 and LC.5 cases respectively.

*3) **Various Granularity Scenarios**:* This section elaborates on additional evaluation cases to demonstrate that the type and granularity of the decoding and pre-processing of the data pulled from the Intel PT tracing can greatly impact the performance of the tracer. That is, we examine the execution time of a full decoding of the entire assembly codes executed (Full decoding), then we chose to only filter and decode all the branches that the binary takes (All branches), and finally we measure the decoding of all the branches within the main function (All main branches). The timing results (in seconds) can be found in TABLE III with each row representing a different binary with incremental codebase complexity and each column denoting a different decoding granularity.

One can observe that the complexity and size of the executed code, greatly impacts the performance of the tracing and decoding of our tracer. For instance, the 4th binary with a full decoding requires 313,66 secs while the 1st and simplest binary with full decoding takes only 5,25 secs. Another important observation is that doing a full instruction tracing and decoding on any assembly command requires considerable time since it decodes the entire chain of execution of commands, even at the level of the operating system. However, if we focus on the branch instructions, we have a reduction in time of around 96%, while further narrowing down to only specific function to trace for its branches (e.g. main) can have even better results.

*4) **Compiler Optimizations**:* In our work we evaluated the effect that compiler optimizations may have on the Intel PT tracing performance. We compiled two of our case studies with the "-O3" compiler option which will enable all possible compile-time optimizations. On the 2nd binary the time for tracing the main function branches before the optimizations was 0,06549 secs and after the optimizations 0,04422 secs, while doing the same on the 3rd binary, the timing before the optimizations was 1,24342 sec, while after 1,30259 sec. It seems that the compiler optimizations will not necessarily lead to better tracing performance, as those results are in the margin of statistical error.

IV. DISCUSSION AND CRITIQUE

This section provides a discussion of our proposed "hybrid" type of tracing solution, while posing some open issues that still need to be considered if CFA is to reach its full potential. Program tracing solutions can sufficiently record run-time information about a program's execution and enable flexible and powerful offline analysis. Therefore, they have become fundamental techniques extensively leveraged in software analysis applications and forensics [18]. Those techniques aim to the generation of the CFG through static program binary testing. However, such offline and rigid forensics analysis methods set several barriers, as they mainly have to ensure that the produced tracing output has not been tampered, while the detection of events takes place after the occurrence of the incident. Thus, we need to investigate and aim for online flexible tracing solutions.

Although detailed tracing and introspection can be resource intensive, it is required for proper and thorough attestation schemes. Thus, we argue that a practical way forward is basically to provide a multi-level detail tracing mechanism that can actually incorporate different types of tracing mechanisms with varying levels of granularity in order to provide much higher scalability. Our evaluation results advocate that eBPF can handle efficiently both the interception and the processing of the traced events, while Intel PT can operate with acceptable overhead when configured to a reasonable granularity level and targeting a specific application scope. Such an optimization allows for better security-performance of the end solution as it allows for a timely response to security incidents. In addition, the proposed "hybrid" tracing approach could allow parallel tracing of different aspects of a running application, having the eBPF and Intel PT tracers acting in parallel, focusing on different aspects of the execution, while sending all the created traced data to the decoder for processing. To this direction, what is proposed, is an attempt for an online tracing and decoding configuration, where during runtime, the tracer feeds the data to the decoder so as to allow the CFG to be generated spontaneously without having to wait program execution completion. In fact, this is a major challenge in the field and the misalignment between the tracer and the decoder has a major impact in the tracing performance.

In order to close this gap, we argue that a well-suited solution for the realisation of the proposed architecture is the use of a ring buffer. A ring buffer provides an ideal FIFO buffer that allows the incoming data to be processed in the correct order and priority and, given the fact that the expected buffer size is predictable, a ring buffer is an optimal solution. In the input of this buffer, the tracer can be attached to feed tracing data as soon as they are available and on the output, the decoder digests data as fast as its capabilities allow and it proceeds with the control flow graph creation.

To the best of our knowledge, the most prominent tracing solutions in the context of CFA, which can also be instantiated in resource constraint systems, assume the existence of additional specialised hardware elements to perform the tracing and CFG extraction. The fact that our approach is based on an embedded feature of widely used Intel processors, renders our solution a pseudo-hardware-based one, as we do not assume the existence of additional hardware. Nonetheless, we need

to highlight the need to move towards pure software-based tracing solutions that will be generic and applicable in a wide range of application domains without the limitations posed by the hardware aspects of devices.

In fact, purely software-based tracing solutions could offer great flexibility and configurability of the tracing mechanism. Crucially, such a solution can be also backwards-compatible with the trusted computing software stack available on most machines today, enabling tracing with high trust assurances. This mode of operation relies on software that resides within the same trust domain of the application to be traced, and the device being inspected. However, we need to consider that the current trusted software stack libraries used to interact with the underline trusted component of a system can be attacked if a host machine is compromised. Thus, it becomes clear that while tracing exhibits strict security and trust requirements is not one and the same with *secure tracing* which is the trust anchor towards secure attestation variants.

In this direction, we need to guarantee the operational integrity of such a critical software component as the tracer. When capitalising on hardware-based solutions for tracing, the trust assurance is delegated to the robustness and security qualities of the hardware. However, for software-based tracing we need to find a trust anchor to meet such strict security requirements. To realise secure tracing, the corresponding enabler needs to be part of the trusted computing base of a system so that it can be protected in an isolated processing environment in which it can be securely executed; without direct dependencies to rest of the system. Hence, in order to overcome the aforementioned limitations posed for realising the secure purely software-based tracing, we need to adopt strong isolation mechanisms and consider secure architectural designs such as RISC-V [19]. The system-level organization of a RISC-V platform can range from a single-core microcontroller to a many-thousand-node cluster of shared-memory manycore architectures. Even small systems-on-a-chip might be structured as a hierarchy of multiprocessors to modularize internal operations and to provide secure isolation.

V. Conclusions

In this work, we presented an efficient and lightweight architecture that effectively extracts rigid control-flow information to be then used for enabling the new generation of control-flow attestation schemes. Our approach is based on the novel reuse of the eBPF and Intel PT technologies for collecting and checking both the configuration and execution properties of a device's codebase while striking a balance among the key requirements of precision, efficiency and transparency. This hybrid solution addresses challenges such as slow decoding and incomplete control traces by leveraging appropriate structures (ring buffers) to optimize the entire tracing process. Our prototype and the evaluation results demonstrate that our architecture can satisfy all the strict performance requirements cementing our vision that such a new breed of tracing mechanisms can be a key enabler for supporting the realization of trustworthy next-generation smart-connectivity SoS.

References

[1] N. Koutroumpouchos, C. Ntantogian, S. Menesidou, K. Liang, P. Gouvas, C. Xenakis, and T. Giannetsos, "Secure edge computing with lightweight control-flow property-based attestation," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 84–92.

[2] B. Kuang, A. Fu, L. Zhou, W. Susilo, and Y. Zhang, "Do-ra: Data-oriented runtime attestation for iot devices," *Computers & Security*, vol. 97, p. 101945, 2020.

[3] D. Papamartzivanos, S. A. Menesidou, P. Gouvas, and T. Giannetsos, "A perfect match: Converging and automating privacy and security impact assessment on-the-fly," *Future Internet*, vol. 13, no. 2, 2021. [Online]. Available: https://www.mdpi.com/1999-5903/13/2/30

[4] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: Control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[5] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead control flow attestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference*, 2017, pp. 24:1–24:6.

[6] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: Lightweight hardware-assisted attestation of program execution," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 106:1–106:8.

[7] F. Kohnhäuser, N. Büscher, S. Gabmeyer, and S. Katzenbeisser, "Scapi: A scalable attestation protocol to detect software and physical attacks," in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2017, pp. 75–86.

[8] H. B. Debes, T. Giannetsos, and I. Krontiris, "Blindtrust: Oblivious remote attestation for secure service function chains," 2021.

[9] T. Zhang, X. Zhuang, S. Pande, and W. Lee, "Anomalous path detection with hardware support," in *Proceedings of the Int. Conf. on Compilers, architectures and synthesis for embedded systems*, 2005.

[10] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, 2012, pp. 1–12.

[11] S. Canakci, L. Delshadtehrani, B. Zhou, A. Joshi, and M. Egele, "Efficient context-sensitive cfi enforcement through a hardware monitor," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2020, pp. 259–279.

[12] R. Ding, C. Qian, C. Song, W. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 131–148.

[13] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 173–184.

[14] B. Larsen, T. Giannetsos, I. Krontiris, and K. Goldman, "Direct anonymous attestation on the road: Efficient and privacy-preserving revocation in c-its," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 48–59. [Online]. Available: https://doi.org/10.1145/3448300.3467832

[15] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 529–540.

[16] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 585–598.

[17] B. Larsen, H. B. Debes, and T. Giannetsos, "Cloudvaults: Integrating trust extensions into system integrity verification for cloud-based environments," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 197–220.

[18] Y. Shoshitaishvili, R. Wang, and S. et. al, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[19] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual volume ii: Privileged architecture version 1.9," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129*, 2016.