Grant Agreement No.: 952697
Call: H2020-SU-ICT-2018-2020
Topic: SU-ICT-02-2020
Type of action: RIA

ASSURED

D3.4: ASSURED REAL-TIME MONITORING AND TRACING FUNCTIONALITIES

Revision: v.1.0

Work package	WP 3
Task	Task 3.4
Due date	28/02/2022
Deliverable lead	MLNX
Version	1.0
Authors	Meni Orenbach (MLNX), Ahmad Atamli (MLNX)
Reviewers	Richard Mitev, Phillip Rieger (TUDA) Thanassis Giannetsos (UBITECH)
Abstract	Deliverable D3.4 focuses on the first release of the software-based Tracer designed in ASSURED for enabling the real-time monitoring of various system properties to be considered for attestation. More specifically, D3.4 puts forth the ASSURED device runtime data and execution stream monitoring and introspection capabilities necessary for tracing the control- and information-flow execution paths needed by the runtime attestation enablers developed in WP3. Dynamic tracing functionalities are provided, as programmable components, 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' behavior for detecting cheating attempts or if any type of exploits to the program and data memory. This provides the trusted anchor 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 safety-critical components
Keywords	Trusted Computing, Dynamic Tracing, Intrusive vs. Non-Intrusive Tracing, Dynamic Memory Acquisition

Document Revision History

Version	Date	Description of change	List of contributors
v0.1	15.12.2021	ToC	Meni Orenbach (MLNX)
v0.2	10.01.2022	SOTA analysis of the various hw- and sw-based tracing mechanisms that exist in the literature. Reasoning behind the design choice of ASSURED to proceed with a urely sw-based tracing solution (Chapter 2)	Meni Orenbach, Ahmad Atamli (MLNX) Ilias Aliferis (UNIS)
v0.3	17.01.2022	First draft of the Tracer architecture, its functional specifications and mode of operation (Chapter 4)	Meni Orenbach, Ahmad Atamli (MLNX) Richard Mitev, Philip Rieger (TUDA) Thanassis Giannetsos (UBITECH)
v0.4	26.01.2022	Description of the instantiation and usage of the Tracer in the context of Control-flow Attestation (Chapter 5)	Ahmad Atamli, Meni Orenbach (MLNX/MLNX) Richard Mitev, Philip Rieger, Marco Chilese (TUDA)
v0.5	29.10.2021	Description of the instantiation and usage of the Tracer in the context of the Configuration Integrity Verification (Chapter 6)	Meni Orenbach, Ahmad Atamli (MLNX) Benjamin Larsen, Heini Bergsson Debes(DTU) Ilias Aliferis (UNIS) Dimitris Karras (UBITECH)
v0.6	10.02.2022	Description of the interaction between the Tracer and the ASSURED Attack Validation component towards the acquisition of system traces in order to identify the exact attack path that an adversary exploited (Chapter 7)	Meni Orenbach, Ahmad Atamli (MLNX) Karthik Shenoy Panambur (BIBA) Thanassis Giannetsos, Dimitris Papamartzivanos (UBITECH)
v0.7	18.02.2022	Finalization of the Tracer architecture and list of the future plans towards the second and final release – including also a detailed evaluation plan for different codebase complexity (Chapter 2 and 8)	Meni Orenbach, Ahmad Atamli (MLNX) Thanassis Giannetsos (UBITECH)
v0.9	24.02.2022	Review the document	Richard Mitev, Phillip Rieger (TUDA) Thanassis Giannetsos (UBITECH)
v1.0	28.02.2022	Finalisation of the document	Meni Orenbach, Ahmad Atamli (MLNX)

Editors

Meni Orenbach (MLNX), Ahmad Atali (MLNX)

Contributors (ordered according to beneficiary numbers)

Heini Bergsson Debes, Benjamin Larsen (DTU)

Richard Mitev, Philip Rieger, Jingru Wang, Marco Chilese, David Koisser (TUDA)

Liqun Chen, Nada El Kassem (SURREY)

Ahmad Atali, Meni Onrebach (MLNX)

Dimitrs Papamartzivanos, Thanassis Giannetsos, Dimitris Karras (UBITECH)

Karthik Shenoy Panambur (BIBA)

Ilias Aliferis (UNIS)

Funded by the Horizon 2020
Framework Programme of the European Union

DISCLAIMER

COPYRIGHT NOTICE

Project co-funded by the European Commission in the H2020 Programme		
Nature of the deliverable:	**R**	
Dissemination Level		
PU	Public, fully open, e.g. web	✓
CL	Classified, information as referred to in Commission Decision 2001/844/EC	
CO	Confidential to ASSURED project and Commission Services	

* R: Document, report (excluding the periodic and final reports)

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

OTHER: Software, technical diagram, etc.

Executive Summary

The design and the documentation of the ASSURED Reference Architecture is presented in D1.2 [9]. D1.2 depicts the identified core interfaces and components that need to be provided towards the integration of the overall ASSURED framework. This deliverable aim is to present the ASSURED Tracer component, which provides the backbone capability to prove the edge devices state is secure to remote verifiers. That is achieved via the different attestation mechanisms considered in ASSURED such that are documented in D3.2 [8].

The ASSURED Tracer is designed to securely capture all information regarding the runtime state of each edge device deployed in ASSURED. For example, runtime information includes the up-to-date configuration state of the device, and per-program executed control flows. The tracer generates traces that are used by the attestation verifiers. The ASSURED risk assessment engine uses this information to calculate the risk level on each device and in the entire ASSURED ecosystem.

ASSURED ensures the trustworthiness of the generated traces by running the tracer in a Trusted Execution Environment. This guarantees the integrity and authenticity of the received traces, which results in the tracer being included in the overall ASSURED Trusted Computing Base. D3.1 [6] designs the protocol for secure and authentic communication between the tracer and the TPM-based Wallet. The latter verifies the authenticity of the traces and their signatures prior to sending them to the verifiers.

In this deliverable, we present the algorithms to securely track control flows, configuration, and variable values in edge devices, aiming for a negligible impact on the performance of programs executing in edge devices. The ASSURED tracer is a software component that can be executed on top of commodity off-the-shelf platforms.

The high-level tracing methods to be considered in the context of ASSURED are the following:

- Control flow tracing
- Configuration integrity tracing
- Attack validation program tracing

In conclusion, the overall purpose of this deliverable is to provide a reference document for the ASSURED Tracer that will explain the underlying mechanism to support the attestation primitives considered within WP3. The deliverable contains a detailed definition of the technical interfaces, the security of the traces, and the tracing methods that need to be supported by the ASSURED tracer.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Towards Efficient Remote Attestation with Software Assisted Multi-level Execution Tracing

In the face of an increasing attack landscape, as described in D1.3 [11] and D2.1 [13], it is necessary to cater for efficient mechanisms to verify software and device integrity for detecting run-time modifications in next-generation "Systems-of-Systems". Recall the latest trend in the attack vectors, as documented by the Open Web Application Security Project (OWASP) [35], where an updated ranking list of Common Vulnerabilities and Exposures (CVEs) was put forth: *It is apparent that memory-related vulnerabilities are becoming more prevalent and lucrative targets to be exploited by adversaries for launching software-based attacks against deployed devices*. Such attacks can range from the exploitation of loopholes due to **security misconfiguration and insecure system design** to **vulnerable & outdated components and cryptographic failures**. The common denominator in all such cases is the lack of **appropriate security hardening across any part of the application stack**: from the secure boot of a system (based on secure, certified, and tested software) to the run-time detection of software and data integrity failures through efficient and effective trustworthiness control design.

In this context, ASSURED offers orchestration features of distributed (remote) attestation enablers [8] for providing enhanced operational assurance and functional safety of the entire SoS-enabled supply chain for checking and assuring the integrity and execution correctness of the deployed safety-critical CPSoS. This defense mechanism enables the safeguarding of both the software and hardware layers covering all phases of devices' 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 behavioral attestation of those safety-critical components of a system** (as defined in D1.3 [11]) providing strong guarantees on the correctness of the **control- and information-flow properties**, thus, enhancing the performance and scalability when composing secure systems from potentially insecure components.

However, most of the existing families of such attestation solutions suffer from the lack of software-based mechanisms for the efficient extraction of rigid system information traces. This limits their applicability to only those cyber-physical systems with the necessary amount of resources for being able to perform detailed code analysis or equipped with additional hardware support. Unfortunately, this approach does not capture the real-time constraints of emerging attestation security enablers that require a detailed dynamic tracing of properties stemming from diverse levels of a system's architecture [36]: kernel shared libraries, low-level code, etc. resulting in an in-depth

investigation of the systems behavior and execution flow towards detecting any cheating attempts or if any type of (non-previously identified) exploits are resident to the memory.

To date, several remote attestation techniques have been proposed to verify the integrity of software or the control-flow of devices. However, most of them are static and verify only the software integrity of devices, and only recently some run-time Control-flow Attestation (CFA) and Control-flow Integrity (CFI) schemes, leveraging control-flow information of a program, have been proposed [8, 25, 28]. With CFA, sophisticated attacks that tamper with state information in the program's data memory (e.g., the stack and the heap) can be detected. CFA mechanisms can be also deployed as part of holistic run-time risk assessment frameworks and contribute towards a more concrete cyber risk quantification. In all of these approaches, however, there is always a compromise between performance, security, and usability. This sets the challenge ahead: *How to provide near real-time low-level code inspection and tracing, thus, capturing the requirements of ASSURED remote attestation while striking a balance between the precision of monitored system traces, efficiency, and transparency?*

1.2 Scope and Purpose

The main goal of this deliverable is to present the design of the **ASSURED (software-based) Tracer component, which includes its architecture, algorithms, and interaction with the other ASSURED components ecosystem towards the provision of detailed systems traces to facilitate the ASSURED attestation toolkit for real-time embedded devices**. Unlike other existing solutions, ASSURED Tracer does not require any **custom hardware extensions**, for offering multi-level execution tracing with the required timing guarantees, and it can operate with minimal trust assumptions as defined in D3.1 [6].

The ASSURED Tracer (*tracer* hereafter), is designed to securely capture all information regarding the runtime state of each edge device. This includes, but is not limited to **tracking control of configured programs, and the entire device configuration state**. The tracer generates *traces*, which depict the necessary system measurements (based on the validation properties of only those safety-critical components) to be used by the attestation verifiers, and calculates the risk level on each device and in the entire ASSURED ecosystem.

The tracer is designed to run on the same device as potentially compromised software. However, to ensure the trustworthiness of the traces it is run in a Trusted Execution Environment (TEE), which provides an isolated execution guaranteed by the underlying hardware. This is essentially the only assumption needed for guaranteeing the integrity and authenticity of the received traces as part of the overall ASSURED Trusted Computing Base (TCB). As described in D3.1 [6] and summarized in Section 4.3, ASSURED has already designed the appropriate protocol for this secure and authentic communication between the tracer and the TPM-based Wallet which is responsible for verifying the authenticity of the traces and their signing prior to being sent to the Verifier. In this TEE, the tracer securely signs the traces to ensure their authenticity before passing them to the TPM-based Wallet. The Trusted Software Stack (TSS) is not part of the "*secure world*" of the TEE, thus, the need to use strong key usage protection policies for safeguarding the integrity of the traces prior to also being sent to the Verifier.

We envision the tracer together with the TEE's operating system to be small in size such that the amount of code - as part of the TCB - is limited, which reduces the overall attack surface on the tracer and does not pose a large fingerprint on the performance.

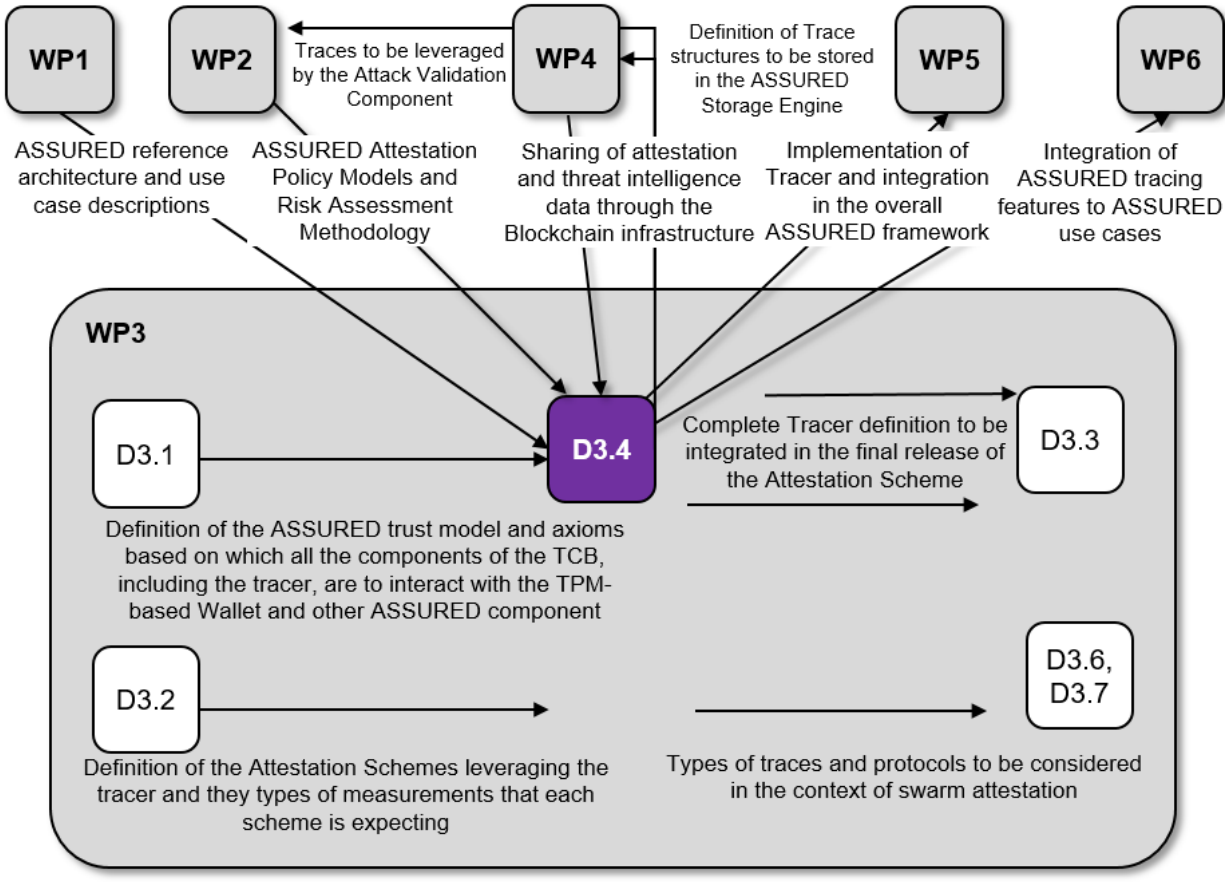


Figure 1.1: Relation of D3.4 with other WPs and Deliverables

In the following chapters, we present the designed algorithms to **securely track control flows, configuration, and variable values** in edge devices, with negligible intrusiveness on the device software, while using commodity off-the-shelf platforms, and with negligible desired impact on software executing in the device. Finally, we present the different interfaces used to communicate with the ASSURED components, including the attestation verifiers and the attack validation component. We also provide samples of traces to depict the traces format and tracing granularity.

1.3 Relation to other WPs and Deliverables

In what follows, Figure 1.1 depicts the relationship of this deliverable with other Work Packages (WPs) as well as other tasks in the same WP(3). As aforementioned, the main focus of this deliverable is the design of the ASSURED software-based, multi-level execution tracer for providing the necessary system measurements to be used during the attestation enablers, as defined in D3.2 [8]. Thus, this document acts as technical guidance for the **monitoring of all validation properties**, defined in D1.3 [11] per use case, and for the **interactions that need to be executed between the Tracer and the other ASSURED components** based also on the overall conceptual architecture defined in D1.2 [9]. in this context, D3.4 also puts forth the detailed description of the entire ASSURED flow of actions and how the Tracer is been triggered and used towards providing the necessary security claims that the Attestation Agent can then use for verifying the level of trustworthiness of a device in the overall service graph chain.

For the latter, and to assure the security of the calculated traces, the design also takes into con-

sideration the trust models and the detailed protocols (defined in D3.1 [6]) for the **secure and authentic communication with the TPM-based Wallet** that, in turn, will sign the traces to ensure their integrity when sent to the Attestation Agent of the Verifier device. Detailed definitions are also provided on the type of traces to be provided per attestation scheme so that they can be integrated into the final Attestation Toolkit. In the same line of activities, besides the attestation process, the Tracer also provides input to the ASSURED Attack Validation component that, defined in WP2, in the case of a failed attestation report, it consumes the produced system traces for being able to perform a more detailed analysis on the exact attack path that was leveraged by the adversary to compromise the target device.

Overall, the outcome of Deliverable D3.4 is intended to support the definition of later activities in the project. In relation to the rest of the WPs of the project, D3.4 serves as a point of reference for the technical developments of the project as it offers a set of directions to each WP. More specifically, D3.4 provides the description of the operation of the Tracer, the outcome of which will be leveraged by the ASSURED attestation mechanisms towards assessing the assurance and trustworthiness level of a “Systems-of-System”. This in turn will be recorded, audited, and shared through the policy-compliant Blockchain infrastructure designed in WP4. Last but not least, WP5 undertakes the development of the integrated framework, and WP6 aim is the validation of the high-level interactions of components in the context of the pilot use-cases.

1.4 Deliverable Structure

This deliverable is structured as follows: After providing necessary background knowledge in **Chapter 2**, we describe in **Chapter 3** the ASSURED System Model comprising the building blocks and hardware and software components, including a discussion about the trusted computing base and its relation to the tracer. Next, in **Chapter 4** we provide an in-depth description of the tracer architecture while presenting the relation to the overall ASSURED ecosystem. Then, in **Chapter 5**, we provide high-level information on the control-flow attestation mechanism, the tracing method to capture control flows, and the interface with the attestation Verifiers. In **Chapter 6**, we provide high-level information on the configuration integrity attestation mechanism, the tracing method to capture loaded libraries and binaries in edge devices, and the interface with the attestation Verifiers. In **Chapter 7**, we provide a high-level description of the attack validation component, followed by a presentation on the tracing of variables defined in the system specification given by an administrator and the interface the tracer and the attack validation component use. Finally, we conclude this deliverable in **Chapter 9**.

Chapter 2

Research Background of Tracing Techniques

As aforementioned, one of the core building blocks in any attestation scheme is the use of a Tracer capable of extracting the necessary system measurements to be used for verification. Such techniques are used to collect **statistical information, performance analysis, dynamic kernel or application debug information, and in general, system audits**. In dynamic tracing, this can take place without the need for recompilation or reboot. In the context of Control-flow Attestation (Chapter 5), a detailed dynamic tracing [28, 36] of the kernel shared libraries, low-level code, etc., and an in-depth investigation of the system's behavior and execution flow will be performed to detect any cheating attempts or if any type of (non-previously identified) exploits are resident to the program and data memory. For instance, consider a device Tx_i that hosts a critical software component C running a set of services $Srv_1, Srv_2, ..., Srv_j$. Based on the pre-defined policies that dictate when the attestation will commence, the trusted anchor in Tx_i will request from the tracing component to record the control-flow of only those safety-critical services (properties) of interest $Srv_k, ...Srv_j$. At the end of the execution, the tracer provides the trusted anchor with the compiled CFG_i that represents the run-time state of Tx_i, again only if those properties of interest need to be attested.

In what follows, we give a state-of-the-art analysis on the most prominent types of tracing techniques and discuss their merits and challenges. In the end, we also proceed with an evaluation between **intrusive and non-intrusive tracing capabilities** in order to highlight the core features of the ASSURED Tracer that allows it to perform an efficient tracing with high accuracy while imposing the least overhead and impact on the device's actual performance.

2.1 Tracer Solutions

Modern attacks aim at compromising software components in systems. Software is traditionally considered more susceptible to attacks due to its fast pace changes and lack of verification as opposed to hardware-based components. TPMs [43] are traditionally used to ensure that devices are in a valid state. However, this only applies to the initial boot state of the devices. Attacks on run-time may compromise the device even though it was initialized in a valid state due to internal vulnerabilities in the deployed software components. Thus, as attacks get more sophisticated, defenses must also follow the same pace. In this context, tracing solutions allows collecting run-time data for the software components that can be used for attestation by remote parties.

Several open-source tracers exist in the literature. Examples include the Unix-based *ftrace tool*

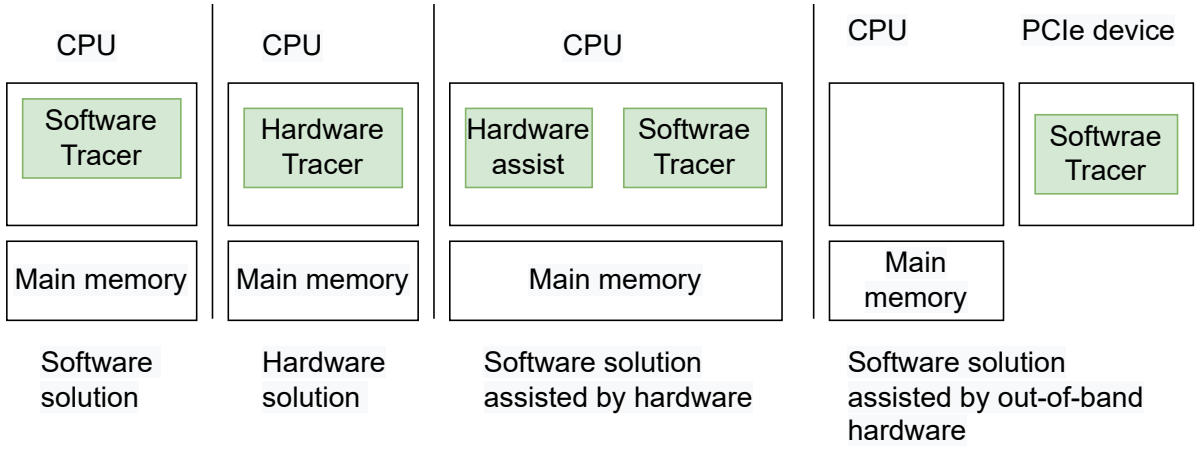


Figure 2.1: Overview of tracing solutions.

that provides static and dynamic tracing. The *SystemTap* tracing tool provides dynamic tracing through the use of Kprobes, Jprobes, and Uprobes [28]. These have traditionally been TRAP-based tracing methods. However, it has been shown that leveraging such techniques consumes a significant amount of resources in the host device for large software runs, thus, making their integration infeasible for the resource-constrained edge devices envisioned in our scenario. Another example of dynamic tracing is DTrace but a visual survey of its code reveals that it offers very limited optimizations, which limits its applicability.

Linux Trace Toolkit Next Generation (LTTng) tracing adds up considerably the collective tracing impact on the target software for long runs, in resource-constrained and high throughput environments, such as embedded network nodes and production servers [41]. An important aspect regarding tracing is the need for filtering due to the large number of generated data [41].

Based on the above, in what follows, we provide details on various tracing approaches - leveraged in remote attestation - as can be seen in Figure 2.1.

2.2 Software Tracing

SMART [22], is a static attestation scheme that establishes a root of trust in the edge device. SMART targets platforms that execute code from external memory. The tracing code and key used to ensure traces authenticity reside in internal read-only memory (ROM) and both are protected by access control policies of a memory protection unit (MPU). Upon receiving an attestation request, SMART executes the tracing code in ROM, which reads a region of code memory and computes an HMAC of the content to be provided in the attestation response. Then the attested code executes atomically.

C-FLAT [2] uses a Trusted Execution Environment (TEE) that hosts a tracing program, which collects data of another application and constructs the control-flows for it towards verification via control-flow attestation. This is a step forward in protecting the software stack compared to TPM-based attestation that only validates the initial device state. Moreover, as the tracing software is isolated by the TEE hardware it maintains the trustworthiness of the collected data. However, the control-flow tracing itself is based on instrumenting the programs such that any control-flow instructions would transition to the TEE. This design decision has a performance hit due to crossing the boundary to the TEE and the intrusive changes to applications that might not apply to all users.

More recently, DIAT [1] was introduced to perform an efficient control-flow tracing. DIAT traces integrity-relevant control-flow events, which are instructions that transfer the flow of a program's execution from the current instruction's address to an address determined at run-time. Thus, DIAT also relies on instrumentation, yet, it filters many control-flow events to reduce the performance impact of the tracing mechanism.

2.3 Hardware Tracing

Encouraged by C-FLAT's success, prior work used hardware-based tracing to improve control-flow run-time collection. For example, LO-FAT [21] introduces new hardware extensions to common processors to allow tracking of control-flow events while performing hash calculations parallel to program execution to report the executed control-flows of programs. Furthermore, LO-FAT supports the control-flow acquisition of legacy programs as it removes the need for binary instrumentation, which is required by C-Flat.

Further building on LO-FAT, Artium [45], is a recent work that showed **time-of-use-to-time-of-check attacks on control-flow attestation**. In a nutshell, an adversary can interleave benign instructions with malicious ones in-between attestation requests. For example, this can be done by replacing the original memory interface with an interface to an attacker-controlled memory controller. Thus, the adversary can execute malicious code when an attestation process is not currently running. Artium solves these attacks by proposing new extensions to the processor, such that like LO-FAT control-flows will be measured, however, Artium also measures every retired instruction. Thus, all code is attested for in the cost of higher performance impact and more intrusive changes to existing processors.

In ASSURED, to encourage adoption of our approach, we focus on off-the-shelf processors, therefore, we do not utilize end-to-end hardware-extension-based tracing mechanisms. Instead, we focus on existing hardware-assisted mechanisms such as ARM Coresight, as discussed next.

2.4 Hardware-assisted Software Tracing

ARM Coresight is a hardware-assisted tracing component that can be used to reduce the performance impact of control-flow tracking, while also removing the requirement for instrumentation of programs. Specifically, with Coresight tracing enabled, the CPU maintains a circular buffer and writes configured set of instructions that were executed by the processor pipeline. A software tracer can use this information with knowledge of the instructions and structure of the program being traced to recover the higher-level control-flow information. We provide the details of this tracing method in Chapter 5.

In the same direction, another hardware-assisted mechanism is Intel Processor Trace (PT). Intel PT is an architecture extension introduced and implemented by Intel processor micro-architectures 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 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 and analysis. However, real-time control-flow attestation implementation also exists. 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 control-flow of traced execution, which is similar to our tracing mechanism with Coresight. Yet, the ASSURED tracer recovers the operating system information through online forensic analysis.

Hardware-assisted tracing solutions such as Intel PT and Coresight generate very large amounts of data when tracing, so sophisticated filtering and optimized parsing are necessary to isolate the relevant information and decode it before it is overwritten by new data (retired instructions). This is where ASSURED comes into play: By leveraging advanced (software-based) parsing techniques, we can extract targeted information and narrow down the context of a trace in a very small amount of time. **These include filtering by a 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 an internal subset of a function**.

2.5 Out-of-band Tracing

Recent advances on out-of-band tracing solutions such as those used by Nvidia AppShield [32] demonstrate a powerful tool to infer **run-time information on systems without exposing the tracer to an attacker**. *The reason is that there is physical isolation between the tracer and the software running on the devices. Thus, even compromising the device including the TEE will not disable the tracing*.

The main idea behind out-of-band tracing is to utilize the hardware to perform **Direct Memory Access (DMA)** and **introspect into the main memory used by software running on the CPU**. This has the advantage of a reduced attack surface and not competing with the software executing in the CPU for hardware resources, thereby limiting the tracer performance impact on the software. However, it requires additional hardware support such as a dedicated PCIe device with sufficient compute capabilities to run the tracer. Despite recent advancements in embedded PCIe devices.

In ASSURED, we choose not to include such a tracing component to encourage adoption by utilizing popular and widely-used platforms accessible to many use cases.

2.6 Intrusive vs. Non-intrusive Tracing

Tracing is a powerful tool to recover high-level information, which can be used to enhance security as performed by the ASSURED ecosystem. *Unfortunately, tracing can also potentially cause a degradation in the overall performance of both programs being traced, and in extreme settings the entire system*.

Traditionally, it is common to consider two forms of tracing: *intrusive* and *non-intrusive*. **Intrusive tracing corresponds to tracers that require changes in the programs being executed**. For example, instrumenting programs through static or dynamic binary rewriting, or even through compile-time transformation passes are ways to introduce new logic to existing programs. This added logic can be used to capture relevant information about the programs, and the entire system's state. **Non-intrusive tracing, however, does not require changes in the programs and uses different methods to capture the necessary information on the programs**. For

example, through hardware capabilities such as ARM Coresight and Intel PT, or through operating systems management features such as signals and traps being introduced into programs execution.

However, trap-based methods are known to incur high-performance impact due to a large number of context switches between real program execution, and the trap logic execution states.

In ASSURED, our goal is to both **reduce the tracing performance impact** and **provide seamless integration with the programs** without forcing developers to change them in advance. This will essentially unlock the possibilities of remote attestation and it would enable the easy employment of such advanced techniques offered by ASSURED. *To the best of our knowledge, ASSURED is one of the first initiative towards providing such a purely software-based Tracer capable of providing high-level of detail and accuracy in tracing a multitude of a system's functional properties.*

Thus, we consider **hardware-assisted tracing capabilities whenever they are available in commodity processors and use a software-based tracer to efficiently recover information made available by the hardware**. Since the hardware is highly optimized and has negligible impact on the system's performance, and the tracer can run in parallel to other programs without interfering with their execution states, we achieve both goals: non-intrusiveness by not changing the programs, and reducing the need for trap-based tracing using hardware whenever it is available.

2.7 ASSURED Software-based Tracing Approach Adoption

Hardware-based tracing is less flexible compared to software-based tracing as the hardware needs to provide support for the tracing and is not as straightforward to make any necessary adaptations (this usually requires additional certification which is not a preferred process by the hardware vendors). Thus, hardware-based tracing alone cannot support all requirements in ASSURED, e.g., for tracing both control-flows and configuration integrity of the platform in an efficient manner. Furthermore, in ASSURED we use off-the-shelf components, thus, some recent advancements in hardware-based tracing cannot be used as they are not implemented by vendors' platforms. Hence, **the decision to provide purely software-based tracing capabilities capable of delivering high accuracy even for low-level tracing down to a single function**.

In ASSURED, **dynamic tracing functionalities** are provided, as **software programmable components**, 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' behavior for detecting cheating attempts or if any type of exploits to the program and data memory. This provides the trusted anchor with the compiled control- and information-flow graphs (CFGs & DFGs) that represent the run-time state of a remote device, against the configuration and execution properties of safety-critical components. ASSURED advanced tracing techniques are software-based and leverage embedded OS introspection agents capable of traversing the entire physical memory of a CPS, via Direct Memory Access, for known execution signatures. These tracing functionalities will be fully programmable, enabling the priority of the ASSURED Framework towards dynamic adaptation of tracing.

Chapter 3

System Model

In this chapter, we summarize the system model, adopted in ASSURED and was first presented in D3.1 [6], and in particular the Trusted Computing Base (TCB) that comprise the architecture of the edge devices. Recall that, as was described in the defined trust models of ASSURED [6], the core requirement for the correct operation of all attestation enablers is the **trustworthiness and authenticity** of the employed Tracer. This is the reasoning behind placing the Tracer as part of the ASSURED Device TCB.

In this context, the high-level architecture of each edge device is depicted in Figure 3.3. As can be seen, the trusted and untrusted components are **isolated via the underlying hardware**. We discuss the full details of the ASSURED system model next and provide also the high-level conceptual architecture of the proposed tracing mechanism in Section 3.1.

3.1 Use of Tracer in ASSURED Ecosystem

Figure 3.1 depicts a high-level overview of the tracer positioning within the ASSURED ecosystem and its interaction with other ASSURED components - especially the **Attestation Agent (as part of the TPM-based Blockchain Wallet [17])** that is responsible for "*reading*" the attestation policies from the ledger, extract the resources (systems properties) to be monitored for then triggering the tracer to initiate its operation. This relationship also includes the **verification and signing of the extracted traces by the TPM-based Wallet** based on the already created Attestation Key (AK). Recall that in order to guarantee the **correctness and authenticity** of the received traces, the usage of the AK is protected through a policy that only accepts traces signed from the (pre-installed) tracer's symmetric key [6].

The other ASSURED component that continuously interacts with the tracer is the **Attack Validation** component (Figure 3.2). As described in D2.7 [14], ASSURED also offers **security forensics and fuzzing capabilities for better resilience and mitigation planning**. More specifically, in case of a failed attestation report, which is an indication of risk, the Attack Validation component collects the system traces so that it can perform a detailed memory fuzzing towards identifying the exact attack path that was exploited by the adversary, thus, possibly leading to the identification of new (zero-day) exploits for which new attestation policies will need to be compiled and deployed. Towards this direction, as can be seen in Figure 3.2, the Attack Validation component may also request more detailed traces (including additional information on the internal memory view of the target device), thus, allowing for a **multi-level detailed tracing**. The motivation behind this approach is that the employed attestation should not target the whole stack of a system's architecture unless there is an indication of suspicious activities. More precisely, the scope of monitoring

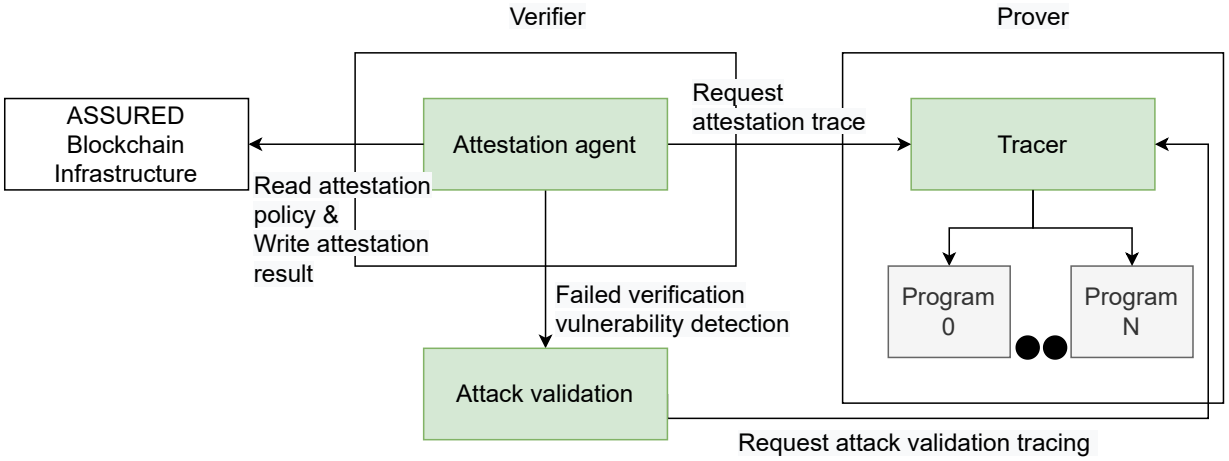


Figure 3.1: High-level Overview of Tracer Interaction with the ASSURED Ecosystem

and tracing granularity should be increased only when additional evidence and information need to be collected, on a detected incident, for the assistance in finding the province of the attack as well as in the development of new enforceable attestation policies that should be able to catch this newly identified threat [36].

As it pertains to the overall ASSURED flow of actions (Figure 3.1), the Verifier is the entity responsible for supporting the different ASSURED attestation schemes [8]. The Verifier, upon reading an attestation policy, initiates a remote attestation process based on the extracted policy, which is populated in the Blockchain infrastructure of ASSURED by the Security Context Broker (SCB) based on the current scheduling policy and risk level assessed per each edge device and the global ASSURED ecosystem. This policy is sent to the attestation agent, which allows invoking the requested attestation task: either CFA or CIV. Additional parameters are also read from the Blockchain, including a fresh nonce and parameters that are required for the attestation task, such as a process identifier that the tracer will trace for the CFA attestation scheme. The attestation agent sends the tracer a request to compute the trace according to the extracted parameters. The tracer computes a trace, signs it, and passes it back to the attestation agent for verification.

Once the traces are received, the Verifier **validates the signature of the traces and performs the attestation validation based on the content of the traces**. For example, a CFA Verifier validates that the control flow represents a benign execution pattern. The attestation results are later written to the Blockchain, which facilitates further attestation schemes for all the deployed edge devices in ASSURED.

Finally, as aforementioned, in case of a failed validation, the Attack Validation component is triggered by the Verifier. The attack validation component then passes requests to the tracer to capture the values of core variables. These variables are also passed as a signed trace, such that the signature can be validated. Finally, the attack validation component uses this data to infer vulnerabilities in the program logic in order to analyze attacks sources and potential remediation.

3.2 Edge Device Hardware

Hardware components: We envision ASSURED edge devices would contain in their package the following hardware components: a Trusted Platform Module (TPM), a TrustZone-supported ARM processor, volatile memory, and non-volatile storage, and finally a network interface card.

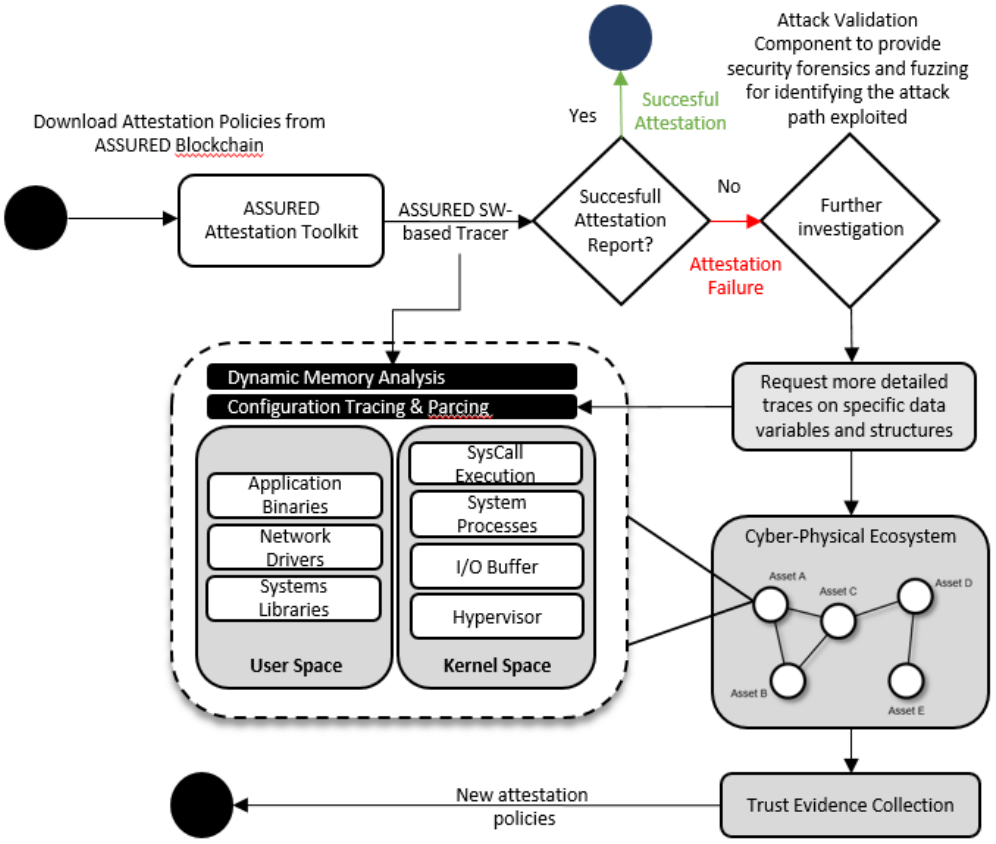


Figure 3.2: ASSURED multi-level detailed tracing

To support efficient and fine-grained control-flow tracing of programs on edge devices we require the processor to support the CoreSight tracing extensions.

ARM TrustZone background: ARM TrustZone [31] provides access control capabilities, enforced by the hardware that lets programs switch between two *worlds*: **a secure world, and a normal world**. Specifically, the processor enforces isolates across the worlds providing a secure execution environment for programs such that programs executing in the normal world cannot leak or tamper with code and data that are part of the secure world.

Internally, the processor includes a non-secure (NS) bit that is used to identify the currently active world. That is, a value of 0 in the NS bit means the processor can only execute programs in the normal world, whereas a value of 1 means the processor can only execute programs in the secure world. Furthermore, the processor partitions the volatile memory between the two worlds. Effectively, each world has access to its region of volatile memory. Yet, programs can opt in to share memory regions, which allows passing requests and responses for computations being made by the secure world to the normal world and vice versa.

We deploy the tracer in the secure world to provide strong security guarantees. Thus, even if the normal world is compromised it does not affect the tracer's functionality and ability to attest programs executing in the normal world.

ARM CoreSight background: The CoreSight architecture provides a system-wide solution for real-time debugging and tracing capabilities enforced by the processor. Coresight has evolved

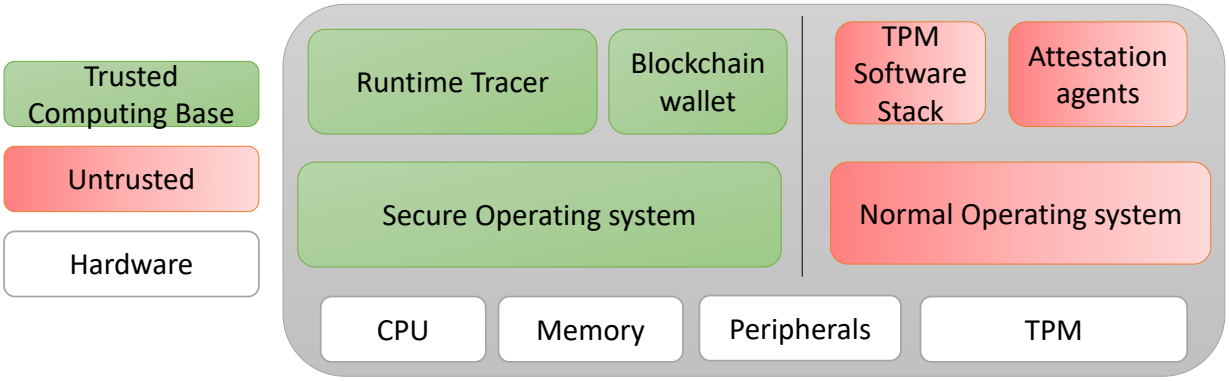


Figure 3.3: Overview of edge device components

and includes different tracing capabilities such as Embedded Trace Macrocells (ETMs), AMBA Trace Macrocells, Program Flow Trace Macrocells (PTMs), and System Trace Macrocells (STMs).

Specifically, Coresight enables configuring the processor to store retired instructions in a circular log buffer in a dedicated region in the main memory. Thus, the tracer can read the memory and based on the instructions infer control-flow decisions made by the traced programs to later create a trace report of control-flow execution towards control-flow attestation (CFA) by a Verifier.
In ASSURED we focus on PTMs and ETMs to provide hardware-backed tracing capabilities. PTM and ETM can be used to only trace instructions which relate to the control flow, thereby reducing the tracing performance impact and the load on the circular buffer.

Trusted Platform Module (TPM): A TPM is a hardware module, which is traditionally used to act as a hardware root-of-trust (RoT) for a platform. In ASSURED, a TPM is used as the building block of the TPM-based Wallet [17] which is responsible for the **correct creation and management of all cryptographic material** (Attestation Key, DAA Key, symmetric communication key, etc.), **key usage protection** (through the specification of adequate policies [6]) as well as the **continuous authentication and authorization of the host device through the issuance and management of Verifiable Presentations** based on credentials that include the necessary attributes needed for performing the various ASSURED operations (and have been validated by the Privacy CA during the Device Registration and Enrollment phase). The latter is used to access smart contracts for the different attestation primitives provided in ASSURED. The ledgers are also used to record the results of the attestation Verifiers to the blockchain.

The TPM is used in ASSURED to provide hardware-based protection, which is traditionally considered more secure compared to software-based protections. Thus, **ASSURED utilizes the Trustzone architecture to combine trusted execution environments with TPM functionality**. Specifically, the TPM provides encryption and decryption for any user data and secure storage. Access to the data is controlled by a securely stored cryptographic key. Furthermore, the TPM includes Platform Configuration Registers (PCRs), a memory location in the TPM that has some unique properties. For example, the size of the value that can be stored in a PCR is determined by the size of a digest generated by an associated hashing algorithm. The TPM technology required by ASSURED use cases is used to enable secure remote attestation and authentication. This enhances the confidentiality and integrity of the exchanged data. Platform authentication will be performed by the TPM and is required in the ASSURED framework to prove that the platform is with a legitimate identity.

This design choice enables the concept of **Zero Trust security principle**, with the need of "*Never Trust, Always Verify*", for which ASSURED bootstraps vertical trust for all devices and users in

the target SoS by enabling **continuous attestation, authorization, and authentication** prior to being allowed to communicate and/or be granted to data or resources. This type of TCB allows the flexibility of being able to **guarantee the correctness of the tracing features of ASSURED (through the TEE)**, as the cornerstone of the entire framework correctness [8], and the correct **(self-) issuance of cryptographic material and verifiable credentials through the use of a TPM as the building block of the ASSURED TPM-based Wallet** [17].

We have to highlight that while ASSURED implementation will be instantiated based on the use of OP-TEE, as the underlying trusted execution environment (Section 3.3.1), and TPM, all of the attestation models and secure data management schemes to be designed (as part of the WP3 and WP4 research activities) should be agnostic on the type of TCB to be considered.

3.3 Edge Device Software Stack

The tracer is a software component, which is part of the Trusted Computing Base (TCB) [6] of the edge device. The tracer continuously introspects the programs executing and collects information to be used as part of the attestation schemes of ASSURED: **control-flow graphs for the control-flow attestation and hashes of code pages for the configuration integrity verification**. The tracer signs the traces in the secure world passes them to the TPM-based Wallet for verification which then forwards them to the Attestation Agent of the Verifier for performing the requested attestation operation. We provide more details on the tracer component in Chapter 4.

3.3.1 Secure World Software

The Open Portable Trusted Execution Environment (OP-TEE), is a widely-used and popular open-source reference implementation for TrustZone-based TEEs. OP-TEE is designed as a companion to a non-secure Linux kernel running on ARM processors. OP-TEE implements TEE Internal Core API v1.1, which is the API exposed to Trusted Applications, and the TEE Client API v1.0, which is the API describing how to communicate with a TEE. Those APIs are defined in the GlobalPlatform API specifications [38]. Furthermore, OP-TEE can run on an emulator as well as on numerous comparatively inexpensive platforms [33].

The non-secure OS is referred to as the Rich Execution Environment (REE) in TEE specifications. It is typically a Linux OS flavor, which is used by the different use cases considered in ASSURED. Yet, it also allows other operating systems to be used together with OP-TEE such as Android. OP-TEE is designed primarily to rely on the Arm TrustZone technology as the underlying hardware isolation mechanism. However, it has been structured to be compatible with any isolation technology suitable for the TEE concept and goals, such as running as a virtual machine or on a dedicated CPU. Finally, OP-TEE provides the following security design goals.

Isolation: The TEE provides isolation from the non-secure OS and protects the loaded Trusted Applications (TAs) from each other using underlying hardware support.

Small TCB: The TEE should remain small enough to reside in a reasonable amount of on-chip memory as found on Arm-based systems.

Portability: The TEE aims at being easily pluggable to different architectures and available HW and has to support various setups such as multiple client OSes or multiple TEEs.

OP-TEE builds on top of the ARM Trusted Firmware, which is used as the basis for many software architectures found on commercially available devices. Furthermore, it is actively maintained and well documented.

In ASSURED, we execute the user programs in the normal world as we describe next. The secure world is unaffected by their execution as the hardware enforces isolation across the worlds. Thus, we place the tracer in the secure world to enforce the trustworthiness of the generated traces that are used by the different Attestation Agents to provide assurance of the edge devices' security.

3.3.2 Normal World Software

In ASSURED, we assume the normal world is using a Linux-based OS. Linux is a widely used, family of Unix-like OSs, which are based on the Linux kernel. We assume the normal world will manage the drivers, e.g., to manage I/O devices such as the network and the storage. Thus, the traces must be sent to external devices through the normal world, and to ensure their trustworthiness we employ cryptographic operations as we discuss next.

Furthermore, as Linux is popularly used, it has support for many libraries and frameworks that enables quick and simple deployment of the different applications considered in the context of ASSURED's use cases. Finally, as the normal world manages all devices, this includes the TPM and thus the TPM software stack (TSS) is entirely placed in the normal world.

TPM Software Stack (TSS): The TSS is a Trusted Computing Group (TCG) software standard that provides a standard API for accessing the functions of the TPM. Any local or remote communication with the TPM-based Blockchain wallet is achieved using TSS. TSS provides all the necessary TPM commands for various crypto primitives. It also reduces the programming complexity of applications that desire to send individual TPM commands. The TSS in the context of ASSURED will be providing the necessary standard API stack for "driving" the TPM towards supporting features such as registration, login, and authentication using trusted hardware/wallet for accessing the ledger, smart contract read/write/execution, management of proofs, e.g., identity, reputation value, status, in consensus algorithm. The TPM will be coordinated by the host software towards forming the TPM-based Blockchain Wallet in the ASSURED framework. The TPM-based Blockchain wallet communicates with the rest of the Blockchain components of the ASSURED ecosystem to securely access ledgers content. Moreover, smart contracts are read and executed through the Blockchain wallet, and their security and privacy are enforced by the TPM.

Attestation Agent: The Attestation Agent is the "*bridge*", running on the host device, for enabling the correct orchestration of the tracer based on the attestation tasks that need to be executed. These can either be Control-flow Attestation (CFA) that analyzes control-flow traces to verify the state of the Prover; Configuration Integrity Verification (CIV) the monitors the configuration state of the prover device; Direct Anonymous Attestation (DAA) which can verify the membership of a device for a group without disclosing privacy-related information; Swarm Attestation which can attest large groups of devices; and, Jury-based Attestation that utilizes multiple Verifiers if the Prover disagrees on the result of the first Verifier.

3.4 Trusted Computing Base (TCB)

The TCB refers to a set of **hardware, firmware and software components** that are supposed to be resistant to any type of attacks specified in the adversarial model and are assumed secure by default. The TCB is required for implementing primitives like attestation that provide security guarantees and is crucial for the security of the entire system [26, 37]. In this section, we provide

a summary of the TCB components in edge devices in ASSURED. The full description is available in D3.1 [6].

Hardware TCB: We assume the SoC hardware components including the processor, system buses, memory, and peripherals as part of the TCB. Moreover, we assume the existence of a TPM and include it in the TCB. We assume the TPM provides trustworthy and correct security engines. ASSURED core services such as attestation and blockchain wallets are implemented based on the TPM services. We assume the ARM processor in the edge devices supports the Trustzone architecture to provide a trusted execution environment (TEE), which enables a hardware-assisted isolated execution environment.

Software TCB: First, we assume the edge devices' firmware as part of the TCB. The firmware is responsible for booting the edge devices, before loading both the secure world operating system and normal world boot loader and operating system. Furthermore, we assume that each boot stage in the firmware measures and validates the next boot stage, and extends the loaded images hashes into the TPM to later provide an attestation report on the loaded software of the system, including the secure world OS image. As mentioned, the firmware implements the security monitor, which provides a secure context switch between the normal and secure world, and prevents leaking the internal states of the secure world to the normal world. The tracer and TPM wallet, which comprises the core functionalities of the ASSURED framework, are crucial to the security of the whole ASSURED architecture and are assumed as part of TCB running inside the secure world. Both are executed as user space programs. The integrity of the tracer and TPM wallet is verified before loading them into the secure world by measuring the deployed binaries and validating them.

Chapter 4

ASSURED Tracer Architecture

A high-level overview of the tracer architecture is depicted in Figure 4.1. The tracer is a software component executing in a Trusted Execution Environment (TEE), and specifically the secure world in ARM Trustzone. Therefore, the tracer is part of each edge device's TCB acting as a trust anchor, which is inaccessible to attackers who potentially can compromise the rest of the device. The tracer runs continuously as a daemon process in the TEE, tracking the volatile physical memory of the device. Furthermore, the tracer has an untrusted interface executing in the normal world. This is the only communication interface exposed to the tracer and is leveraged by the Attestation Agent for controlling which tracing policy should be invoked by the tracer (depicting which specific system validation properties to be traced [11]) and receive the traces to be used towards attestation verification.

Please note that the use of ARM Trustzone is the current instantiation of the tracer. However, we have to highlight that the design of the tracing capabilities of ASSURED can operate within any type of TEE that offers isolation capabilities so that tracer is assumed trusted.

4.1 Adversary Model

We assume a strong threat model where the adversary may compromise programs outside of the secure world. The adversary may try to leak or tamper with data being processed in the normal world. However, the adversary cannot tamper or leak data in the secure world. Under these assumptions, the tracer guarantees the generated traces to be correct and represent the state of the normal world programs. Using the traces, the attestation agents can detect attacks and report them to the ASSURED cloud backend framework.

Examples of attacks that can be considered in ASSURED for which the detection will leverage produced traces are the following:

- **Security Misconfiguration:** A malicious modification in the configuration of a device, enabling access by a malicious party. This is relevant to all use cases, since all devices need to be protected from threats resulting from an untrusted configuration. This is mitigated by employing the newly developed Configuration Integrity Verification (CIV) [8] based on types of traces as described in Chapter 6.
- **Vulnerable and Outdated Components:** Outdated components are notoriously susceptible to security threats and attacks, since they may be loaded with firmware or application software that has not been patched in order to be protected from newly identified threats.

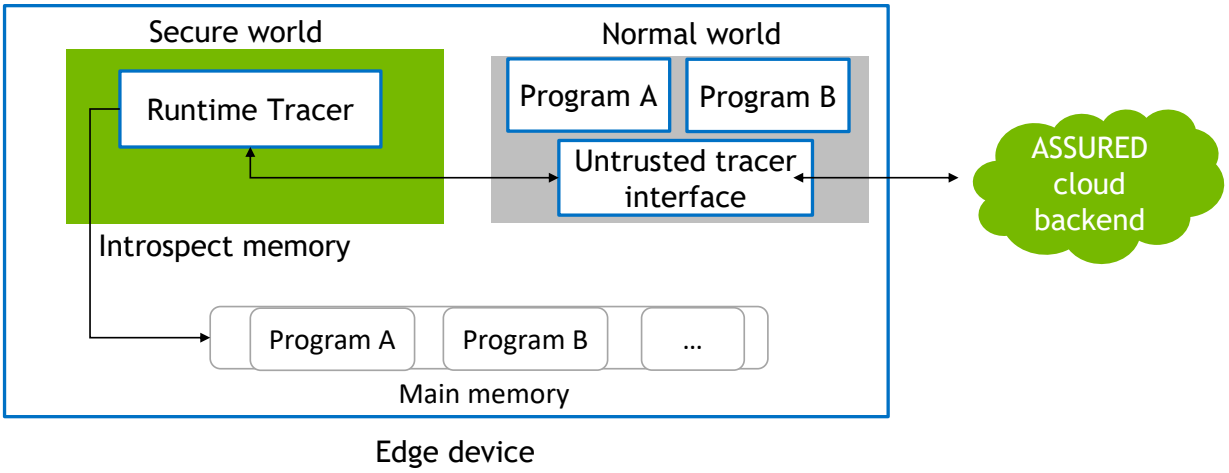


Figure 4.1: Overview of the tracer

This falls under the category of software vulnerabilities that can be mitigated by attestation services, verifying that the components possess the latest software versions.

- **Insecure Design:** Refers to architectural flaws related to design methodologies that cause vulnerabilities. Therefore, this category does not refer to a specific kind of vulnerability, but design and coding practices that make devices and systems more susceptible to various kinds of attacks. These can be detected with the help of attestation, so that they can be patched in future versions of the software.
- **Cryptographic Failures:** Encompasses failures and attacks related to cryptographic functions and cryptographic primitives. This is of particular importance in the smart satellite use case, which employs a variety of cryptographic schemes and protocols for communication of the base station with the deployed satellites and can be mitigated by the ASSURED traffic verification and attestation services.

4.2 Operational Model

In what follows, we give a detailed overview of the steps performed by the tracer as depicted in Figure 4.1. We first give a description of the internal actions performed, as building blocks, prior to showcasing the complete chain of actions/events that take place in ASSURED (Section 4.4) through a concrete example from one of the modeled use cases.

Initialization: The tracer is pre-configured with specific device information: **symbol names and their corresponding offset in memory for both the normal world's operating system and programs that are executed in the normal world**. This information is received by the System Administrator as part of the *system description* that needs to be provided per each software asset when registered in the ASSURED Risk Assessment component. Such information is needed so that the appropriate risk graph can be calculated [15] based on which an optimized set of attestation policies will be calculated that will also contain the tracing policies to be deployed to the edge devices.

Memory acquisition: We allow the tracer to operate both in the secure world with enhanced security guarantees inherited by the TCB (enforcing the isolated execution environment) as well in the normal world, which simplifies the deployment of the tracer.

In the normal world, we assume the existence of a Unix-based operating system. We use a kernel module to expose access to the physical memory of the platform via `ioctl()` requests. Effectively, it acts similar to a `memcpy()`-like interface passing requests to read a region of the physical memory starting at a requested address and a given byte-length storing the content into a buffer in the address space of the requesting application. **Note, this is only possible to normal world operation mode since communication with kernel modules cannot be secured when made by programs executing in the secure world**.

Instead, in the secure world operational mode, we assume OP-TEE is used as the secure world operating system. *However, this is only used as a reference point for the implementation of the tracer. Our design is agnostic on the type of trusted execution environment to be used*. The OP-TEE operating system has access to the entire physical memory of the platform enforced by the ARM trusted firmware. In this mode, we provide an extension to the OP-TEE operating system to map a requested region in the normal world's part of physical memory to the secure world. This is done in per-page (4 KB) granularity in a streaming fashion. That is, each page is mapped, read into a software buffer, and unmapped. Thus, we gain access to the normal world's memory in the secure world and can introspect and analyze it to provide secure traces as discussed next.

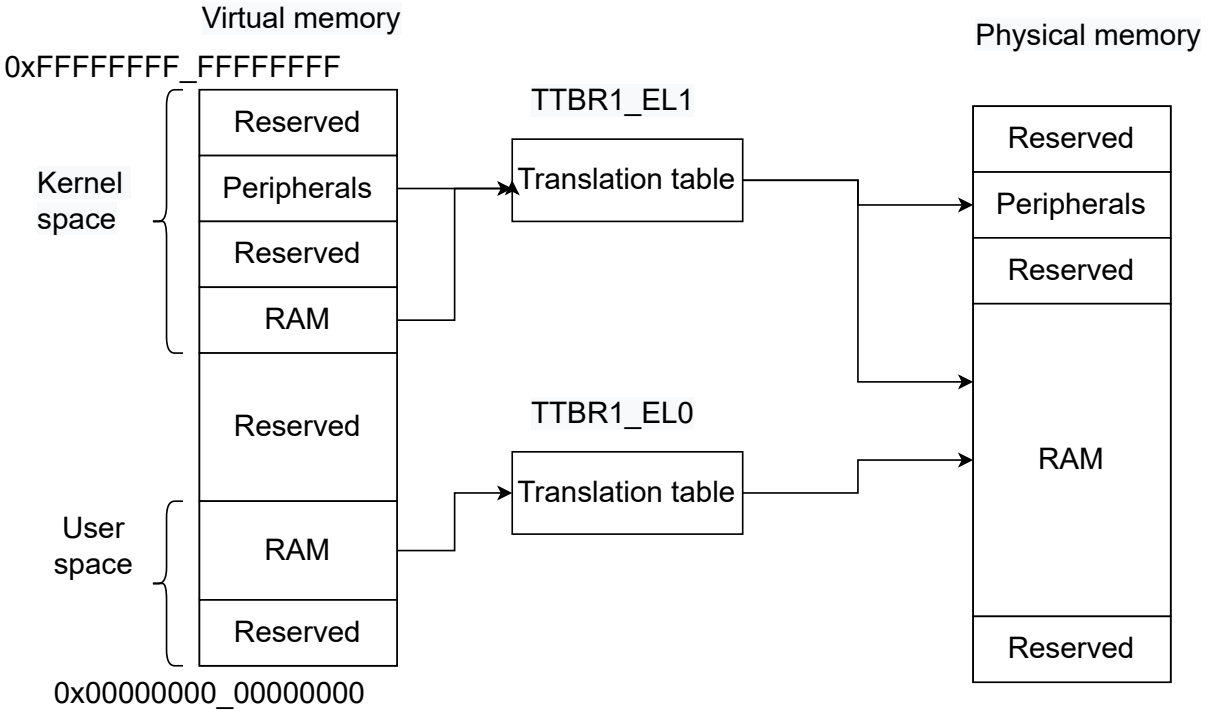


Figure 4.2: Overview of the page table mapping virtual-to-physical pages

Virtual-to-Physical Translations: Interestingly, access to the physical memory is insufficient to provide control-flows of applications and configuration integrity of the platform. The reason is that programs and operating systems operate in virtual address space [24]. Current processors provide paging capabilities through virtual-to-physical page tables that map each virtual address to a physical address, thereby creating two address spaces (see Figure 4.2). Specifically, each memory access instruction that a program issues undergoes a page walk translating the virtual address into a physical one, in page granularity (4 KB region). This translation is performed by current processors with different architectures, such as ARM and x86. This includes the type of micro-controllers assumed in ASSURED; *this is not a strong assumption made by the project*

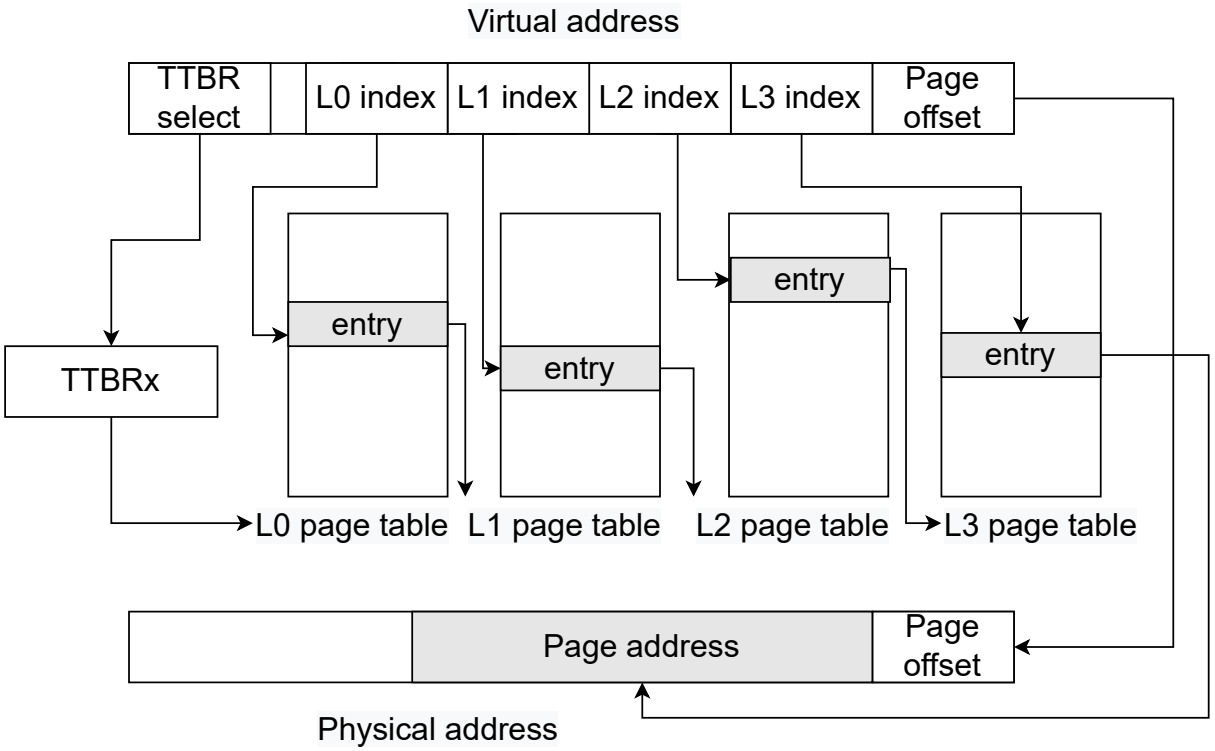


Figure 4.3: Overview of address translation from virtual to physical addresses in ARM processors.

as ARM is currently considered the most prominent type of processor leveraged in commodity embedded systems encountered in many application domains. However, in ASSURED, this will also be considered in other types of processing units and operating systems such as in the "*Secure Aerospace*" use case where a specific type of operating system is considered called KUB-OS [18]. Overall, this involves using the virtual address as an indexer to a hierarchical page table (see Figure 4.3). The processor is combined with a hardware mechanism to perform the page table walk efficiently and caches the translation in a structure called the translation look-aside-buffer (TLB) for faster access to future memory access instructions to the same page. The tracer implements the same page walking mechanism, yet, in software that is running as part of ASSURED TCB.

For the tracer to analyze the control-flow or configuration integrity of programs executing in the platforms, it must know the exact structure of the page tables used by the edge devices. The tracer starts by using a specific known address of a symbol that was embedded into it during the initialization phase - *this is something that can also be read from the Blockchain (through the provided System Description)*. This essentially represents the starting point of the tracer for a specific codebase (software function) as defined by the downloaded attestation policy. The symbol is for the `init_task`, the first task that the Linux operating system executes. Interestingly, this address is in virtual memory and the content of the physical memory is always the same as it describes the initial task, which has the same name, process identifier, etc. The tracer uses this information and partitions the main memory into 4 KB regions representing pages. The tracer then iterates over all regions considering each page as a candidate address for the page table root address. Each candidate is treated as the entry point to the page table, and the tracer implements a software page walk, thereby translating the address from virtual to physical. A successful translation of the `init_task` address and reading the content of the task validating it is correct means the tracer found the correct page table address and can use it to translate future

virtual addresses.

To conclude, the output of this phase is the physical location of the first level of the page table used by the Linux kernel. This address is used by the tracer to translate virtual addresses into physical ones, before accessing the content stored in these addresses. The latter is used by the tracer to infer information on the system, specifically on software executing in the normal world as explained next.

Recovering System-Wide Information: Utilizing this exact symbol information and virtual-to-physical translation capability the tracer can recover high-level semantic information about the edge device and programs executing in it. It does so by using apriori knowledge of the operating system services and their respective implementation. For example, to recover the currently active processes in an edge device the tracer uses the `task_struct` structure defined as part of the Unix-based OS kernel. The `task_struct` contains information about a specific process such as its unique identifier, its name, its virtual memory addresses, etc. Furthermore, the `task_struct` acts as a linked list node in a Unix-based OS and, thus, contains a pointer to the next `task_struct` node. While these pointers contain virtual memory addresses, the tracer armed with the knowledge of virtual-to-physical address translation can traverse the linked list, perform software address translation and recover the details of all currently running processes on the device.

Similarly, the tracer can **infer information about the state of the normal world and programs executing in it**. For example, using **symbol information and virtual address translation** the tracer can access data structures stored in the Linux kernel memory region that are used to track loaded kernel modules. Using data stored in these data structures, the tracer can infer which kernel modules are currently in use. This is one of the core innovations in ASSURED that also enables us to perform **multi-level execution tracing**; including both the monitoring of the loaded binary hashes (to be used in Configuration Integrity Verification - Chapter 6) and low-level tracing of mission-critical processes even at the kernel level (to be used in Control-Flow Attestation - Chapter 5). The outcome of the tracing process is in all cases the **extraction of the corresponding CFGs (in the context of CFPA) and/or binary hashes (in the context of CIV)**. The motivation behind this multi-level approach is that the employed attestation should not target the whole stack of a system's architecture unless there is an indication of suspicious activities. More precisely, the scope of monitoring and tracing granularity should be increased only when additional evidence and information need to be collected, on a detected incident, for the assistance in finding the province of the attack as well as in the development of new enforceable attestation policies that should be able to catch this newly identified threat.

Furthermore, as we discuss more in Chapter 6, the tracer can also iterate over all active processes in the normal world and read the data structures containing information on their mapped virtual address spaces to acquire all loaded libraries and binaries. This is used as part of the configuration integrity verification attestation scheme with a verifier that checks whether the in-memory state of the platform matches the expected state that was configured to be deployed in the edge devices.

Continuous Non-Intrusive Tracing: The tracer is configured to continuously run in the secure world and traces the normal world. Therefore, the tracer's secure world component, which reads the normal world's memory content does not require changes to the normal world's software stack. We envision the tracer would run in parallel (on a separate CPU core) and result in a negligible impact on the performance of programs executing in the normal world. The reason is that the tracer memory accesses might only affect micro architecture caches and buffers in the processor due to additional memory accesses being performed.

Furthermore, as the tracing is non-intrusive, a malicious adversary, controlling programs in the normal world, is unaware whether the tracing is currently active or not (thus, bypassing also the limitations of current techniques based on code instrumentation). Furthermore, the active tracing method, e.g., CIV, or CFA, cannot be detected by the adversary since it is completely isolated in the secure world. Indeed, the isolated execution guarantees, offered by the secure world, allow the invocation of efficient and powerful tracing components. The tracer does not require changes in programs executing in the normal world. For example, C-FLAT [2] required intrusively instrumenting programs to log their executed control flows towards support for control flow attestation. However, with the ASSURED tracer, we aim to track the control-flows in a non-intrusive manner, thus, achieving high performance with minimal overhead.

Tracing Policy: The tracer maintains an active security policy that should be enforced on the device. This policy is effectively what triggers the tracing mechanism, that is currently employed, to detect specific properties depicting the state of the device that need to be attested. As aforementioned, these properties can either be configuration properties or execution behavioral properties - depending on the type of threat that we want to detect. **The output of the tracing is a trace report that is passed to the respective attestation Verifier**.

As described in D1.2 [9], this tracing policy is part of the optimized set of attestation policies outputted by the Policy Recommendation Engine [12] based on the identified threats and risk level for each asset (of the SoS) as well as the entire service graph chain. Essentially, they hold the type of validation properties that need to be attested during run-time in order for the devices to produce the necessary **security claims to prove their level of trustworthiness**. For instance, the name of a specific software binary, function, or a data variable that needs to be monitored in terms of control-flow execution or the IDs of any internal process that try to update this specific variable. All these illustrate possible properties to be traced during run-time (defined as resources in the MSPL-based attestation policies as defined in D2.2 [12]). For the envisioned use cases, examples of validation properties to be attested against specific attacks vectors were documented in D1.3 [11]. This mapping set the scene for the actual experimentation and evaluation scenarios and metrics defined in D6.1 [18].

All the attestation policies, including the definition of the tracing policies, are securely deployed to all devices through the ASSURED Blockchain infrastructure [7]. As described in D4.2! [16] and D4.5 [17], the TPM-Wallet of each device gets notified of a new policy defined, and if it is destined for this device, then it connects to the Blockchain Peer and makes a request for extracting the policy. The policy is then interpreted by the Attestation Agent of the device (acting as the Verifier) and will trigger the Attestation Agent of the Prover device for initiating the tracer for extracting the requested measurements. This communication is performed through a TLS channel. The Attestation Agent provides the only interface to the tracer and communicates with the tracing components over shared memory to pass the tracing policy (type of property to be monitored) to the secure world. We note an adversary controlling the normal world and compromising this interface: However, she cannot attack the confidentiality and integrity guarantees of the traces as the traces are signed with a nonce passed by the Verifier Attestation Agent (extracted through the execution of the $createNonce$ smart contract function [7]). Thus, the attestation agent may detect any violation in the traces due to a bad signature or incorrectly signed nonce passed back to it. The adversary may perform denial-of-service (DoS) attacks, but these are considered out-of-scope in ASSURED since they can be easily identified if there is no reply to the attestation request made by the Verifier. Finally, the attack cannot also alter the tracing policy (change the list of properties to be attested) since the policy is signed by the Verifier's TPM-based Wallet (using its Attestation Key).

Tracing for Attestation: Once a security policy is set and a request for initiating a tracing process for an attestation report is sent to the tracer, it employs the respective tracing mechanism. The tracing uses the virtual-to-physical translation capabilities of the tracer to recover information on the platform based on different tracing data structures. Each data structure is different and the method to recover information differs depending on the tracing type. In ASSURED, we consider tracing for CFA, which is described in Chapter 5, and tracing for CIV, which is described in Chapter 6. The generated traces are signed, as explained next, and sent to the attestation Verifiers. The Verifiers validate both the signature and the traces content. Validating the signature is used to ensure the traces are trustworthy and come from an authentic tracer, and validating the content assures the secure state of the device. For example, in the case of CFA, the Verifier validates the trace does not contain malicious control-flows in the traced program.

Error handling: In case a tracing method fails, either for CIV or CFA, it signals that the device might have been compromised. Therefore, the Attestation Agent sends a signed failed trace to the Verifier, together with a memory dump trace to the attack validation component. The latter is used for performing a more in-depth analysis of the actual attack path that was exploited by an adversary in order to be able to then identify adequate mitigation measures expressed through new attestation policies [14].

Recall that the output of the CFA will provide a Boolean decision regarding a systems configuration and execution integrity with respect to the properties that were attested. If the output is "YES", then this reflects the appropriate statements on the correct and trustworthy execution of the deployed devices (**Operational Correctness**); otherwise, a "NO" attestation report reflects that a deviation from the normal device behavior was detected and the tracer forwards the monitored system traces (that led to this failed attestation) to the Attack Validation component for further processing. This in turn will be fed to the Risk Quantification for the re-calculation of the overall risk and threat vector and for the re-configuration of the security policies.

Towards this direction, the ASSURED Risk Assessment enhancement [15], of the overall RA framework, will be responsible for performing this re-calculation based on the following techniques: a) the **Backwards Chaining** to resolve the given set of constraints and b) the **Cascading Effect** Analysis.

4.3 Traces Security

As mentioned previously and also elaborated in D3.1 [6], one of the core interactions from the Tracer is with the TPM-based Wallet of the host device that acts as the root-of-trust. The main trust requirement that we have for all of the traces (leveraged in any of the ASSURED attestation enablers) is that they **originate from an authentic and valid Tracer that is running in the *secure world* of the same device**. This is crucial for the correctness of the operation considering that all traces, sent to the Attestation Agent, must traverse through the normal (insecure) world, as it has control over communication peripherals such as the network interface card for relaying them to the Attestation Agent of the Verifier device.

To ensure the trustworthiness of the traces, the tracer must rely on **cryptographic operations such as signatures** and the **enforcement of key usage policies through the TPM**. These essentially require placing trust in both the tracer, to be able to manage its cryptographic material, as well as the TPM for safeguarding the use of the Attestation Key to sign only authentic traces. Recall that, as was described in D3.1 [6], it is very challenging to add the entire Trusted Software Stack (TSS) (used for interacting with the TPM) as part of the TCB since this would have a

huge impact on the performance of the entire software stack: As part of the TSS, there are a lot of interactions with system libraries that will reside outside the TCB and, thus, any exchanged data would have to be "*sanitized*" prior to being processed. Furthermore, all traffic is exchanged through an I2C channel (between the host and the discrete TPM chip) which again needs to run in the "*normal world*".

Thus, we use a traditional TSS execution stack running as part of the "*normal world*" with the minimum set of privileges required. Essentially, it means that an adversary (as part of a compromised host) may attempt to manipulate the traces during transit from the tracer to the TPM or even spoof traces, which limits the applicability of the TPM to sign traces on behalf of the tracer. Compounding this issue, in D3.1 [6], we proposed three distinct solutions to this problem so as to enable secure and authentic communication between the tracer and the TPM.

We note that all solutions require placing a unique, asymmetric secret key - for the tracer - with its private part being accessible only to the secure world and public part known to everyone. This key can be provisioned during the manufacturing process, and securely embedded as part of the edge device's hardware, e.g., by placing it in the edge device's fuses. This is the key used for the communication between the tracer and the TPM, denoted as *tracer key*(TRC_{priv}). It is used for signing the traces prior to being sent to the TPM. The TPM, during also its certification to the Privacy CA (as part of the Device Enrolment and Registration phase [16]), creates the appropriate key usage policies for "*binding*" the use of the Attestation Key, as a restrictive signing key, to only sign traces that have in turn been signed with TRC_{priv}. This allows us to make sure that the Attestation Agent of the Verifier is dealing with authentic traces since they will have been signed by the TPM-based Wallet under an Attestation Key (AK) of a TPM with a valid and certified Endorsement Key (EK).

Traces Security	Description
Tracer Signature	in this context, we don't use the tracer key for the communication between the tracer with the TPM-based Wallet but for the communication between the tracer (as part of the Prover device) with the Attestation Agent of the Verifier device. Essentially, we use the tracer key to sign the traces on behalf of the edge device. The traces are then sent to the attestation Verifier, which can verify they were signed correctly by the key accessible only to the secure world. The traces cannot be forged by an adversary as we assume the adversary has no access to the private part of the secret key. This solution does not rely on the TPM and places trust that the secret key is never leaked by software executing in the secure world. However, we note in this case the public part of TRC_{priv} needs to be known to all of the edge devices comprising the service graph chain in a way that the device can verify its authenticity (aligned with conventional Public-key Infrastructures (PKIs)).
Locality-based Protection	To weaken this trust assumption, that the public part of the tracer key needs to be pre-deployed to all edge devices, in the following two schemes we rely on the TPM-based Wallet to sign the traces on behalf of the edge device. However, as aforementioned, in this case, it is important to make sure that the traces are authentic and not the result of an attack; *either by manipulating the actual traces or spoofing the traces by launching a* fake *tracer running as part of the normal world*. The first approach leverages one feature of the TPM platform called *locality*: If enabled, this flag forces the TPM to accept commands to be executed only by processes that have higher privileges running in the secure world. it essentially enforces a sort of access control to who can interact with the TPM - and these can only be processes running in the secure world, thus, the tracer.

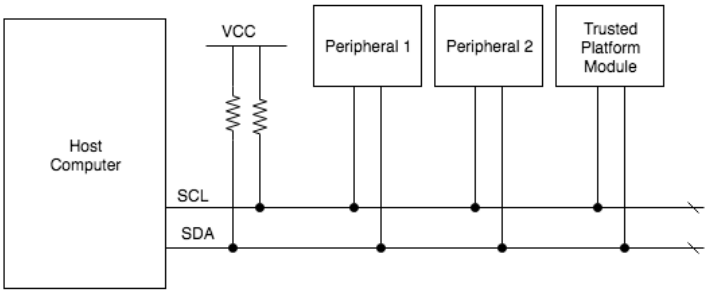
	As can be seen in the figure below, this approach leverages the UEFI drivers for the communication between the host device and the TPM so that only commands with the correct *locality* (exchanged through the SCL bus) will be processed by the TPM. As also mentioned in D3.1 [6] this is a rather interesting approach that the consortium is planning to investigate further. However, there is one caveat to consider: *How to make sure that an adversary is not using tailored interposers for manipulating TPM response packets in order to trigger parsing bugs in the host side TPM drivers which essentially can manipulate the locality flag and disable it.* 
Traces Integrity & Authentication	In this scheme, the TPM signs the traces for the edge device before they are sent to the attestation verifier. The challenge in this approach is that the adversary may control the communication between the tracer and the TPM. We overcome this challenge by constructing an Attestation Key (AK) in the TPM that is "*binded*" to be used for signing only authentic traces that, in turn, have been signed by the tracer's TRC_{priv}. This is essentially a key usage policy, as described in D3.1 [6], that enables the TPM to release the AK only when signed traces are received whose signature can be verified using the known and certifiable public part of the TRC_{priv}.

Table 4.1: ASSURED Offered Solutions for Traces Security & Authenticity. The current implementation follows the third approach as is also depicted in D4.2 [16] where the entire protocol has been fleshed out

4.4 Running Example of the Complete ASSURED Tracing Flow

Having defined all of the functional specifications and internal mode of operation for the ASSURED Tracer, in what follows, we give an example of how the Tracer is engaged within ASSURED through a concrete instantiation. The detailed sequence of actions executed is depicted in Figure 4.4. *Please note that in the following example we consider a tracing policy for monitoring the executional behavior (through control-flow graph traces) of specific software properties of a device to be leveraged by the ASSURED Control-Flow Attestation mechanism (Chapter 5).* However, the sequence of steps executed and the interactions remain the same irrespective of the type of attestation task executed - it's only the type of system properties to be traced that differ as orchestrated by the Attestation Agent (Section 3.3.2)

We consider a running example of smart connected devices in a manufacturing floor which comprise an ecosystem of thousand of Program Logic Controllers (PLCs). Each PLC must attest to each other in a web of interconnected nodes that propagate trust from edge nodes (usually sensors) to more central management nodes (IoT Gateway). The example follows the functionality of collision avoidance (between the workers and the robots) in which the PLCs attached to the robots estimate the location coordinates of the user (based on data they are getting from wear-

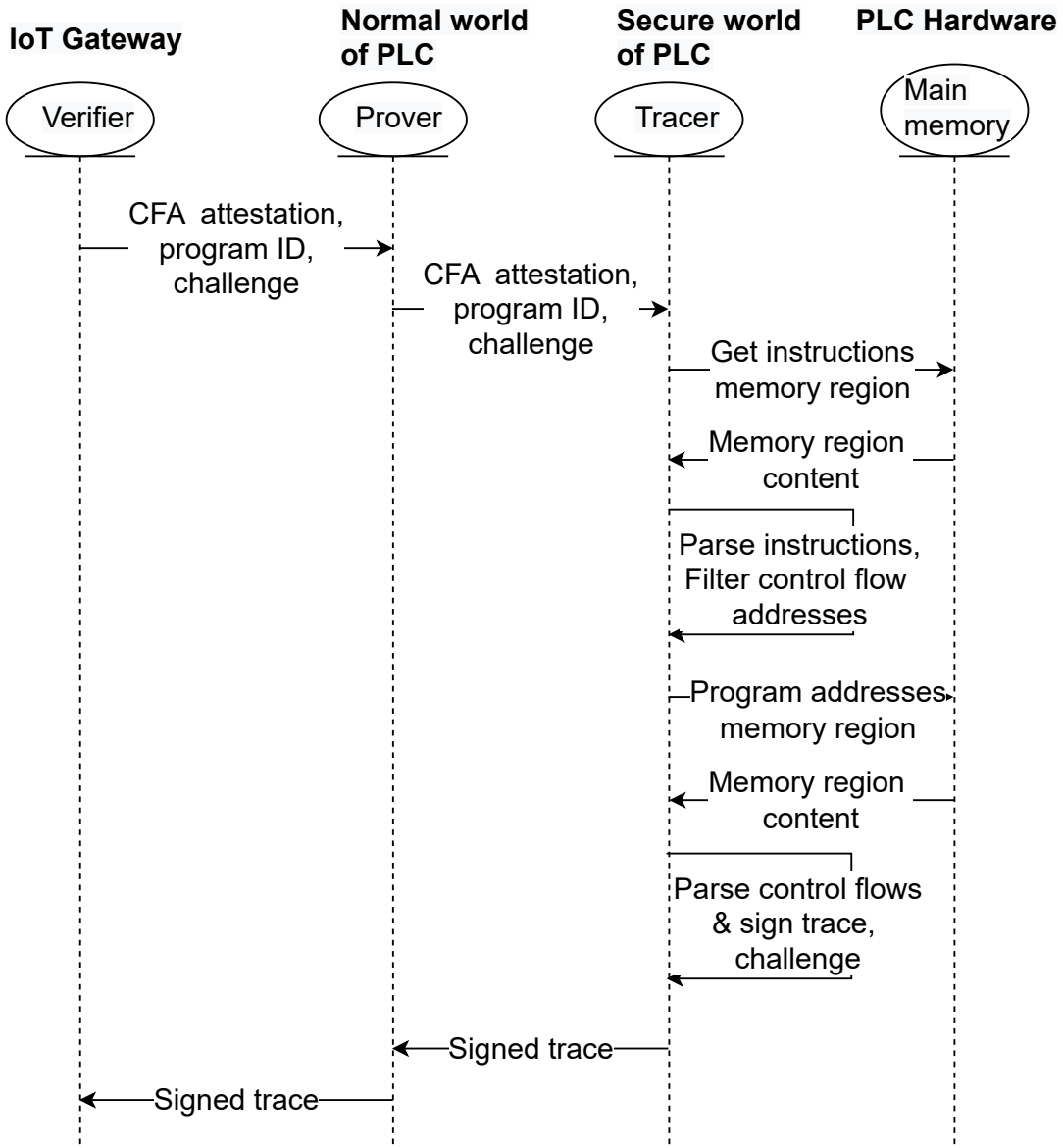


Figure 4.4: The Sequence of Actions for Control-Flow Tracing

able sensors embodied to the workers). Thus, such data needs to have strong correctness and integrity guarantees necessitating the continuous attestation of the location estimation function. The Real Monitoring System (RTM) [10], depending on the status of the deployed wearable sensors, will take action and apply the safety breaks for the robot accordingly. Let a code snippet that runs in such PLCs contain a buffer overflow vulnerability (Figure 4.5). In this example, the IoT Gateway is the Prover and the PLC is the Verifier. The PLC must attest to its running integrity before it is trusted by the IoT Gateway: if the properties received from the attestation are consistent with the S_1, S_2, S_3 control-flow state sequence, then the attestation process is successful and any data reported from the PLC (as received by the sensor) is trusted. Any other set of properties that might result from any deviating control-flow like S_1, S_2, S_4 will be rejected and will flag the PLC as compromised and untrusted.

The properties should contain just enough details to figure out any control-flow changes. For instance in Figure 4.5, we demonstrate a basic code injection scenario where the control flow is altered. The normal execution flow should start from the function entry (S_1 / ReceiveMsg.receive()),

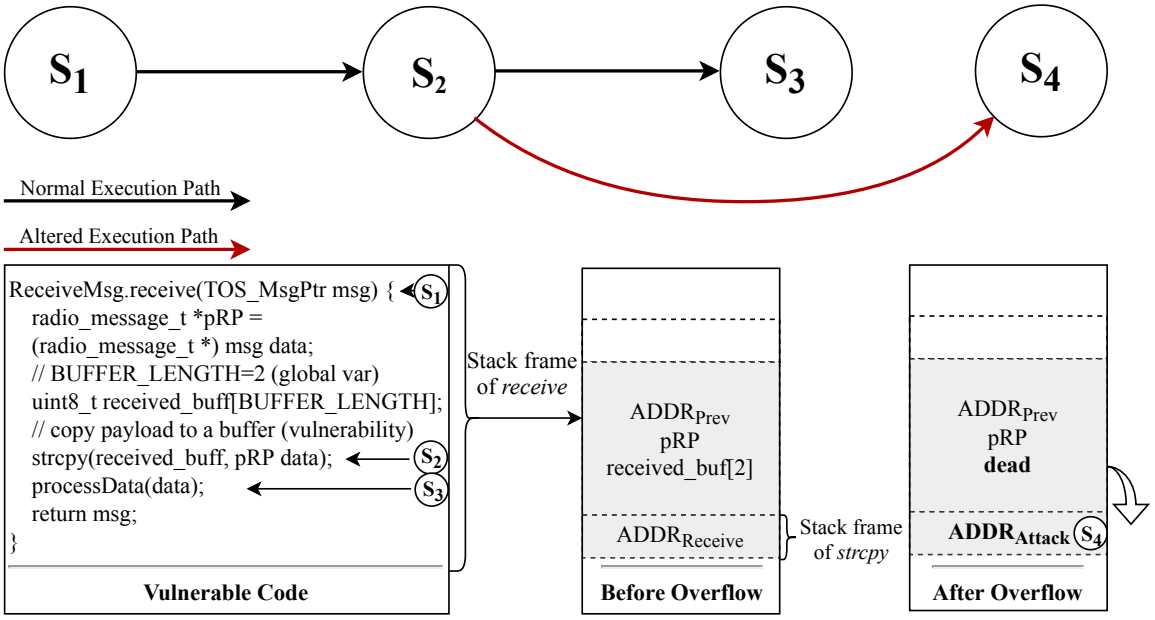


Figure 4.5: Normal and Altered Control Flows

continue to a memory sensitive function (S_2 / strcpy()) and finish with a data processing function (S_3 / processData()). In the scenario of code injection, the attacker will overflow the stack frame up to the point that she can overwrite the return address of the function with the address of S_4 ($ADDR_{Attack}$) which is the function that she will redirect the execution flow to. The system we propose, stores a healthy image of the control flow in the form of its properties, and afterward, during the run-time of the application, it is able to detect any changes made from code injection attacks like the aforementioned one.

As aforementioned, Figure 4.4 depicts the sequence diagram of all actions that take place for correctly triggering the Tracer to provide the correct measurements for the defined software function property. As can be seen, the IoT Gateway (as the Verifier) initiates the attestation process, based on the policy it has read from the Blockchain infrastructure, by sending the type of **attestation task, resources (system properties) to be attested, and the attestation challenge (nonce)** to the Prover PLC device. All this information is packaged in a message that is signed by the AK of the Verifier's TPM-based Wallet so as to guarantee its authenticity and timeliness. The nonce is also extracted through the Blockchain so as to be able to have a random but reproducible nonce that any entity wanting to verify the correctness of the entire attestation process should be able to re-create it - this is based on the hash of one transaction block on the ledger [12, 19] whose ID is been kept in the **private data collection**.

Once the Attestation Agent of the PLC receives the type of attestation task to be executed (CFA in this context) and the name of the resource to be monitored ($receive()$ function), it then triggers the Tracer to initiate the tracing. Upon this event, the tracer loads all the instructions from the memory region and starts parsing the executed low-level instructions so as to be able to extract the actual control-flow graph. Recall that, as aforementioned, the innovation behind the ASSURED tracer is on exactly this parsing: how to be able to efficiently (in the order of milliseconds) combine all the monitored memory information in a readable control-flow graph. Once the control-flow graph is extracted, it is then signed by the tracer's symmetric key $\longrightarrow$ passed to the PLC's TPM-based Wallet for verification and signing based on the AK $\longrightarrow$ Forwarded to the Attestation Agent of the IoT Gateway $\longrightarrow$ Verified (in terms of signature) by the TPM-based Wallet $\longrightarrow$ Verified by the CFA.

Chapter 5

Control-Flow Tracing

5.1 Control-flow Attestation

control-flow Attestation (CFA) is about being able, in a two parties system, to detect any kind of manipulation of control-flow Graph (CFG), which describes the flow of execution on an application.

The verification process consists of two actors:

1. A Verifier, the trusted party, that assures the integrity of the CFG;
2. A Prover, an untrusted party, who provides the traces that have to be verified.

Thus, in ASSURED, each edge device contains a tracer that acts as the Prover and sends control-flow traces to a Verifier agent (see Figure 5.1). The Verifier utilizes a deep neural network (DNN), trained on a specific application, to detect incorrect control-flows made by applications. More in detail, before deploying each introspected application we first trace it to generate benign traces, which are used to train the DNN in a semi-supervised way. Semi-supervised training is useful as it does not require malicious traces for training the model and makes the model not attack-specific, which means that the model will be able to detect any kind. For detecting the attacks we rely on the fact that the model will perform poorly on malicious data, according to a specific metric (i.e., the values for that metric for malicious data will significantly differ from benign data).

After application deployment, the Verifier can detect traces that differ significantly from the training data and are therefore flagged as malicious traces. Using only benign traces for the training, in addition to a scheme that automatically collects benign traces, e.g., by using techniques like fuzzing, allows the system to adapt dynamically to changes of the application and automatically train for new applications. However, it is important to notice that the Verifier is reliable and responsible only for a specific application at a specific fixed version, and in any case of update, it has to be retrained, as the control-flow graph of the application may be changed.

In ASSURED, the attestation process is either initiated by a Prover that wants to show its trustworthiness to other components or is initiated by the attestation toolkit based on the defined policies, e.g., if a new device joins the system. After the control-flow trace is evaluated, the Verifier sends the attestation result to the TPM wallet.

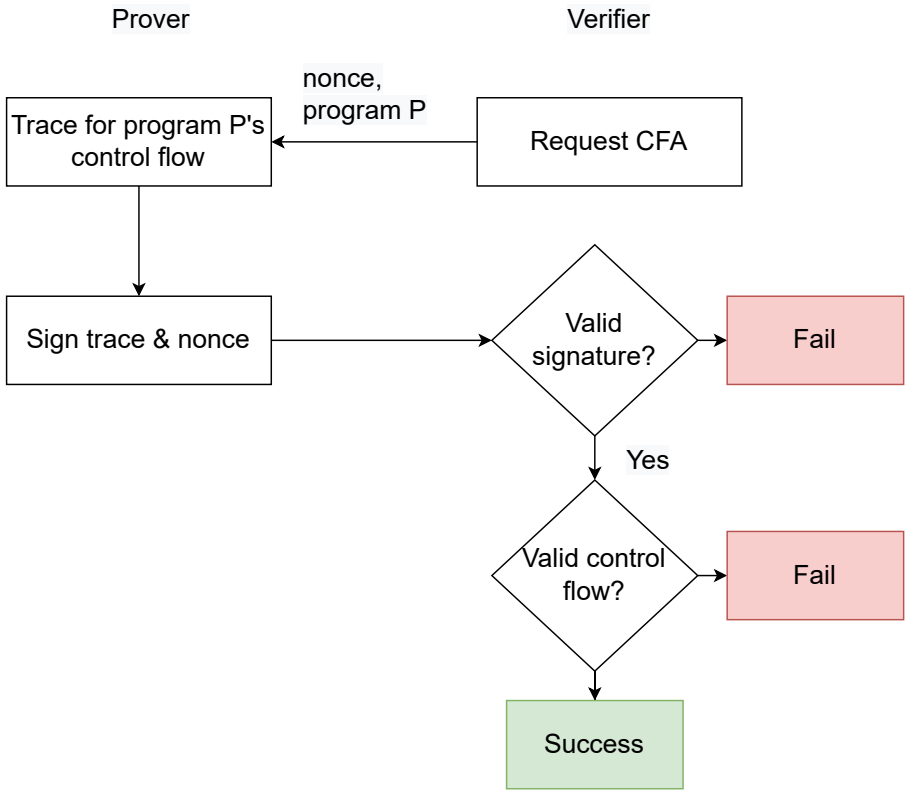


Figure 5.1: control-flow attestation flow.

5.2 CFA Tracing Building Blocks

5.2.1 Program Composition

A compiled program's code can be represented by its Control-Flow Graph (CFG), which encapsulates all possible program executions by modeling the legal control-flow between all of the program's statements. However, since not all statements affect the control flow, we typically fractionate the statements into maximal-length sequences of branchless statements that ultimately end in a branch, jump, or predicated operation. We denote each such sequence as a basic block (BBL) and have the CFG model the control-flow only between program BBLs. Let $CFG = (N, E)$ denote a directed graph, where each node $n_i \in N$ corresponds to one BBL, and each edge $e = (n_i, n_j) \in E$ denotes a possible transfer of control from n_i to n_j. We refer to edges corresponding to (direct and indirect) jumps and calls as forward edges and returns as back edges. We further label any node n_i a final node ($n_{\blacktriangleleft}$) if it has no reachable nodes and an entry node ($n_{\triangleright}$) if it is unreachable. Finally, we consider any continuous sequence of edges a legal execution path if it connects a node $n_{\triangleright}$ to $n_{\blacktriangleleft}$ (denoted $n_{\triangleright} \rightsquigarrow n_{\blacktriangleleft}$) in the CFG.

More formally, the CFG is a directed graph, $G = (N, E)$, where each node $n \in N$ corresponds to a program statement or BBL, and each edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control from block n_i to block n_j. We refer to edges corresponding to jumps (direct and indirect) and function calls as forward edges, and function returns as back edges. We call any node n a final node ($n_{\blacktriangleleft}$) if it has no reachable nodes and an entry node ($n_{\triangleright}$) if it is unreachable. Any consecutive sequence of edges that connect a node $n_{\triangleright}$ to $n_{\blacktriangleleft}$ (denoted $n_{\triangleright} \rightsquigarrow n_{\blacktriangleleft}$) and comprises exclusively of edges that exist in a CFG is considered a legal execution path.

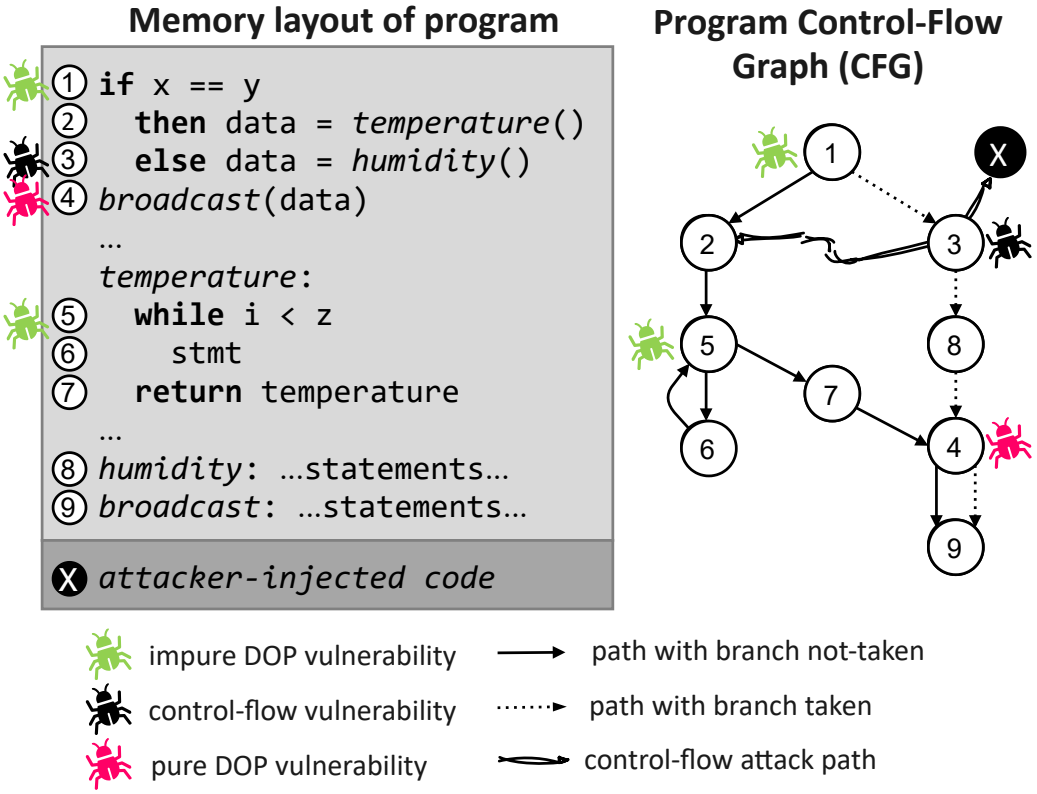


Figure 5.2: Abstract view of a program's CFG and threats.

5.2.2 Runtime Attacks

We continue with a description of major runtime attack classes that can induce harmful behavior by exploiting subtle software vulnerabilities to corrupt a program's control and data planes (based on the analysis already performed in D1.3 [11]). As a running example, consider the simple program in Fig. 5.2.

Control-based attacks. The most common attacks target a program's control plane to execute unintended code by explicitly diverting its execution path. There are essentially two variants: code injection and code-reuse attacks. With code injection, an adversary crafts and injects a payload into a device's memory and redirects a benign program's control flow to execute the payload [23, 44]. As an example, consider that in Fig. 5.2 an adversary has injected node n_X and diverted the program's control-flow from (n_3, n_8) to (n_3, n_X), resulting in the execution of code in n_X instead of n_8. However, being an early attacking methodology, code injection is easily defeated using common mechanisms such as Data-Execution Prevention (DEP), where executable memory regions cannot be written to during runtime. For the latter variant, however, it gets more difficult. Without injecting code, code-reuse attacks reuse existing program code to achieve some unintended behavior–using control plane maneuvers such as Return-Oriented Programming (ROP) [40], Jump-Oriented Programming (JOP) [3], and Counterfeit Object-Oriented Programming (COOP) [39].

With ROP and JOP, an adversary fabricates a new program by stitching together a chain of benign pieces of existing code (called gadgets) that end in either function returns (ROP) or indirect jumps or function calls (JOP). The chain is then written into memory (e.g., through a stack overflow vulnerability), where, once it is triggered (e.g., from replacing a function's return address with that of the first gadget), the gadgets execute in sequence. For example, in Fig. 5.2, the adversary

launches a simple ROP attack which diverts the control-flow from (n_3, n_8) to (n_3, n_2) to effectively execute code in the other branch.

Non-control-data attacks. Another class of attacks exists, called non-control-data attacks, which corrupt data variables to make programs yield unexpected outputs or indirectly drive program execution down unexpected or unauthorized paths. The attacking methodology behind non-control-data attacks is the application of Data-Oriented Programming (DOP) [4,27] which we can call impure or pure, depending on whether the program execution path is influenced (impure) or only data variables are altered with no effect on the path (pure). However, due to the difficulty of verifying a program's data flow at the Verifier, CFA schemes generally disregard data-oriented attacks altogether or assume that verification is done locally at the Prover. This is something that will be investigated in the second release of the ASSURED control-flow attestation enabler.

5.3 Tracing Method

The ASSURED methodology for extracting such control-flow traces can be seen in Figure 5.3. Recall that existing software-only approaches such as C-FLAT [2] require changing the original program via an instrumentation framework to collect information on retired control-flows. Specifically, Figure 5.3 depicts a set of assembly instructions, which are part of the `main()` function. These instruction includes memory access instructions such as `mov`, `ldr` and `str`, branch instructions such as `b.le` and `b`, and function invocations such as `bl foo` which invokes the `foo()` function (excluded from the figure for space considerations). To execute this program, the OS loads it into the main memory. As part of the process creation, the operating system populates a page table containing the virtual-to-physical translations of this program's pages. The tracer can access the main memory and use stored data on both the entire system and specifically for this program that is populated by the operating system and additional components such as ARM Coresight tracing to recover information needed by the attestation Verifiers. The full tracing details are described in the next chapters.

In contrast, in ASSURED, using a tracer that introspects memory to recover this information has several advantages. First, there is no need for **program instrumentation**, which is not always possible for different types of deployment environments. Second, the instrumentation may hinder performance as it intrusively changes the program behavior. For example, C-FLAT instrumentation interposed every control-flow instruction to transition into the secure world to trace the retired control-flow gadget and return to the normal world to continue the program execution. In ASSURED, we envision the **tracer would continuously run and introspect normal world programs such that it will not impact performance**. Finally, such instrumentation calls may be skipped due to hijack attacks, such as ROP and JOP as described in Section 5.2.2, and before the control-flow attestation takes place to detect them. With a tracer, skipping tracing is not possible.

Removing the need for instrumentation can be achieved via specialized hardware. For example, LO-FAT [21] required custom processors that track control-flows. In ASSURED, we envision supporting control-flow attestation on commodity processors such as ARM and RISC-V, which are popularly used for embedded devices. We hope this would enable mass usage as this is one of the most prominent types of micro-architectures for commodity embedded devices. We note that RISC-V processors do not include a Trustzone architecture and therefore do not support a secure and normal world separation. However, RISC-V processors include another form of a trusted execution environment, e.g., Keystone [30], which can be used by future versions of the tracer to achieve an isolated execution environment.

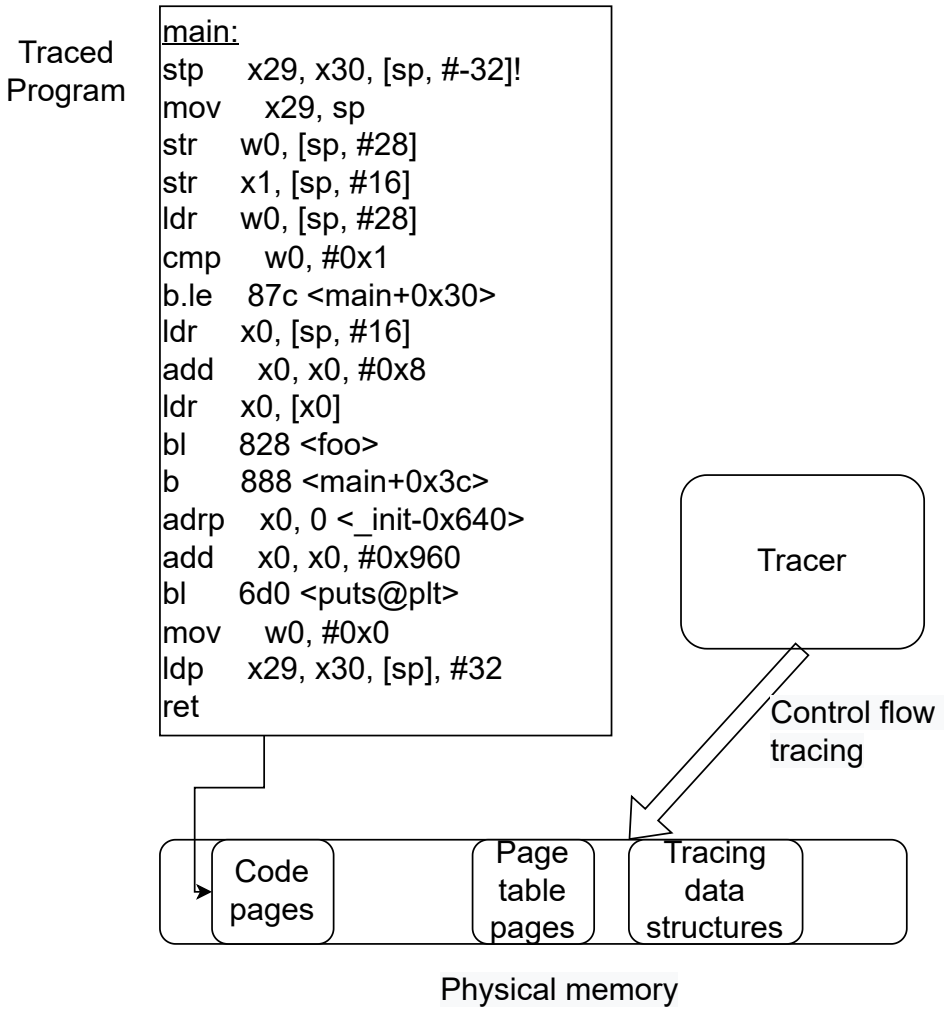


Figure 5.3: Overview of CFA tracing.

In ASSURED, the tracer **does not rely on instrumentation and instead traces programs control-flow via the following three methods**.

Software Stack Trace Reconstruction. We employ state-of-the-art forensics techniques to recover stack traces of invoked functions using register information stored in the physical memory [34,42]. We assume that Linux is used by the normal world in edge devices. The Linux kernel operates such that it stores registered information at the bottom of the kernel stack on privileged mode switches; for instance, due to system calls invocation. The user context (registers values) is required by the operating system to correctly restore the registers when returning to the program that executes in the unprivileged mode. The kernel also stores the user program stack pointer register as one of the saved register values. The stack pointer register points to the top of the stack and matches the latest user program function that was invoked. To recover the full stack trace the tracer can traverse the user stack from top to bottom using the frame pointers. Specifically, the calling convention in Linux stores the frame pointer before the return address on the stack, and potential argument if used by the function. The frame pointer points to the previous frame pointer and so forth up to the bottom of the stack. To conclude, the tracer identifies the frame pointer and uses it to traverse the entire user stack, recovering all the functions called by the user.

We note that this approach does not require any special hardware support. However, there are

Validation entity	Format	Description
CFA	[(symbol_0, library), ... (symbol_n, library)]	List of tuples corresponding to visited symbols and the library or binary they are part of.

Table 5.1: CFA traces format.

two main challenges for constructing an accurate control-flow trace. First, the **control-flows only include functions**. much like a stack trace obtained through a debugger. However, an adversary may also attempt to change programs execution to different basic blocks. For example, execute an `else` statement block instead of an `if` statement block in the C language. The second limitation of this approach is that **mode switches between user programs and the kernel** may not be frequent enough, which would cause some control-flows to not be detected at all. For example, short-lived functions would be missed from the control-flow traces created. Moreover, if the program being traced continues to execute in another core in the normal world, it may change the values of the registers, causing the tracer to read stale values and construct incorrect traces. Therefore, we also consider using more **intrusive software-based tracing and non-intrusive hardware-based tracing** as discussed next.

Operating system-assisted software tracing. For platforms that do not support the Coresight technology, we defer the tracing to using the `ptrace` Linux capability. `ptrace` provides a mechanism for a tracer to observe and control the execution of another process while examining and changing the process's memory and registers. Traditionally, `ptrace` is primarily used to implement breakpoint debugging and system call tracing. However, in ASSURED, we observe that it can also be used to track the control-flows of traced programs. Specifically, it allows the tracer to interrupt the traced program in instruction-level granularity. Thus, the tracer can read the instructions of the program, and for control-flow instructions, it would capture the destination address and generate a complete control-flow trace. Note, this method, unlike using Coresight impacts the executing program performance due to the operating system generating interrupts for the program. However, we note that this method does not require intrusive changes to the program, for example through instrumentation like prior work did [2]. To reduce the performance impact when the Coresight hardware is available we envision the tracer would use the hardware-assisted tracing method, which is described next.

Hardware-assisted software tracing. We envision using the ARM Coresight tracing technology to configure the processor to always emit branch instructions to a circular buffer in a predefined physical memory location known to the ASSURED tracer. To recover the control-flow graph of a program, The ASSURED tracer would continuously introspect this buffer and recover the branch instructions executed by the processor. The ASSURED tracer would infer the branches that are corresponding to the program to be traced for the control-flow attestation and using symbol information it has on the binary and libraries of the program recover the functions and basic blocks that were executed during the program run. Using this traced information the ASSURED tracer would emit the control-flow graph to a signed trace that would be used by the attestation Verifier.

To conclude, in ASSURED we are envisioning the usage of hardware-assisted tracing to support the CFA mechanism. In the context of WP6, we will evaluate the use cases with the different tracing methods described and analyze programs' latency with and without tracing enabled. This would allow us to choose the best method according to the available hardware for the different considered use case applications.

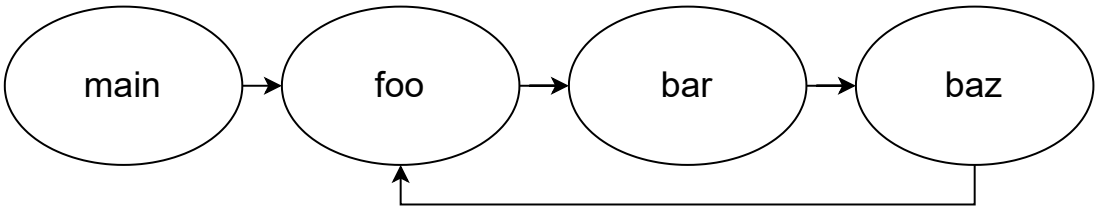


Figure 5.4: control-flow graph of a simplified program

5.4 Trace Output

The CFA traces generated by the tracer follows the format depicted in Table 5.1. Specifically, each trace contains an ordered sequence of symbols and the libraries or binary they are part of. This sequence represents the current control-flow graph the application is using. For example, consider the following program, which is depicted in Listing 5.1. For this program, we illustrate the control-flow graph in Figure 5.4.

```
int not_called() {
    system("/bin/bash");
}

int baz(int i) {
  return foo(i-1);
}

int bar(int i) {
  return baz(i);
}

int foo(int i) {
  if (i ==0) {
    return 0;
  }

  return bar(i)+1;
}


int main(int argc, char* argv[]) {
    char buffer[100];
    if (argc > 1) {
        strcpy(buffer, argv[1]);
    }
  return foo(10);
}
```

Listing 5.1: Sample program to demonstrate how control-flow tracing can be used to detect malicious behavior.

Next, we present a benign trace captured by the tracer. The trace represents a valid control flow of the program.

```
{[
```

```
    {
        "bar+0x14": "test"
    },
    {
        "foo+0x28": "test"
    },
    {
        "baz+0x18": "test"
    },
    {
        "bar+0x14": "test"
    },
    {
        "foo+0x28": "test"
    },
    {
        "main": "test"
    }
]}
```


Unfortunately, the program depicted in Listing 5.1 contains a buffer overflow vulnerability. The `main()` function uses an argument passed to it and copies it to a buffer that is 100 bytes in size and is allocated on the stack. An attacker can use this to trigger a return-oriented programming (ROP) attack. In a nutshell, `strcpy()` would overrun the buffer and corrupt values on the stack. Careful construction of the argument value can cause the return address from the main function to be altered to a different location. For example, to the `not_called()` function, which gives an attacker executable privileges. We note that this is a simple example, yet, ROP attacks are known to use similar buffer overflow vulnerabilities to gain similar general execution capabilities on platforms.

Fortunately, a trace under an attack, which we refer to as a malicious trace would contain the `not_called()` function as illustrated next. Such a trace file differs from the benign traces that represent correct execution and valid control-flows of the program. Thus, a control-flow Verifier can use the malicious trace to detect an attack.


```
{[
    {
        "not_called": "test"
    },
    {
        "baz+0x18": "test"
    },
    {
        "bar+0x14": "test"
    },
    {
        "foo+0x28": "test"
    },
    {
        "main": "test"
    }
]}
```

5.5 Verifier Interface

The Verifier communicates with the tracer through the ASSURED Attestation Agent interface that is essentially the "*bridging component*" that extracts the signed traces from the TPM-based Wallet (after received by the tracer). The communication between the Verifier and the Attestation Agent is based on a web service exposing a REST API interface in the tracer part in the normal world. The web service allows a remote Verifier to request traces generation and in turn, the tracer computes the control-flow traces and signs them.

Thus, in the overall security pipeline of ASSURED (Figure 3.1), the Verifier will initiate the remote attestation process based on the attestation policy read from the Blockchain infrastructure [17]. This policy will be interpreted by the Attestation Agent of the Verifier who will extract the type of attestation task to be executed, the exact software properties to be traced as well as the nonce to be sent to the Attestation Agent of the Prover for initiating the attestation process (*attestation challenge*). This is for allowing the Prover to verify that this attestation request originates from a valid Verifier device.

Assuming the device can verify the signature, the device instructs the trusted tracer to trace the properties defined, obtaining the trace-set $\mathcal{N}$ which is to be signed by the Attestation Key. As described in D4.2 [16], this allows the TPM-based Wallet to safeguard the usage of the AK: Recall that to use the attestation key, an authorized policy by the SCB must be satisfied. This allows the usage of this restrictive key to sign the traces if and only the device is at an expected configuration state based on what was verified during the device registration and enrollment phase. This allows to essentially perform, first, a *local attestation* on the correct configuration state and then continue with the execution of the control-flow attestation, thus, enabling stronger assurance claims.

After receiving a trace via the web service interface, the Verifier first validates the traces signatures (through its TPM-based Wallet) as described in Section 4.3. More specifically, the TPM-based Wallet checks the validity of the signature which implies both the correct configuration state of the Prover and the authenticity of the traces (originated by a valid tracer). If the signature verification fails an error is reported and it is detected as an attack. If not, the Verifier moves on to verify the trace report. The control-flow verification entails comparing the generated control-flow with the valid control-flows that the Verifier was trained with [8]. Similarly to verifying the signatures, if the control-flow verification fails it is detected as an attack and reported.

The process of verification starts after the traces are received. This process starts with the preprocessing phase, which takes care of preparing the traces for the Deep Learning Model which is responsible for the final verification. The model that is used for attesting traces has been pre-trained on benign traces so that it will be able to recognize all the possible malicious trace states. Then, according to a defined metric, the model will perform poorly on malicious data and optimally on benign data. According to these scores, the system is able to detect whether the traces received are malicious or not.

Chapter 6

Configuration Integrity Tracing

6.1 Configuration Integrity Verification

Besides the tracing of execution behavioral measurements, as described in D3.2 [8], ASSURED is also providing attestation enablers capable of verifying the correct configuration of a device through its entire lifecycle [20, 29]

The goal of the Configuration Integrity Verification (CIV) [8, 29] scheme is to enable the verification of the correct configuration of the target: the loaded binaries and libraries in the system. More specifically, the goal is to support the creation of **trust-aware service graph chains with verifiable evidence on the integrity assurance and configuration correctness of all comprised devices**. It is the first step towards a new line of security mechanisms that enables the provision of Configuration Integrity Verification, during both load- and run-time, by providing fine-grained measurements in supporting supply chain trust decisions, thus, allowing for a much more effective verification towards building a global picture of the entire service graph integrity.

CIV is building on the Integrity Measurement Architecture (IMA) and EVM features of the Linux kernel and leverages the ASSURED tracer for being able to monitor the integrity of a loaded binary. It monitors the information flows between TCB processes and those outside the TCB and can prevent violations or record them in the TPM-protected IMA measurement list. CIV introduces a concept of *digest lists* to limit the reporting of measured software only to the case when that software is unknown (not added to the digest list). This approach ensures predictable PCR values and reduced usage of the TPM and, consequently, reduced performance impact. It also introduces Simple Remote Attestation (Simple RA), to minimize the effort of integrating Remote Attestation in existing distributed architectures, by using implicit attestation over existing secure protocols (e.g. TLS), while addressing the lack of dedicated standard attestation protocols and thus mitigating interoperability concerns.

6.2 Tracing Method

A high-level overview of the CIV tracing method is depicted in Figure 6.2. The tracer generates traces that represent all the binaries and their dependent libraries that are loaded in the memory of the edge device. The traces, in turn, are used by a Verifier to detect any configuration violation from the expected valid configuration state. This proves for example that an adversary did not load an invalid program or changed any of the code pages of deployed programs as meant by the end-user.

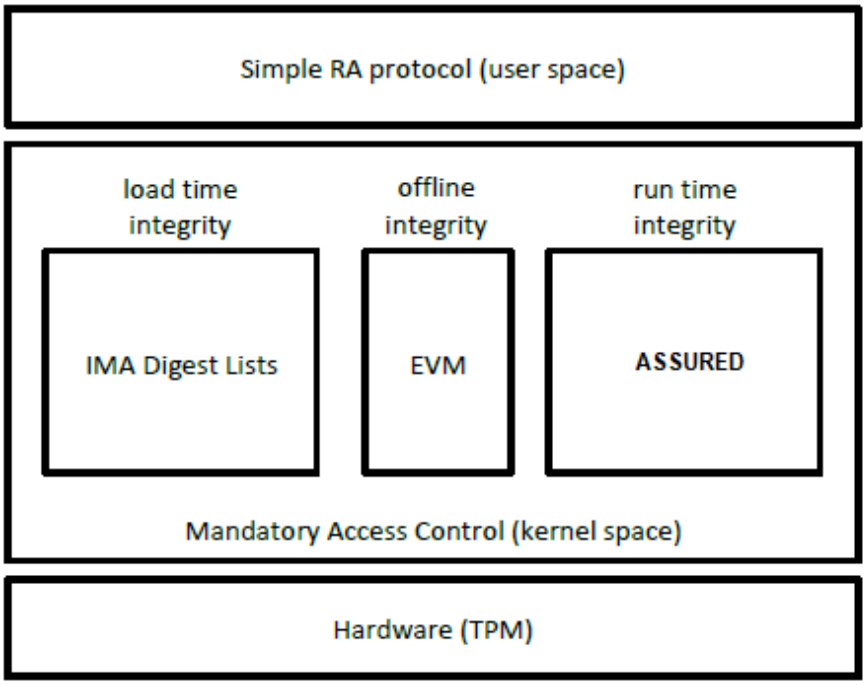


Figure 6.1: CIV Architecture

To support CIV, the tracer measures all the code pages belonging to a given program, which includes the program binary and associated libraries. The tracer computes the measurement based on a well-established cryptographic hash function: SHA-256. The tracer iterates over all active processes on the device and infers the in-memory addresses of all the code pages belonging to the program and its dependent libraries. Tracking the processes is explained in Chapter 4. To infer the pages' addresses, the tracer utilizes knowledge of the Linux operating system's virtual memory management of processes. Specifically, the Linux operating system stores information about the memory regions of a process in the per-process structure `task_struct` in the `mm` field. The tracer iterates over all the virtual memory areas stored in the `mm` field, which is a structure on its own. These areas contain the base and end addresses of contiguous virtual addresses and their backing file. For anonymous memory, the file does not exist, but for libraries and programs, the file contains the exact path of the in-storage file from which the contents are loaded into memory. Thus, the tracer can translate the virtual-to-physical addresses of all these regions and compute the hash representing each page. Effectively, for the CIV attestation, the tracer acts as the Prover. The corresponding Verifier gets the signed trace and validates the measurement matches a pre-established measurement value for the programs and their dependent libraries that were deployed on the device.

6.3 Trace Output

The CIV traces generated by the tracer follows the format depicted in Table 6.1. For example, the following sequence represents the configuration of an edge device with a design, as depicted in Figure 6.3. Specifically, the device contains two libraries: `libc-2.27.so` with a size of three pages, and `ld-2.27.so` with a size of a single page. The tracer infers the location of each page loaded in memory for the libraries, reads the content of the page, and computes a hash of 64 bytes using the SHA-256 algorithm. The produced trace thus contains a set of hashes representing the

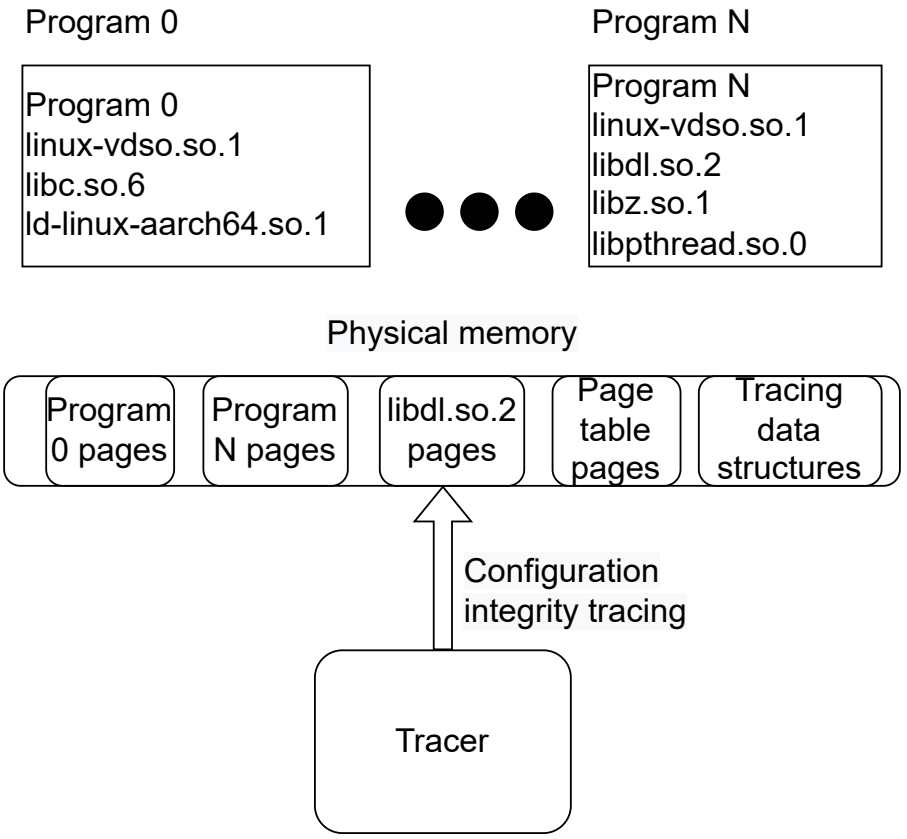


Figure 6.2: Overview of CIV Tracing

in-memory configuration of the device as follows.

```
{[
  {
    "name": "libc-2.27.so",
    "hashes" : [
      { "0" : "0721718c63188fe6ed74462222b4af4177e518e3ac2457cf9d5570137de77
          e0a" },
      { "1" : "f7a3bcec228f707044edf6d3be796adccd5f3c8f993f41b036beb10fd69d7
          d75" },
      { "2" : "16b1f2de728277b5b92a10c7a78e9fa347f42cca84195fc7fa584b99f91
          def0d" },
    ]
  },
  {
   "name" : "ld-2.27.so",
   "hashes" : [
      { "0" : "ad7facb2586fc6e966c004d7d1d16b024f5805ff7cb47c7a85dabd8b48892
          ca7" },
    ]
  }
]}
```

Validation entity	Format	Description
CIV	[lib_i: [hash, [invalid byte_0,.., invalid_byte_n]]	List binaries and libraries. Each list node is a tuple with the binary name, its corresponding measured hash value followed by a list of invalid bytes that are zeroed out before the measurement.

Table 6.1: CIV traces format.

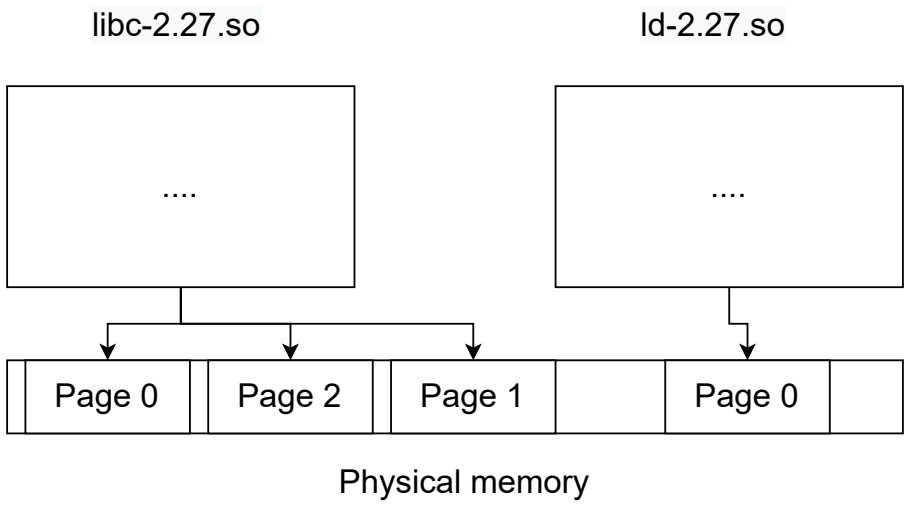


Figure 6.3: Exemplary Device Configuration with Libraries loaded in Physical Memory

6.4 Verifier Interface

The Verifier communicates with the tracer through the Attestation Agent interface following a similar flow as the one presented in the context of CFA (Section 5.5). This communication is based on a Unix socket. That is, similarly to the CFA tracing, the tracer maintains a service based on the socket in the normal world. The service lets the CIV Verifier (through its Attestation Agent) request CIV traces generation and in turn, the Prover tracer computes the hashes over all loaded libraries and programs in the edge device. Finally, the tracer signs the generated traces and sends them to the Prover's TPM-based Wallet for verification and signing with the produced AK prior to being forwarded to the TPM-based Wallet of the Verifier.

After receiving a trace over the Unix socket, the Verifier first validates the traces signatures as described in Chapter 4. If the signature verification fails an error is reported and it is detected as an attack. If not, the Verifier moves on the verify the trace report. For the CIV this entails comparing each page hash with the expected valid hash of the corresponding library, or program that was scheduled to be deployed in the device. Similar to verifying the signatures, if the configuration verification fails it is detected as an attack and reported through the ASSURED backend services.

Chapter 7

Attack Validation Tracing

The main objective behind the attack validation component is to validate the cause of an already detected attack (based on a failed attestation result) on an edge device by subjecting attacks on virtual counterparts and employing security forensics so as to be able to identify the exact attack path that was exploited by the adversary. **Validating the attack refers to indicating which component or sub-system is affected by the attack, and how the attack affects the overall behavior of the system.**

More specifically, during run-time, the low-level properties, that will be attested by the ASSURED attestation agents, and the **Attestation Report will contain the final verdict**. In case of failure, a more **in-depth investigation of the systems behaviour is needed in order to to identify the exact attack vector and for the re-configuration of the security policies**. Towards this direction, we have proposed a novel solution based on the use of security forensics [14]: The Attack Validation component comprises of a **virtual representation of all deployed physical devices, each one of which receives (in real-time) the monitored system properties as extracted by the ASSURED software-based Tracer loaded in each device.** The Tracer essentially outputs the measurements (for the properties of interest) that, in the case of a failed attestation report, can be leveraged by the Attack Validation component for performing detailed **concolic testing and fuzzing mechanisms** in order to identify the exact attack path that was exploited by an attacker. This, in turn, can lead to the identification of zero-day exploits that when given as input to the RA Engine it can re-calculate the overall risk graph for then been able to identify the appropriate mitigation measures in order to upkeep the desired assurance and the required details for post-attack investigation.

Projecting attacks on virtual counterparts allows testing various scenarios without any damage to equipment, devices or systems also allows greater observability, , for the sake of aggregating monitoring data and generating a composite view of complex installations. In ASSURED, the Attack Validation component considers vulnerabilities that compromise the behavior of Programmable Logic Controllers (PLCs) and Industrial Control PCs (IPCs). For example, vulnerabilities in the configuration, due to invalid inputs, and vulnerabilities in the code.

The virtual PLCs and IPCs form the behavioral representation of an actual system. The Attack Validation component uses the different values acquired either by the tracer or the fuzzing engine to simulate a complete process execution of the PLCs and IPCs. The resulting outputs are read and compared with the expected values as described in a *system description*. A mismatch of an expected value with a traced value is used to detect an attack.

The system description represents the behavior description of the system provided by the system administrator. Upon detection of an intrusion, a root cause analysis is performed to find the

Validation entity	Format	Description
Attack validation	[name, type, value]	List of traced variables. Each entry contains the variable name, the variable type, and the traced value

Table 7.1: Attack validation traces format.

source of compromise in the system. The detection and root cause analysis components do not only find potential vulnerabilities in the systems but also forecast the effect the aforementioned vulnerabilities have on the entire process.

7.1 Tracing Method

As mentioned, the attack validation component relies on the tracer to track a configurable set of variables' values to match them with pre-configured valid values, which allows attacks detection and vulnerabilities discovery. To that end, the tracer utilizes the **symbols and debug information of the applications executing in the host device**.

More specifically, the tracer reads the debug data structures in memory, such as ELF symbol table sections to infer the existing variables, and their addresses in the virtual memory. The tracer can then employ its **software virtual-to-physical address translation logic** to acquire the physical addresses. Upon attack detection, the tracer reads the content of the corresponding memory region for the variable. Furthermore, based on the variable type, the tracer can reconstruct the specific value of the variable according to the actual raw bytes read from the memory region.

Optional - program state tracing: In ASSURED, we also consider leveraging the Attack Validation component to detect code-level vulnerabilities, and not only behavioral vulnerabilities related to the end to end program logic. To that end, the component must know of the latest state of the program when an attack occurred. Using the program state information, we can synthetically emulate this state to recover information on the vulnerability itself that caused the program to reach this invalid state.

For the second release of the tracer and Attack Validation component, where code analysis will be considered, an optional improvement is to include tracing of the program state after attacks are detected. **This program state would include data the Attack Validation component considers as important to simulate attacks.** For example, the current active stack frames, the process identifier, the instruction pointer, etc. This this pertains to the **resilience and mitigation planning** as a complementary service of the overall ASSURED Attack Validation component: It may also request more detailed traces (including additional information on the internal memory view of the target device), thus, allowing for a **multi-level detailed tracing**.

7.2 Trace Output

The traces generated for the Attack Validation component are in a JSON format, with the contents structure being depicted in Table 7.1. Specifically, the tracer monitors the values of variables after an incident is detected to be passed to the attack validation component. The exact variables to be traced are stated in a system description map.

For example, consider the following system description map.

```
components:
  - ref: &p_gain_var
    name: "controller p gain variable"
    kind: "var"
    bounds:
      physical:
        lower: "0"
        upper: "10"
        type: "double"
        unit: ""
      operational:
        lower: "0"
        upper: "10"
        type: "double"
        unit: ""
    children:

  - ref: &i_gain_var
    name: "controller i gain variable"
    kind: "var"
    bounds:
      physical:
        lower: "0"
        upper: "10"
        type: "double"
        unit: ""
      operational:
        lower: "0"
        upper: "10"
        type: "double"
        unit: ""
    children:

...
```

An example for an output trace corresponding to the above system description map is the following.

```
[
 { "controller p gain variable", "double", 5.0 },
 { "controller i gain variable", "double", 3.1 },
]
```

7.3 Tracer Interface

The Attack Validation component is triggered by the tracer-generated outputs [14]. Specifically, the tracer periodically collects variable values according to the specification available in the system description and publishes this information to the attack validation component via a Kafka broker.

The communication takes place through Apache Kafka which is an open-source distributed event streaming framework that is used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Internally, Kafka stores

key-value messages sent by processes. The data can be configured to be partitioned into different partitions within different topics. Kafka runs on a cluster of one or more servers, which are also referred to as brokers. Kafka distributes the partitions of all topics across all the brokers. Additionally, partitions are replicated to multiple brokers. This architecture allows Kafka to deliver massive streams of messages in a fault-tolerant fashion.

More specifically, Attack Validation receives traces (during run-time), from the physical device, through the ASURED software-based Tracer that is responsible for monitoring and extracting all the necessary system measurements (for the properties of interest) to be used by the attestation enablers. This tracing is a continuous process and in the case of a failed attestation report, then the output of the Tracer is sent to the Attack Validation component for further processing; i.e., identification of possible attack path by generating mutation of the received run-time traces. More detailed traces can also be requested if needed. The Attack Validation provides multiple interfaces to generate different mutation values for acquired traces using “Mutational fuzzing” for identifying different attack paths (a priori) by injecting these mutated traces into Virtual PLCs and IPCs where they comprise the behavioral equivalent of a physical system. Later, after the execution of the program logic in Virtual PLCs and IPCs, the system states such as input, output, and process image table are read. The image table is a memory location (Array or list) containing values of different state variables. After reading system states, this is compared with the states described in System Description profiles, defined by the System Administrator, to detect any possible violation. System description profiles represent the behavior logic of the system containing information such as state variables and their data type and operation and physical limits. On detecting such violations, an analysis is made to find the attack path. This Virtual Host-based Intrusion Detection and Analysis *cannot only find potential vulnerabilities in the systems but also project their impact on the entire process (of the service graph chain) illustrating how such shallow and deeper vulnerabilities can affect the level of trustworthiness of the overall ecosystem*.

Chapter 8

Current Status and Future Plans

8.1 Current Implementation Status & Research Plan towards Second Release

The implementation and setup details of the ASSURED Tracer can be found on the project's Github repository [5]. In what follows, we describe the current implementation status and depict the features that we intend to continue looking into towards the next release in order to improve the tracing capabilities and support for the ASSURED ecosystem and the envisioned use cases.

8.1.1 Current Implementation Status

At the time of writing this deliverable, the tracer implementation closely follows the architecture described in Chapter 4. Specifically, the tracer is partitioned into two components, a process that is awaiting requests from the attestation agents in the normal world and a secure world service. Both software communicate over a shared memory in the normal world, acting as an event queue. That is, requests are published on the queue, allowing the secure world to process them, generate traces according to the request and return the response (signed traces) over the same shared memory.

The current prototype currently supports the control-flow tracing as described in Chapter 5 based on `ptrace`. We plan to look into the more efficient Coresight tracing in the next release. Moreover, the prototype supports the CIV tracing as described in Chapter 6. *Please note, however, the prototype does not yet support traces signatures. We plan to investigate the different signature schemes as part of the next release as described next.*

To simplify the tracer integration process, the current prototype supports three modes of operation. First, the one described in Chapter 4 running the tracer in Trustzone via OP-TEE. Second, running the tracer as a standard Linux process, which enables running the tracer in devices that do not support the Trustzone architecture. Finally, the tracer can run in a virtualized environment via QEMU, which emulates a processor that supports Trustzone. We deploy the tracer prototype in the secure world as traditionally. Yet, instead of tracing programs running in the virtualized normal world, it traces programs running in the bare-metal device by integrating a communication channel between the virtualized environment and the bare-metal environment.

8.1.2 Research Plan towards Second Release

The following features are already planned to be integrated with the second release of the ASSURED Tracer:

Attack Validation Tracing: We plan to support the variable tracing as described in Chapter 7, and optionally to trace the program state if the attack validation component would enable code-level vulnerabilities analysis.

Coresight-assisted Tracing: We plan to support the Coresight-assisted tracing for CFA as described in Chapter 5. We plan to further evaluate the tradeoff of the different control-flow acquisition methods, including the performance impact on programs and the integration challenges.

Traces security: We plan to evaluate the different signature schemes as described in Chapter 4. Specifically, we envision evaluating both the performance impact due to passing traces to the TPM, the overall throughput of signing traces via the TPM vs. the device's processor.

8.2 Discussion

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

Although detailed tracing and introspection can be resource-intensive, it is required for proper and thorough attestation schemes. Thus, we argue that a practical way forward is basically to provide a multi-level detail tracing mechanism that can actually incorporate different types of tracing mechanisms with varying levels of granularity in order to provide much higher scalability. *However, performance still remains an issue as it is directly linked to the complexity of the codebase traced.*

Thus, one could argue that the performance of the tracer is crucial to the security of the edge device. Consider for instance tracing features that are able to provide measurements way after an incident has occurred. Such a tracer would provide traces at a slow rate, which in turn would affect the number of attestation requests being served. A quick attacker may take advantage of such a gap to perform the attacks in between each tracing cycle. Therefore, in ASSURED, we aim to optimize the tracer for the different tracing methods supported and provide seamless integration with programs running on the edge devices.

Unfortunately, however, the **tracing performance depends on the complexity of the loaded binaries**. That is, the more control-flows that are part of a program require larger traces construction, and more data points (control-flow events) to trace and decode. We note that in WP6 in ASSURED we plan to evaluate use cases with different levels of complexity. Specifically, the applications evaluated would include a diverse set of control-flows. Tracing each application would

provide a tracing latency, which would enable us to better understand the relation between the tracer performance and the complexity level of the application.

Ultimately, building an efficient tracer is an open research problem. Prior work aimed to reduce tracing latency by utilizing hardware-based tracers [21,45]. However, hardware alone is inflexible, which limits such tracers' capabilities. For example, tracing only for control-flows and not for configuration integrity.

In ASSURED, we aim to utilize hardware acceleration in commodity processors for control-flow tracing (via ARM Coresight). However, we use a software-based tracer that controls hardware acceleration to optimize tracing performance where possible and use the software for the flexibility of acquiring traces for different properties representing the system.

Chapter 9

Conclusion

This deliverable provides a detailed description of the ASSURED tracer. It focuses on the tracer architecture and the integration of the tracer within the entire ASSURED ecosystem.

The deliverable begins with a review of the state-of-the-art tracing mechanisms establishing a relation between them and the goals and assumptions for the ASSURED project. This review aims to justify the software tracing mechanism proposed in ASSURED. Specifically, the tracing must be flexible enough to support the different attestation mechanisms envisioned by ASSURED to enforce the security state of the interconnected system-of-systems. Furthermore, software-based tracing facilitates adoption as it does not rely on unique processor features, which enables deploying the ASSURED tracer on commodity ARM processors that are widely available on numerous platforms. Additionally, software-based tracing can be combined with trusted execution environments technology to provide trustworthy tracing even in the face of powerful adversaries who gained control of other parts of the device.

The deliverable continues with a summary of the envisioned edge device model as described in D3.2 [8]. This includes the hardware components that must be part of each edge device, such as an ARM processor supporting the Trustzone technology, and a TPM. We also provide details regarding the software components that execute on edge devices and focus the discussion on their relationship with the tracer. Finally, the assumed TCB of each edge device is summarized based on the definition provided in D3.1 [6].

In addition, in the deliverable, we provided a thorough description of the tracer architecture. The algorithms used to recover semantic information on the platform, including memory acquisition both in normal world operation mode and in the secure world operation mode. We presented the virtual-to-physical translation, which is the cornerstone to trace high-level semantic information such as control flows and the configuration of the platform.

We also provided a summary of the control-flow attestation and configuration integrity verification mechanisms that are used in ASSURED to infer the risk level of each edge device. We provided details on methods used to capture the control flows and device configuration, examples of traces, and the interface the tracer shares with the verifiers. Finally, we concluded D3.4 with a high-level description of the attack validation component, and how the tracer can supplement it by tracing specifically chosen variable values to be used as inputs to infer attack details.

List of Abbreviations

Abbreviation	Translation
AE	Authenticated Encryption
ABE	Attribute-based Encryption
AK	Attestation Key
CA	Certification Authority
CFA	Control-flow Attestation
CFG	Control-flow Graph
CIV	Configuration Integrity Verification
CSR	Certificate Signing Request
DAA	Direct Anonymous Attestation
DLT	Distributed Ledger technology
EA	Enhanced Authorization
EK	Endorsement Key
GSS	Ground Station Server
MSPL	Medium-level Security Policy Language (MSPL)
NMS	Network Management System
Privacy CA	Privacy Certification Authority
Prv	Prover
PCR	Platform Configuration Register
PLC	Program Logic Controller
RA	Risk Assessment
RAT	Remote Attestation
SCB	Security Context Broker
SoS	Systems of Systems
SSR	Secure Server Router
S-ZTP	Secure Zero Touch provisioning
TC	Trusted Component
TLS	Transport Layer Security
TPM	Trusted Platform Module
Vf	Virtual Function
VM	Virtual Machine
Vrf	Verifier
WP	Work Package
TCG	Trusted Computing Group
ZTP	Zero Touch Provisioning

References

[1] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *NDSS*, 2019.

[2] Tigist Abera et al. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proceedings of the 2016 ACM SIGSAC CCS Conf.*, pages 743–754, 2016.

[3] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[4] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[5] The ASSURED Consortium. ASSURED tracing. https://gitlab.com/assured_project/assured-tracing.

[6] The ASSURED Consortium. Assured attestation model & specification. Deliverable D3.1, November 2021.

[7] The ASSURED Consortium. Assured blockchain architecture. Deliverable D4.1, November 2021.

[8] The ASSURED Consortium. Assured layered attestation and runtime verification enablers design & implementation. Deliverable D3.2, November 2021.

[9] The ASSURED Consortium. Assured reference architecture. Deliverable D1.2, May 2021.

[10] The ASSURED Consortium. Assured use cases & security requirements. Deliverable D1.1, February 2021.

[11] The ASSURED Consortium. Operational sos process models & specification properties. Deliverable D1.3, September 2021.

[12] The ASSURED Consortium. Policy modelling & cybersecurity, privacy and trust constraints. Deliverable D2.2, November 2021.

[13] The ASSURED Consortium. Risk assessment methodology & threat modelling. Deliverable D2.1, November 2021.

[14] The ASSURED Consortium. Assured collective threat intelligence analysis & forecasting framework. Deliverable D2.7, February 2022.

[15] The ASSURED Consortium. Assured runtime risk assessment framework. Deliverable D2.3, February 2022.

[16] The ASSURED Consortium. Assured secure distributed ledger maintenance & data management. Deliverable D4.2, February 2022.

[17] The ASSURED Consortium. Assured tc-based functionalities. Deliverable D4.5, February 2022.

[18] The ASSURED Consortium. Evaluation framework & demonstrators planning. Deliverable 6.1, February 2022.

[19] The ASSURED Consortium. Security context broker specification and smart contract definition & implementation for policy enforcement. Deliverable D2.2, February 2022.

[20] Heini Bergsson Debes, Thanassis Giannetsos, and Ioannis Krontiris. BLINDTRUST: oblivious remote attestation for secure service function chains. *CoRR*, abs/2107.05054, 2021.

[21] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[22] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *Ndss*, volume 12, pages 1–15, 2012.

[23] Thanassis Giannetsos and Tassos Dimitriou. Spy-sense: Spyware tool for executing stealthy exploits against sensor networks. In *Proceedings of the 2Nd ACM Workshop on Hot Topics on Wireless Network Security and Privacy*, HotWiSec '13, pages 7–12, 2013.

[24] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[25] Tooba Hasan, Akhunzada Adnan, Thanassis Giannetsos, and Jahanzaib Malik. Orchestrating sdn control plane towards enhanced iot security. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 457–464, 2020.

[26] James Hendricks and Leendert van Doorn. Secure bootstrap is not enough: Shoring up the trusted computing base. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, page 11–es, New York, NY, USA, 2004. Association for Computing Machinery.

[27] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, P. Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[28] Nikos Koutroumpouchos, Christoforos Ntantogian, Sofia-Anna Menesidou, Kaitai Liang, Panagiotis Gouvas, Christos Xenakis, and Thanassis Giannetsos. Secure edge computing with lightweight control-flow property-based attestation. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 84–92, 2019.

[29] Benjamin Larsen, Heini Bergsson Debes, and Thanassis Giannetsos. Cloudvaults: Integrating trust extensions into system integrity verification for cloud-based environments. In *Computer Security*, pages 197–220, Cham, 2020. Springer International Publishing.

[30] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[31] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016.

[32] Nvidia. NVIDIA DOCA APP SHIELD: Shield Your Host Services with Adaptive Cloud Security. https://resource.nvidia.com/en-us-linely-whitepaper/doca-app-shield-solution-brief.

[33] OP-TEE. OP-TEE: Platforms supported. https://optee.readthedocs.io/en/latest/general/platforms.html.

[34] Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kazuhiko Ohkubo. Building stack traces from memory dump of Windows x64. *Digital Investigation*, 24:S101–S110, 2018.

[35] OWASP Top 10: Open Web Application Security Project. OWASP Top 10 2021, 2021. https://sharedassessments.org/blog/owasp-top-10-open-web-application-security-project/ [Online; accessed 28-December-2021].

[36] Dimitrios Papamartzivanos, Sofia Anna Menesidou, Panagiotis Gouvas, and Thanassis Giannetsos. Towards efficient control-flow attestation with software-assisted multi-level execution tracing. In *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, pages 512–518, 2021.

[37] Bryan Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, pages 9:1–9:6, Berkeley, CA, USA, 2008. USENIX Association.

[38] Global Platform. TEE Internal Core API Specification v1.1. https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[39] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[40] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[41] Suchakrapani Datt Sharma and Michel Dagenais. Enhanced userspace and in-kernel trace filtering for production systems. *Journal of Computer Science and Technology*, 31(6):1161–1178, 2016.

[42] Edwin Smulders. User space memory analysis. Master's thesis, University of Twente, 2013.

[43] TCG. Trusted platform module (tpm). https://trustedcomputinggroup.org/work-groups/trusted-platform-module, 2020. Accessed: 2020-12-01.

[44] Giannetsos Thanassis, Dimitriou Tassos, and Prasad Neeli R. Weaponizing wireless networks: An attack tool for launching attacks against sensor networks. In *Black Hat Europe 2010*, Barcelona, Spain, April 12-15, 2010.

[45] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391. IEEE, 2017.