DIHyper: providing lifetime Hypervisor Data Integrity

José Lopes, José Martins, Adriano Tavares and Sandro Pinto
Centro Algoritmi
University of Minho, Portugal

{jose.lopes, jose.martins, atavares, sandro.pinto}@dei.uminho.pt

***Abstract*—Virtualization is being widely adopted in modern embedded systems, due to its security advantages in isolating multiple and heterogeneous operating systems. Security by isolation is a well established strategy for achieving security goals such as data confidentiality, integrity, and availability; however the successful attacks against virtualization infrastructures have raised seriously questions regarding the trustworthiness of existing hypervisors. In this paper we present DIHyper, a lightweight approach which endows a TrustZone-assisted hypervisor with a self-protecting capability to provide lifetime data integrity. The implemented runtime mechanism is able to protect the system against non-control-data attacks. The experimental results demonstrate DIHyper can be effectively enabled with approximately 4% performance overhead.**

***Index Terms*—DFI, Data flow, integrity, security, TrustZone.**

I. Introduction

Virtualization has been used as a key enabling technology for coping with the ever growing complexity and mixed-criticality of modern embedded systems, due to the possibility of consolidation and isolation of multiple operating systems on the same hardware platform. Several embedded industries, ranging from consumer electronics to aerospace, and automotive to industrial control systems, share an upward trend for integration, due to the common interest in building systems with reduced size, weight, power and cost (SWaP-C) budget [1,2].

Security requirements and functionalities are currently being implemented using the isolation primitives provided by hypervisors, based on the premise of increased reliability from a reduced Trusted Computing Base (TCB); however, for high-assurance systems, this has been proven to be insufficient [3,4]. If hypervisors are vulnerable, security functionalities might be disabled by attacks originated from the guest OSes. Recent research efforts towards virtualization technology and security point out one common problem: the need for reconciling security with virtualization, the need for real trustworthy hypervisors [4]–[7].

As security researchers have been focusing on protecting against code-injection and code-reuse attacks, attackers moved their attention to a more subtle class of attacks: non-control-data attacks [8]. These attacks are perpetrated while still following the legitimate control-flow of a program. Program's data can be divided into two classes: control data and non-control data. A multitude of attacks with growing complexity, such as JOP [9], has been developed targeting control-data and, consequently, the control-flow. As control-flow integrity countermeasures starts becoming widespread, modern OSes and hypervisors remain vulnerable against non-control-data attacks. Non-control-data attacks are not recent; however, they maintain an ever-increasing relevance [10,11].

There are two paradigms driving research on secure software and systems. The first approach consists in eliminating software bugs (e.g., buffer overflow, dangling pointers) at compile-time. Here, formal verification gains relevance, being the only known way to ensure that a system is free of programming errors. For example, seL4 [12] provides formal-proof of functional correctness for a microkernel OS. However, for this purpose, several restrictions are imposed on micro-kernel design and implementation. A second avenue of work considers the former methods either insufficient or inadequate and proposes the implementation of runtime defense mechanisms [13]–[18]. HyperSEntry [13] is a runtime integrity monitor that only captures persistent changes to the hypervisor. Other memory introspection-based solutions such as VMWatcher [14] and Lares [15] suffer from this same problem of only capturing persistent changes programs' data. Despite being designed to protect a guest OS, they follow the same technique. Other methods, aiming at total or partial memory protection (e.g., Cyclone [16], BaggyBounds [17] and DFI [18]), are also interesting approaches, although sometimes prohibitive due to severe non-deterministic behavior or performance overhead.

Ensuring hypervisor's lifetime integrity poses a real challenge. Boot-time integrity can be enforced using a secure boot mechanism (e.g., TPM, ARM TrustZone); however, runtime integrity poses another set of problems and challenges. The hypervisor is expected to be vulnerable, which demands the agent to analyze system's dynamics, ideally, without interfering with the hypervisor itself or imposing stringent design or implementation constraints.

In this paper we present the design, implementation and evaluation of DIHyper, a runtime monitor which reliably establishes the continuous data integrity of a TrustZone-assisted hypervisor. Our experience indicates DIHyper's code size is small and its integration in commodity hypervisors is straightforward, as it does not require specific hardware support. Evaluation with application benchmarks show that the integrity protection can be effectively enabled with approximately 4%

performance overhead.

II. Background

This section overviews essential background about the ARM TrustZone technology (Section II-A) and the TrustZone-assisted hypervisor adopted in this work (Section II-B).

A. ARM TrustZone

TrustZone is a hardware security-oriented technology implemented in ARM application processors (Cortex-A), covering the processor, memory, peripherals, interrupts and bus. At the heart of TrustZone is the concept of secure and non-secure worlds. These two worlds are completely hardware isolated, with non-secure software blocked from directly accessing secure world resources. The current world in which the processor runs is determined by the Non-Secure (NS) bit. A switch between the two worlds can be bridged via software referred to as the secure monitor, which runs in high-privileged processor mode, the *monitor mode*. To enter the monitor mode, a new privileged instruction, SMC (*Secure Monitor Call*), was introduced. TrustZone allows system designers to add a TrustZone Address Space Controller (TZASC). This component extends isolation to the memory infrastructure, allowing partition of dynamic random-access memory (DRAM) into different secure and non-secure memory regions. The TrustZone-aware Memory Management Unit (MMU) provides a distinct MMU interface per world, enabling each world to have a local set of virtual-to-physical memory address translation tables. The isolation is also present at the cache-level. Devices can be configured as secure or non-secure through the TrustZone Protection Controller (TZPC). The Generic Interrupt Controller (GIC) also supports the coexistence of secure and non-secure interrupt sources.

B. μRTZVisor

μRTZVisor [5] stands for microkernel real-time TrustZone-assisted hypervisor, and is an extended version of RTZVisor [19] following a microkernel-like architecture and an object-oriented approach. μRTZVisor targets security from the outset, by applying a secure development process. Contrarily to existing microkernel-based solutions, μRTZVisor is able to run nearly unmodified guest OSes, while, contrarily to existing TrustZone-assisted solutions, μRTZVisor is able to provide a high degree of functionality and configurability. μRTZVisor places also strong emphasis on the real-time support. The hypervisor was enhanced with a scheduling policy based on time domains. These time domains can have different priorities and are scheduled according to a preemptive, round-robin schema. Experiments were conducted using both synthetic and application benchmarks; on both cases results were similar, with a slight increase in degradation under realistic workloads (application benchmarks). To the best of our knowledge, RTZVisor and μRTZVisor have proven to be the unique TrustZone-assisted hypervisors allowing the coexistence of multiple and completely isolated OSes on the same hardware platform without ARM's Virtualization Extensions (VE) support.

III. DIHyper: Design

DIHyper is designed to provide a secure and self-protected runtime data integrity monitor for μRTZVisor using commodity hardware available in current ARM processors. In this section, the design goals, threat model and assumptions are discussed. Then, the proposed system approach is overviewed.

A. Design Goals

Designing a hypervisor with tight security requirements for safety-critical and/or real-time applications can be a demanding task. Performance and resource utilization are important metrics for embedded software. Furthermore, due to real-time constraints, determinism is often paramount in embedded systems, creating an extra challenge when designing any security countermeasure oriented towards these systems. DIHyper seeks the following goals.

Detection of non-control-data attacks This class of attacks is the focus of this work. DIHyper must detect attacks compromising μRTZVisor's integrity, which can compromise its availability. Control-data attacks can also be considered when modifications to code pointers occur.

Self-protection DIHyper executes at the same privilege level as μRTZVisor, with the highest privilege in the software stack. This requires a self-protection mechanism against attacks aiming at DIHyper itself.

Deterministic behavior The proposed security countermeasure must not lead to non-deterministic behavior while μRTZVisor is executing normally.

Maintainability DIHyper must easily adapt itself to newer versions of μRTZVisor. Its deployment process must be automated to create a complete and sound data integrity protection.

B. Threat Model and Assumptions

The adversary model consists of attackers able to exploit memory corruption vulnerabilities capable of reading and writing arbitrary locations in memory. To successfully launch an attack, attackers can inject and execute their own code or misuse existing one. More importantly, this threat model considers non-control-data attacks; thus, an attacker can commence an attack without ever modifying program's control-flow. This model assumes that a Control-Flow Integrity (CFI) implementation is being used and control-flow attacks are out of the scope of this work.

The hardware is considered trustworthy. A secure boot mechanism and MMU hardware must be in place. Secure boot is essential to guarantee the integrity of the μRTZVisor hypervisor and DIHyper, at boot-time. The hypervisor code is assumed to be vulnerable, due to its implementation in C++. In this model, it is assumed that the attacker, failing to compromise the hypervisor, will attempt to attack the deployed security mechanism. Hardware-related attacks are out of the scope of this threat model.

C. Proposed Approach

Figure 1 depicts the proposed runtime defense mechanism against non-control-data attacks. Executing at the same privilege level as the hypervisor, a *Remote Monitor* is introduced

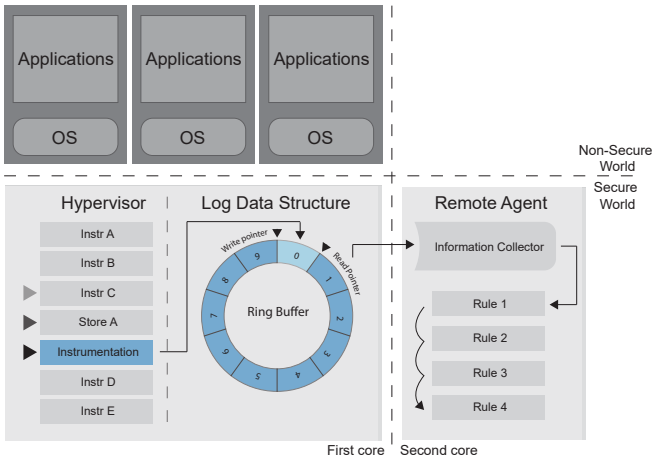

Figure 1. **Concurrent execution of an integrity monitor and μRTZVisor.** Data is collected through instrumentation and analyzed in the Remote Monitor, enforcing a developer-defined specification.

to intermittently verify data integrity rules, provided at design time. The specification aims to ensure lifetime correctness for all sensitive static data structures which, in turn, safeguard μRTZVisor's runtime data.

The Remote Monitor is isolated from μRTZVisor, leveraging the MMU to write-protect sensitive data and enforce a W$\oplus$E policy, where pages are configured either as writable or executable. Further, the introduction of a CFI scheme ensures that the target program follows a statically-defined Control Flow Graph (CFG). The Remote Monitor's *Information Collector* retrieves execution traces, generated by the instrumented μRTZVisor, containing information about write operations to sensitive static variables used to enforce the data integrity rules. This execution traces consist of both values written to critical variables as well as addresses used for indirect write operations. This scheme follows a lazy approach, as information is only collected when a write to a critical static variable occurs, in order to minimize performance overhead.

The *Ring Buffer* works as an intermediary, transferring information from the hypervisor to the Remote Monitor. Through compile-time instrumentation, extra instructions are injected into the hypervisor, logging sensitive information to the Ring Buffer.

IV. DIHyper: Implementation

Figure 2 presents the modifications to the compilation chain of μRTZVisor. First and foremost, μRTZVisor is instrumented, at compile-time, by an extended version of GCC. Then, the code for the Remote Monitor is automatically generated, using a developer-provided data integrity specification. At last, the Remote Monitor is inserted into μRTZVisor, through binary patching.

Delving into the specifics, two extensions or plug-ins were added to the GCC compiler: the *Type Analyzer* and the *Instrumentation Pass*. The former analyzes every developer-defined data type (e.g, struct/class definitions). Essentially, it creates a textual representation of the program's memory layout. The layout, containing developer-defined data types and offsets is then stored in a file (*Memory Layout*). This information is subsequently used to translate a data integrity specification to C++, creating the Remote Monitor. The Instrumentation Pass analyzes the code currently under compilation, in its intermediate GIMPLE IL representation, injecting instrumentation as required. Instrumentation assumes the form of inline assembly. Likewise, the Instrumentation Pass logs its operations to a file (*Instrumentation Metadata*), used by the *Rule Mapper* to generate the Remote Monitor. The Instrumentation Pass and Type Analyzer are independent entities.

The Rule Mapper maps a data integrity specification to C++, using the previously generated files as well as the developer-provided specification itself. Data integrity rules must be mapped to C++ since they are abstractly defined by the developer. Lastly, the Remote Monitor code is compiled and a binary blob is inserted into the μRTZVisor's executable. This is the final step in the compilation chain.

A. DI Specification

Data integrity rules can be created as a result of the instrumentation process. The devised rule types emerged from the developer's security requirements to secure the current version of the μRTZVisor hypervisor. Currently, there are four types of rules that can be defined; however, this can be extended as required.

- **Immutable array element** - Allows to define an element of an array as read-only. This array can be defined inside classes or it can be the static object itself.
- **Immutable Element** - This rule is a generalization of the previous. In this case, classes' member variables (i.e., fields) can be defined as read-only.
- **Integer Range** - Defines a range for the value of an integer variable.
- **Bit Values** - Enforces individual bit values on memory mapped registers.

Data integrity rules are field-sensitive, meaning they can be applied to classes' member variables or structures' fields. The rules provide finer granularity over commonly available hardware (i.e., MMU), when enforcing memory access policies (e.g., defining read-only memory areas). Write-protecting an internal part of a continuous memory block is a difficult task without close to word-level granularity to define access policies, except by changing program's memory layout which can introduce new errors or incompatibilities with legacy code. Being field-sensitive, this method is more granular than other software-based approaches such as WIT [20].

The state of memory mapped registers must be taken into consideration as μRTZVisor interacts directly with the hardware. The Bit Values rule type allows to specify individual bit values when updates to such registers occur.

Rule definition is ongoing work, aiming to create a concrete Domain Specific Language (DSL). In Figure 2, these rules are defined in the *Abstract Specification* text file, which is

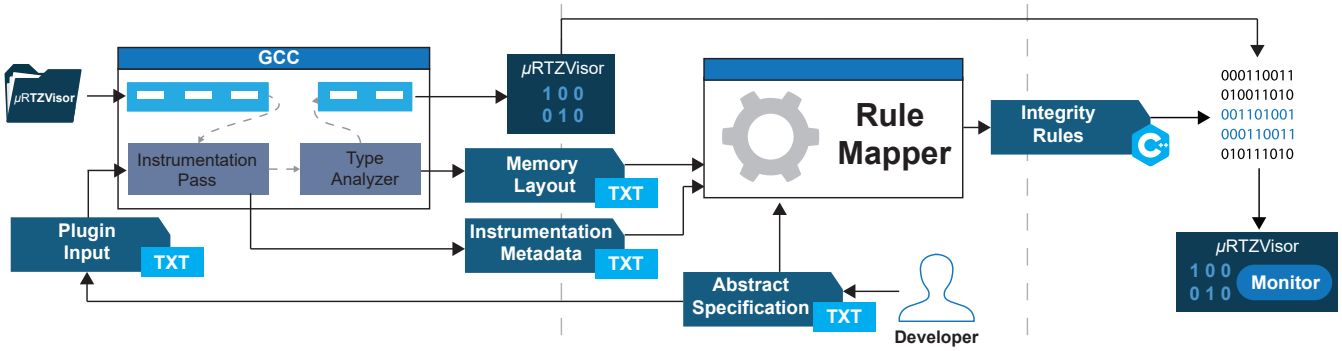

Figure 2. **Modifications to μRTZVisor's compilation chain to insert the Remote Monitor.** Inserting two plug-ins in GCC to instrument the code under compilation. Introduction of a Rule Mapper Java program to translate an abstract data integrity specification to C++, generating the Remote Monitor.

then used by the Instrumentation Pass to determine where the logging procedure instrumentation must be inserted.

B. Hypervisor Instrumentation

Instrumentation is unavoidable to record the execution traces of μRTZVisor into the *Log Data Structure*, depicted in Figure 1. With two options available, instrumentation at compile-time was chosen instead of binary patching. Compile-time instrumentation consists of extending the compiler with extra functionalities to generate supplementary code. With full access to μRTZVisor's source code, the semantic information provided by the compiler and its intermediary representations can be leveraged to automate this process of inserting new instructions. The Instrumentation Pass is a compiler pass.

The logging procedure instrumentation is inserted as inline assembly thus, being platform specific. The algorithm begins by loading the value of the Write Pointer, a pointer containing the index of the next location to be written in the ring buffer data structure. The logged data is stored after performing the arithmetic to calculate the address of the new log entry. In the end, the Write Pointer is incremented to point to the next log entry. The log has a statically defined size for each entry. While there is only one log, it can be divided into two virtual logs: the Value Log and the Address Log. The former stores a copy of the data written into a sensitive data structure. The latter collects data used in indirect write operations such as addresses or/and copies of integer variables used to index arrays. Listing 1 denotes an indirect write, extracted from μRTZVisor, to a sensitive static object. To log this operation, the runtime value of `guest_num`, represented in the optimized GIMPLE IL representation in Listing 2 as `guest_num.0_19`, will be copied to the Value Log. Similarly, the variables used to resolve the indirection (`this_6` and `prehitmp_81`) are logged as well. Instrumentation is inserted after line 3 in Listing 2.

When instrumentation is inserted, an unique identifier is associated with that code block and logged as the code executes, allowing to identify the origin of new log entries in the Remote Monitor. Logged data can be categorized as: the written value, the address of the object, variables used to index an array and the identifier. A new log entry always contains the

```
1  void GuestManager::GuestCreate(GuestConfig const
       &config){
2  static int32_t guest_num = 0;
3
4  Guest &rguest = guestList[guest_num];
5  rguest.id = guest_num;
6  ...
7  return;
8  }
9
```

Listing 1. C++ code extracted from μRTZVisor.

```
1  ...
2  guest_num.0_19 = guest_num;
3  MEM[(struct Guest
       &)this_6(D)].guestList[prephitmp_81].id =
       guest_num.0_19;
4  ...
5
```

Listing 2. Snippet of GIMPLE code equivalent to lines 4 and 5 of Listing 1.

identifier; however, the other fields are optional. For example, as constants are written to sensitive data structures the written value is not copied. Similarly, if there are no indirections in the write operation, only the written value gets copied to the log. Information about the inserted instrumentation is stored in the *Instrumentation Metadata* text file, depicted in Figure 2, to be subsequently used by the Rule Mapper. By logging both values and sources of indirection - in write operations -, it is possible to perform either sanity or bound checking, respectively.

C. Remote Monitor

The Remote Monitor is a standalone bare-metal application, distinct from the hypervisor and executing in a separate core from μRTZVisor, allowing to concurrently perform security verifications. With the data generated from the compilation of μRTZVisor, the Rule Mapper can generate the verification code to enforce the developer-defined data integrity rules, at runtime. Firstly, the Rule Mapper generates code to resolve ambiguities on the accessed memory location while performing bound checking. For this purpose, it uses both the identifier and the values stored in the Address Log. Furthermore, for this approach to be field-sensitive, bound checking is performed in any indirect write to any class member variable of a

sensitive static object. Then, rules are translated to their C++ representation and enforced when write operations to specific memory locations occur.

The Remote Monitor and the μRTZVisor executables must be linked into a single binary image, as depicted in Figure 2. A new section is included in the μRTZVisor object file, using the GNU obj-copy utility. At runtime, the MMU is used to write-protect both the log, except when instrumentation is being executed, and the Remote Monitor code from modifications originated in μRTZVisor. Regarding TrustZone, only the Secure MMU interface is configured to protect these memory regions, since an attacker must violate the VMM software to access this sensitive memory regions and the VMM is protected by DIHyper (complemented with a CFI implementation). This extra layer of protection tries to mitigate attacks to the μRTZVisor hypervisor targeting the security mechanism.

V. Evaluation

DIHyper was evaluated on a Zybo platform targeting a dual ARM Cortex-A9, running at 650 MHz. The evaluation focused on security (Section V-A) and performance (Section V-B). To evaluate security, we conduct a security analysis. To evaluate the performance overhead, we used the MiBench Benchmark Suite.

A. Security Analysis

In this section, we evaluate the security of our solution by summarizing and explaining how two security invariants must be enforced:

- **Tamper proof** - Attackers should not manipulate either the data or code of the Remote Monitor. This includes the logs - shared between the target program (μRTZVisor) and the Remote Monitor - and the instrumentation.
- **Non-bypassable** - Attackers cannot bypass the logging instrumentation.

μRTZVisor, using TrustZone technology, provides the first line of defense against attacks originated in VMs, either aiming at the VMM or the Remote Monitor. So, an attacker must exploit a memory corruption vulnerability while the processor is in Monitor mode - VMM privilege level - to be able to break the security invariants. If such an exploit occurs, the devised security mechanism has memory protections in place that ensure the first invariant. Namely, the MMU enforces access control policies on memory. Tampering with the Remote Monitor is only possible by disabling the MMU or modifying the translation tables. Disabling the MMU requires special instructions that are inserted with the instrumentation and are not available as a function. Furthermore, an attacker cannot disable the MMU without re-enabling it as the instrumentation inserted in critical writes is an atomic basic block (i.e., does not possess any branch instructions). Log data structures are read-only, also enforced by MMU. Log write protections are disabled by the inserted instrumentation and re-enabled shortly after the log update. While attackers can target the page table, their efforts would be fruitless, as the physical addresses of the page table are not mapped in the page table itself. Changing the base address for the page table also requires specific assembly instructions. By adding all instrumentation after the write operation and due to the aforementioned atomicity, invariant 2) is enforced. An attacker cannot perform a write and avoid instrumentation as branch instructions are not present.

Complementing DIHyper with a CFI implementation allows to thwart attacks either to data- or control-planes of the VMM. The proposed approach aims at mitigating non-control-data attacks. Considering code pointers as critical variables that must be protected, it also provides control-flow protection to a certain extent. However, the stack is completely disregarded as it is not considered a source of expressive non-control-data attacks. CFI arises as a complement to further secure the control-plane of μRTZVisor.

Novel non-control-data attacks, such as DOP [10], are mitigated in their expressiveness as an attacker is not able to arbitrarily modify a sensitive static variable. However, due to the granularity of the proposed approach these attacks are still possible. For example, the stack can be corrupted to perform operations on spurious data without diverging from the legal control-flow. Although that would go undetected, once an indirect access tried to access critical variables without permission, it would most likely be stopped by the MMU. Furthermore, attacks aiming at data assigned to critical variables are eliminated as that data is further analyzed by the Remote Monitor. Additionally, the CFI implementation allows to detect further control-flow related attacks.

A special characteristic of this log-based protection scheme is the detection of transient attacks. In this type of attacks, the adversary may cause harm and then hide its traces. With this scheme, this kind of attacks are not possible, as every write to critical variables and variation in the control-flow are registered for further analysis. Periodic integrity verification tools can only detect permanent integrity damage. For example, methods using introspection techniques, such as HyperSEntry, are invoked periodically to analyze the memory of a monitored entity.

B. Performance

To evaluate the performance overhead introduced by DIHyper, an application benchmark was executed to measure runtime overhead, using the MiBench Embedded Benchmark Suite. μRTZVisor was assigned to the primary core, while DIHyper was configured to run on the secondary core. The evaluation focused on the MiBench's automotive suite. Figure 3 depicts the results of running the benchmarks on a non-instrumented version of μRTZVisor (normalized and highlighted in a darker blue) and on the proposed solution with DIHyper (light blue). According to Figure 3, DIHyper introduces a performance overhead of approximately 4%. This remains the same throughout the entire test suite as all applications only use μRTZvisor's scheduling capabilities.

A set of tests was also conducted to measure the cost of IPC-related hypercalls with and without DIHyper. The focus on this specific set of hypercalls is due to their relevance on

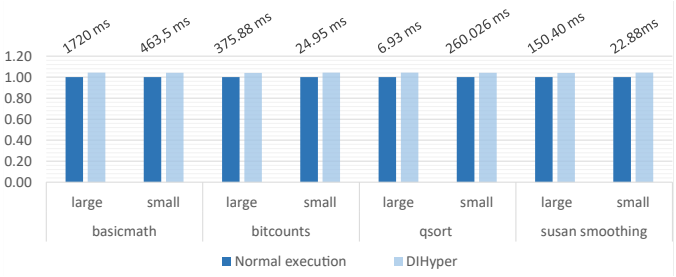

Figure 3. MiBench Automotive Benchmark results, comparing the virtualized execution of a FreeRTOS guest against one with DIHyper.

Table I
IPC-RELATED HYPERCALLS EXECUTION TIME (μS).

Port Operations	Non-Instrumented	Instrumented
SendMsg	4.36	4.37
RecvMsg	4.17	4.18
Notify	2.36	2.38
SendReceive	5.49	5.51
SendReply	3.70	3.70
ConfigPort	3.65	3.66

the μRTZVisor's microkernel implementation. All hypercalls, except those that perform copy of data should have a constant cost; however, due to our focus on the latter we assume a message size of 64 bytes. The used data integrity specification does not consider data transferred between guest OSes as crucial for μRTZVisor's integrity. Runtime overhead is reduced when performing such data intensive operations, which are the most common.

VI. Conclusion

Virtualization has been largely adopted in the embedded systems domain to ensure robust isolation and fault containment among systems with different criticalities. However, the successful number of attacks against virtualization infrastructures have raised seriously concerns regarding the trustworthiness of existing hypervisors. In this paper we presented the development of DIHyper, a runtime monitor which reliably establishes the continuous data integrity of a TrustZone-assisted hypervisor (i.e., μRTZVisor). We have described a self-protecting solution capable of detecting the most recent non-control-data attacks and which can coexist with other control-flow integrity mechanisms. DIHyper contributes with increased security for μRTZVisor.

Work in the near future will mainly focus in the implementation of response mechanisms which aim at restoring system's integrity, whenever a security violation is detected through DIHyper. Going forward, our security mechanism will be extended to cover dynamic data as well, providing total protection over the data-plane.

VII. Acknowledgments

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT - *Fundação para a Ciência e Tecnologia* within the Project Scope: UID/CEC/00319/2013.

References

[1] G. Heiser, "Virtualizing embedded systems: why bother?" in *Proceedings of the 48th Design Automation Conference*. ACM, June 2011, pp. 901–905.

[2] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive e/e-systems," in *9th IEEE International Symposium on Industrial Embedded Systems*. IEEE, June 2014, pp. 189–198.

[3] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, "IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing*, vol. 21, no. 1, pp. 40–47, Jan 2017.

[4] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 401–412.

[5] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "μRTZVisor: A Secure and Safe Real-Time Hypervisor," *Electronics*, vol. 6, no. 4, 2017.

[6] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 203–216.

[7] G. Cicero, A. Biondi, G. Buttazzo, and A. Patel, "Reconciling Security with Virtualization: A Dual-Hypervisor Design for ARM TrustZone," in *Proceedings of the 18th IEEE International Conference on Industrial Technology*, 2018.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium*, vol. 14, August 2005.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[10] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 969–986.

[11] I. Díez-Franco and I. Santos, "Data is flowing in the wind: A review of data-flow integrity methods to overcome non-control-data attacks," in *International Conference on EUropean Transnational Education*. Springer, 2016, pp. 536–544.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd Symposium on Operating Systems Principles*, 2009, pp. 207–220.

[13] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 38–49.

[14] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[15] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, May 2008, pp. 233–247.

[16] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[17] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.

[18] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006, pp. 147–160.

[19] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems," *IEEE Comp. Arch. Letters*, vol. 16, no. 2, pp. 158–161, 2017.

[20] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *IEEE Symposium on Security and Privacy*, May 2008, pp. 263–277.