FutureTPM

D4.5

Runtime Risk Assessment, Resilience and Mitigation Planning

Project number:	779391
Project acronym:	FutureTPM
Project title:	Future Proofing the Connected World: A Quantum-Resistant Trusted Platform Module
Start date of the project:	1st January, 2018
Duration:	36 months
Programme:	H2020-DS-LEIT-2017

Deliverable type:	Other
Deliverable reference number:	DS-06-779391 / D4.5/ 1.0
Work package contributing to the deliverable:	WP 4
Due date:	December 2020
Actual submission date:	8th February, 2021

Responsible organisation:	UPRC
Editor:	Koutroumpouchos Nikolaos
Dissemination level:	PU
Revision:	1.0

Abstract:	The aim of this deliverable is to firstly present and analyse the FutureTPM dynamic and multi-level tracing solution that it provides for the monitoring of a device's configuration and execution behavioural properties. Secondly, there is a detailed documentation of the design, implementation and performance evaluation of the two underlying technologies used; namely eBPF and IntelPT tracing capabilities. The final aim of this deliverable is to offer a constructive critique of the FutureTPM proposed optimization and set some open questions for further optimizations of real-time device data and execution stream processing and monitoring functionalities.
Keywords:	Multi-Level Tracing, eBPF, Intel PT

The project FutureTPM has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779391.

Editor

Koutroumpouchos Nikolaos (UPRC)

Thanassis Giannetsos (DTU)

Contributors (ordered according to beneficiary numbers)

Koutroumpouchos Nikolaos, Xenakis Christos, Veroni Eleni (UPRC)

Sofianna Menesidou, Dimitris Papamartzivanos (UBITECH)

Roberto Saschu (HWDU)

Disclaimer

Executive Summary

As described in Deliverables D4.2 and D4.4, 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". Compounding this issue, the FutureTPM security enablers (Control-Flow Property-based Attestation and Zero-Touch Configuration Integrity Verification) demonstrate the practice of validating the configuration correctness and execution path that a program takes during run-time. Such validation needs to take place in real-time in order to be an effective countermeasure. This is, however, a difficult task to achieve, as it requires the extraction and analysis of information about the behaviour of a service during run-time. Many solutions have been proposed to address this issue but they tend to make compromises between performance, security and usability. **Clear and precise data on the effectiveness and performance of available tools is required in order to produce more effective and efficient real-time CFA solutions.**

In this direction, the FutureTPM Risk Assessment framework (in its first release) provided a **dynamic multi-level detailed tracing solution with a varying granularity for the amount and the type of data gathered.** This is achieved by leveraging a two-tier granularity tracing technique that initially monitors the targeted binary with an efficient and highly scalable tracing technology named "extended Berkeley Packet Filter" (eBPF); which allows for the tracing of the configurational integrity of the executing binary without posing any significant overhead (around 6% additional time on the traced functions).

In the second and final version of the FutureTPM RA engine, these eBPF hooks are enhanced with more thorough **tracing capabilities based on the use of the emerging IntelPT tracing mechanism**. This programmable component provides near **real-time low-level code inspection**, thus, capturing the strict constraints of SoS-enabled ecosystems, as envisioned in FutureTPM. More specifically, it allows for much more detailed tracing of the binary up to a per-assembly-command monitoring that ultimately enables the efficient extraction of the respective control flow graphs. The results from both developed tracers are always compared against stored reference values through a device attestation scheme and are essentially used to ensure that the security-critical operations of a device are doing what they are supposed to and are not deviating from the expected behaviour. The dynamicity of the tracer usage is **enforced through policies** that define exactly which tracer and how the two tracers are deployed; in a "normal operational mode" only the eBPF tracer is running. However, if an attestation fails, then new policies are deployed which will attach new Intel PT agents and more advanced eBPF hooks allowing for a better understanding of why the initial attestation failed; i.e., identify new vulnerabilities or points of intrusion.

The aforementioned tracing capabilities are used in the remote attestation protocols that are described in D4.4 (Attestation-by-Quote and Attestation-by-Proof). **All in all, in this deliverable, there is a thorough presentation and analysis of the dynamic tracing capabilities of FutureTPM.** This includes a design, implementation, and performance analysis of each of the tracers used with insights on their interworking and assessment of their correct and efficient functionality. Additionally, there is a critique of the FutureTPM used tracing solution and an investigation of how the tracing procedure could be further optimized to allow for even more advanced tracing schemes.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The main goal of this deliverable is to present the final release of the **FutureTPM tracing solution** as part of the overall Runtime Risk Assessment, Resilience and Mitigation planning.

More specifically, it provides the details and underpinnings of the **FutureTPM Device Data and Execution Stream Tracer** (Figure 1-1) for tracing the control- and information-flow execution paths (CFG and DFG) needed by the runtime attestation enablers presented in D4.2 [2] and D4.4 [3]. Such techniques are used to collect statistical information, performance analysis, dynamic kernel or application debug information and in general **system audit traces and evidence** that can be used by the attestation enablers for identifying any integrity or execution correctness violations. In FutureTPM, dynamic tracing functionalities are provided, as programmable components, enabling the continuous monitoring of kernel shared libraries, low-level code, system calls, shared data and memory address space, etc., and the in-depth investigation of the systems' behaviour and execution flow for detecting any cheating attempts or if any type of (non-previously identified) exploits are resident to the program and data memory. This provides the trusted anchor (i.e., TPM) with the compiled control- and information-flow graphs (CFGs & DFGs) that represent the runtime state of a remote device, against the configuration and execution properties of only those safety-critical components, as identified and modelled in the deployed attestation policies, extracted from the FutureTPM RA Engine.

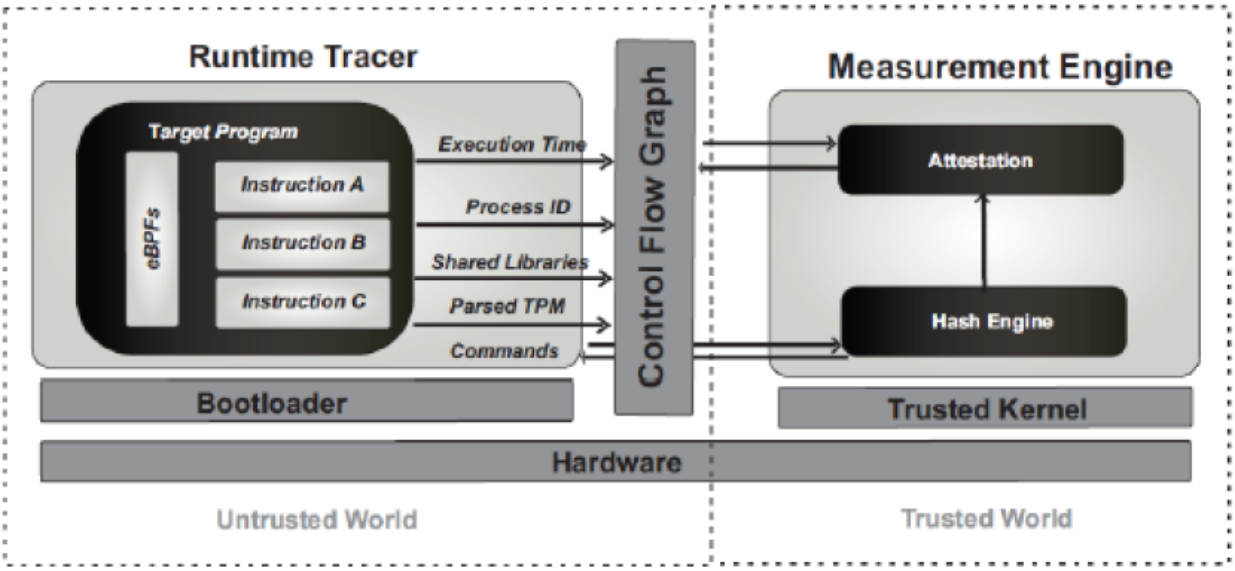

Figure 1-1: FutureTPM Device Data and Execution Stream Tracer

FutureTPM advanced tracing techniques are based on the novel use of: (i) lightweight eBPF [23] execution hooks capable of providing near real-time low-level code inspection, thus, capturing the strict constraints of SoS-enabled ecosystems, and (ii) Intel PT introspection agents [13] capable of traversing the entire physical memory of a CPS, via Direct Memory Access, for known execution signatures which can then be used to navigate to the relevant information to be traced – this allows for fast tracing to take place without affecting the normal system execution. **These tracing functionalities are fully programmable, thus, enabling another priority of the FutureTPM Risk Assessment framework towards dynamic adaptation of tracing and introspection tasks based on new attestation policies that may be identified and enforced during runtime.**

Therefore, the high-level overview of this deliverable is focusing on this dynamic tracing capabilities coupled also with a (first to the best of our knowledge) detailed evaluation of the two

aforementioned approaches. We: (i) first, analyse how the tracing process can escalate depending on the security level conformance that need to be achieved (i.e., properties to be attested and whether the focus is on configuration or executional behaviour integrity), (ii) then proceed to elaborate on the initial eBPF (extended Berkeley Packet Filter) tracing and monitoring with a detailed analysis of its performance and deployment details, and (iii) finally, present and analyse the usage of the low-level detailed tracing capabilities based on the Intel PT technology. **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 are enforced which will automatically deploy the Intel PT tracer which will in turn gather more detailed data on how, why and which 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.

1.1 Scope and Purpose

As aforementioned, the main purpose of this document is to consolidate, formally define and evaluate the FutureTPM dynamic tracing solution. The goal is to create a coherent analysis of the two levels of tracing mechanisms produced (eBPF and Intel PT) while also defining how the dynamicity of the solution will function in the context of FutureTPM; *escalated tracing will be triggered and enforced alongside the standard tracing and monitoring*.

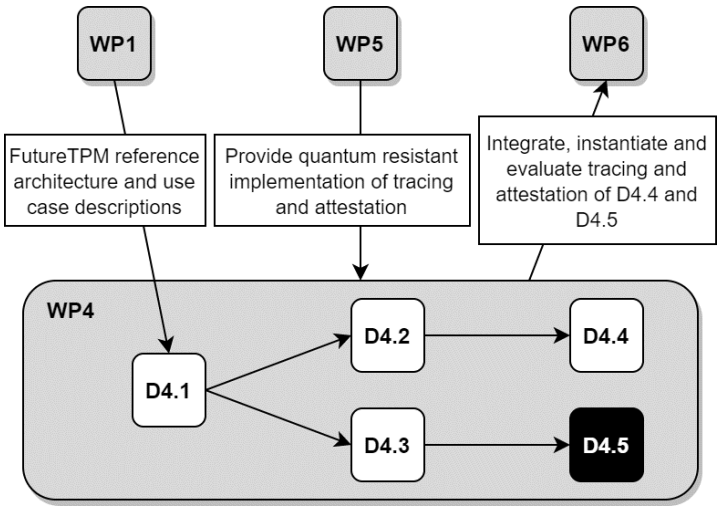

Figure 1-2: Relation to other deliverables and other work packages

1.2 Relation to other WPs and Deliverables

Figure 1-2 reflects on the relationship of this deliverable to the other activities conducted in the context of Work Package (WP4) as well as the other project work packages. More specifically, D4.5 is an extended version of D4.3 [1] and represents the final version of the FutureTPM RA Engine presented in D4.2 [2]. More specifically, the interactions as described in Figure 1-2 are:

- The connection with D4.4 is a hand-in-hand relationship due to the fact that the attestation and configuration integrity verification enablers, described in that deliverable, are based on the correct instantiation and execution of the compiled device data and execution stream tracer components presented here;
- This deliverable is the second and final version of D4.3 where we initially described the concept behind the multi-level detailed tracing approach of FutureTPM based on which

we now elaborate on the integral components, APIs, interfaces coupled with technical, implementation and performance documentation;
- The connection with WP5, on the implementation of the QR Future TPM platform, is the fact that all presented attestation and tracing functionalities leverage the quantum resistant trusted platform module as the underlying trust anchor;
- The connection with WP6 is the implementation and evaluation of the attestation and tracing capabilities in the context of the three envisioned use cases.

Our goal is to **enhance run-time software integrity and trustworthiness with a scalable and decentralized solution eliminating the need for federated infrastructure trust**. Based on our findings, we also posit open issues and challenges, and discuss possible ways to address them, so that the challenge of efficient tracing does not hinder the deployment of intelligent edge (trust) computing systems.

1.3 Deliverable Structure

The structure of this deliverable is as follows: Chapter 1 gives an overview of the focus of this documents coupled with a high-level overview of the core properties achieved by the FutureTPM tracing solution produced. Chapter 2 summarizes the dynamic tracing approach adopted by the FutureTPM RA Engine alongside with background information on its mode of operation. Chapter 3 and Chapter 4 constitute the core of this work by presenting the underpinnings of the two leveraged tracing mechanisms, eBPF execution hook and Intel PT introspection agents, respectively. They put forth a detailed analysis of their workflow of actions as well as an evaluation of their performance and any overhead posed on the underlying device for extracting the target control- and data-flow graphs. To the best of our knowledge, this is the first complete evaluation and comparison performed in the literature between such two different “tracing directions” in an attempt to propose a hybrid solution taking advantage of the benefits of both worlds. Finally, in Chapter 5 there is a discussion and critique of these tracing performance optimization techniques and future directions. Closing in Chapter 6 there is the conclusion of the deliverable where key takeaway messages are presented with a coherent overall summarization of the deliverable.

Chapter 2 FutureTPM Dynamic Multi-Level Detailed Tracing

Towards our vision in future proofing the connected world by having a QR TPM as the trust anchor, the FutureTPM consortium investigated on complementary building blocks that can work in synergy with the QR TPM for enabling security and trust in the envisioned decentralised architectures.

Towards this direction, the FutureTPM overall framework provides advanced risk assessment functionalities for extracting security policies that reflect attestation specifications for properties that are verified on the devices with the usage of the attached QR TPMs. Additionally, FutureTPM investigates how to perform real time risk assessment updated by zero-day and newly identified vulnerabilities.

The main goal is to specify models capturing all those safety-critical components, that need to coexist and be securely executed in a platform with shared hardware and software resources (such as caches and central memory bus), and their connection to the overall system behaviour. This will allow for the identification of only those functions (such as algorithms, control and device operational logic elements) with the highest criticality level, that need to be continuously verified in real-time, thus, enabling the better specification of core device properties, accessed by these functions, to be attested.

The adoption of the risk assessment process aims to deliver a well-rounded framework that can be used to protect future decentralized system architectures using as a solid base the well-established methodologies of the field, but by enhancing them accordingly to meet the needs of the assessment of TPM-enabled architectures. More specifically, the use of TPMs enables a wide range of protective mechanisms of the trusted computing realm than can be used as possible countermeasures and extend the arsenals of defenders, in their effort to ensure that risk levels are kept under acceptable threshold. In this direction, our developments in the context of WP4 led to the definition a risk assessment conceptual flow shown in Figure 2-1 that can steer the assessment conduction during the whole life cycle of a system, starting from the design phase of the system and through its runtime phase.

In what follows, we first present an overview of the methodology adopted towards the implementation of the FutureTPM Risk Assessment Framework. Deliverables D1.2 [4] and D4.1 [5] presented the guidelines for the FutureTPM Risk Management phase, while D4.2 [2] and D4.3 [1] highlighted the various design aspects and specifications. The FutureTPM project has developed a generic model to identify all possible threats and vulnerabilities, and a risk assessment methodology that transparently augments the security, of not only the QR TPM, but also of the whole ecosystem of deployed devices through the quantification of the overall risk and threat vector that, in turn, enables the identification of the optimal defence strategy tailored to calculated cyber-risks. The risk assessment methodology is comprised of a **design-time phase**, where an initial risk graph is created, and a **run-time phase** described in D4.3 [1], where the risk graph is updated (taking into consideration newly identified threats and vulnerabilities based on the output of the Control-Flow Property-based Attestation (CFPA)) toolkit in order to achieve the desired security and privacy properties. These properties are instantiated as policies that have to be enforced during the life cycle of the CPS. Overall, FutureTPM Risk Assessment Framework is used to:

- **identify** and **measure** all relevant cyber threats including TPM threats.
- **predict** all possible attacks/threats paths and patterns.
- **evaluate** the **individual**, **cumulative,** and **propagated** vulnerabilities.
- estimate the existence of **zero-day exploitable** vulnerabilities.
- **assess** the possible impacts.

- **derive** and **prioritize** the corresponding risks.
- **formulate** a proper mitigation strategy.

The whole framework is based on UBITECH's OLISTIC risk management suite [6], which is in position to maintain a digital reflection of the cyber-physical ecosystem in the form of interdependency graphs that hold together all the interrelated assets of the assessed environment. Based on this, the risk assessor is engaged in a semi-automated process where attestation policies are enforced to the deployment in order to attest its operational integrity using the TPM as the trust anchor in this process. Given the attestation outcome, the assessor can be sure for the operational correctness of the attested components of the deployment and in cases where the operational correctness is not verified, the assessor can proceed to the deployment of additional policies and checks to further investigate the security status during runtime.

Figure 2-1 depicts the high-level overview of the Risk Assessment execution workflow. **High-level security and trust policies, based on the produced risks, are created and interpreted to low-level behavioural attestation policies.** In this way, fine-grained trust assumptions are defined and are refined iteratively. Such fine-grained trust assumptions can be used to precisely delimit the contextual interactions under which the TPM's security guarantees are proved to hold, and can be monitored, or even partially enforced, through the Control-Flow Attestation Toolkit. These low-level policies are received by the Policy Decision Point, and through the Policy Enforcement Point are deployed and enforced to the host devices ecosystem. Finally, the low-level properties will be attested by the CFPA and the Attestation Report will contain the final verdict. For more information regarding the Continuous Cyber-Risk Assessment, one can refer to D4.2 [2], D4.3 [1] and D6.1 [7].

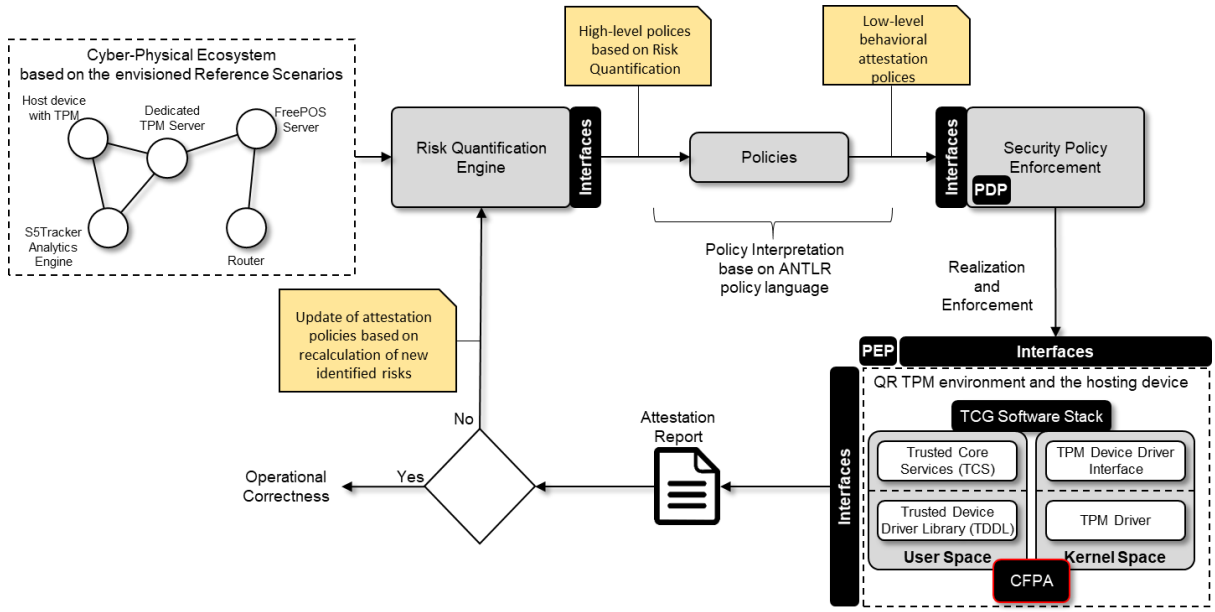

Figure 2-1: Risk Assessment Conceptual Flow

In order to support real time risk assessment and to have a continuous up to date attestation, the FutureTPM project (in the context of D4.2) described the workflow of **control-flow attestation trust extensions** and, in the context of D4.4, presented a newly designed set of **configuration integrity verification enablers**. These two are extreme variants of remote attestation covering different aspects of the operational lifecycle of the device. The goal was to design secure remote attestation and execution services, targeting both the software and hardware layers and covering all phases of a device's execution; from the trusted boot and integrity measurement of a CPS, enabling the generation of static, boot-time or load-time evidence of the system's components correct configuration (Configuration Integrity Verification

(CIV)), to the runtime behavioural attestation of those safety-critical components of a system providing strong guarantees on the correctness of the control- and information-flow properties (as modelled in T1.4), thus, enhancing the performance and scalability when composing secure systems from potentially insecure components.

- **Software Attestation:** Novel attestation mechanisms providing unforgeable evidence about the state of software executing on a CPS. Emphasis is placed on attestation based upon the configuration and execution properties of the target device that were identified in D3.2. More specifically, the focus is on enforcing the remote attestation requirements that were specified as part of the attestation policies, extracted from the FutureTPM EA Engine, through the precise attestation of the execution path of the application running on the target CPS. The goal is to enhance the currently existing environments (with standalone or single devices) with well-known static attestation methods and extend them to a dynamic attestation of "Systems-of-Systems" by measuring a program's execution path. More concretely, the developed attestation mechanisms will incrementally authenticate (via a keyed hash) specific execution paths (i.e., branch commands), that are efficiently and continuously monitored by the deployed tracing programmable components.
- **Hardware Attestation:** This is one important building block for truly secure attestation, especially when considering attacks or processes that may alter or replace the trusted component. We therefore also want to investigate by what means authenticity of hardware can be attested. This is done via the developed QR TPM.

In the context of secure remote attestation, a basic component is to be able to monitor and trace the measurements of the state of the device (configurational or behavioural states) and compare and verify them against pre-recorded safe reference values and states. When it comes to such advanced tracing mechanisms, the FutureTPM project (in D4.3) has described the workflow of multi-level detailed tracing which solves the literature-defined challenge of efficient and lightweight monitoring, and tracing of processes. Therefore, the FutureTPM Device Data and Execution Stream Tracer provides a "hybrid" approach, by leveraging two of the most prominent tracing mechanisms currently been proposed enabling the continuous monitoring of kernel shared libraries, system calls, shared data and memory address space, etc., and the in-depth investigation of the systems' behaviour for detecting cheating attempts or if any type of exploits to the program and data memory

As a first step, in the overall workflow of actions, we trace the order of execution of commands especially in the TSS that essentially provides the aspects for the correct configuration of the underlying TPM. This is achieved by leveraging the very scalable and lightweight eBPFs. If the attestation enablers fail to properly verify some aspects of the defined policies (based on the eBPF-based extracted graphs), in order to be able to have a more detailed and low-level tracing so as to understand what went wrong (such as the point of intrusion that potentially was exposed by an attacker), we trigger the deployment of more detailed introspection agents leveraging the new generation of Intel PT tracing functionalities which can actually provide the low-level control-flow graphs of the system's execution. To summarize, we propose the usage of lightweight eBPFs for integrity verification of the configuration (D4.4) and of the more advanced Intel PT tracers to monitor the control flow attestation (D4.2).

FutureTPM performs the risk assessment process that identifies the attestation policies for the integrity verification (correct device state) which is divided in the verification of configurational properties integrity and the verification of the correct behavioural aspects of the device. **To perform this type of attestation, we need to have access to the current state of the device which ultimately is provided through tracing techniques.** The main challenge here is to have the efficient near-real-time tracing, during runtime, without having a large overhead (completely depleting the device resources). In this context, we have to consider what we are aiming to trace; that is in integrity verification we need to trace a static property (easy to monitor) while in the behavioural tracing we need to trace something dynamic (hard to monitor). With this in mind, we need to identify a hybrid tracing model that can accommodate both of

these two peculiarities. This exactly is what is proposed and implemented by FutureTPM which is presented in the following Chapter 2.1.

2.1 Multi-Level Detailed Tracing based on eBPFs and Intel PT Programmable Components

As described in the previous chapter and also depicted in Figure 2-2, there does not yet exist a comprehensive design nor an effective as well as efficient implementation for enabling dynamic tracing. Moreover, existing tracing-based attestation approaches are limited to single device attestation, whereas in FutureTPM we target attestation of systems-of-systems to address the emerging class of interconnected embedded devices in the Internet-of-Things. To provide strong security assurance in this context, we bridge this gap within FutureTPM and developed novel dynamic attestation mechanisms, particularly focusing on behavioral-based attestation dynamic properties of software and hardware for systems-of-systems, through the deployment of **eBPF execution hooks** and **Intel PT based system call tracing**.

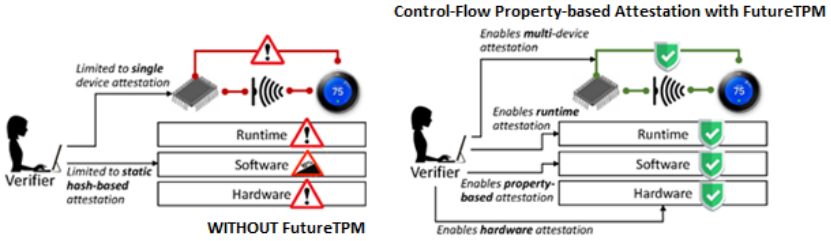

Figure 2-2: FutureTPM Methodology and Advancement to Remote Attestation

These kernel hooks, initially, are identified as low-level behavioral properties and are deployed to trace system calls. **The goal of this initial tracing is to monitor system calls, produce the necessary CFGs and finally acquire the attestation report.** In the case of a failed attestation report, FutureTPM has defined a methodology for increasing the level of monitoring in order to collect additional evidence and information on the incident for the assistance in finding the province of the attack as well as in the development of new enforceable policies that should be able to catch this newly identified threat that caused the attack in the first place. This escalated tracing will be based on Intel PT, a technology that is capable of decoding execution codes on-the-fly during runtime and allow for a detailed introspection of what is done by the binary. More details on the Intel PT utilization in the context of FutureTPM will be analyzed in Chapter 3. **Effectively, we propose a novel solution for multi-level detailed tracing that, depending on the situation, monitors the security-critical components of each system in different levels in order to upkeep the desired assurance while also providing the required details for post-attack investigation.**

This multi-level detailed tracing is implemented in a semi-automatic manner. Towards this direction, there are two possibilities explored within FutureTPM. The first is the automatic deployment of new, richer, programmable eBPF hooks after the reception of a failed attestation report. In this case, the security analyst(s) (performing the offline investigation) assess the attestation report and determines whether more information is needed for identifying the cause and type of attack - in which case she also defines the set of policies describing the type of information/evidence to be collected. After this process, new programmable eBPF kernel hooks are seamlessly and automatically deployed to the target device for fulfilling these policies. As previously noted, the eBPF is implemented in the kernel, thus, needs a userland control agent for remote control and exportation of the data. Such agents are implemented as part of the FutureTPM RA framework. Overall, this approach enables us to get the necessary evidence without affecting the performance of the target system since these rich eBPF hooks in cooperation with the Intel PT tracing (that will incur a higher penalty in the system execution as more system calls need to be monitored in near real-time) will be deployed. However, there is

the inherent limitation of a large attack time window: Since the detailed tracing commences after the attestation process has failed, this means that the attack is already in place which in turn results to the additional requirement of "letting" the attack to run further (for a larger amount of time) in the device until the detailed tracing has finished. Depending on the application in hand, such an approach (which will of course require the isolation of the attacked device during the evidence collection so as other adjacent assets are not affected) might not be feasible.

Compounding this issue, another approach is the deployment, during design-time, of eBPF hooks already capable of enhanced monitoring and tracing of most of the operational system calls. In this case, only those execution hooks that are necessary for tracing the CFGs to be attested will be active during the run-time risk assessment phase (i.e., including the execution of the CFPA mechanism) and the enriched eBPFs will be activated only if the evidence collection is triggered; thus, resulting to the transmission of the necessary information about the device that got compromised, vulnerabilities found in any of the internal system components or their configurations, configuration changes, etc. This approach while less efficient, as the penalty of increased tracing is incurred from the beginning, does not necessitate a large attack time window. Further investigation on the feasibility and usability of these two mechanisms were done during the first testing phase of the FutureTPM demonstrators and are documented in this deliverable (D4.5).

Summarizing the process of the FutureTPM tracing hooks deployment, multi-level detailed tracing and trust evidence collection mechanisms, the conceptual workflow is depicted in Figure 2-3.The process of this deployment is as follows:

1. An attack or malfunction occurs causing an enforced policy to fail as detected by the output of the Control-Flow Property-based Attestation mechanism.
2. The security analyst(s) assess if she needs additional information related to the attack incident. If she can directly deduce what was the cause of the attack, then the process ends here.
3. The security analyst(s) defines the required information (through adequate policies) and directly deploys (Case 1 in Figure 2-3) or activates (Case 2 in Figure 2-3) the enriched eBPF execution hooks for collecting the necessary additional information.
4. All the information collected is fed to the FutureTPM Run-time Risk Assessment framework for performing a re-calculation of the overall risk and threat vector (considering Backwards Chaining and Cascading Effect Analysis). Compilation and enforcement of new attestation policies that should be able to cover it in the future.

The two main key points of this method is that the security analyst will either require or not any additional tracing. The standard tracing kit will be running on the end device (prover) generating attestations of its control-flow graphs and sending them to the verifier. When a policy fails, then the security analyst will read the attestations generated by the prover and will assess if she needs additional tracing to identify the source of the event. Depending on her choice she will either end the evidence collection process or deploy/activate any additional tracing required to gather the required data. For example, when a buffer overflow attack occurs that will alter the control flow graph (and thus, failing the corresponding behavioral property policy) of the monitored application, then there might be the need for additional information such as what memory regions where overwritten, what memory regions where over-read, or which entity started the malfunction. On the other hand, when an unidentified application is found to run on a restricted device (thus failing the corresponding configuration property policy), then from the already existing attestation the security analyst can black-list this application without any additional information. In both cases, new policies will be compiled that will be able to catch this attack before it happens and prevent it from breaching the security of the end device.

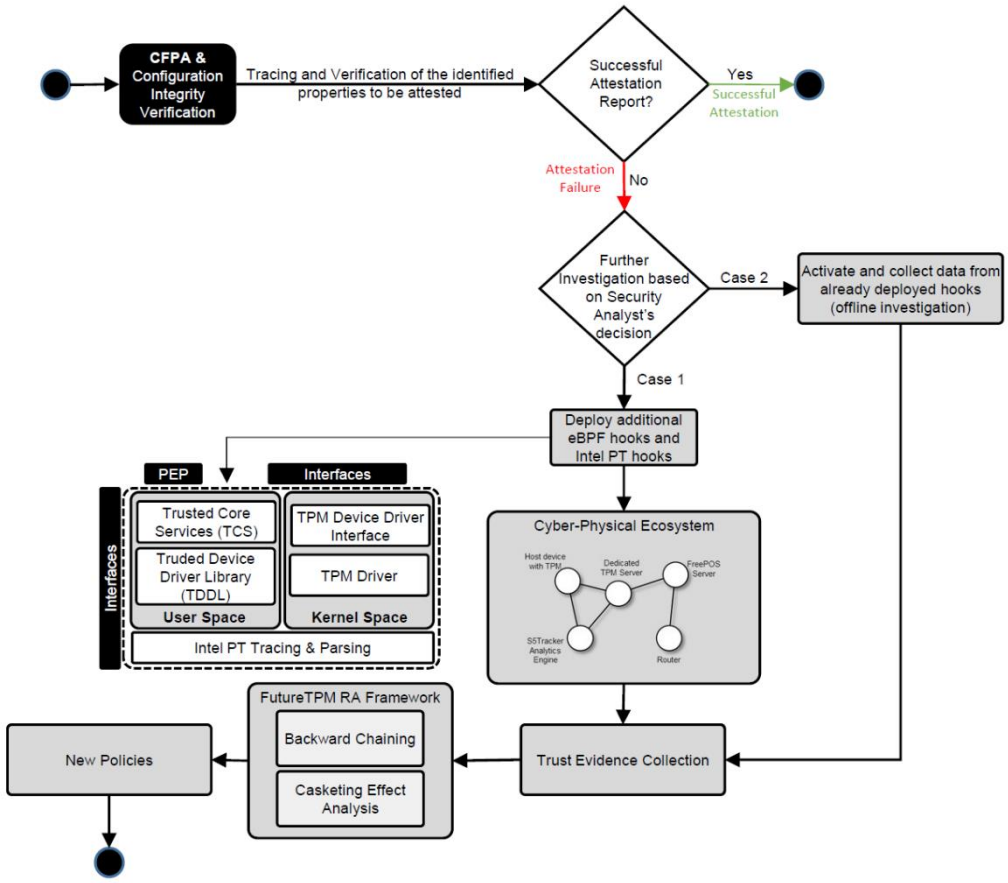

Figure 2-3: Conceptual Workflow of Multi-level Detailed Tracing & Trust Evidence Collection

In order to support periodic or on-demand attestation actions, it is necessary to collect and analyse low-level code information during run-time. This type of monitoring is based on the integration of advanced tracing techniques. The CFPA Runtime Tracer performs a set of traces, including the **eBPF-based tracing** for QR-TPM sequence of commands and the binaries hashing for Configuration Integrity Verification (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). Note that, these two tracing mechanisms are deployed in order to provide execution traces from multiple perspectives. In fact, eBPF-based tracing can provide deep insights to the execution of TSS commands and infer the configuration integrity of critical software components, while Intel-PT-based tracing captures low-level functionalities in an efficient way.

To achieve auto configuration on the necessary deployed trace agents during runtime, the CFPA Runtime Tracer communicates with the Attestation Agent, which receives the attestation specifications, including the tracing configurations, from the Attestation Engine. Overall, both the eBPF and Intel-PT Tracers have the following inputs and outputs.

Input: Trace Configurations (included in an attestation policy)
Output: Traces, Extracted CFGs

2.2 Formal Definition of Property Tracing

Consider a program P, defined as a sequence of N code blocks B_0, ..., B_{N-1}, such that each block is a memory address offset at the beginning of a sequence of 0 or more non-branching instructions, followed by 1 branching instruction in the source binary of P. Let $B_0 \rightarrow B_1$ denote a

control-flow transition between two states; more specifically, from the last instruction of B_0 to the first instruction of B_1. Unless otherwise specified, it is assumed that a block identifier on the left-hand side of → refers to the last instruction of the identified block, and an identifier on the right-hand side refers to the first instruction of that block.

To specify a different instruction within a block, the following notation can be used: Given blocks B_0 = 0x1000, B_1 = 0x1005 and B_2 = 0x100a, each consisting only of single-byte instructions for the purpose of demonstration, it can be said that $B_0 \rightarrow B_1$ (+x). Here, x is a non-negative integer and less than the size of code block B_1 in bytes. This denotes a transition from 0x1004, the last instruction of B_0, to the instruction at offset x in B_1.

Let CFG(P) be the complete and correct control-flow graph of P, the set of all legitimate control-flow transition in P, and E(P) be the control-flow graph of a single execution of P. For such graphs, it is helpful to think of code blocks as states in a state-machine and branches as transitions between them. S denotes the start state of an execution and E denotes the end state. In order to perform real-time CFA, E(P) must be extracted and each control-flow transition verified as a member of CFG(P) during runtime of P. E(P) is valid if and only if $E(P) \subseteq CFG(P)$. This can also be written $\forall x \in E(P)(x \in CFG(P))$, indicating that all elements of E(P) are also elements of CFG(P).

2.2.1 Control-Flow Graph Structure

Consider the following example: Figure 2-4 shows a simple C program which loops 10 times, calling a function *foo* in every other iteration of the loop. It is the program P in this example, consisting of 6 blocks, B_0,.., B_5. Each of these blocks represents a list of linear instructions followed by a branching instruction. *For simplicity, the focus will be entirely on user defined code and any additional code generated by the compiler will be ignored.*

```
1  int foo()
2  {
3      return 0;
4  }
5
6  int main (int argc, char* argv [])
7  {
8      int a = 0;
9
10     for (int i = 0; i < 10; i++)
11     {
12         if (a % 2 == 0)
13         {
14             foo();
15         }
16
17         a++;
18     }
19
20     return 0;
21 }
```

```
1  0:1129: endbr64 // foo
2  0:112d: push   %rbp
3  0:112e: mov    %rsp,%rbp
4  0:1131: mov    $0x0,%eax
5  0:1136: pop    %rbp
6  0:1137: retq
7  1:1138: endbr64 // main
8  1:113c: push   %rbp
9  1:113d: mov    %rsp,%rbp
10 1:1140: sub    $0x20,%rsp
11 1:1144: mov    %edi,-0x14(%rbp)
12 1:1147: mov    %rsi,-0x20(%rbp)
13 1:114b: movl   $0x0,-0x8(%rbp)
14 1:1152: movl   $0x0,-0x4(%rbp)
15 1:1159: jmp    1177 <main+0x3f>
16 2:115b: mov    -0x8(%rbp),%eax
17 2:115e: and    $0x1,%eax
18 2:1161: test   %eax,%eax
19 2:1163: jne    116f <main+0x37>
20 3:1165: mov    $0x0,%eax
21 3:116a: callq  1129 <foo>
22 4:116f: addl   $0x1,-0x8(%rbp)
23 4:1173: addl   $0x1,-0x4(%rbp)
24 4:1177: cmpl   $0x9,-0x4(%rbp)
25 4:117b: jle    115b <main+0x23>
26 5:117d: mov    $0x0,%eax
27 5:1182: leaveq
28 5:1183: retq
```

Figure 2-4: Control-Flow Example Program P

First, CFG(P) is generated. P should always start executing at the beginning of the main function at address offset 0x1138. This can be written formally as $S \rightarrow B_1$. Since B_1 ends with an unconditional branch to offset 0x1177, belonging to B_4, $B_1 \rightarrow B_4$ (+8) is written to CFG(P). The process has now entered the for-loop and is about to check whether its condition is met. If i < 10, it will jump to offset 0x115b. Since this is a conditional branch, both $B_4 \rightarrow B_5$ and $B_4 \rightarrow B_2$ are written. In B_2, the condition a mod 2 = 0 is evaluated, and a conditional jump back to B_4 follows. Therefore, $B_2 \rightarrow B_4$ and $B_2 \rightarrow B_3$ are written. Since B_4 has already been covered, only B_3 needs to be examined, which ends with a call to the foo function at offset 0x1129. $B_3 \rightarrow B_0$ is written. For B_0, the locations of corresponding call instructions must be considered. In this case, there is only one call to that function. From this, $B_0 \rightarrow B_4$ is written, denoting the transition from the called function to the instruction at 0x116f since it immediately follows the corresponding call at 0x116a. Finally, the process enters B_5, which ends with a return instruction which marks the end of P. This is simply written as $B_5 \rightarrow E$. It has now been determined that CFG(P) = {$S \rightarrow B_1$, $B_1 \rightarrow B_4$ (+8), $B_4 \rightarrow B_5$, $B_4 \rightarrow B_2$, $B_2 \rightarrow B_4$, $B_2 \rightarrow B_3$, $B_3 \rightarrow B_0$, $B_0 \rightarrow B_4$, $B_5 \rightarrow E$}. This CFG is illustrated in Figure 2-5.

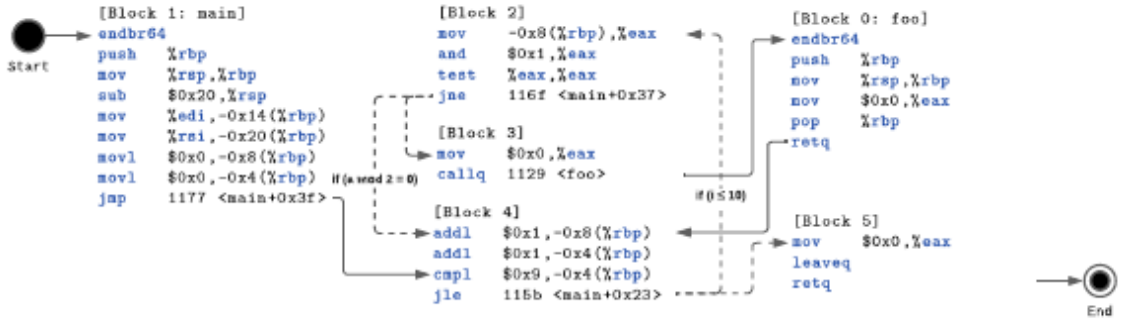

Figure 2-5: Visual Representation of CFG(P)

Now, E(P) can be generated and validated against CFG(P). Execution starts at 0x1138, at the start of B_1, so $S \rightarrow B_1$ is written to E(P). This can be validated: $(S \rightarrow B_1) \in$ CFG(P), so the execution continues, combining the extraction and validation steps. From there, the process unconditionally jumps to 0x1177 in B_4, so $(B_1 \rightarrow B_4 (+8)) \in$ CFG(P) is written. Now, the states of the integer variables i and a must be tracked. At this point, they both equal 0, so the condition i < 10 evaluates to true. The conditional branch is therefore taken and $(B_4 \rightarrow B_2) \in$ CFG(P) is written. The next condition is a mod 2 = 0, and the value of a has not changed, so it also evaluates to true. So $(B_2 \rightarrow B_3) \in$ CFG(P) is written. Next, the function call can simply be written as $(B_3 \rightarrow B_0) \in$ CFG(P), followed by $(B_0 \rightarrow B_4) \in$ CFG(P). a and i are incremented by 1 and the loop repeats. This time, a mod 2 = 0 evaluates to false, so the process jumps from B_2 to B_4. $(B_2 \rightarrow B_4) \in$ CFG(P) is verified. These two iterations are repeated until the loop exits, so there is nothing more to write, assuming the program runs correctly. The final evaluations, $(B_4 \rightarrow B_5) \in$ CFG(P) and $(B_5 \rightarrow E) \in$ CFG(P) hold, so it can be said that E(P) = {$S \rightarrow B_1$, $B_1 \rightarrow B_4$(+8), $B_4 \rightarrow B_2$, $B_2 \rightarrow B_3$, $B_3 \rightarrow B_0$, $B_0 \rightarrow B_4$, $B_2 \rightarrow B_4$, $B_4 \rightarrow B_5$, $B_5 \rightarrow E$}, and E(P) $\subseteq$ CFG(P). Therefore, E(P) is valid.

Suppose that, during this process, E(P) = {$S \rightarrow B_1$, $B_1 \rightarrow B_4$(+8), $B_4 \rightarrow B_2$, $B_2 \rightarrow B_3$, $B_3 \rightarrow B_0$, $B_0 \rightarrow B_4$, $B_2 \rightarrow B_4$} has been written. The function foo is currently executing at B_0. This time, a malicious actor has found some way to corrupt the stack, and when foo returns, the process does not transition to B_4. Instead, P starts executing some function in libc or another external library. This can have serious consequences and cause P to behave in unexpected ways. For the purposes of this example, the target block shall be called B_x, and, thus, $B_0 \rightarrow B_x$ is written. It is immediately apparent that $(B_0 \rightarrow B_x) \notin$ CFG(P), and E(P) becomes invalid. The fault has been detected and can be dealt with.

Such a control flow graph will be generated for the behavioural monitoring of the use cases of FutureTPM, namely Secure Mobile Wallet and Payment, Activity Tracking and Device Management. For example, in the use case of the mobile device e-Payment of FutureTPM, we want to attest the correct behaviour of the mobile phone when it connects to the POS application in order to execute a financial transaction. We want the steps taken within this functionality to be verified before allowing the execution of the transaction. These steps are essentially translated to the branches that the internal binary will take towards achieving this functionality. More specifically, in the e-payment use case, these steps as defined in D6.5 that include the correct functionality of the FIDO registration, the FIDO authentication, and the management of bearer and financial tokens; these are the functionalities that we want to be able to trace and attest for their operational correctness.

Chapter 3 High-Level Tracing Leveraging eBPF Execution Hooks

As we have already described the (multi-level) dynamic tracing and attestation mechanisms of FutureTPM, in the remaining part of this deliverable we focus on the **implementation and evaluation details of the two types of tracers that we leverage: the eBPF execution hooks and the Intel PT introspection agents**. The aim is to give the details of how these two techniques operate in tandem when it comes to the underlying models and algorithms but also their detailed evaluation. In this chapter we start with the more scalable and efficient eBPF tracer. In what follows, we evaluate the hardware FPGA-based TPM provided to us by Infineon since we have already presented the tracing of the SW-based TPM in D4.3 [1]. Since the TPM is FPGA-based, the connection to it is made through a network interface with the usage of the TCP/IP stack. This fact forces us to make an adjustment to the traditional tracing and add another layer that is able to capture and analyse the TCP/IP protocol packets. This will unquestionably add some latency to our measurements but as we will see, even with the network overhead, the eBPF tracing is very efficient and able to provide static and very fast timings even for the worst-case scenarios (when tracing quantum resistant algorithms).

The eBPF runtime tracer was introduced in D4.1 [5] and D4.2 [2] with a further elaboration on the dynamic multi-level tracing of FutureTPM in D4.3 [1]. In the context of the final release of the runtime risk assessment, resilience, and mitigation planning, we provide more implementation details, while we also elaborate on the integration points and its usage. Furthermore, we also present the timing results we measured from the execution of core QR algorithms as they are implemented in the HW-based QR TPM. The designed tracer is built on top of BPF Compiler Collection (BCC), which is a toolkit for creating efficient kernel tracing and manipulation programs. The tracer is logically divided into two parts. The BPF code for tracing the kernel is written in C language, and the user-space tool is written in Python. In order to develop a dynamic tracing tool, one needs to be aware of the kernel or application internals. That is, in the context of FutureTPM, the eBPF tracing focuses on the internal commands used for interacting with the TPM.

3.1 Implementation Path Report

The designed tracer takes advantage of the Virtual File System (VFS) layer, which provides an interface for accessing files and directories from different devices and file systems. The tracer focuses on the *vfs_open, vfs_read* and *vfs_write* functions of the VFS interface [8] in order to intercept (hook) the TPM function invocations and resource accesses. More specifically, the C-based BPF interceptor traces the input and returned parameters generated by the invocation of the aforementioned functions. This process reveals the exact instructions and the corresponding parameters launched by the TPM Software Stack (TSS) to the TPM. Once these parameters are traced in the kernel space, they are parsed by the Python-based implementation, which undertakes their transition to the user-space and their matching with the exact TPM commands. To do so, the Runtime tracer is aware of the structure of the TPM Library and is able to manipulate all the TPM commands. A detailed documentation of the TPM command structures is given in [9]. The TPM structure included in the Python-based implementation has been extended in order to include the new QR TPM commands.

The pseudocode snippet given in Figure 3-1, gives an overview of the execution flow of the eBPF Runtime Tracer. The tracer takes as inputs the kernel functions to be intercepted, a process ID and a character device corresponding to the application of interest and the BPF program to be executed. The workflow consists of three discrete phases which are described below.

Algorithm 1: FutureTPM eBPF tracer

Input : kernel_function, pID, character_device, bpf_code
Output: traced_tpm_commands

```
/* Initialisation phase                                                    */
1  b ← initialize_BPF(bpf_code, pID, character_device)
2  b.JIT_compile()                    // BPF code verification and compilation in kernel
/* eBPF instantiation phase                                                */
3  kprobes ← create_kprobes(kernel_function)          // dynamic tracing of a function call
4  kretprobes ← create_kretprobes(kernel_function)  // dynamic tracing of a function return
5  b.attach_hooks(kprobes, kretprobes)                    // introduce hooks to BPF
/* Tracing phase                                                           */
6  b.open_perf_buffer()                                          // open BPF buffer
7  while true do
8  |  traced_event ← b.perf_buffer_poll()              // get tracing event from BPF buffer
9  |  tpm_command ← tpm_parser(traced_event)  // returns command header & parsed command
10 |  write(tpm_command)
11 end
```

Figure 3-1: FutureTPM eBPF Runtime Tracer – Instantiation, Initialization and Tracing Phases

Initialization phase:

- **eBPF initialisation:** The BPF Compiler Collection (BCC) is used for compiling the BPF program. After the compilation, the bytecode is pushed to the kernel and is executed by an in-kernel sandboxed virtual machine and uses its own instruction set (line 1 Figure 3-1). The initialisation function provides a reference (b) to the user space code which can be used for interacting with the BPD program in the kernel and attach on-demand tracing functions. (Figure 3-2)

```
bpf = BPF(src_file="fpga_hook.c", debug=0)
```

Figure 3-2: eBPF Initialisation

- **Code verification and JIT-compilation:** Silently, and in the background, the kernel runs a verifier on the bytecode to make sure the program is safe to run. The kernel JIT-compiles the bytecode to native code and attaches it in the in-kernel sandboxed virtual machine for execution (line 2 Figure 3-1).

eBPF instantiation phase:

- **Kernel probes creation:** To enable dynamic tracing of a kernel function call, *kprobes* and *kretprobes* are created for the kernel function provided as input to the tracer. The former is used for the interception upon the function invocation, while the latter is used for the interception the function return (lines 3-4 Figure 3-1).
- **Attaching hooks:** By attaching kprobes and kretprobes to the BPF running program instance dynamically, we instrument the interception of events when the kernel function is called (line 5 Figure 3-1). After, this step the eBPF program has been instantiated including the necessary hooks and every function invocation will be traced and logged. (Figure 3-3)

```
bpf.attach_kprobe(event="tcp_v4_connect", fn_name="trace_connect_v4_entry")
bpf.attach_kretprobe(event="tcp_v4_connect", fn_name="trace_connect_v4_return")
bpf.attach_kprobe(event="tcp_set_state", fn_name="trace_tcp_set_state_entry")
```

Figure 3-3: eBPF Hook Attachment

- **Event Handlers:** Each attached probe has a corresponding handler function; namely trace_connect_v4_entry, trace_connect_v4_return and trace_tcp_set_state_entry. Every

time that a TCP connection is taking place in the kernel, we are fetching the pid (process ID) of the process that initialized that connection with the help of a predefined eBPF helper function. The pid is then pushed to the hash table map "connectsock". Accordingly, on the return of the TCP connection kernel routine, we are making a look up at the hash table map to verify that we have hooked the right process (the one that triggered the TCP connection). Afterwards, we store the name of the process and the IPv4 addresses of the receiver (FPGA) and the sender (our local system), as a tuple in another hash table map "tupleid_ipv4" for later use. These two processes are implemented through the trace_connect_v4_entry and trace_connect_v4_return handler functions and are depicted in Figure 3-4.

```
int trace_connect_v4_entry(struct pt_regs *ctx, struct sock *sk)
{
  u64 pid = bpf_get_current_pid_tgid();
  //##FILTER_PID##
  // stash the sock ptr for lookup on return
  connectsock.update(&pid, &sk);
  return 0;
}
int trace_connect_v4_return(struct pt_regs *ctx)
{
  int ret = PT_REGS_RC(ctx);
  u64 pid = bpf_get_current_pid_tgid();
  struct sock **skpp;
  skpp = connectsock.lookup(&pid);
  if (skpp == 0) {
      return 0;       // missed entry
  }
  connectsock.delete(&pid);
  if (ret != 0) {
      // failed to send SYNC packet, may not have populated
      // socket __sk_common.{skc_rcv_saddr, ...}
      return 0;
  }
  // pull in details
  struct sock *skp = *skpp;
  struct ipv4_tuple_t t = { };
  if (!read_ipv4_tuple(&t, skp)) {
      return 0;
  }
  struct pid_comm_t p = { };
  p.pid = pid;
  bpf_get_current_comm(&p.comm, sizeof(p.comm));
  tuplepid_ipv4.update(&t, &p);
  return 0;
}
```

Figure 3-4: Tracer Connect Event Handler

When the TCP connection stabilizes, meaning that it is at a certain state, then another kernel routine takes place and the function displayed below (Figure 3-5) handles the breakpoint. We are fetching the saved data from the hash table based on the *struct sock* that was used for the specified kernel routine and then we store the desired fields in a struct (*evt4*) that is "pushed" inside an eBPF table (*tcp_ipv4_event*) which is used for pushing out custom event data to user-space via the perf ring buffer (*perf_submit()*).

```
int trace_tcp_set_state_entry(struct pt_regs *ctx, struct sock *skp, int state)
{
  if (state != TCP_ESTABLISHED && state != TCP_CLOSE) {
      return 0;
  }
  u8 ipver = 0;
  if (check_family(skp, AF_INET)) {
      ipver = 4;
      struct ipv4_tuple_t t = { };
      if (!read_ipv4_tuple(&t, skp)) {
          return 0;
      }
      if (state == TCP_CLOSE) {
          tuplepid_ipv4.delete(&t);
          return 0;
      }
      struct pid_comm_t *p;
      p = tuplepid_ipv4.lookup(&t);
      if (p == 0) {
          return 0;       // missed entry
      }
      struct tcp_ipv4_event_t evt4 = { };
      evt4.ts_ns = bpf_ktime_get_ns();
      evt4.type = TCP_EVENT_TYPE_CONNECT;
      evt4.pid = p->pid >> 32;
      evt4.ip = ipver;
      evt4.saddr = t.saddr;
      evt4.daddr = t.daddr;
      evt4.sport = ntohs(t.sport);
      if (ntohs(t.dport)!=2321){
        return 0;
      }
      evt4.dport = ntohs(t.dport);
      evt4.netns = t.netns;
      int i;
      for (i = 0; i < TASK_COMM_LEN; i++) {
          evt4.comm[i] = p->comm[i];
      }
      tcp_ipv4_event.perf_submit(ctx, &evt4, sizeof(evt4));
      tuplepid_ipv4.delete(&t);
  }
  // else drop
  return 0;
}
```

Figure 3-5: TCP Set State Handler

- **Socket Attachment:** Hooking the FutureTPM tracer to the TCP connection that every TSS command creates is not enough. The packets transferred through each connection need to be further analyzed as these packets will contain the payload that needs to be extracted in order to acquire a structured information of the TPM commands. To accomplish that, another function is attached (which is going to be used for filtering of incoming packets) to a socket which in turn is attached to the interface which is used for the communication with the hardware based TPM. (Figure 3-6)
  The shown commands are responsible for assigning a file descriptor (*socket_fd*) to the eBPF function and then attaching it to the socket. Creating such a socket, we specify the domain (AF_PACKET), socket type (SOCK_RAW which translates to attaching to a raw socket) and protocol (IPPROTO_IP due to the fact that the packets we are interested are encapsulated inside IPv4 packets).

```
function_extract_packets = bpf.load_func("extract_packets", BPF.SOCKET_FILTER)

BPF.attach_raw_socket(function_extract_packets, interface)

socket_fd = function_extract_packets.sock

sock = socket.fromfd(socket_fd, socket.PF_PACKET, socket.SOCK_RAW, socket.IPPROTO_IP)

sock.setblocking(True)
```

Figure 3-6: TCP Socket Attachment

Every time a packets arrives at the specified interface; it is available for processing through the extract_packets function which is shown in Figure 3-7. With the *cursor_advance()* function we gain access to a pointer inside the payload of the packet. We first check that it is of "Ethernet" type and that it has the proper header length. If the conditions are not met, we drop the current packet as it is not of interest. Afterwards, through the *cursor_advance()* function we move inside the TCP payload and we check whether the source or destination port of the packet is the specified by Infineon port which handles external connections (2321) on the FPGA TPM. When this requirement in not met, we once again drop the packet as we are certain that is not a packet of interest.

```
int extract_packets(struct __sk_buff *skb){
  u8 *cursor = 0;

  struct ethernet_t *ethernet = cursor_advance(cursor, sizeof(*ethernet));

  if (!(ethernet->type == 0x0800)){
    goto DROP;
  }

  struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));

  u32 tcp_header_length = 0;
  u32 ip_header_length = 0;
  u32 payload_offset = 0;
  u32 payload_length = 0;

  ip_header_length = ip->hlen << 2;

  if (ip_header_length < sizeof(*ip)) {
    goto DROP;
  }

  void *_ = cursor_advance(cursor, ip_header_length - sizeof(*ip));

  struct tcp_t *tcp = cursor_advance(cursor, sizeof(*tcp));

  if ((tcp->src_port != 2321) && (tcp->dst_port != 2321))
  {
    goto DROP;
  }

  // KEEP:
  return -1;

  DROP:
  return 0;
}
```

Figure 3-7: TCP Packet Extract

Tracing phase:

- **Fetching traced events:** The tracing process pushes evens into a buffer (*open_perf_buffer()*). The tracer, after gaining access to that buffer (line 6), is constantly pulling traced events out of it (lines 7-8). However, as mentioned previously, one needs to be aware of the kernel or application internals in order to "translate" the traced event and extract a structured representation of the executed commands. Furthermore, we

also initialize the TpmParser() class that includes all the subclasses of the TPM commands, in order for each of those classes to be ready for initialisation. (Figure 3-8)

```
tpm_parser = TpmParser()

bpf["tcp_ipv4_event"].open_perf_buffer(print_ipv4_event)

while True:
    bpf.perf_buffer_poll()
```

Figure 3-8: TPM Command Parse

- **Unmask TPM commands:** In order to unmask the traced events and acquire a structured information of the TPM commands sequence, the tracer parses the events by utilising a library that contains the TPM structure [9] and produces a structured information of the TPM command header and the command's details (line 9). Once the tracing even is parsed, the tracer logs the executed TPM command in the user space (line 10).

  Every time a TCP connection between our system and the FPGA is made (and the corresponding event takes place), the *print_ipv4_event* is called. Now our task is to extract the correct payload which is the one that is given to the hardware based TPM in order to process the specified command (write command) and the one that the TPM itself produces and sends back as a result (read command). This payload exists in packets of the TPM protocol (the TPM protocol is encapsulated inside the TCP protocol).

  The first information collected is the as pid and name of the process that triggered the TCP connection in order to have a more complete output. This information is included in the event structure of Figure 3-9.

```
def print_ipv4_event(cpu, data, size):
    event = bpf["tcp_ipv4_event"].event(data)
```

Figure 3-9: eBPF Initial Information Collection (pid, process name)

In order to find the packets with the aforementioned payloads, we need to recognize the packets which are sent for every TPM command. This is achieved by searching for certain patterns that exist within the TPM SEND_COMMAND or the SESSION_END packets. (Figure 3-10)

```
SEND_COMMAND = r"\x00\x00\x00\x08"
SESSION_END = r"\x00\x00\x00\x14"
```

Figure 3-10: TPM Search Terms to Identify the Start and the End of a Sequence

The TPM packet that consists of the first sequence of bytes shown in Figure 3-10 in the TPM segment, is sent from the TPM Software Stack (TSS) and is the first TPM packet that will be sent to the FPGA hardware TPM, meaning that a sequence of other TPM packets is going to be sent in order to complete the TPM command. Respectively, the second sequence of bytes is included in the last TPM packet, in the TPM segment and it signifies that the TPM command has been executed and the TCP connection is about to close. With the help of these 2 packets, we are able to intercept the sequence of the TPM packets and to find the correct payloads.

At this point, the socket attached to the appropriate interface is read and interpreted. The packets that arrive at the socket are those who have been kept after the filtering that was done in the eBPF function *extract_packets.* By setting the maximum size of bytes that will be reading for every packet to 4096, we make sure that we will not have any data loss due to the fact that the maximum packet size is 1514 bytes by default. After conducting some processing, we are left with the "*payload_for_print*" string which consists of the bytes in the TPM segment. (Figure 3-11)

```
while True:
    packet_str = os.read(socket_fd, 4096)

    mode = ''
    acc = ""
    data_to_print = ''
    packet_bytearray = bytearray(packet_str)

    ETH_HLEN = 14

    total_length = packet_bytearray[ETH_HLEN + 2]
    total_length = total_length << 8
    total_length = total_length + packet_bytearray[ETH_HLEN + 3]

    ip_header_length = packet_bytearray[ETH_HLEN]
    ip_header_length = ip_header_length & 0x0F
    ip_header_length = ip_header_length << 2

    tcp_header_length = packet_bytearray[ETH_HLEN + ip_header_length + 12]
    tcp_header_length = tcp_header_length & 0x0F
    tcp_header_length = tcp_header_length >> 2

    payload_offset = ETH_HLEN + ip_header_length + tcp_header_length + 20

    payload_string = packet_str[(payload_offset):(len(packet_bytearray))]

    payload_for_print = toHex(payload_string)
```

Figure 3-11: Packet Parsing

- **Interpreting Quantum Resistant Commands:** Normally the write and read payload can be transferred through one TPM packet. That is not the case though when the TPM command interpreted is using quantum resistance algorithms. The processing then is altered, and we are going to explain exactly the steps for handling such cases.
  Every time we come across a SEND_COMMAND packet, we set a Boolean flag to True so that we know that a sequence of packets following a TPM packet is taking place. On the contrary, when we come across a SESSION_END packet, we set this flag to false so that we know that the TCP session is about to end. The *packet_count* variable increments only when we are inside a TCP session and when the desired payloads that we want to extract are not fragmented (when we are not using quantum resistant algorithms in a command). This process is depicted in Figure 3-12.

```
if payload_for_print == SEND_COMMAND:
    flag = True

if payload_for_print == SESSION_END:
    flag = False
    packet_count = 0
    break

if (flag and fragmented is not True):
    packet_count+=1

if (fragmented_seq):
    fragmented_seq = False
    continue
```

Figure 3-12: QR Command Parsing with Workaround for the Large Size of the Packet

When we execute a TPM command using quantum resistance algorithms the write and read payloads in the TPM segments of the packets are exceeding the maximum packet size and the network layer has to fragment the total payload in more packets than one. In this scenario, we have to accumulate the payload to a buffer until the packet that we

come across is less than the maximum packet size. Also, we have to increment the packet number that the packet with the read payload. For every maximum size TPM packet (containing the write payload), a TCP re-transmission packet has to be sent, so the TPM packet with the read payload is going to be delayed by two packets. The code for implementing the aforementioned process is shown below in Figure 3-13.

```
# case where the write to tpm needs fragmentation
if(total_length==1500 and packet_count==2):
    if(fragmented):
        previous_payload += payload_for_print
        packet_of_read_payload = packet_of_read_payload + 2
    else:
        previous_payload += payload_for_print[20:]
        packet_of_read_payload = packet_of_read_payload + 2
    fragmented = True
    continue

if(packet_count==2 and total_length!=1500):
    fragmented = False
```

Figure 3-13: Handling of Fragmented Packet

At this point we have to store the final payload to the buffer that is going to be sent to the Tpm_Parser() in order for the command to be parsed at the network layer. (Figure 3-14)

```
if (packet_count == packet_of_write_payload):
    start_ts = time.time()
    #case where the write payload is not fragmented
    if not previous_payload:
        write_buf = payload_for_print[20:]
    else:
        previous_payload += payload_for_print
        write_buf = previous_payload
    acc+=write_buf
    previous_payload = ""
    mode+='W'
```

Figure 3-14: Storing of the Final Write Payload

Accordingly, we are handling the case for the read payload based on the resulting variable *packet_of_read_payload*. (Figure 3-15)

```
if (packet_count == packet_of_read_payload):
    end_ts = time.time()
    if(total_length == 1500):
        previous_payload += payload_for_print
        fragmented = True
        fragmented_seq = True
        continue
    if(fragmented):
        previous_payload += payload_for_print
        if(previous_payload[-24:] == TPM2_SUCCESS):
            read_buf = previous_payload[:-16]
        else:
            read_buf = previous_payload
        fragmented = False
        fragmented_seq = False
    else: # case where the payload packet has not been fragmented
        read_buf = payload_for_print
    acc+=read_buf
    mode+='R'
```

Figure 3-15: Storing of the Final Read Payload

For the last step, after we have made some modification to the data (payload) we send them to our parser, with the necessary information provided. We also, print the result with a help message. (Figure 3-16)

```
if mode == 'W':
    tpm_parser.tpm_command = data
    tpm_parser.offset = 0
    parsed_command = tpm_parser.parse(event.pid,'W')
    if parsed_command:
        data_to_print += "[i] MANAGED TO PARSE COMMAND SEND TO TPM!!\n"
        data_to_print += pformat(parsed_command)
elif mode == 'R':
    tpm_parser.tpm_command = data
    tpm_parser.offset = 0
    parsed_command = tpm_parser.parse(event.pid,'R')
    if parsed_command:
        data_to_print += "[i] MANAGED TO PARSE COMMAND RECEIVED FROM TPM!!\n"
        data_to_print += pformat(parsed_command)

data_to_print+= "\n"+"-"*177
print(data_to_print)
```

Figure 3-16: Send all Data to the Parser with Auxiliary Information

The aforementioned workflow of the eBPF Runtime Tracer execution is also depicted in the sequence diagram of Figure 3-17. The diagram provides an abstract representation of the interactions between the user land environment of the tracer with the subcomponents of the system's kernel and the BPF program.

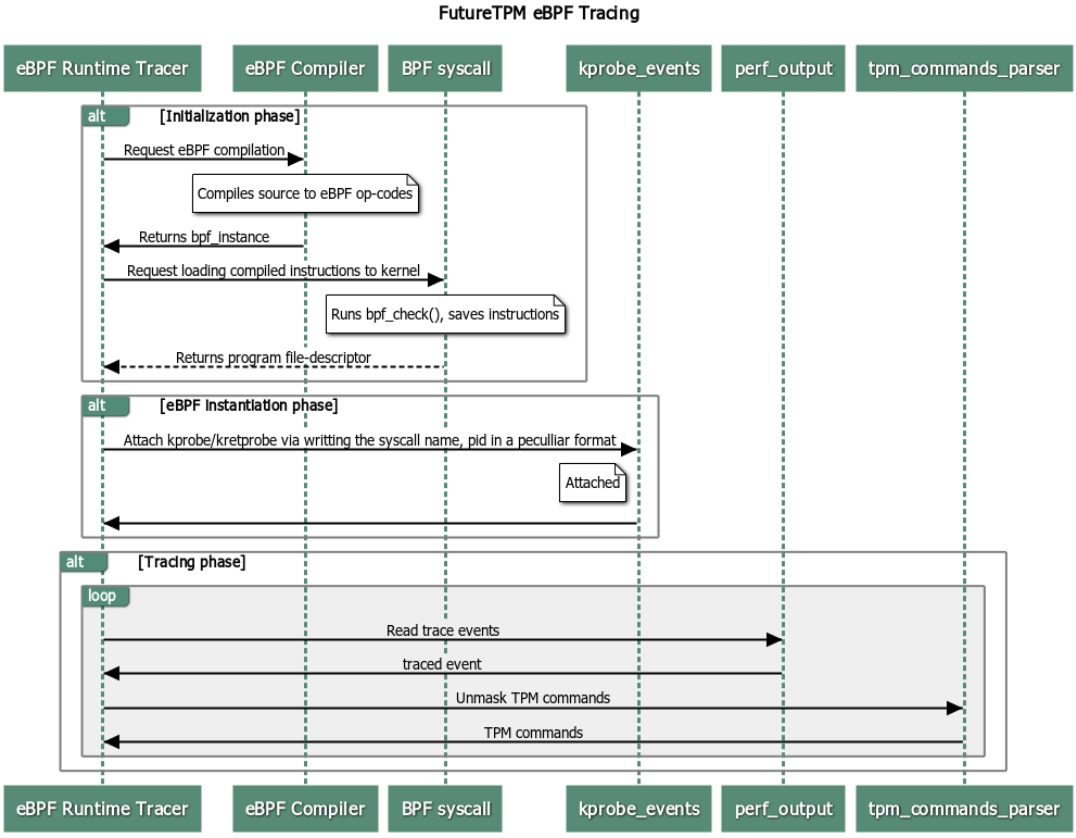

Figure 3-17: FutureTPM eBPF Runtime Tracer – Workflow of Actions

Apart from intercepting the TPM commands, the eBPF Runtime Tracer proceeds to the generation of the CFGs based on the captured sequences of traces, where the CFG nodes represent command executions and the edges the control-flow transitions. That is, this representation can be used both for generating the measurements database, i.e., the points of reference for the legitimate behaviour of an application running on the QR TPM-enabled systems, and for attesting the execution behaviour of an application at runtime. The process

and the properties of the CFG generation have been discussed in D6.2, while further details will be given in Chapter 4.2.

Overall, the eBPF Runtime Tracer has the following inputs and outputs.

Input: Trace Configurations

Output: Traces, Extracted CFG

An indicative example of a traced TPM command which is published in the *trace* topic of Kafka is given in Table 1.

Table 1: Example of traces published to trace topic in JSON format.

Trace topic
{ "tpmcomand":"tpm_create", "algorithm":"SHA256", "lenght":"256", "parameters":"t", "digest": "ebcf86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08", "timestamp": "2020-03-28 24:23 12 904", "executestatues": "success" }

3.2 eBPF Tracer Timings

In this chapter, we put forth the implementation path that we followed for architecting the evaluation of the eBPF tracer based on extracted timings that we measured in the context of TPM command execution and eBPF tracing with the usage of the network connected physical TPM. More specifically, we measured various command execution times on an actual hardware TPM which was connected through a network interface, and we measured the time that the command required to execute, the time that was required for the data to travel through the network interface and the time that our eBPF tracer required to decompile the captured data into human-readable assembly code. It is note-worthy that the TPM is connected through a higher-than-normal latency network interface whereas a normal TPM would be connected with a much faster and responsive bus (LPC or SPI). The fact that the entire TPM has to go through the TCP/IP stack increases the timings and we expect that the presented timings would be decreased in a scenario where the TPM is properly connected to the device. Additionally, we use an external tool to capture the network data (Wireshark) which further adds to the added delay that we expect that will be absent from a more common setup.

Next, we provide a more detailed presentation of the workflow of the executed commands throughout the entire process of the execution and tracing. When a TPM command is executed, the TSS takes on the task of marshalling the data. That is when our eBPF tracer starts its execution by intercepting the commands at the network layer of our system. 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. Afterwards, the executed TPM command is processed and the resulting data are transferred by sending the TPM packets through the network interface and the socket that was created by the TSS. Once again, our eBPF tracer intercepts and unmasks the payload of the TPM command. Then the data are fetched by the TSS, which in turn un-marshals the payload in order for the tracing process to terminate.

For demonstration reasons, we are going to execute a TPM command within the TSS with the "-v" parameter enabled (for producing a verbose output). The resulting output is depicted in Figure 3-18.

```
Extending 32 bytes from file into 1 banks
TSS_Execute: Command 00000182 marshal
TSS_Execute_valist: Step 1: initialization
TSS_Execute_valist: Step 2: authorization 0
TSS_Execute_valist: session 0 handle 40000009
TSS_Execute_valist: Step 5: command encrypt
TSS_Sessions_GetDecryptSession: Found 0 decrypt sessions at 0
TSS_Execute_valist: Step 6 calculate HMACs
TSS_HmacSession_SetHMAC: Step 6 session 40000009
TSS_Execute_valist: Step 7 set command authorizations
TSS_Execute_valist: Step 8: process the command
TSS_AuthExecute: Executing TPM2_PCR_Extend
TSS_Socket_Open: Opening 192.168.0.16:2321-mssim
TSS_Socket_SendCommand: TPM2_PCR_Extend
 TSS_Socket_SendCommand length 65
 80 02 00 00 00 41 00 00 01 82 00 00 00 10 00 00
 00 09 40 00 00 09 00 00 00 00 00 00 00 00 01 00
 0b 46 d4 8c 7e 17 0a 71 ca 9e 1f c7 e1 77 e5 7b
 53 75 df c4 3a 44 c9 65 4b 18 97 ce b1 92 e0 21
 50
 TSS_Socket_ReceiveCommand length 19
 80 02 00 00 00 13 00 00 00 00 00 00 00 00 00 00
 01 00 00
TSS_Execute_valist: Step 9 get response authorizations
TSS_Execute_valist: Step 10: process response authorization 40000009
TSS_Execute_valist: Step 13: response decryption
TSS_Sessions_GetEncryptSession: Found 0 encrypt sessions at 0
TSS_Execute: Command 00000182 unmarshal
TSS_Execute: Command 00000182 post processor
TSS_Socket_Close: Closing 192.168.0.16-mssim
pcrextend: success
```

Figure 3-18: Output of the PCR Extend Function Invocation

Based on the above figure, the workflow of the command is demonstrated, but only at the application layer, omitting the tracing steps and the process on the FPGA. The executed binary essentially runs an authorization transaction with various steps that include HMAC calculation, encryption/decryption, and signature/verification with the quantum resistant algorithms BLISS and NewHope. To measure both the execution time of the TPM commands as well as the tracing time we did the time measurements in the following stages.

Stages of timings:

- *Stage 1*: INITIATE – TSS Data Marshal

  At this stage, we measure the time it takes for the command from the start of its execution until right after the TSS marshals the data. For this to happen, we have inserted a patch inside the TSS source code in order to extract the desired timing. On the tss.c file and in more particular the TSS_Execute function we declare our variables which will store the time. Then we execute the command which stores the time in our variable right after the marshalling of the data was made. (Figure 3-19)

```
rc = TSS_Marshal(tssContext->tssAuthContext,
                 in,
                 commandCode);
clock_gettime(CLOCK_REALTIME, &ts);
```

Figure 3-19: Storage of TSS Marshal Timing

- *Stage 2*: TSS Data Marshal – eBPF Write Packet Interception

  At this stage we measure the time it takes for the command right after the marshalling of the data until we intercept the packet with the write payload at the network layer.

- *Stage 3*: eBPF Read Packet Interception – Command Unmask

  At this stage we measure the time from the interception of the read payload (after the processing was done at the TPM) at the network layer, until our command had been unmasked and parsed with the help of our TPM structure.

Before we present the measured timings, it is important that we define the properties of interest; that is the actual timing of a function to be executed, the time required by the tracer to monitor the executed function and how these two timings compare so as to find any possible overhead (if the tracer timing is significantly larger than the function execution time). The timing results can be found in Table 2 where the performance of tracing various TPM commands is depicted. We measured the timings of basic TPM commands such as "*TPM_CC_Startup*" and advanced commands that include the FutureTPM-used quantum resistant algorithms NewHope and BLISS. The overall timings of Stage 1 are in the order of milliseconds; a reasonable result since data marshalling and TSS command preparations are quick although some of the operations are more complicated given the added requirements of quantum resistant algorithms. It is of

note here that the PCR extend command took more time in Stage 1 than any other quantum resistant algorithm, an indication that the new algorithms do not take any additional toll on this part of the process. In Stage 2, the time required from the data marshalling to the interception of the outbound write packet (from the TSS to the TPM) is around 0.24 seconds and, finally, the Stage 3 timings which are rather fast, thus, validating the efficiency of this part of the FutureTPM Tracing Component (sub 0.005 seconds) with the only exception being that of the creation process of a NewHope key. The Stage 3 measurements provide a strong evidence that the eBPF tracer that we implemented is capable of unmasking TPM commands rapidly and efficiently without substantial overheads.

We evaluate the measured timings by using the properties of interest we defined. The actual execution time of the function is the one found in Stage 1 plus the one identified in Stage 2 that essentially begins right after the function commences its execution and terminates before the command is sent to the TPM; thus, it measures the execution time within the TSS. On the other hand, in Stage 3 we measure the 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 seconds, Stage 2 (on average) takes 0.236163 seconds and the Stage 3 takes (on average) 0.014625 seconds. Following the methodology depicted in Figure 3-20, we conclude that the **eBPF solution has an average overhead of 6.017825% which proves the fact that eBPFs execution hooks are very efficient and lightweight.**

$$Stage\ 1 + Stage\ 2 = 0.243028\ sec$$

$$\frac{Stage\ 3}{Stage\ 1 + Stage\ 2} \times 100 = 6.017825\%\ Overhead$$

Figure 3-20: Measurement of eBPF Overhead

Additionally, we would once again like to notice that the entire setup is based on a network-attached TPM, something that would further decrease the measured time of both Stage 2 but also Stage 3 since all of these measurements are affected by the added delays that occur from the TCP/IP stack and the usage of the Wireshark program to capture the network packets (the reading and writing commands have to go through the network between the FPGA hardware TPM and our host device). Furthermore, the measured timings are mostly static, something that stems from the fact that we have not added any new commands as part of the TPM algorithm portfolio since the overall goal of FutureTPM is to enable the migration to a QR TPM ecosystem with the trusted component being able to act as a *"plug-and-play"* replacement of the current TPM 2.0. **This shows that the eBPF tracer has the same operational timings irrespective of the complexity of the traced function**; *something that would provide even smaller overheads in more complicated functions*. All in all, we argue that the eBPF tracer is a very efficient solution and ideal for our use case where we continuously monitor the function execution, and we required a low overhead approach.

Table 2: eBPF Tracer Timings

TPM Command	Stage 1	Stage 2	Stage 3
TPM_CC_Startup	0.007531 sec	0.256136 sec	0.000118 sec
TPM_CC_CreatePrimary (NewHope)	0.007523 sec	0.246996 sec	0.137073 sec
TPM_CC_Create (NewHope)	0.008129 sec	0.246298 sec	0.003777 sec
TPM_CC_Create (Bliss)	0.008110 sec	0.240702 sec	0.006160 sec

TPM Command	Stage 1	Stage 2	Stage 3
TPM_CC_Load	0.003019 sec	0.245251 sec	0.004703 sec
TPM_CC_NewHope_Enc	0.007668 sec	0.229365 sec	0.001704 sec
TPM_CC_PCR_Extend	0.008806 sec	0.226297 sec	0.003735 sec
TPM_CC_NewHope_Dec	0.007701 sec	0.238259 sec	0.000154 sec
TPM_CC_Quote (Bliss)	0.002381 sec	0.219403 sec	0.002293 sec
TPM_CC_VerifySignature (Bliss)	0.008388 sec	0.233758 sec	0.001026 sec
TPM_CC_Shutdown	0.006260 sec	0.215326 sec	0.000140 sec

Chapter 4 Low-Level Detailed Tracing Leveraging Intel PT Introspection Agents

After covering the initial tracing technique using the very lightweight and scalable eBPF tracing hooks, we will now focus on the enhanced tracing solution of FutureTPM leveraging the Intel PT introspection agents. The main difference in the mode of operation between the Intel PT and the previously used eBPFs, is that the former will **trace the behavioural aspects of the traced binary and create the exact control flow graphs followed while the eBPF provided a static configurational integrity tracing**. Intel PT tracing provides a deep and low-level assembly monitoring and tracing while it is also able to create the exact control flow graph of the binary traced. This is achieved by monitoring in real-time the binary and capturing low-level executional and behavioral information. This information is then loaded into the decoder that is responsible for translating this information to actual assembly structures from which the control-flow graph is generated. The basic aspect that we are investigating by leveraging Intel PTs is how fast we can generate these control-flow graphs from the start of the tracing to the actual CFG generation in order to assess how effectively the control flow attestation will work. This is the intuition behind the FutureTPM Control-Flow Property-based Attestation. Where property is a safety-critical-function that needs to be attested, based on the security attestation policies extracted by the FutureTPM RA Engine. The idea is to minimize the code base that we attest and extract its control flow graph. What affects the execution timing of extracting the control flow graph, as will be described in the later sections, is **dependent on the order of complexity of the function of the property that we want to be able to trace and extract its control flow graph**. One important aspect that we are evaluating here is how Intel PT behaves under such different orders of complexity. To the best of our knowledge this is one of the **first comprehensive evaluations that try to investigate the core aspects that affect the execution of the Intel PT**.

As already aforementioned, if the calculated risk (based on the output of the Risk Quantification Engine) is not acceptable or the result of an attestation report has failed (based on the output of the CFPA), then a new configuration and security policy needs to be enforced. More specifically, the configuration policy includes a high-level access control policy or new interpreted low-level execution properties that needs to be attested. In the case there is a need to enforce a new policy, there is also a possibility to update the tracer and deploy new eBPF hooks and Intel PT tracing automatically (Case 1 of Figure 2-3). This chapter focuses on this automatic deployment of such tracing techniques.

This process will work according to an Event-Condition-Action pattern, where deployment events are triggered by the failure of an attestation report or a calculated risk with high impact, being above a threshold, and actions entail the deployment and/or modification of the eBPF execution hooks (monitored data, frequency, granularity, filtering, etc.), the deployment of Intel PT tracing hooks, re-configuration of the control-flow attestation policies, etc. The description of data to be monitored and collected and the automatic deployment of the kernel hooks are expressed by policies, which also represent the “smartness” of the FutureTPM Secure Tracing component and encompasses both reaction and prevention actions as well as defensive policies.

This Secure Tracer instantiates and configures these programmable execution hooks to be deployed to the envisioned ecosystem of devices. For instance, if the focus is mainly against remote memory and data exploitation attacks, the tracer will instantiate the agents for monitoring system calls that try to access the data variable of interest (in the device’s memory) and also are capable of compiling in real-time the necessary Control-Flow and Data-Flow Graphs (CFGs and DFGs, respectively) to be then attested by the CFPA. This controller is also capable of reprogramming the execution environment of existing eBPF hooks, leveraging

various means of abstraction. **The main technical challenge here is the translation of the evidence-related policies (definition of evidence to be collected) into configurations and code for the heterogeneous set of security hooks.** In the current architecture, this is realized by selecting pre-defined programs and configuration files from an internal library, but the long-term ambition would be the definition of dynamic code generation and run-time compiling.

All these control elements and tracing components are part of the Security Policy Enforcement mechanism that is highly interdependent with the FutureTPM RA framework. In the current implementation, eBPF hooks that are instantiated during design-time to capture the requirements of the extracted control-flow attestation policies can be deployed as part of the defence strategies of the OLISTIC-based risk assessment toolkit.

The Security Policy Enforcement strategy [10], [11], [12] implements a policy-driven approach for allowing proofs of a system's integrity based on the attested properties. This architecture specifies how composition of large scale "Systems-of-Systems" is to be controlled via layered and cross-domain attestation decisions. The outputs of the FutureTPM RA framework dictate the control-flow attestation policies required to mitigate the identified risks. Such policies are expressive, deployable, and enforceable within the Security Policy Enforcement architecture and may be dynamically updated if the attack graph (produced and maintained by the RA framework) is amended with new types of vulnerabilities.

4.1 Intel® Processor Tracing Building Blocks

4.1.1 Overview

Intel PT is an architecture extension introduced and implemented by Intel processor microarchitectures from Broadwell and Apollo Lake onward. **Intel PT provides hardware support for tracing code execution with minimal CPU overhead.** It defines a set of model-specific registers (MSRs) that the operating system can use to enable and configure tracing and exposes a stream of packets (see Chapter 4.1.2) containing compressed execution information of the traced binary. This packet stream can be captured and written into a memory buffer for further analysis. The most common use case is recording the Intel PT trace stream to disk for post-mortem decoding [13], though real-time control flow attestation implementation also exist [14]. To supplement the Intel PT trace, the operating system can also provide sideband information, such as linked binaries and context-switching when tracing multiple threads on the same CPU. This information can be used to correlate the Intel PT trace with the linked binaries to reconstruct the traced execution.

Intel PT generates very large amounts of data when tracing, so some filtering is necessary in order to isolate the relevant information. For this reason, Intel PT supports multiple ways of narrowing down the context of a trace. These include filtering by user- or kernel-space, process, and instruction pointer (IP) addresses. Thus, it is possible to narrow the trace context down to a single function, or even part of a function [15].

4.1.2 Packets

The packet stream generated by Intel PT consists of packets of various types, encoding information about the processor's behaviour. In this section, the relevant packet types are listed and briefly explained below. Some packets types and sub-types, including ones denoting software-defined PT events and CPU power state changes, have been omitted for brevity [15].

- **PSB: Packet Stream Boundary** These packets are generated at regular intervals and act as markers for the decoder to synchronize with the packet stream.
- **PIP: Paging Information Packet** During tracing, if the CR3 register is modified, a PIP packet is generated. This allows the decoder to differentiate between processes in a trace.

- **TSC: Time-Stamp Counter** This encodes the lower 7 bytes of the software visible time-stamp counter at the time of generation and can be used to correlate the Intel PT packet stream with other timestamped events.
- **CBR: Core Bus Ratio** At the beginning of a trace, and every time the CPU frequency is changed, a CBR packet is generated. It encodes the CPU's core frequency relative to the system bus frequency.
- **OVF: Overflow** When the processor drops packets due to an internal buffer overflow, it generates an OVF packet to inform the decoder of this loss of data.
- **TNT: Taken Not-Taken** These packets contain information on whether conditional branches were taken during execution, each branch represented as a single bit. They are instrumental in reconstructing the control-flow of a program execution.
- **TIP: Target Instruction Pointer** For various branching events - including (but not limited to) returns, interrupts and exceptions - a TIP packet is generated, containing the target instruction pointer of that branch. Real-Time Execution Tracing with Intel PT 5.
- **FUP: Flow Update Packet** In the case of interrupts, exceptions, and other events for which the source instruction pointer cannot be reliably determined, a FUP packet is generated, containing that address.
- **MODE** These packets encode important information about the current execution context of the trace, including execution mode (16-, 32- or 64-bit) and TSX transaction events.

4.1.3 Intel libipt

Intel provides an open-source reference Intel PT decoder library implementation called libipt. The library includes sample implementations of several utilities: a packet dumper, trace disassembler, trace test generator and several utility scripts for working with recorded traces.

Layers

When it comes to decoding Intel PT traces, libipt offers several different layers of abstraction for interpreting the Intel PT packet stream:

- **Packet** On this layer, the output of the decoder will consist of raw Intel PT packets. This will expose all of the information gathered by Intel PT and thus enables very detailed analysis of the recorded execution.
- **Event** Also referred to as the query layer, the event layer encodes combinations of packets into higher level CPU events for a slightly broader and more abstract overview of CPU activity.
- **Instruction** This layer combines a trace with either sideband information or image sections parts of the executable files the execution of which was recorded to reconstruct the entire recorded execution in terms of CPU instructions. While more abstract than the event and packet layers, the instruction layer provides a comparatively legible view of the recorded execution.
- **Block** The block layer further abstracts the instruction layer into blocks of instructions. These blocks encode continuous regions of executed instructions, making this layer ideal for CFA. It operates faster than the instruction layer, though more post-processing is required in order to analyse individual instructions within blocks.

4.1.4 Linux perf

Intel PT support in the Linux kernel was added in version 4.2 via perf, a set of tools for performance monitoring and execution tracing. This allows Linux user-space applications to gather and inspect Intel PT trace data. More importantly for this research, the underlying kernel module allows a process with sufficient privileges to interact with Intel PT directly via a set of system calls and IOCTL requests. This enables the development of a simple tracing program which records only to system memory [16].

4.2 Technical Building Blocks of the FutureTPM Intel PT Tracing

This section covers the technical foundation on which the solution is built, such as the relevant Intel PT components and key concept definitions.

4.2.1 Intel PT Components

As stated in Section 4.1, the trace output of Intel PT is highly compressed. To achieve this, Intel PT relies on a well-defined set of principles and core design elements intended to minimize its output while still guaranteeing full reproducibility of the traced execution. One core assumption is that all binary code run during the trace is available when decoding, along with information on how it was loaded. Based on this assumption, Intel PT can focus on encoding only the information required to reconstruct the execution flow from the source binaries [15].

Trace Output

Intel PT supports several output schemes, including a contiguous region of physical memory, a table of physical addresses and various platform-specific trace transport subsystems. Whichever scheme is chosen, Intel PT bypasses the CPU cache and writes directly to memory instead, which is a key factor in its low performance overhead.

The output of Intel PT is organized into packets, as introduced in Section 4.1.2. These packets contain various information about the traced execution. Some are more relevant to this endeavour than others, since only a few provide information about control-flow directly. For example, PIP packets are of little interest in our case, since the solution is only meant to trace a single process. TSC and CBR packets are similarly irrelevant for the operation of this solution, since TSC is intended for correlation with timestamped data and we only deal with static binaries, and CBR is primarily an indicator of system performance and does not yield any information about control-flow.

Following is a detailed description of the data encoded in each of the packet types directly relevant to single-process CFA:

- **OVF** Overflow packets are fixed, header-only packets, generated when Intel PT encounters an internal buffer overflow. This indicates packet loss to the decoder. It is immediately followed by either a FUP or a TIP packet, indicating the IP at which tracing resumes after Intel PT recovers. The structure of OVF is shown in Table 3.

Table 3: OVF Packet

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	1	1	1	0	0	1	1

- **PSB** Packet Stream Boundary packets are fixed, header-only packets, designed to be unambiguously recognizable in the trace stream. They are generated at fixed, configurable intervals, and serve as synchronization points for the decoder. The PSB packet is followed by zero or more status-only packets, also referred to as PSB+, which may include TSC, TMA, PIP, VMCS, CBR, MODE and FUP packets. these status-only packets are not ordered and do not signify a change of state or relate to a specific instruction. These can be used by the decoder during synchronization to determine the state of the CPU at the time the PSB was generated. The PSB and PSB+ packets are then followed by either a PSBEND packet, another fixed, header-only packet, or an OVF packet in the case of an overflow during PSB+ generation. The structure of PSB and PSBEND is shown in Table 4 and Table 5, respectively.

Table 4: PSB Packet

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	1	0
2	0	0	0	0	0	0	1	0
3	1	0	0	0	0	0	1	0
[...]								
14	0	0	0	0	0	0	1	0
15	1	0	0	0	0	0	1	0

Table 5: PSBEND Packet

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	1	1

- **TNT** Taken-Not-Taken packets come in two variants: short and long. Each encodes up to a given number of conditional branches or compressed return instructions - 6 for short and 47 for long - as single bits each. The bit values indicate whether the corresponding branches were taken. Table 6 and Table 7 show the structure of these packets. Bits in grey comprise the packet header and the rest are the payload.

Table 6: Short TNT Packet

	7	6	5	4	3	2	1	0
0	0	0	1	B_1	B_2	B_3	B_4	0

Table 7: Long TNT Packet

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	1	0	0	0	1	1
2	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}	B_{29}	B_{30}	B_{31}
3	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}
4	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}
5	1	B_1	B_2	B_3	B_4	B_5	B_6	B_7
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Due to the complex rules governing the generation and ordering of different packet types, TNT packets are often generated before a TNT buffer can be filled during execution tracing. The bits following the header denote the previous N conditional branches or compressed return instructions on the form Bn = {0, 1}, BN being the youngest branch and B1 being the oldest, followed by a 1-bit. The trailing 1-bit denotes the end of the TNT packet.

- **TIP** Target IP packets are generated when control-flow transfers to an IP which is not obtainable from the source binary. It encodes the target IP of the branch, the size of which varies based on the mode of execution and IP compression. IPs may be compressed by comparing their higher-order bytes to those of the previous IP and encoding only the differing lower-order bytes of the address to save bandwidth. The structure of TIP is shown in Table 8.

Table 8: TIP Packet

	7	6	5	4	3	2	1	0
0	IPBytes			0	1	1	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

Two additional variants of TIP exist. They are TIP.PGE and TIP.PGD. These are identical to regular TIP in every way apart from the header bits. PGE stands for Packet Generation Enabled, and TIP.PGE encodes the target IP of an instruction which brings execution inside the trace context, enabling packet generation. Conversely, PGD stands for Packet Generation Disabled, and TIP.PGD encodes the target IP of an instruction which causes execution to leave the trace context, disabling packet generation. These variants are shown in Table 9 and Table 10.

Table 9: TIP.PGE Packet

	7	6	5	4	3	2	1	0
0	IPBytes			1	0	0	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

Table 10: TIP.PGD Packet

	7	6	5	4	3	2	1	0
0	IPBytes			0	0	0	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

The IPBytes field indicates how the target IP is encoded. Values of 110 and 011 indicate an uncompressed IP. A value of 000 indicates that the IP was suppressed. This may appear in a TIP.PGD packet, when execution leaves the trace context via an indirect branch, such as a system call instruction. The values 100, 010 and 001 signify different levels of compression. 100 indicates the six lower-order bytes from the target IP, 010 indicates four target IP bytes and 001 indicates two.

- **FUP** Flow update packets can arguably be called another variant of TIP, being distinguishable from TIP only by their header, but it can be useful to distinguish them conceptually from the TIP variants. FUP encodes a source IP for some event or instruction and is always generated as part of some larger context. They can be preceded immediately by a MODE.TSX packet, in which case FUP indicates the IP to which the

MODE.TSX applies. Otherwise, they are generated at the beginning of some compound event, followed by a series of other packets pertaining to the encoded IP, and finally a TIP or TIP.PGD packet. Such a sequence indicates a branch from the FUP-encoded IP to the TIP-encoded IP. The structure of FUP is shown in Table 11.

Table 11: FUP Packet

	7	6	5	4	3	2	1	0
0	IPBytes			1	1	1	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

- **MODE** Mode packages indicate various information about the operating mode of the CPU during tracing. They come in two variations: MODE.Exec and MODE.TSX. MODE.Exec encodes the execution mode of the CPU (16-, 32-, or 64-bit), which is essential information for the decoder. They always precede TIP or TIP.PGE packets and are generated when the execution mode changes or when tracing resumes. MODE.TSX packets are specific to Intel's Transactional Synchronization Extensions and indicate changes of TSX state during execution. The structure of these packets is shown in Table 12 and Table 13.

Table 12: MODE.Exec Packet

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	0	0	0	0	0	0	Mode	

Table 13: MODE.TSX Packet

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	0	0	1	0	0	0	Mode	

The mode field has different meanings for each variant. In both variants, the value 11 is undefined. In MODE.Exec packets, each combination of bits signifies a different mode of execution: 01 indicates 64-bit mode, 10 indicates 32-bit mode and 00 indicates 16-bit mode. The same applies to MODE.TSX: 01 indicates execution within a transaction or transaction start, 10 indicates an aborted transaction and 00 indicates execution outside a transaction or transaction complete.

Change of Flow Instructions

Intel PT makes use of the concept of basic code blocks, contiguous regions of code executed in order. These need not be traced. Certain instructions and events may change the control-flow of a program. Collectively, these are known as Change of Flow Instructions (COFI) and form the main focus of attention for Intel PT's trace output. This term encompasses branch instructions and events such as exceptions and interrupts. Different COFI require different handling when generating the trace output. The three basic categories of COFI govern how they are handled when encountered.

- Direct transfer COFI These include conditional branches and unconditional direct jumps. The relative target addresses of direct transfers are embedded in the instructions themselves and can therefore be obtained directly from the source binary. For conditional branches, Intel PT only encodes a single bit in a TNT packet, indicating whether the branch was taken. Unconditional direct jumps are not encoded in the trace unless they cause a transfer into - or out of the trace IP filter region.

- Indirect transfer COFI These include indirect jumps, calls, and returns, branches which resolve their target addresses at runtime from memory or a register. Since the target address cannot be obtained from the source binary, Intel PT encodes them in TIP packets. In most cases, the target IP of a return instruction can be derived from the call stack and source binary. In those cases, when a corresponding call instruction has been recorded since the previous PSB packet, return instructions are compressed to a single 1-bit in a TNT packet.
- Far transfer COFI These include exceptions, interrupts, traps, TSX aborts, system calls and other instructions which are not covered by the other categories. Intel PT generates a TIP packet for each far transfer encountered. Some far transfers are asynchronous events, and thus their source IPs cannot be accurately derived from the source binary. For those events, Intel PT generates FUP packets containing the IP at which the event occurred.

Trace Filtering

Intel PT supports three different methods of trace filtering. These filtering mechanisms control the circumstances under which tracing is enabled. This is extremely useful as it can be used to limit the scope of the trace. This can greatly reduce the performance overhead of the tracing itself and, more importantly, the decoding. The following is a short description of each filtering capability and how they are utilized in this solution.

- **CPL filtering** Current Privilege Level (CPL) filtering allows IPL to enable and disable tracing based on the privilege level of the traced execution. This filter has three settings: CP L > 0, CP L = 0 and regardless of CPL. In practice, this means that we can configure Intel PT to trace only user-space code (CP L > 0), kernel-space code (CP L = 0) or all code. For the purposes of this research, the focus shall be on user-space code (CPL filter set to CP L > 0).
- **CR3 filtering** Control Register 3 (CR3) points to the page tables for the currently executing process. It also contains a process-context identifier, which can be used to uniquely identify a running process. Filtering by CR3 value therefore allows Intel PT to enable and disable tracing based on which process is currently being executed. Effectively, this means that we can limit the trace to a single process.
- **IP filtering** As mentioned in Chapter 4.1.1, Intel PT also allows us to filter by the currently executing instruction pointer. This allows very fine-grained control over the scope of the trace, down to the instruction level. It enables the isolation of a single function, or even part of a function for tracing. This is particularly useful for this solution, since it enables greater control over the control-flow complexity of the traced execution.

4.3 Architecture of the FutureTPM Intel PT Tracing Solution

Intel Processor Trace (Intel-PT) is an extension of the Intel architecture that collects information about software execution such as control flow, execution modes and timings, and formats it into highly compressed binary packets. It is a relatively new kernel-based subsystem that provides, among other functionalities, a framework for hardware level analysis and is available to the 7th generation of Intel processors. In the context of FutureTPM tracing, Intel-PT is used in conjunction with the perf Linux analysis tool [17] for better tracing capabilities. More precisely, the tracing procedure is based on Intel-PT to collect the tracing packets and the “perf record,” “perf report” and “perf script” commands to collect additional information. Since the “perf record” command is very generic, filtering is required to remove unnecessary information. After the creation of the trace output (perf.data), further processing to transform the data in a human readable form is necessary by utilizing the “perf report” command. In addition, by combining the “perf script” and the XED x86 Encoder/Decoder, the traces can be disassembled. This disassembled output will be used to generate the control flow graph (CFG). Table 14, summarizes the used tracing perf commands for the hardware intel_pt Performance Monitor Unit (PMU) events. PMU hardware events are CPU specific and documented by the CPU

vendor (e.g., Intel). The overall flow of Intel PT Tracing as described previously is depicted in the following Figure 4-1.

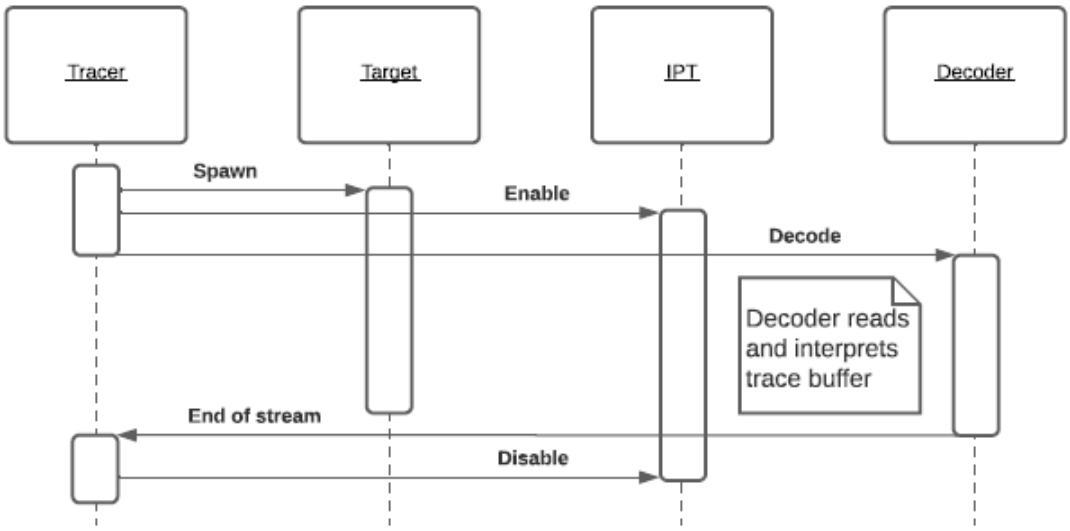

Figure 4-1: FutureTPM Intel PT Tracing Workflow

Overall, the design of the process is as follows:

1. The tracer executes the target program in a new process;
2. The tracer enables Intel PT, filtering down on the target process;
3. Intel PT starts populating the trace buffer with packets;
4. The tracer starts the decoder;
5. The decoder starts reading from the trace buffer;
6. The decoder starts reconstructing the target's control-flow;
7. The target process finishes;
8. The decoder hits the end of the trace stream;
9. The tracer disables Intel PT and exits.

While the tracer and decoder processes belong to the same program, it is useful to manage them as separate entities. The tracer is the component responsible for the setup and configuration of other agents while the decoder is responsible for reading and decoding the Intel PT packet stream.

Table 14: Intel PT tracer - Perf commands

Perf commands syntax	Description
perf record -e intel_pt//u -b --filter 'filter main @ example' --pid=18066	Captures the Intel-PT PMU hardware events and saves them to the perf.data file
perf report -D > raw_data	Translates the traces stored on perf.data to human readable form (ASCII) and stores them to raw_data
perf script --ns --itrace=i1t -F +insn | /path/to/xed/obj/wkit/examples/obj/xed -F insn: -I -64 > decoded_insn.dump	Disassembles the events that Intel-PT captured and stored on raw_data to hexadecimal commands using the XED x86 Encoder Decoder

Figure 4-2 and Figure 4-3, depict the content of the perf report command output (human readable traces). In the context of FutureTPM tracing, the main focus is on the MTC, TNT and TIP.PGE packets. MTC packets are timing packets necessary for time reference of the trace event. A TNT packet contains information of whether a conditional branch was taken or not, with a byte of value "T" in the first case and "N" otherwise. The TIP.PGE packet is necessary to locate the exact branch that each packet refers to, as it discloses the target (as a memory address, e.g.: 0x8755) for the conditional branch we are examining each time. Moreover, in order to exploit the return addresses that TIP.PGE packets provide, it is necessary to identify

the conditional branch that the address refers to, based on the gcc compiler. To do so, we know that the "if" instruction set translated by the gcc compiler for x86_64 systems is **mov dword ptr [rbp-0xfc4], eax**. Given this statement, we are in position to identify, for every TNT packet, whether the condition of the statement was fulfilled, and which execution path was followed. An example is given in Figure 4-3 where we visualize the output of tracing a simple application which invokes two internal methods.

```
.
. ... Intel Processor Trace data: size 12784 bytes
.  00000000:  02 82 02 82 02 82 02 82 02 82 02 82 02 82 02 82 PSB
.  00000010:  00 00 00 00 00 00                               PAD
.  00000016:  19 67 60 0a 08 23 12 00                         TSC 0x1223080a6067
.  0000001e:  00 00 00 00 00 00 00 00                         PAD
.  00000026:  02 73 7b 88 00 30 00 00                         TMA CTC 0x887b FC 0x30
.  0000002e:  00 00                                           PAD
.  00000030:  02 03 24 00                                     CBR 0x24
.  00000034:  02 23                                           PSBEND
.  00000036:  59 10                                           MTC 0x10
.  00000038:  59 11                                           MTC 0x11
.  0000003a:  59 12                                           MTC 0x12
.  0000003c:  59 13 00 00 00 00 00 00                         MTC 0x13
.  00000044:  00 00                                           PAD
.  00000046:  19 d3 7b 0a 08 23 12 00                         TSC 0x1223080a7bd3
.  0000004e:  00 00 00 00 00 00 00 00                         PAD
.  00000056:  02 73 b7 88 00 30 00 00                         TMA CTC 0x88b7 FC 0x30
```

Figure 4-2: Intel-PT traces translated to human readable form with perf report command (a)

```
.  00000640:  59 8f                                           MTC 0x8f
.  00000642:  31 e0 86                                        TIP.PGE 0x86e0
.  00000645:  01 00 00                                        TIP.PGD no ip
.  00000648:  31 fd 86                                        TIP.PGE 0x86fd
.  0000064b:  04                                              TNT N (1)
.  0000064c:  01                                              TIP.PGD no ip
.  0000064d:  31 2c 87                                        TIP.PGE 0x872c
.  00000650:  04                                              TNT N (1)
.  00000651:  01                                              TIP.PGD no ip
.  00000652:  31 55 87                                        TIP.PGE 0x8755
.  00000655:  04                                              TNT N (1)
.  00000656:  01 00                                           TIP.PGD no ip
.  00000658:  31 2c 87                                        TIP.PGE 0x872c
.  0000065b:  04                                              TNT N (1)
.  0000065c:  01                                              TIP.PGD no ip
.  0000065d:  31 55 87                                        TIP.PGE 0x8755
.  00000660:  04                                              TNT N (1)
.  00000661:  01                                              TIP.PGD no ip
.  00000662:  31 2c 87                                        TIP.PGE 0x872c
.  00000665:  04                                              TNT N (1)
.  00000666:  01 00                                           TIP.PGD no ip
.  00000668:  31 55 87                                        TIP.PGE 0x8755
```

Figure 4-3: Intel-PT traces translated to human readable form with perf report command (b)

CFG Generation from Intel PT traces: The disassembled traces will be used to generate the CFG. To this end, we need to identify the "leaders", i.e., instructions that start a basic block (e.g., the first instruction, the target of a conditional or unconditional goto/jump and the instruction that immediately follows a condition of unconditional goto/jump). Finally, the nodes and the edges of the CFG can be defined based on the identified "leaders" and the memory addresses of the conditional branches.

```
(base) pstavrianos@ubiadmin-GP62M-7RDX:~/Desktop/cfa_git/Control-flow-attestation$ ./tree_creation_final.py
0x562980518698 0x56298051866a
0x5629805186b6 0x56298051869a
0x562980518711 0x5629805186b8
0x56298051873c 0x562980518713
0x562980518755 0x56298051873e
0x562980518775 0x562980518757
0x562980518791 0x562980518777
0 [1] [2, 1]
1 [0] [0]
2 [0, 4, 5] [6, 3]
3 [2] [5, 4]
4 [3] [2]
5 [3] [2]
6 [2] []
```

Figure 4-4: CFG Generation from Intel PT traces

Given the output of the "leaders", it is easy to then generate the control flow graph of the binary by disassembling the binary code in the addresses – as depicted in Figure 5-34 - and creating a graph that showcases the flow of the branching commands. This graph is sufficient to understand whether the binary has followed an acceptable and verifiable path or another unpredictable and possibly malicious or malfunctioning execution branch. To present the capabilities of our tracer we created a set of case studies of increasing control flow graph complexity and traced their execution with the resulting graphs presented in Chapter 5. Additionally, we also applied our tracer on a deterministic (static control flow graph) binary and on a non-deterministic (non-static control flow graph) binary so as to show the different results based on the different types of input which our tracer is able to catch. The final case study follows a vulnerable application with a buffer overflow attack available and we show how our tracer will present a binary under attack and what control flow graph will be generated.

4.4 Notable Design Choices

This section aims to detail and justify certain design choices that were made in the development of this FutureTPM Secure Tracer component.

4.4.1 Intel PT Setup and Compatibility

Trace Buffer Type: When allocating the trace output buffer in memory, Intel PT provides two options: **a ring buffer and a linear buffer.** These are distinguished during allocation by the protection flags selected. A read-only buffer is treated as a ring buffer, meaning that when the buffer is full, Intel PT will keep writing from the beginning of the buffer. A read-write buffer, however, is treated as a linear buffer. When a linear buffer is full, all subsequent packets are dropped. The ring buffer can be very useful as it effectively allows Intel PT to keep tracing indefinitely. However special considerations need to be made when implementing a decoder operating on a ring buffer to ensure that it keeps reading from the start of the ring buffer, like Intel PT does, while tracing. Though this functionality was deemed unnecessary for the achievement of the test plan, the ring buffer was used, only without the necessary decoder customizations required to make full use of it.

Trace Buffer Size: The size of the trace buffer is a very important factor and depends heavily on the amount of tracing data we expect to generate. Initially, when assessing the efficacy of the solution, the buffer size was set to 4KB, the lowest possible value, which proved to be more than necessary for accommodating all of the efficacy test cases (see Chapter 5.2). The performance test cases, however, presented a more difficult decision. It was decided to leave this up to trial and error during performance testing.

Some compatibility limitations became apparent during the design phase and early development of this solution. The decision was made, in accordance with the stated design principles (see Section 4.3 and 4.4), to address such when necessary while executing the test plan as laid out in Chapter 5.1. 64-bit support Both Intel PT and libipt have built-in support for 32-bit binaries. However, when customizing the binary loading functionality of libipt, several issues with address extraction arose when attempting to port the changes made to the 64-bit ELF loader to the 32-bit one. Since all tests would be run on 64-bit binaries on a 64-bit processor, these efforts were quickly abandoned as unnecessary.

Chapter 5 FutureTPM Intel PT Tracing Evaluation

5.1 Evaluation Plan

This chapter details an evaluation of the effectiveness and performance of the FutureTPM Secure tracer component. Test cases are written as simple C programs to provide a minimal codebase for covering each test's requirements. 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. The performance was measured against two scaling profiles: **high-complexity and low-complexity.** *Complexity refers here to the target test cases' branch densities.* The maximum complexity, in this context, would be a program consisting exclusively of branching instructions while the minimum would be one containing no branching instructions aside from the final return.

Each performance metric will relate to two scaling factors: the number of executed instructions in the target program and a secondary scaling factor which will reduce the effective complexity of the trace in different ways for the high-complexity and low-complexity tests. This secondary scaling factor shall reduce the number of instructions executed outside the trace context in the case of high-complexity, leaving a larger portion of the target's runtime untraced, and the number of non-branching instructions in the case of low-complexity codebase, effectively lowering the target's branch density.

Seven test cases were written - numbered t_0 to t_6 - to test the decoder's ability to identify and indicate different control-flow structures. The following section provides an overview of these scenarios, along with a discussion on the extracted key performance indicators.

5.1.1 Test Case t_0 – Control

This test case represents the minimum control-flow complexity within the main execution function of a service. It simply enters and returns with no branching in between the start and termination program commands. Figure 5-1 shows the source code of the target program.

```c
/**
 * Case 0
 *
 * Linear flow
 */
int main (int argc, char* argv [])
{
  return 0;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
mov    %edi,-0x4(%rbp)
mov    %rsi,-0x10(%rbp)
mov    $0x0,%eax
pop    %rbp
retq
```

Disassembly of main function

Figure 5-1: Test case t_0 target program

The CFG, as shown in Figure 5-2, is the simplest possible non-empty CFG: A single block of code with uniquely defined entry and exit points.

Figure 5-2: Test case t_0 control-flow graph

As shown in Figure 5-3, there are only two control-flow events to report here:

1. Execution enters the trace context at the start of main.
2. Execution leaves the trace context via a return statement.

```
$ bin/tracer 1 "$(pwd)/bin/t0" main
>ENTER @ 0x55856f7b0129                    # int main(...)
<RET   @ 0x55856f7b013e                    # return
```

Figure 5-3: Test case t_0 trace output

The output is exactly as expected. The > symbol indicates that execution has entered the trace context, and the following ENTER is a placeholder for the branch instruction class, since previous execution was not traced, and the decoder has no way of analysing the preceding branch instruction. The address following the @ symbol is the virtual runtime address of the main function. It consists of a base address at which the binary is loaded, determined at runtime by address space layout randomization (ASLR) and the main function symbol offset. The < symbol in the second event denotes a context exit and RET signifies the return instruction, followed by the instruction pointer.

5.1.2 Test Case t_1 - Single Conditional Branch

Test case t_1 has a very simple branching structure: a single if-statement. The condition is set such that it always evaluates to “false” and the conditional branch skipping the contents of the statement is taken. Figure 5-4 shows the source code of the target program.

```
/**
 * Case 1
 *
 * Single conditional jump
 */
int main (int argc, char* argv [])
{
    int a = 0;

    if (a > 0)
    {
        a++;
    }

    return a;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
mov    %edi,-0x14(%rbp)
mov    %rsi,-0x20(%rbp)
movl   $0x0,-0x4(%rbp)
cmpl   $0x0,-0x4(%rbp)
jle    1149 <main+0x20>
addl   $0x1,-0x4(%rbp)
mov    -0x4(%rbp),%eax
pop    %rbp
retq
```

Disassembly of main function

Figure 5-4: Test case t_1 target program

The CFG, as shown in Figure 5-5, illustrates the single conditional branch in this program. Note that the condition evaluated is the logical inverse of the if-condition in the source code. If the inverse condition holds, we jump past the code within the if-statement.

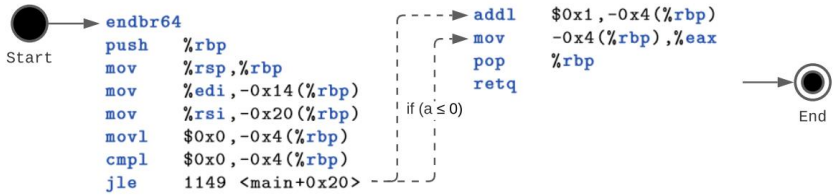

Figure 5-5: Test case t_1 control-flow graph

In addition to the output of t_0, this case also produces a branch within the trace context, as shown in Figure 5-6. Note that the instantiation of the variable a is not registered, as it produces linear control-flow and is, thus, ignored in this context.

```
$ bin/tracer 1 "$(pwd)/bin/t1" main
>ENTER @ 0x55e07f81c129                          # int main(...)
|JUMP   @ 0x55e07f81c143 target: 0x55e07f81c149  # if (a > 0)
<RET    @ 0x55e07f81c14d                         # return
```

Figure 5-6: Test case t_1 trace output

The focus here is on the second line of the output. The | symbol means that the branch occurs completely within the trace context - i.e., both the branching instruction and its target are located within the bounds of the context - and JUMP indicates a jump-class instruction. Further details could be displayed here, as many different types of jump instructions exist, but differentiation among instruction classes has been kept at a high level for simplicity.

5.1.3 Test Case t_2 – Loop

This test case adds a loop in the underlying codebase. **It is a very simple while-statement that runs twice and contains no additional branching structures.** It is important to note here that the loop starts with an unconditional jump instruction to the set of instructions that evaluate the loop condition and jump to the top of the loop's contents based on that condition. The initial, unconditional jump is ignored by the decoder as it is predictable with static analysis and is therefore unnecessary to profile the runtime control-flow of the program. Figure 5-7 shows the source code of the target program.

```
/**
 * Case 2
 *
 * Loop x2
 */
int main (int argc, char* argv [])
{
    int a = 0;

    while (a < 2)
    {
        a++;
    }

    return a;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
mov    %edi,-0x14(%rbp)
mov    %rsi,-0x20(%rbp)
movl   $0x0,-0x4(%rbp)
jmp    1145 <main+0x1c>
addl   $0x1,-0x4(%rbp)
cmpl   $0x1,-0x4(%rbp)
jle    1141 <main+0x18>
mov    -0x4(%rbp),%eax
pop    %rbp
retq
```

Disassembly of main function

Figure 5-7: Test case t_2 target program

The CFG, as shown in Figure 5-8, shows the basic control-flow structure of a while-loop: an unconditional jump followed by the contents of the loop, the evaluation of the while condition and a conditional jump. The unconditional jump branches to the comparison instruction, after which the conditional branch evaluates the while-condition, branching to the top of the loop's contents on a true value.

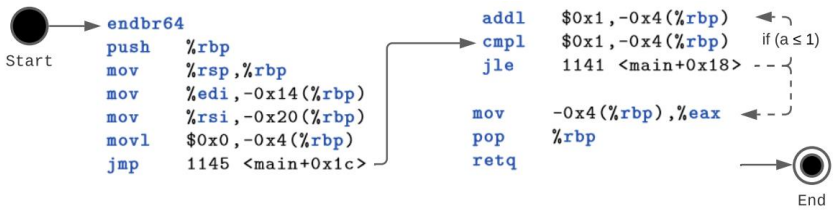

Figure 5-8: Test case t_2 control-flow graph

The loop can be seen in Figure 5-9 as two identical jump events. In the first case, a = 0, so the condition a < 2 evaluates to true and the conditional jump branches to the top of the loop. The second time, a = 1, and the loop repeats. The third time, however, the condition a < 2 evaluates to "false" and the jump is not taken.

```
$ bin/tracer 1 "$(pwd)/bin/t2" main
>ENTER @ 0x55c0a9a8a129                              # int main(...)
|JUMP  @ 0x55c0a9a8a149 target: 0x55c0a9a8a141      # while (a < 2)[0]
|JUMP  @ 0x55c0a9a8a149 target: 0x55c0a9a8a141      # while (a < 2)[1]
<RET   @ 0x55c0a9a8a14f                              # return
```

Figure 5-9: Test case t_2 trace output

Conditional branches not taken are not displayed, for the same reason as unconditional branches within the context: they do not add any information to the runtime control-flow profile that could not be gained via static analysis. Therefore, the two jump events displayed each represent a conditional jump taken to the front of the loop.

5.1.4 Test Case t_3 - Function Call

Test case t_3 adds a function call to the control-flow structure of the program. It is a simple integer function that always returns 0. Note that the added function lies outside the tracing context. Figure 5-10 shows the source code of the target program.

```
/**
 * Case 3
 *
 * Call+ret
 */
int func()
{
    return 0;
}

int main (int argc, char* argv [])
{
    return func();
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    %edi,-0x4(%rbp)
mov    %rsi,-0x10(%rbp)
mov    $0x0,%eax
callq  1129 <func>
leaveq
retq
```

Disassembly of main function

Figure 5-10: Test case t_3 target program

The CFG, as shown in Figure 5-11, is very simple. There are no conditional branches, just a single function call. When finished, the function returns to the instruction immediately following

the call to it. Note, however that the function is abstracted out of this diagram, because the function lies outside the trace context and is as a black box by the tracer.

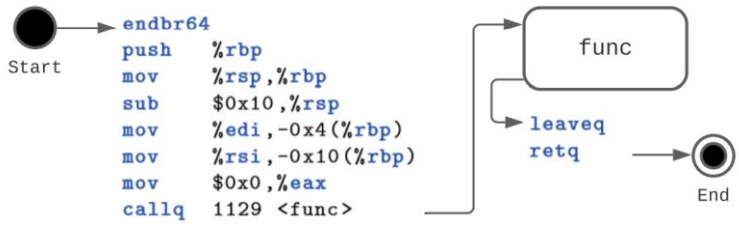

Figure 5-11: Test case t_3 control-flow graph

This illustrates a call outside the trace context, which produces an interesting result. Now there are effectively two entry - and exit points in the main function as shown in Figure 5-12.

```
$ bin/tracer 1 "$(pwd)/bin/t3" main
>ENTER @ 0x5579a2b9f138                          # int main(...)
<CALL  @ 0x5579a2b9f150 target: 0x5579a2b9f129   # func()
>ENTER @ 0x5579a2b9f155                          # return from func()
<RET   @ 0x5579a2b9f156                          # return
```

Figure 5-12: Test case t_3 trace output

As before, > and < denote entry into - and exit from the trace context, respectively. The second and third lines of output (CALL and ENTER) denote the function call and return to the context of the main function, immediately followed by the return out of the main function.

5.1.5 Test Case t_4 - Conditional Branch Within Loop

Now that all the basic branch classes have been shown to display as expected, it is time to see what happens when they are combined. **This test case combines t_1 and t_2 with a single if-statement inside a loop.** Figure 5-13 shows the source code of the target program.

```
/**
 * Case 4
 *
 * Conditional jump inside loop
 */
int main (int argc, char* argv [])
{
    int a = 0;

    while (a < 2)
    {
        a++;

        if (a > 2)
        {
            a++;
        }
    }

    return a;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
mov    %edi,-0x14(%rbp)
mov    %rsi,-0x20(%rbp)
movl   $0x0,-0x4(%rbp)
jmp    114f <main+0x26>
addl   $0x1,-0x4(%rbp)
cmpl   $0x2,-0x4(%rbp)
jle    114f <main+0x26>
addl   $0x1,-0x4(%rbp)
cmpl   $0x1,-0x4(%rbp)
jle    1141 <main+0x18>
mov    -0x4(%rbp),%eax
pop    %rbp
retq
```

Disassembly of main function

Figure 5-13: Test case t_4 target program

The CFG, as shown in Figure 5-14, shows neatly how t_1 and t_2 are combined in terms of control-flow. There are two conditional jumps following comparison instructions. The if statement from t_1 can be seen nested within the while-loop from t_2.

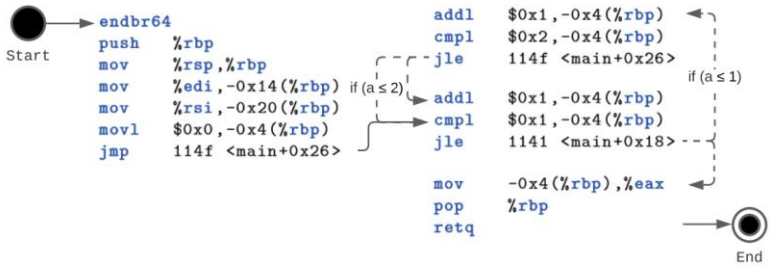

Figure 5-14: Test case t_4 control-flow graph

The output shown in Figure 5-15 clearly indicates the combination of these control-flow structures.

```
$ bin/tracer 1 "$(pwd)/bin/t4" main
>ENTER @ 0x55bb47919129                              # int main(...)
|JUMP  @ 0x55bb47919153 target: 0x55bb47919141      # while (a < 2)[0]
|JUMP  @ 0x55bb47919149 target: 0x55bb4791914f      # if (a > 2)
|JUMP  @ 0x55bb47919153 target: 0x55bb47919141      # while (a < 2)[1]
|JUMP  @ 0x55bb47919149 target: 0x55bb4791914f      # if (a > 2)
<RET   @ 0x55bb47919159                              # return
```

Figure 5-15: Test case t_4 trace output

For each of the two iterations of the loop, two jump events are detected: one for the loop itself and another for the inner conditional jump. These can be differentiated based on their instruction - and target addresses.

5.1.6 Test Case t_5 - Function Call Within Loop

This test case combines t_2 and t_3 with a function call inside a loop. This should cause execution to repeatedly leave and re-enter the trace context. Figure 5-16 shows the source code of the target program.

```
/**
 * Case 5
 *
 * Call inside loop
 */
int func()
{
    return 0;
}

int main (int argc, char* argv [])
{
    int a = 0;

    while (a < 2)
    {
        a++;

        func();
    }

    return a;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %edi,-0x14(%rbp)
mov    %rsi,-0x20(%rbp)
movl   $0x0,-0x4(%rbp)
jmp    1162 <main+0x2a>
addl   $0x1,-0x4(%rbp)
mov    $0x0,%eax
callq  1129 <func>
cmpl   $0x1,-0x4(%rbp)
jle    1154 <main+0x1c>
mov    -0x4(%rbp),%eax
leaveq
retq
```

Disassembly of main function

Figure 5-16: Test case t_5 target program

The generated CFG is as shown in Figure 5-17.

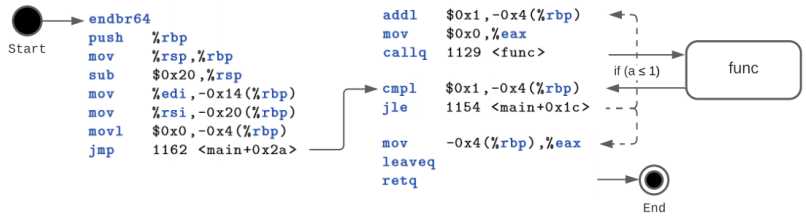

Figure 5-17: Test case t_5 control-flow graph

As shown in Figure 5-18, the output shows the expected combination of the control-flow structures displayed in t2 and t3.

```
$ bin/tracer 1 "$(pwd)/bin/t5" main
>ENTER @ 0x55c302c0d138                          # int main(...)
|JUMP  @ 0x55c302c0d166 target: 0x55c302c0d154   # while (a < 2)[0]
<CALL  @ 0x55c302c0d15d target: 0x55c302c0d129   # func()
>ENTER @ 0x55c302c0d162                          # return from func()
|JUMP  @ 0x55c302c0d166 target: 0x55c302c0d154   # while (a < 2)[1]
<CALL  @ 0x55c302c0d15d target: 0x55c302c0d129   # func()
>ENTER @ 0x55c302c0d162                          # return from func()
<RET   @ 0x55c302c0d16c                          # return
```

Figure 5-18: Test case t_5 trace output

For each iteration of the loop, the decoder outputs a jump to the top of the loop contents, followed by a context exit via CALL and re-entry following a return from the external function.

5.1.7 Test Case t_6 - Function Call and Conditional Branch Within Loop

Now, the only thing left to test is the combination of all of these branch types. **This test case does just that, combining t_4 and t_5 to form a simple program with a loop containing a conditional branching instruction and a function call.** Figure 5-19 shows the source code of the target program.

```
/**
 * Case 6
 *
 * conditional jump and call inside
     loop
 */
int func()
{
    return 0;
}

int main (int argc, char* argv [])
{
    int a = 0, b = 0;

    while (a < 2)
    {
        a++;

        if (a > 2)
        {
            a++;
        }

        func();
    }

    return a;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %edi,-0x14(%rbp)
mov    %rsi,-0x20(%rbp)
movl   $0x0,-0x8(%rbp)
movl   $0x0,-0x4(%rbp)
jmp    1173 <main+0x3b>
addl   $0x1,-0x8(%rbp)
cmpl   $0x2,-0x8(%rbp)
jle    1169 <main+0x31>
addl   $0x1,-0x8(%rbp)
mov    $0x0,%eax
callq  1129 <func>
cmpl   $0x1,-0x8(%rbp)
jle    115b <main+0x23>
mov    -0x8(%rbp),%eax
leaveq
retq
```

Disassembly of main function

Figure 5-19: Test case t_6 target program

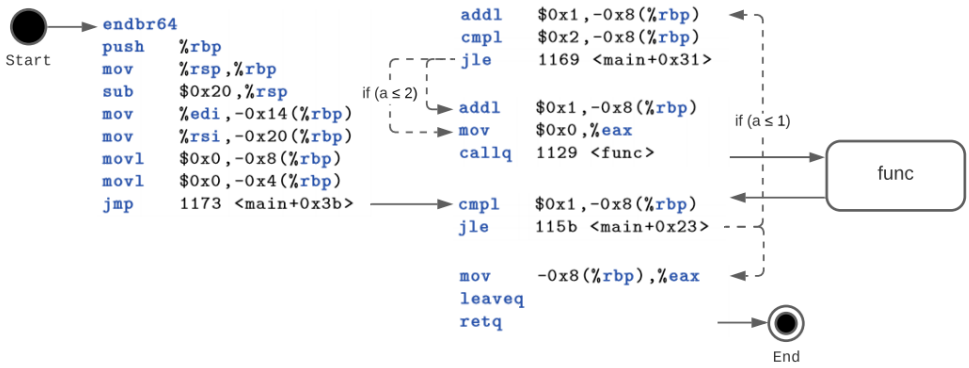

Figure 5-20: Test case t_6 control-flow graph

Again, the output - as shown in Figure 5-21 - displays the control-flow of the program as a combination of its constituent structures.

```
$ 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
```

Figure 5-21: Test case t_6 trace output

Each iteration of the loop is displayed as before, with the addition of both the single conditional branch and the function call and subsequent context re-entry.

5.2 Performance Evaluation

This chapter presents the results of the performance tests evaluating the additional overhead of each test case. For both high- and low-complexity testing, the secondary scaling factor has been set at five exponential increments: 1, 10, 100, 1000 and 10000. This will give a clear picture of the overall effect the tracing process has on the scaling characteristics of this approach. The primary scaling factor will range from 1000 to 1000000 in increments of 1000. This strikes a balance between fine-grained test results and speed of testing.

5.2.1 Memory Usage

The primary factor in the solution's memory requirements is the size of the trace buffer. The trace buffer must be a power of two 4KB pages long, and the setup time for the tracer increases drastically as the buffer size is increased, disproportionately affecting the performance overhead of the lower scale factor test cases. The required buffer size was determined by trial-and-error across the range of scaling factors tested. Table 15 shows threshold values for the primary scaling factor coupled with the required buffer size.

Table 15: Trace buffer size model

Threshold	Pages	Buffer size
1000	2	8KB
2000	4	16KB
3000	8	32KB
6000	16	64KB
12000	32	128KB
24000	64	256KB
48000	128	512KB
96000	256	1MB
190000	512	2MB
380000	1024	4MB
760000	2048	8MB

5.2.2 High Complexity Scenario

The target program t_7 is designed to facilitate the two scaling factors selected. **It contains two 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 program expects two command line arguments denoting the number of iterations for the main loop and the external loop, respectively. These are the primary and secondary scaling factors. The source code of the target program t_7 is shown in Figure 5-22.

```c
/**
 * Case 7
 *
 * Scalability test:
 * Variable loops inside and
     outside context
 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int func(int loops)
{
    int a = 0;

    for (int i = 0; i < loops; i++)
    {
        a++;
    }

    return a;
}

int main (int argc, char* argv [])
{
    int a = 0;
    int scale = atoi(argv[1]);
    int extScale = atoi(argv[2]);

    for (int i = 0; i < scale; i++)
    {
        func(extScale);
    }

    return 0;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %edi,-0x14(%rbp)
mov    %rsi,-0x20(%rbp)
movl   $0x0,-0xc(%rbp)
mov    -0x20(%rbp),%rax
add    $0x8,%rax
mov    (%rax),%rax
mov    %rax,%rdi
callq  1050 <atoi@plt>
mov    %eax,-0x8(%rbp)
mov    -0x20(%rbp),%rax
add    $0x10,%rax
mov    (%rax),%rax
mov    %rax,%rdi
callq  1050 <atoi@plt>
mov    %eax,-0x4(%rbp)
movl   $0x0,-0x10(%rbp)
jmp    11d6 <main+0x5d>
mov    -0x4(%rbp),%eax
mov    %eax,%edi
callq  1149 <foo>
addl   $0x1,-0x10(%rbp)
mov    -0x10(%rbp),%eax
cmp    -0x8(%rbp),%eax
jl     11c8 <main+0x4f>
mov    $0x0,%eax
leaveq
retq
```

Disassembly of main function

Figure 5-22: Test case t7 target program

Figure 5-23 to Figure 5-27 show graphs of the execution times of both the standalone target program and the tracing process across the range of primary scaling values.

1 iteration out-of-context: This test case is close to the worst-case-scenario: maximum control-flow complexity. Each iteration of the main loop consists of a jump and a call which returns after a single iteration of the external loop, making for an extremely branch-dense

execution. A graph of the results is shown in Figure 5-23. These results show a deviation from 0.2 to 4.14 milliseconds for the target program, and 5.76 to 5397.76 milliseconds for the tracer. The performance overhead, as expected, is rather ranging from 26.42x to 2184.38x.

10 iterations out-of-context: This test case showcases the effect of increasing the time it takes for the out-of-context function to return on the performance overhead. As depicted in Figure 5-24, the results show a promising trend when we look at the raw data. The target program timings range from 0.22 to 15.8 milliseconds, while the tracing times remain relatively static at 5.97 to 5480.87. This is to be expected, since as the runtime increases with the secondary scaling factor, the amount of trace data to decode does not. The performance overhead is still very large, but now an order of magnitude smaller, at 26.12x to 409.13x.

100 iterations out-of-context: This test case continues the same trend, but this time the difference in the standalone target execution times is more pronounced, as can be seen in Figure 5-25. Once again, the difference can be better appreciated by looking at the raw data. Target program timings range from 0.34 to 148.07 milliseconds, while the tracing times again remain about static at 6.22 to 5337.76. The performance overhead is now at 17.29x to 38.51x.

1,000 iterations out-of-context: As expected, the trend continues and the difference in standalone execution times becomes even more pronounced in this test case, as shown in Figure 5-26. Target program timings now range from 1.5 to 1342.46 milliseconds, while the tracing times still remain about static at 6.03 to 5449.92 milliseconds. Performance overhead is 2.69x to 3.24x.

10,000 iterations out-of-context: In this test case, we seem to pass a threshold for the secondary scaling factor. The time spent executing instructions outside the trace context is allowing the decoder to catch up with the end of the trace buffer's contents. This issue, and the solution used here, are discussed in Chapter 6.1.1. Briefly put, when the decoder catches up with the tracer, it must note its position in the buffer, re-synchronize to the previous 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 Figure 5-27. The prominent thing to note here is the greatly increased variance in the tracing execution time. This is correlated with the number of bytes the decoder must re-read after re-synchronizing. Target program timings now range from 14.2 to 13002.38 milliseconds, with tracing taking from 18.07 to 17060 milliseconds. The performance overhead incurred ranges from 1.09% to 35.54%.

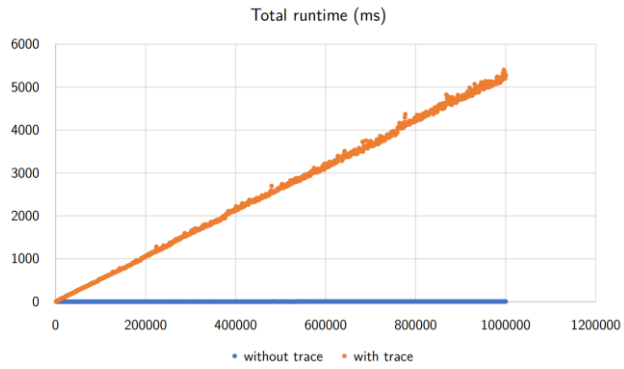

Figure 5-23: 1 iteration out-of-context

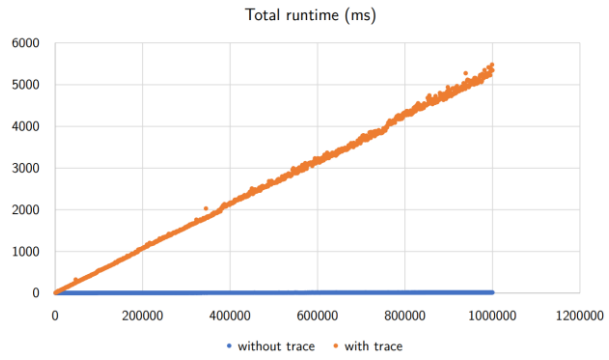

Figure 5-24: 10 iterations out-of-context

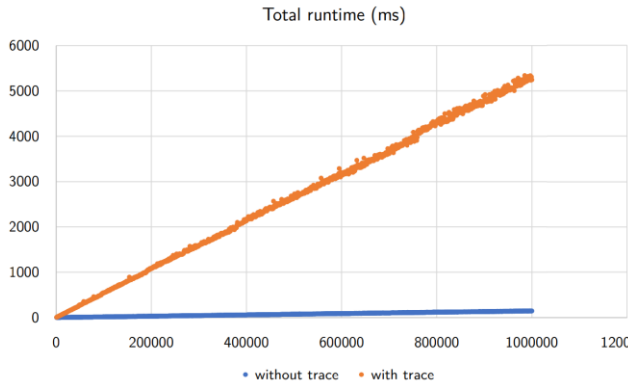

Figure 5-25: 100 iteration out-of-context

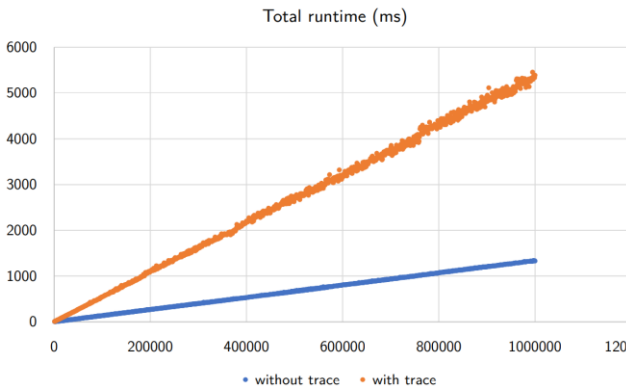

Figure 5-26: 1,000 iterations out-of-context

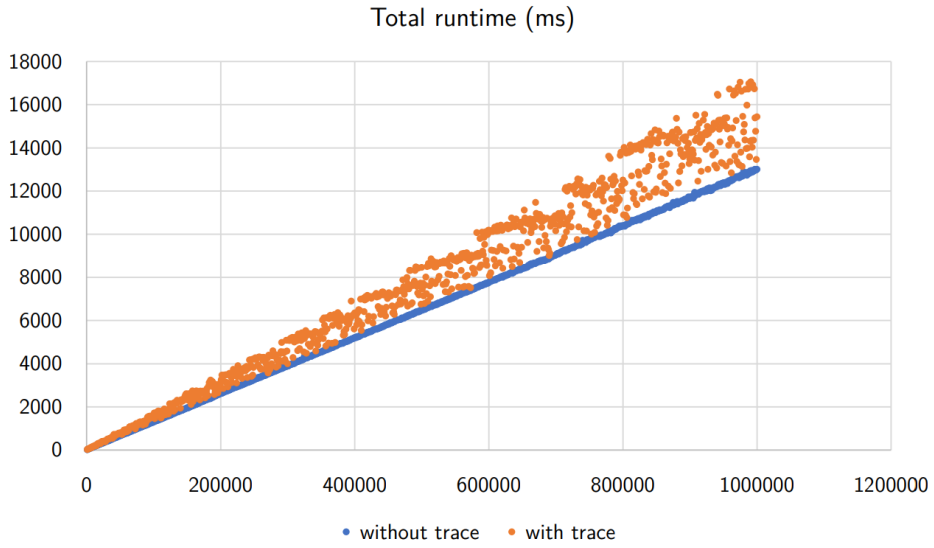

Figure 5-27: 10,000 iterations out-of-context

5.2.3 Low Complexity Scenario

The target program t_8 is designed to scale by the two selected factors. It contains a single loop in the main function. As in the high-complexity testing, the primary scaling factor controls the number of iterations in the main loop. On the other hand, the secondary scaling factor is the number of non-branching instructions between branches inside the trace context. The secondary factor must therefore be decided at compile-time and, thus, t_8 accepts only one parameter. The source code of the target program t8 is shown in Figure 5-28.

For legibility and conciseness in the source code, macros were added to allow multiple repetitions of the same instruction. The ADD instruction on line 21 is equivalent to a++ in this program. It was modified between test runs according to the secondary scaling factor. 1: ADD, 10: X(ADD), 100: C(ADD), 1000: M(ADD) and 10000: X(M(ADD)). Figure 5-29 to Figure 5-33 show graphs of the execution times of both the standalone target program and the tracing process across the range of primary scaling values.

```c
/**
 * Case 8
 *
 * Scalability test:
 * Variable loops with linear code
     inside context
 */
#include <stdlib.h>

#define ADD __asm__("addl $0x1,-0x8
    (%rbp)")
#define X(a) a;a;a;a;a;a;a;a;a;a;
#define C(a) X(X(a))
#define M(a) X(C(a));

int main (int argc, char* argv [])
{
    int a = 0;
    int scale = atoi(argv[1]);

    for (int i = 0; i < scale; i++)
    {
        ADD;
    }

    return 0;
}
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
mov    %edi,-0x14(%rbp)
mov    %rsi,-0x20(%rbp)
movl   $0x0,-0x8(%rbp)
mov    -0x20(%rbp),%rax
add    $0x8,%rax
mov    (%rax),%rax
mov    %rax,%rdi
callq  1050 <atoi@plt>
mov    %eax,-0x4(%rbp)
movl   $0x0,-0xc(%rbp)
jmp    118a <main+0x41>
addl   $0x1,-0x8(%rbp)
[...]
addl   $0x1,-0xc(%rbp)
mov    -0xc(%rbp),%eax
cmp    -0x4(%rbp),%eax
jl     1182 <main+0x39>
mov    $0x0,%eax
leaveq
retq
```

Disassembly of main function

Figure 5-28: Test case t_8 target program

1 instruction per loop: Similarly, this test case shows very large performance overheads. In every iteration of the loop, only one non-branching instruction separates the jump instructions, in addition to the three instructions involved in incrementing the iterator and comparing it to the scale factor. Figure 5-29 shows a graph of these test results. It is notable that these graphs show much clearer thresholds in the execution times of the tracer where the trace buffer size is increased. This is due to the low-complexity test cases taking less time to terminate than the high-complexity ones, and the runtime increase caused by allocating a larger buffer is constant. The execution time of the standalone target program ranges from 0.2 to 3.78 milliseconds, while the tracer runtime ranges from 2.52 to 1722.46 milliseconds, incurring performance overheads between 11.61x and 1189.63x.

10 instructions per loop: This test case, while similar to the previous one when it comes to the produced graph, it shows a decrease in the performance overhead incurred by tracing in raw numbers. A graph of these test results is shown in Figure 5-30. The target program execution time ranges from 0.23 to 14.75 milliseconds and the tracer runtime is between 2.36 and 1748.74 milliseconds. These tests indicated still large performance overheads between 9.25x and 130.1x.

100 instructions per loop: This test case shows visible growth in the standalone runtime of the target program, as seen in Figure 5-31. The target program execution times ranged from 0.37 to 142.84 milliseconds, with tracing times from 2.67 to 1757.86 milliseconds, and performance overheads between 3.2x and 12.35x.

1,000 instructions per loop: This test case showed some interesting results. As shown in Figure 5-32, the tracer execution time appears to follow one of two clearly defined trends. This is a result of the same effect we observed in the last test case of Chapter 5.2.2. At this secondary scaling factor some executions result in the decoder catching up to the tracer and requiring re-synchronization, increasing the performance overhead, while others do not. The standalone execution times of the target program ranged from 1.71 to 1413.17 milliseconds, with tracer execution times between 5.07 and 2761.13 milliseconds. This incurred performance overheads between 11.76% and 196.62%.

10,000 instructions per loop: This test case, unlike the previous ones, did not show a clear decrease in the performance overhead. This appears to be related to the way the decoder re-synchronizes to the trace stream. The effect appears to be more pronounced in the case of low-complexity code, since the target process spends orders of magnitude more time executing code within the trace context, and thus generates more trace data. This means that, on average, the decoder will need to re-read a larger amount of trace data per re-synchronization. (see Figure 5-33). The target execution times were on the range 14.44 to 13895.4 milliseconds, while the tracing times ranged from 30.65 to 34583 milliseconds, resulting in performance overheads between 24.63% and 175.59%.

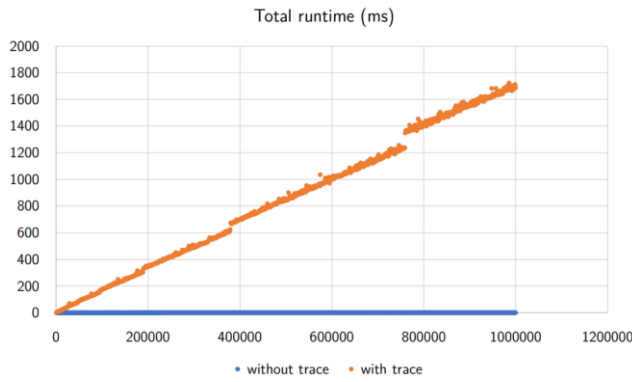

Figure 5-29: 1 instruction per loop

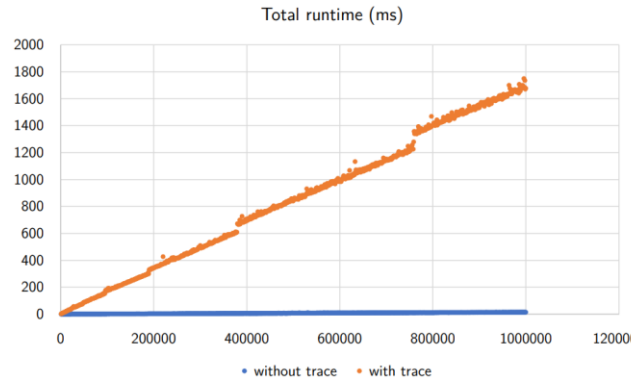

Figure 5-30: 10 instructions per loop

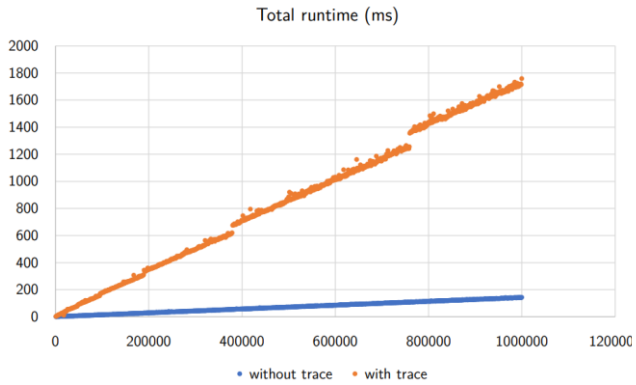

Figure 5-31: 100 instructions per loop

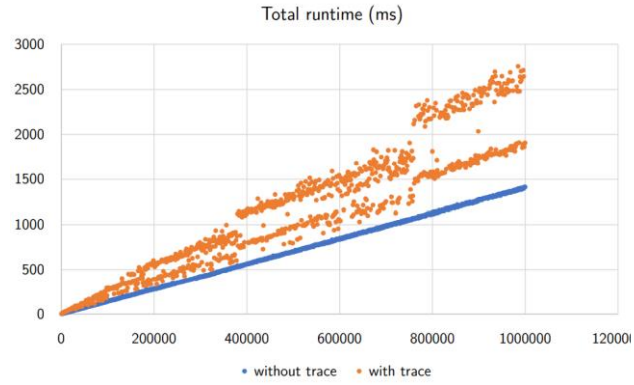

Figure 5-32: 1,000 instructions per loop

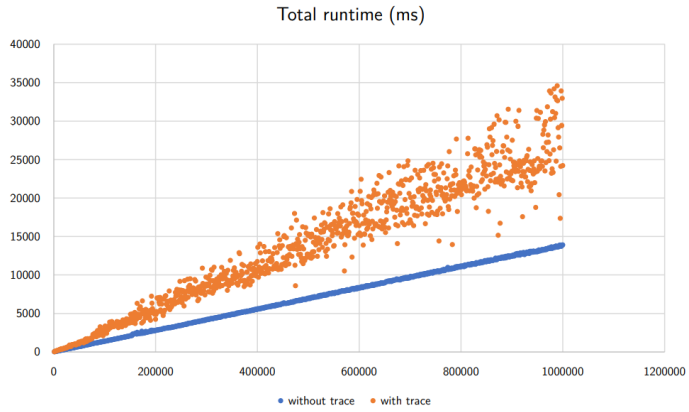

Figure 5-33: 10,000 instructions per loop

5.3 FutureTPM Intel PT Tracing Timings and Additional Test Cases

The additional timing evaluation of this solution is intended to provide a more detailed view on the practical viability and applicability of using Intel PT in the context of real-time CFA. Multiple sample programs with varying control-flow complexity where written to test the efficacy and efficiency of the solution. The sample programs are used to show whether it accurately detects different types of branches and measure its scaling characteristics. The performance testing process will focus on two primary metrics:

- **Performance overhead:** Measure the running times of the target applications with and without the analysis tool. This will give an approximation of the performance overhead incurred by tracing and decoding.
- **Scalability:** Determine how the control-flow complexity of the target program impacts the performance overhead of the analysis. This shall be done by correlating the performance overhead with a linear scaling factor for the target's complexity.

The type and granularity of the initial decoding and preprocessing of the data pulled from the FutureTPM Intel PT tracing greatly impacts the performance of the tracer. The basic distinction between the types of decoding that we perform is whether we choose to decode the entire execution flow, or just the branches of the binary and/or all the execution flow branches that can go into the system and any other external libraries. By branches we mean any jump, return, or call assembly operations that are present within the defined context. First, 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 time the decoding of only all the branches within the main function (All main branches). The timing results can be found in Table 16 with each row representing a different binary (Binaries 1-4) and each column having a different decoding granularity; the table contents show the timings in seconds.

The most common use case will just require the monitoring of the inner-binary branches so as to ensure the proper operation of the security sensitive function since the malfunction of outer binaries is considered secure and not necessarily in the scope of the tracing. Essentially, when monitoring a specific binary, we want to know whether this binary is executing as expected. If we want to have a more in-depth monitoring for some reason, the option to monitor all the execution, then the FutureTPM solution allows for dynamically changing the level of decoding based on new tracing policies that will be enforced on the device.

Table 16: Timings of the Intel PT Tracer for Various Granularity Scenarios (Full Decoding, All Branches and All Main Branches)

	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

The first observation we make is that the complexity and size of the executed code, greatly impacts the performance of the tracing and decoding of our tracer: For instance, the fourth binary with a full decoding requires 313.66655 seconds while the first and simplest binary with full decoding takes only 5.2515 seconds. **The second and most important observation is that as we can see from the results in the above table, doing a full instruction tracing on any command assembly executed and then decoding it takes a large amount of time since it decodes the entire chain of execution of commands even in**

the level of operating system, while if we specify that we only care about the branch instructions, we have a reduction in time of around 96% while further narrowing down and only selecting a specific function to trace for its branches (main), we get even lower results but not as big as the previous comparison.

In general, we could apply the instruction decoding on any of those tracing methods, either decoding only the branches or decoding only the branches in a specific function. Therefore, the amount of extra time needed to decode the instructions depends on the amount of data the tracing has captured. Here decoding the instructions is not necessary to see if any unwanted paths have been taken by the binary, as the difference in the above outputs alone without the decoding is sufficient enough to detect them. With this information in combination with the timings for the tracing of just the branching operations of the binary, we conclude that the detailed tracing with the Intel PT technology of FutureTPM is efficient and robust without any considerable overheads.

5.3.1 Compiler Optimizations

Another possibility we tested was the effect of the usage of compiler optimizations that might 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 and then run again our tests. On the second binary the time taken to run the Intel PT trace on the main function branches before the optimizations was 0.06549 and after the optimizations 0.04422 while doing the same on the third binary, the timing before the optimizations was 1.24342 while after 1.30259. As ***we can see those results are small or even in the margin of statistical error***, it is obvious therefore that ***doing any compiler optimizations will not necessarily mean better tracing performance***, as this ***depends on the nature of the binary traced and how much it can be optimized***. This performance could create smaller differences if the code were more complicated or unnecessarily complicated, where the compiler could create binaries that do not follow the execution flow of the code and have created a more optimized execution flow.

5.3.2 Nondeterministic Examples

In this chapter we will show an example use case of generating adjacency list graphs by parsing the data of our tracing output and creating a structure that will represent the execution flow. This structure is essentially a machine-readable control flow graph that is used to identify divergences in the execution of binaries. The examples shown below have either a deterministic or a nondeterministic nature, meaning that all of them have conditional branches, but for some, this condition is known at compile time, so it can be used directly in the assembly by the compiler or the condition depends on an external factor that we cannot predict beforehand, such as user input, system events or random data. The aim of this case study is to present how different control flow are determined and identified by the FutureTPM tracer.

Case 1 – Deterministic Binary

The first case is identical to the second binary of the previous case studies, where an integer is overflown by iteratively adding numbers to it. This iteration has a conditional branch that continuously checks if the number added to the variable is larger than the max 16-bit unsigned integer. For every execution of the binary, this branch will always take a predetermined path and thus we expect that the generated control flow graph will be static. The source code can be found in Figure 5-34 while the disassembled binary can be found in Figure 5-35.

```cpp
#include <iostream>

int main() {
    int16_t result = 0;

    for (uint_fast32_t i = 0; i < UINT16_MAX; ++i) {
        result += i;
    }
    std::cout << "Overflowed result=" << result << std::endl;
    return 0;
}
```

Figure 5-34: Source Code for the Integer Overflow Binary

```
main + 0            0x00001179      push rbp
main + 1            0x0000117a      mov rbp, rsp
main + 4            0x0000117d      sub rsp, 0x10
main + 8            0x00001181      mov word [var_ah], 0
main + 14           0x00001187      mov qword [var_8h], 0
main + 22           ; CODE XREF from main @ 0x11ae
main + 22        ┌─> 0x0000118f      cmp qword [var_8h], 0xfffe
main + 30        ┌──< 0x00001197      ja 0x11b0
main + 32        │   0x00001199      mov rax, qword [var_8h]
main + 36        │   0x0000119d      mov edx, eax
main + 38        │   0x0000119f      movzx eax, word [var_ah]
main + 42        │   0x000011a3      add eax, edx
main + 44        │   0x000011a5      mov word [var_ah], ax
main + 48        │   0x000011a9      add qword [var_8h], 1
main + 53        └──< 0x000011ae      jmp 0x118f
main + 55        ┌──> 0x000011b0      lea rsi, str.Overflowed_result ; 0x2004 ; "Overflowed result="
main + 62        │   0x000011b7      lea rdi, obj.std::cout         ; sym..bss
                 │                                                 ; 0x4080
main + 69        │   0x000011be      call sym std::basic_ostream<char, std::char_traits<char> >& std:
>&, char const*) ; sym.imp.std::basic_ostream_char__std::char_traits_char_____std::operator____std::char_
main + 74        │   0x000011c3      mov rdx, rax
main + 77        │   0x000011c6      movsx eax, word [var_ah]
main + 81        │   0x000011ca      mov esi, eax
main + 83        │   0x000011cc      mov rdi, rdx
main + 86        │   0x000011cf      call sym std::ostream::operator<<(short) ; sym.imp.std::ostream:
main + 91        │   0x000011d4      mov rdx, rax
main + 94        │   0x000011d7      mov rax, qword [method.std::basic_ostream_char__std::char_traits
raits_char] ; [0x3fd0:8]=0
main + 101       │   0x000011de      mov rsi, rax
main + 104       │   0x000011e1      mov rdi, rdx
main + 107       │   0x000011e4      call sym std::ostream::operator<<(std::ostream& (*)(std::ostream
main + 112       │   0x000011e9      mov eax, 0
main + 117       │   0x000011ee      leave
main + 118       └   0x000011ef      ret
```

Figure 5-35: Disassembly of the Integer Overflow Binary

01	$ python graph.py
02	>> {0: {53}, 53: {53, 30}, 30: {69}, 69: {86}, 86: {107}, 107: {118}}

Figure 5-36: Generated CFG Data for the Integer Overflow Binary

Above, in Figure 5-36, we have an example graph generated for the binary in hand. Our python tool generates this dictionary in which every key is the *index* of a branch located inside the function we are tracing, while our values are a set of those *indexes* effectively creating a linked list. As we can see, every branch instruction has an index number represented in the form “main+53” for example. The first key of the generated structure is 0, which is the start of the function, has a value of 53 where a branch took place. This branch moved us at some location in memory and after executing the instructions found there, we had another jump that happened again at index 53 (first value of index 53 is 53, indicating a loop), and finally after this loop finished we observed a jump at index 30 which moved us out of this assembly loop. Notice the loop in the disassembler as denoted by the arrows on the left of each memory location. For every execution of the binary, the generated structure will always stay the same as we expected since it has no external inputs which change the control flow path.

Case 2 – Non-Deterministic Binary

This example contains a simple switch case which reads a number from the command line arguments and passes that to a switch statement with 4 branches (Figure 5-37). The generated control-flow graphs are depicted in Figure 5-39 (based on the disassembled code of Figure 5-38). By following the same procedure to determine the control flow path, we notice that even though the graphs start by following a static pattern, we see around in the middle a small deviation, specifically at index 126. So, we come to the conclusion that at the offset 126 (validated from the disassembled code of Figure 5-38) the binary essentially makes the decision for which branch to take based on the input of the user; it is the switch case instruction. Depending on which branch in the switch statement was executed we see that the program executed a different branch each time with a varying control flow path. In a real-world scenario, the security analyst should have added each of those control flow paths as acceptable so as to prevent any false positives in the attack detection.

```
#include <string>
#include <iostream>

int main(int argc, char* argv[])
{
    (void)argc;

    int user_int = 2;

    try {
        user_int = std::stoi(argv[1]);
    } catch (const std::exception& e) {
        std::cout << "Need a number argument -- Exception: '" << e.what() << "'\n";
    }

    switch (user_int) {
        case 1: {
            std::cout << 111;
            break;
        }
        case 2: {
            std::cout << 222;
            break;
        }
        case 8: {
            std::cout << 888;
            break;
        }
        default: {
            std::cout << user_int;
            break;
        }
    }
    std::cout << std::endl;
}
```

Figure 5-37: Source Code of the Non-Deterministic Binary

5.3.3 Vulnerable Binary

To further elaborate on how an attack is detected by the Intel PT tracer, we created a simple vulnerable application in C, and we developed an exploit for it that uses code reuse techniques (ROP) to take control of the execution flow and spawn a shell. The binary would normally read something from stdin, then echo it back and finally it would read again something from stdin and store it in a buffer. For the echoing part we used a format string vulnerability such that we can have enough leaks to bypass all the modern binary protections such as ASLR, PIE and Stack cookie and for the reading part we simply overflowed a buffer of 1024 characters. What this vulnerability does essentially is that it alters the control flow path, and we expect that the resulting traces will show this difference.

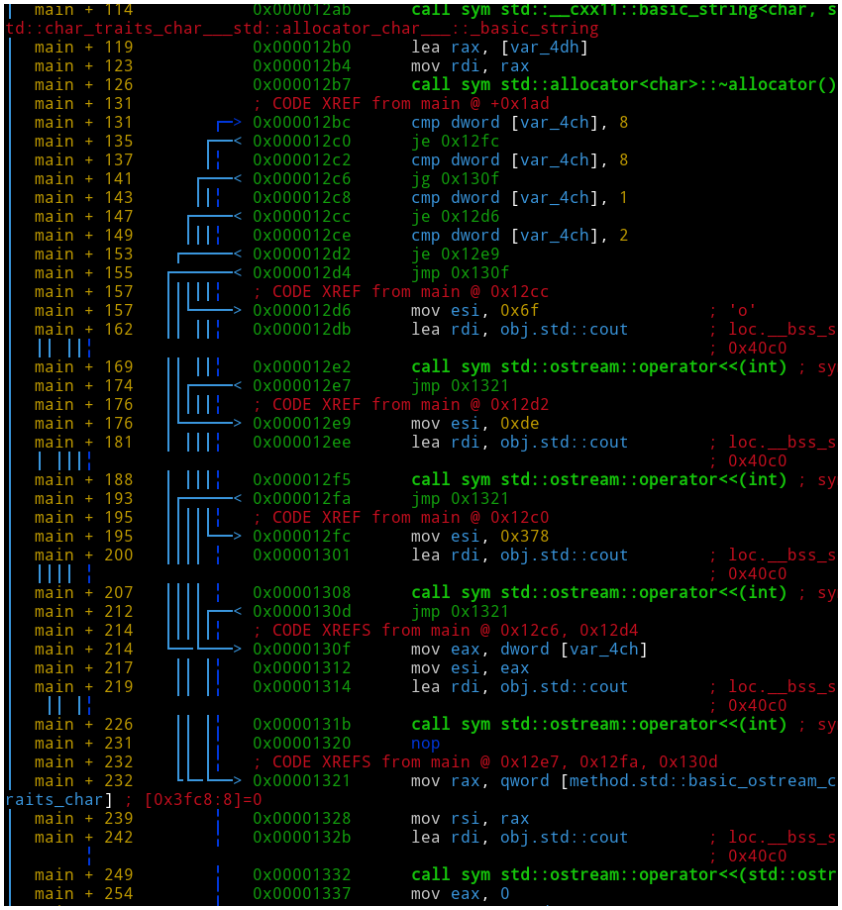

Figure 5-38: Disassembled Non-Deterministic Binary

01	$ python graph.py
02	{0: {47}, 47: {77}, 77: {99}, 99: {114}, 114: {126}, 126: {135}, 135: {207}, 207: {212}, 212: {249}, 249: {272}, 272: {466}}
03	$ python graph.py
04	{0: {47}, 47: {77}, 77: {99}, 99: {114}, 114: {126}, 126: {147}, 147: {169}, 169: {174}, 174: {249}, 249: {272}, 272: {466}}
05	$ python graph.py
06	{0: {47}, 47: {77}, 77: {99}, 99: {114}, 114: {126}, 126: {141}, 141: {226}, 226: {249}, 249: {272}, 272: {466}}

Figure 5-39: Various CFG Data Generated for the Non-Deterministic Binary

Running the binary normally and attaching to its process ID to run the trace, we got every time the same output filtering only on the main function (

Figure 5-40). This is normal since no conditional branches are dependent on any input from the user.

0	[unknown]
55870306927d	main+0x4a
0	[unknown]
558703069287	main+0x54
0	[unknown]
558703069293	main+0x60
0	[unknown]
55870306929e	main+0x6b

Figure 5-40: Control Flow of the Vulnerable Binary in Normal Execution

When doing the same, but running our exploit, we received a different control flow path as depicted in

Figure 5-41. The new control flow path is missing two instruction, which indicates that the binary execution has ended prematurely, an indication of an attack or bug. This diverging control flow graph will propagate an alarm throughout the FutureTPM risk analysis framework and will trigger any set up rules.

0	[unknown]
55870306927d	main+0x4a
0	[unknown]
558703069287	main+0x54
558703069287	main+0x54

Figure 5-41: Control Flow of the Vulnerable Binary when Exploited.

Chapter 6 Critique and Open Questions

In this final chapter of the deliverable, we would like to provide a discussion and critique of this "hybrid" type of tracing solution used by the FutureTPM project while also posing some open questions and challenges. Program tracing solutions can faithfully record runtime 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. Most tracing solutions provide an offline approach that first gathers data during the execution of monitored functionalities and then decode the gathered data to infer possible violations or other issues. **This approach is more widely adopted but a consensus in the bibliography has shown that tracing is still cumbersome, and we argue that it could be further optimized by utilizing parallelism techniques in an online approach as it was proven by the FutureTPM solution and its performance assessment.**

Although the fact is that detailed tracing and attestation can be resource intensive, it is required for proper and thorough attestation schemes (as the ones produced in the context of D4.4). In FutureTPM we have shown a methodology that provides a feasible and relatively scalable solution that advances in relation to the aforementioned offline tracing techniques. We argue that the way forward is what is proposed by the FutureTPM project; **to basically provide a multilevel detail tracing mechanism that can actually integrate and incorporate different types of tracing mechanisms with varying levels of granularity and detail in order to provide much higher scalability.** A further optimization also implemented within FutureTPM is to move away from the one-off decoding and use the benefits of parallelization in a scheme that allow the start of the decoding while the application is still running and not wait for it to finish.

Such an optimization allows for better security-performance of the end solution as it allows for a timely response to security incidents or system misconfigurations. Essentially, what is proposed and implemented, is an attempt to an **online tracing and decoding configuration**, where during runtime, the tracer feeds the gathered data to the decoder so as to allow the control flow graph to be generated spontaneously without having to wait for the finish of the traced program execution. In the case of FutureTPM, this greatly decreased the decoding time of the Intel PT tracer especially in the case of full decoding where the decoding starts during the execution time and it is ready much quicker.

Furthermore, such a parallel tracer may also perform detailed or heavily instrumented analysis of an application in parallel with a performance or lightly instrumented version of the application while also in real time allows for the decongestion of the tracing throughput by sending all its data to the decoder. Both versions of the application may operate on the same input stream, which would be responsible for managing this data and flagging it appropriately for the decoder to manage. This way, in the FutureTPM scenario of two-tier tracing, it would allow for a multi-tracing solution that would be able to trace two versions of the application, one with the eBPF tracer and one with the Intel PT tracer, in parallel while sending all the created trace data to the FutureTPM decoder for processing.

The idea behind this architecture is based on the basic computing building block of a ring buffer which is a well-suited solution for such a scenario. First of all, a ring buffer provides an ideal FIFO buffer that allows the incoming data to be processed in the correct order and priority and secondly 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 is be attached, and it feeds 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 (Figure 6-1). Of course, special measures should be taken for overflow occurrences and other issues which is a matter to be further researched as they will be presented in Chapter 6.1 with other open challenges.

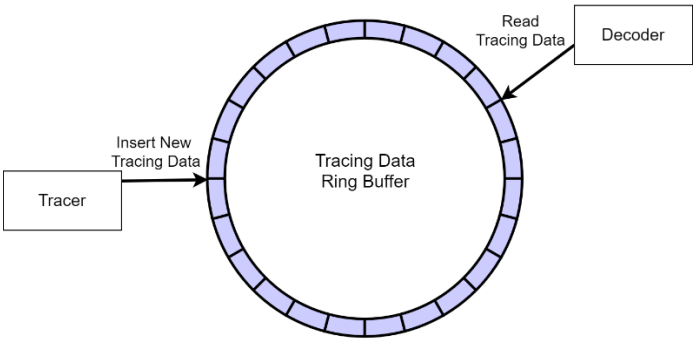

Figure 6-1: Usage of a Ring Buffer for Parallel Tracing Data Parsing

Tracing technologies such as the Intel PT tracer was designed for offline investigation of the data gathered, and the exact implementation details should be closely examined. What we expect generally and have observed in our measurements is that in the ideal scenario where the tracing process creates the required data rate, the decoding process will be T_1 seconds quicker where T_1 is the execution time of the traced process. This fact is visually depicted in Figure 6-2 where we show that if the data generated by the tracer in t_0 is enough to start the decoding process, then the decoder will be able to start its functionality also from t_0 and finish at $t_3=T_2$ whereas in the current solution, the decoder will have to wait T_1 to start and finish at $t_3=T_1+T_2$.

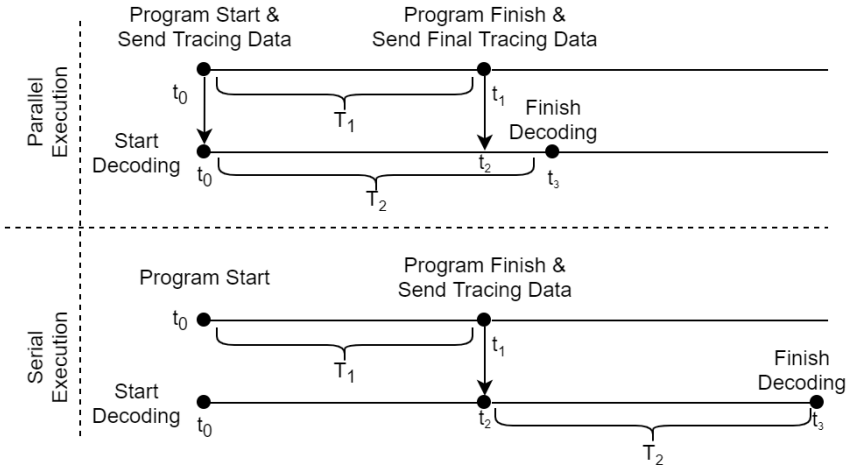

Figure 6-2: Performance Benefits of Parallel Tracing vs. Serial Tracing

6.1 Open Questions and Challenges

Several challenges and open questions were identified during the development of the FutureTPM parallel tracing solution, the most prominent are described in this chapter.

6.1.1 Concurrency issues

The target child process to be traced must be spawned in order to obtain its process ID. This is a prerequisite to the instantiation of the Intel PT file descriptor with the proper process filtering. Further initialization is then required after the initial Intel PT setup. This, and the fact that the tracer and target run in parallel, posed several challenges in the development of an Intel PT-based real-time CFA tool.

Target process exiting before setup is finished. The most immediate issue encountered as a result of this was that the target process tended to exit before the tracer finished reading the mapping information required to populate the decoder's image cache. This caused the tracer to

crash or hang. The solution of choice was to use ptrace to set the target process up such that it hit a breakpoint immediately at the start of execution. This allowed the tracer to finish the required setup before instructing the target process to continue.

Decoder running into empty buffer when filtering. Another concurrency issue was encountered only when we started to include the IP filter. When filtering on a function, for example, the Intel PT buffer remains empty until the process IP enters the filter region. This caused the decoder to discover an empty buffer, immediately report the end of stream status and quit. Due to the way ptxed is set up, this was easy to address. Rather than synchronizing once at the start of decoding, we synchronize repeatedly until the status is not end of stream. This means that the decoder will wait for input to appear in the buffer and start decoding as soon as it does.

Decoder catching up to tracer. One issue arose when experimenting with the runtime scaling of a function outside the IP filter context. The observed function was set up to call an external function from inside an observed loop. The called function took much longer to execute than the in-context instructions between calls to it. The result of this was that the decoder progressed through the trace output faster than new Intel PT packets could be generated from within the context. Eventually, the decoder reached the end of the packet stream and exited, though the process was still executing.

This issue was mitigated to some extent by detecting whether the target process was still executing before processing pending events on the buffer. If an end of trace event was encountered and the process was still running, the decoder goes through the following steps to re-synchronize:

1. Determine the current decoder offset.
2. Determine the current block count within that offset.
3. Determine the previous synchronization point (PSB packet).
4. Set the decoder's position to the previous synchronization point.
5. Resume decoding.
6. Skip processing blocks until the previous offset and block count have been reached.

This is expected to produce some additional overhead, since events still need to be reprocessed and blocks re-fetched in order for the decoder to work its way back from the preceding synchronization point to its previous position in the buffer, but only in cases when it would otherwise have exited prematurely. An edge case remains; when the target program terminates just before the decoder checks its status, but Intel PT is still writing to the buffer when the decoder raises the end of trace event, causing packets to be dropped and the decoder to exit. This appears to be rare enough, however, that testing may still proceed if the test script is made to detect the edge case and retry the trace if necessary.

Tracer overwriting decoder's position. The final issue of this type arose during performance analysis, when scaling up the tracing workload and measuring the performance overhead. At some point, the tracer is going to reach the end of its buffer. It does not stop tracing there, but continues writing from the beginning of the buffer, overwriting the existing trace. Eventually, it will catch up with the decoder and overwrite its current position. This causes the decoder to stop prematurely. The solution to this was to increase the size of the buffer to support the workloads being tested.

6.1.2 Populating the image cache

Though little modification was required to read the trace buffer from memory, it appeared that reading the sideband information from memory - in order to match packets to code - would require much more extensive changes to ptxed. The alternative was to load image sections directly into the decoder from the executable files. Since ptxed already supported this, the only required addition was the automatic detection of linked code libraries and extraction of executable section offsets.

The first step was to determine what to load. This is achieved using the ldd command in Linux to determine the expected number of linked binaries during runtime. Then, the tracer allows the target

process to step through its initialization, block by block, monitoring the target's maps file until the expected number of files has been linked. Once all the required files have been identified, the offsets of their executable sections must be determined. For this, the tracer uses a customized version of the load_elf utility included with libipt, which reads this information directly from Executable and Linkable Format (ELF) files. It then uses ptxed's built-in functionality to load the raw executable sections into the cache.

6.1.3 Inconsistent block boundaries

Two variations of this issue were observed during development:

1. Blocks ended on non-branching instructions, regardless of IP filter config.
2. Blocks failed to end on branching instructions, depending on IP filter config.

The former was easy to handle, as the decoder only needed to determine whether the last instruction of the block was an indirect or conditional branch, and whether that branch was taken.

The latter, however, posed a bigger challenge. It seemed to be an issue with Intel PT's handling of IP filters, as it arose only when using certain filter configurations. Branches leading outside the filter range were ignored in some cases, leading to blocks extending outside the filter range, containing branching instructions that should have marked the end of that block and caused Intel PT to temporarily disable tracing.

The first approach to this issue was to pass the IP filter range to the decoder and check whether each block ended within the range. If it did not, the block was truncated at the last instruction within the filter range. This was an expensive operation, since it required the decoding of every instruction up to and including the branching instruction, in addition to the final instruction of the block. While this was an effective workaround, it violated the design principle of minimizing post-processing, as described in Chapters 4.3 and 4.4.

The problem turned out to be the result of a known bug in Intel PT itself. When tracing is disabled via a direct, unconditional branch, the resulting TIP packet may not contain the target IP. This bug was originally documented in erratum SKL014. Luckily, libipt has a workaround for this. If the IP filter configuration used by Intel PT is provided in the decoder configuration, along with the correct CPU details, the decoder can use this information to bind the TIP packet to the correct instruction and determine the correct end of the block.

6.1.4 Migrate to purely software-based tracing capabilities.

Every tracing solution so far is hardware-based (like CFLAT). Our proposal with Intel PT is pseudo-hardware-based because we assume the existence of an Intel processor. **What we need is the move towards a pure software-based tracing solution that will be generic and applicable in a wide range of application domains.**

6.2 Other optimizations

According to [18], few tracers have paid attention to the size of traces and corresponding overheads introduced to offline analysis, as well as the Control Flow Graph support. This work presents a solution called ATOS, an efficient tracing solution, to address these issues. It adaptively adjusts the granularity of tracing while conservatively preserving the essential execution information. They implemented a prototype of this solution and evaluated it on various benchmarks. The results have shown that their solution can greatly reduce the size of a trace and accelerate offline analysis, while preserving the execution states and supporting existing applications seamlessly. For example, using ATOS, the trace produced by an application called CryptoHunt is reduced by 46 times, while the analysis time is reduced by 34 times. This solution resembles the one used by FutureTPM but instead of changing the tracing technology, it changes the granularity of the existing tracing technique to scale depending on the needs of the system.

One optimization that ATOS utilizes is loop identification and optimization. A loop, in general, is a repeated sequence of instructions. Loops are one of the root causes of the oversized traces. To reduce the size of a trace, ATOS performs loop-level tracing once the instruction list of a loop iteration has been determined. The challenge here is identifying loops during runtime tracing. ATOS first statically analyses the target program to identify candidate loops (including nested loops) and then monitors runtime code modification and generation to recognize new loops. Moreover, the authors have refined the loop-level tracing process to handle complicated loops, that is, ones that have a very high cyclomatic complexity.

Another optimization is proposed by [19] in a solution called Fay which shows that modern techniques for high-level querying and data-parallel processing of disaggregated data streams are well suited to comprehensive monitoring of software execution in distributed systems. Fay also demonstrates the efficiency and extensibility benefits of using safe, statically verified machine code as the basis for low-level execution tracing. Finally, Fay establishes that, by automatically deriving optimized query plans and code for safe extensions, the expressiveness and performance of high-level tracing queries can equal or even surpass that of specialized monitoring tools.

C-FLAT is a remote runtime CFA tool for embedded systems. It requires modifications to the target binaries, adding so-called trampolines at each branch instruction, so that they inform the tracer each time a branch is encountered. An external, trusted device handles the actual attestation from there. The attestation is stateful, i.e., it is able to match an entire execution path to its CFG. This unloads a large portion of the performance overhead to external hardware. What remains on the target machine is linear overhead correlated with the number of branch instructions in the target program [20].

LiteHAX is a remote runtime CFA tool for low-end embedded systems. It attests not only the control-flow of a program, but also its data flow, allowing for the detection of a very wide variety of attacks. Like C-FLAT, it does require additional trusted hardware to function. In a case study, the researchers measured a ≈ 15% performance overhead on the prover device [21].

CFIMon is a tool that leverages Intel Branch Trace Store (BTS) functionality - in combination with performance counters - to validate branches locally at runtime with very little overhead. It operates without special modifications to program binaries and requires only a compatible Intel processor to function. However, it is limited in that it only validates branch targets without context, and cannot detect high level semantic control-flow violations [22].

Finally, GRIFFIN by uses Intel hardware tracing functionality in execution monitoring and CFI enforcement with a hybrid approach. Requiring a compatible Intel processor to function, it uses specifically defined security policies for validation. These policies can be course-grained, providing a lookup table of legitimate branch destination addresses, allowing for very fast validation of branch destinations. To balance security and performance, GRIFFIN also supports fine-grained policies. These denote legitimate branches in terms of both source and destination addresses. Validating against this is much more secure, due to the fact that illegitimate control-flow may still branch to legitimate destination addresses, but also much slower. In order to extract the source address of the branch the control-flow must be reconstructed from the trace. GRIFFIN is capable of switching between **course- and fine-grained policies at runtime to minimize the performance overhead and lag between the occurrence and detection of control-flow violations during execution** [14].

Chapter 7 Summary and Conclusion

Summarizing this deliverable, **we underlined the need for efficient tracing and tracking of mission critical applications on a detailed low-level so as to understand and timely detect attack attempts or malfunctions of the executable.** With this tracing approach we have created a binary monitoring solution that not only catches known threats but is also capable of detecting zero-day vulnerabilities that could affect the executable in hand.

This was produced in the context of the FutureTPM Risk Assessment framework towards supporting a **dynamic multi-level detailed tracing solution with a varying granularity for the amount and the type of data gathered.** This is achieved by leveraging a two-tier granularity tracing technique that initially monitors the targeted binary with an efficient and highly scalable tracing technology named "extended Berkeley Packet Filter" (eBPF); which allows for the tracing of the configurational integrity of the executing binary without posing any significant overhead (around 6% additional time on the traced functions). These execution hooks were also enhanced with more thorough **tracing capabilities based on the use of the emerging IntelPT tracing mechanism**. This programmable component provides near **real-time low-level code inspection**, thus, capturing the strict constraints of SoS-enabled ecosystems, as envisioned in FutureTPM.

These kernel hooks, initially, are identified as low-level behavioural properties and are deployed to trace system calls. The goal of this initial tracing is to monitor system calls, **produce the necessary CFGs and finally acquire the attestation report.** In the case of a failed attestation report, FutureTPM has defined a methodology for increasing the level of monitoring in order to collect additional evidence and information on the incident for the assistance in finding the province of the attack as well as in the development of new enforceable policies that should be able to catch this newly identified threat that caused the attack in the first place. This is what the integration of Intel PT is currently offering. Effectively, we propose a novel solution for multi-level detailed tracing that, depending on the situation, **monitors the security-critical components of each system in different levels in order to upkeep the desired assurance while also providing the required details for post-attack investigation.** This multi-level detailed tracing is implemented in a semi-automatic manner. Towards this direction, there are two possibilities explored within FutureTPM. The first is the automatic deployment of new, more rich, programmable eBPF hooks after the reception of a failed attestation report. In this case, the security analyst(s) (performing the offline investigation) assess the attestation report and determines whether more information is needed for identifying the cause and type of attack - in which case she also defines the set of policies describing the type of information/evidence to be collected. After this process, new programmable ebPF kernel hooks are seamlessly and automatically deployed to the target device for fulfilling these policies. As previously noted, the eBPF is implemented in the kernel, thus, needs a userland control agent for remote control and exportation of the data. Such agents are implemented as part of the FutureTPM RA framework. Overall, this approach enable us to get the necessary evidence without affecting the performance of the target system since these rich eBPF hooks (that will incur a higher penalty in the system execution as more system calls need to be monitored in near real-time) will be deployed. However, there is the inherent limitation of a large attack time window: Since the detailed tracing commences after the attestation process has failed, this means that the attack is already in place which in turn results to the additional requirement of "letting" the attack to run further (for a larger amount of time) in the device until the detailed tracing has finished. Depending on the application at hand, such an approach (which will of course require the isolation of the attacked device during the evidence collection so as other adjacent assets are not affected) might not be feasible.

Compounding this issue, the other approach adopted by FutureTPM is the deployment, during runtime, of Intel PT introspection agents already capable of enhanced monitoring and tracing of most of the low-level functional calls. In this case, only those execution hooks that are necessary for tracing the CFGs to be attested will be active during the run-time risk assessment phase (i.e.,

including the execution of the CFPA mechanism) and the enriched agents will be activated only if the evidence collection is triggered; thus, resulting to the transmission of the necessary information about the device that got compromised, vulnerabilities found in any of the internal system components or their configurations, configuration changes, etc. This approach while less efficient, as the penalty of increased tracing is incurred from the beginning, doesn't necessitate a large attack time window.

Chapter 8 List of Abbreviations

Abbreviation	Translation
EA	Enhanced Authorisation
AES	Advanced Encryption Standard
AK	Attestation Key
BBA	Binary-Based Attestation
BIKE	Bit Flipping Key Encapsulation
BLISS	Bimodal Lattice Signature Scheme
CFB	Ciphertext Feedback
CIV	Configuration Integrity Verification
CPU	Central Processing Unit
CSR	Certificate Signing Request
DAA	Direct Anonymous Attestation
DSA	Digital Signature Algorithm
EA	Enhanced Authorization
eBPFs	enhanced Berkeley Packet Filters
ECC	Elliptic Curve Cryptography
FPGA	Field Programmable Gate Array
HMAC	Hash-based MAC
ICT	Information and Communication Technologies
IMA	Integrity Measurement Architecture
IntelPT	Intel Processor Trace
IoT	Internet of Things
KEX	Key Exchange
LDAA	Lattice-based Direct Anonymous Attestation
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
NMS	Network Management System
NTT	Number-Theoretic Transform
PBA	Property-Based Attestation
PCA	Privacy Certification Authority
PCRs	Platform Configuration Registers
PKE	Public Key Encryption
PQ	Post-Quantum
PQC	Post-Quantum Cryptography
QR	Quantum Resistant
RNG	Random Number Generator
RSA	Rivest–Shamir–Adleman
SAPiC	Stateful Applied Pi Calculus
SHA	Secure Hash Algorithm
SoC	System-on-chip
S-ZTP	Secure Zero Touch Provisioning
TC	Trusted Components

Abbreviation	Translation
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCTI	Command Transmission Interface
TLS	Transport Layer Security
TPM	Trusted Plaform Module
TSS	TPM Software Stack
TTP	Trusted Third Party
U2F	Universal 2nd Factor
UC	Universal Composability
XOF	Extendable-Output Function
XOR	Exclusive OR

Chapter 9 Bibliography

[1] “FutureTPM Deliverable D4.3: Runtime Risk Assessment, Resilience and Mitigation Planning – First Release.”.

[2] “FutureTPM Deliverable D4.2: FutureTPM Risk Assessment Framework - First Release.”.

[3] “FutureTPM Deliverable D4.4: FutureTPM Risk Assessment Framework - Final Release.”.

[4] “FutureTPM Deliverable D1.2: FutureTPM Reference Architecture.”.

[5] “FutureTPM Deliverable D4.1: Threat Modelling & Risk Assessment Methodology.”.

[6] *OLISTIC Risk Assessment Platform*. UBITECH.

[7] “FutureTPM Deliverable D6.1: Technical Integration Points and Testing Plan.”.

[8] “Overview of the Linux Virtual File System — The Linux Kernel documentation.” https://www.kernel.org/doc/html/latest/filesystems/vfs.html (accessed Dec. 01, 2020).

[9] “Trusted Platform Module Library Part 2: Structures.” [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.07-2014-03-13.pdf.

[10] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, “The ponder policy specification language,” in *International Workshop on Policies for Distributed Systems and Networks*, 2001, pp. 18–38.

[11] “TCG TNC Federated TNC,” *Trusted Computing Group*. https://trustedcomputinggroup.org/resource/federated-tnc/ (accessed Dec. 01, 2020).

[12] “Trusted Network Connect: Open Standards for Integrity-based Network Access Control,” *Trusted Computing Group*. https://trustedcomputinggroup.org/resource/trusted-network-connect-open-standards-for-integrity-based-network-access-control/ (accessed Dec. 01, 2020).

[13] “Processor Tracing.” https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html (accessed Jan. 15, 2021).

[14] X. Ge, W. Cui, and T. Jaeger, “Griffin: Guarding control flows using intel processor trace,” *ACM SIGPLAN Not.*, vol. 52, no. 4, pp. 585–598, 2017.

[15] “Intel® 64 and IA-32 Architectures Software Developer Manuals,” *Intel*. https://www.intel.com/content/www/us/en/develop/articles/intel-sdm.html (accessed Jan. 15, 2021).

[16] “Perf tools support for Intel® Processor Trace - Perf Wiki.” https://perf.wiki.kernel.org/index.php/Perf_tools_support_for_Intel%C2%AE_Processor_Trace (accessed Jan. 15, 2021).

[17] “Perf Wiki.” https://perf.wiki.kernel.org/index.php/Main_Page (accessed Dec. 01, 2020).

[18] H. Sun, C. Zhang, H. Li, Z. Wu, L. Wu, and Y. Li, “ATOS: Adaptive Program Tracing With Online Control Flow Graph Support,” *IEEE Access*, vol. 7, pp. 127495–127510, 2019.

[19] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, “Fay: Extensible distributed tracing from kernels to clusters,” *ACM Trans. Comput. Syst. TOCS*, vol. 30, no. 4, pp. 1–35, 2012.

[20] T. Abera *et al.*, “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.

[21] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, “Litehax: lightweight hardware-assisted attestation of program execution,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[22] Y. Xia, Y. Liu, H. Chen, and B. Zang, “CFIMon: Detecting violation of control flow integrity using performance counters,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.

[23] IOVisor, “BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more,” 2018. [Online]. Available: https://github.com/iovisor/bcc#tools.