Towards Host Intrusion Detection For Embedded Industrial Systems

Marine Kadar
SYSGO GmbH, Germany
marine.kadar@sysgo.com

Sergey Tverdyshev
SYSGO GmbH, Germany
sergey.tverdyshev@sysgo.com

Gerhard Fohler
TU Kaiserslautern, Germany
fohler@eit.uni-kl.de

Abstract—
Original Equipment Manufacturers now embed hardware virtualization in car equipments to reduce costs and hardware complexity, while allowing more functionalities, such as connectivity. This evolution forces the cohabitation of distinct criticality domains on the same hardware, reaffirming the need for security. Because of the trade-off between performance and system overall complexity, deploying security becomes a challenging balancing act. Host Intrusion Detection Systems (HIDS) security protects the behavior of a program at runtime: it monitors the program execution flow to distinguish threats from benign activity. This paper presents a novel runtime security solution for embedded mixed-criticality systems, which integrates HIDS in a partitioned system based on Multiple Independent Levels of Security (MILS) architecture. Our HIDS monitors a program’s execution by observing both hardware and software signals; there is to our knowledge no HIDS providing such precise representation of program execution.

Index Terms—Intrusion Detection, MILS, Mixed-Criticality

I. INTRODUCTION

Information Technology (IT) security is an active research topic, for protecting generic computers and networks, avoiding malicious programs from running on the protected machine. For example, a threat can be originated from a competitor attempting to steal company sensitive data, or from an adversary trying to control safety critical equipment such as a nuclear plant or a train. While contemporary IT security implements widespread techniques such as antivirus, intrusion detection systems (IDS), or system information and event management, comparatively few solutions apply to embedded systems. IT solutions can not directly run on embedded system, because both system platforms differ: embedded systems generally induce limited hardware resources and their architecture (hardware and software) varies much for different use-cases. However today, embedded systems need robust security, as they become more open, more complex, and thus more exposed to threats. Security is even more essential for critical equipment such as in-vehicle embedded systems: whenever a system security is compromised, there is also a threat on system safety.

Because of continuous hardware performance improvement, modern cars integrate many services. Previous distinct electronic control units (ECU) now coexist on a same hardware inside virtual machines. Embedded systems are now interconnected: cars can be accessed remotely. These new system characteristics increase the attack surface. Thus, the risk of threat increases. The attack can also directly come from the owner of the car. Hence, security must be deployed locally on the target.

Host Intrusion Detection System (HIDS) is a runtime security method for controlling the behavior of a monitored software during its execution to detect any malware activity. In this paper, we present a HIDS for embedded mixed-criticality systems, based on the observation of both hardware and software signals; this representation of system execution has to our knowledge not yet been investigated. We discuss the local deployment of the framework, and propose a methodological approach for integrating this technology in a safety critical environment, using MILS system design properties.

Section II describes the context of this study. We discuss related work in section III. Section IV presents our security solution. Then, section V further analyzes its integration in a mixed-criticality environment, while section VI focuses on challenges for deployment in business production. Finally, section VII summarizes this paper.

II. CONTEXT DEFINITION

A. System Model

Figure 1 provides an example of automotive system architecture. The hardware is a generic embedded platform with limited resources and networking ability. In the example, the system runs real-time critical user-level applications such as the car cockpit display and GPS trajectory calculation software, as well as a security critical application for software update from the network. It also includes a non-critical Android guest OS for in-vehicle infotainment. In this context, we generalize the definition of monitoree as a software to monitor in user space. This program can be a complex workload such as guest OS, as well as real-time critical software. It is typically a business logic program provided by a third-part. The source code or specification is not systematically available; the security solution must assimilate it as a black box. The MILS Separation Kernel [1] ensures the strong separation of the user-level software on the hardware and provides secure communication channels. Section III develops further this security implementation.
B. Threat Model

We consider intrusions occurring inside a user-level software, which alters runtime behavior of this victim. Because of system connectivity, the network is a potential entry point, as illustrated on Figure 1. The attack can also be the result of a physical attack such as malicious memory bit flips, or software memory corruption. The threat is then a malicious deviation in runtime execution of the program we want to protect.

C. System Asset

In this work, our objective is to protect user-level applications exposed to the threat model described in section II-B. Hence, our asset is the integrity of the monitoree’s execution behavior.

Let \( p \) be a program, we define its execution graph \( G(p) = (S(p), B(p)) \), where \( G(p) \) models the execution path of \( p \) by a sequence of intermediate abstract states given in \( S(p) \). \( B(p) \) defines all possible transitions between two states in the graph: for every execution path \( e(p) \), \( e(p) \subseteq B(p) \). We define a golden model \( GM(p) \) to describe all trustworthy execution paths (equation 1). The execution path integrity is guaranteed if and only if \( e(p) \subseteq GM(p) \). We can then summarize our asset as the following invariant on the complete program execution with equation 2.

\[
G(p) \subseteq B(p) \\
\forall t. e(p)(t) \subseteq GM(p)
\]

III. RELATED WORK

A. Security by Design with Partitioning

Partitioning, one security by design method, defines distinct criticality domains of execution. Its asset consists in the separation of the distinct domains, to guarantee basic security properties such as confidentiality, integrity, and availability. The Multiple Independent Layers of Security (MILS) architecture is one direct application. It implements both hardware and software separation mechanisms to isolate the defined criticality domains, as well as secure communication channels to control the information flow transiting between the different domains’s interfaces. It embeds the notion of separation kernel [2], an execution domain composed of all system software running privilege operations.

B. Runtime Security with Host Intrusion Detection Systems

However, some intrusions target assets which are not protected by the security by design. For example, Return Oriented Programming attack was first highlighted in [3]: the authors successfully run a shellcode by manipulating the program control flow to execute a malicious chain of functions from the C standard library. Such attack indeed does not corrupt the victim program’s enclave. However, the program behaves differently from how it is expected to execute, to eventually compromise its own execution and possibly affect other system components. Because we target complex virtualized systems including multiple software and hardware runtime interactions, such dynamic attacks are in scope.

Host Intrusion Detection Systems (HIDS) represent a runtime security method to monitor the global execution of monitored applications running from user space. HIDS define a baseline of execution for a specific application, to distinguish benign operation sequences from malicious activity. During program execution, HIDS is a runtime service, which first collects traces of execution, analyze them in a second step, to finally infer a probability of intrusion.

1) Modeling System Interactions: The HIDS collects data from the monitoree to represent its interactions with other software components, i.e. OS services, and its activity on the hardware, e.g. memory stress, processor execution time, etc.

A software interaction from a virtual machine (VM) can either be a direct OS service request, or a communication channel with another VM. In the separation kernel architecture, two user-level VM can not directly interact, as the communication goes through the kernel stack. Since late nineties with [4], many research work have highlighted the security potential of system call tracing based HIDS applied to generic computers.

Recent research has emphasized the applicability of Hardware Performance Counters (HPC) tracing to detect malicious exploits: e.g. [5] shows how to detect Spectre [6] by observing cache activity at runtime. Krishnamurthy et al. [7] use HPC for anomaly detection in real-time multi-threaded processes. As a main issue, the restricted amount of simultaneous available HPC limits the coverage of all possible hardware interactions. For instance, ARMv8 Cortex-A processors have to our knowledge up to 6 counters per core, despite a total of more than 40 configurable events [8].

2) Detecting the Threats: We assimilate the traces as words in a language where sentences are sequences of a program execution. The execution time and execution order properties in the trace sequences motivate the use of Natural Language Processing (NLP) techniques for predicting a future trace from a preceding sequence. Finally, the trace prediction is compared to the actual trace: the detector identifies an anomaly above a predefined deviation threshold. For example, [9] applies a subclass of recurrent neural networks to detect anomalies in system call streams.
IV. INTRUSION DETECTION IMPLEMENTATION

A. Combining Security Layers

Because it is deployed locally on the monitored target, the HIDS alone can not guarantee system security: for the intrusion detection to be trustworthy, the attacker should not be able to access the HIDS. We propose to ensure the external observability of the monitoree by integrating the HIDS service in the separation kernel. HIDS is then part of the asset protected by a secure system design.

Therefore we propose a system security solution integrating the HIDS concept in a MILS system architecture. MILS architecture protects the interfaces between the different system layers: user-level programs, kernel services, and hardware components. HIDS technique defends the integrity of user-level programs at runtime. The HIDS itself runs locally on the system. Its security basic properties such as confidentiality, integrity, and availability are part of MILS architecture asset. We use a MILS solution in this work, without any modification.

B. Mapping HIDS to Execution Integrity Formal Definition

1) Intrusion Definition: We map the concept of HIDS to the execution graph formalization described in section II-C. An intrusion corresponds to a broken execution integrity invariant (equation 3). According to section II-C, the HIDS should then spot any unauthorized transition between two execution states at runtime. It can control the asset recursively at runtime (equation 4). Thus, each transition between two states is compared to the golden model, while the previous path has already been checked.

\[ \exists m, b_m \in e^{(p)} \land b_m \not\in GM^{(p)} \] (3)

\[ \forall t > 0, \exists m \in [1, M], b_m \in B^{(p)} \land e^{(p)}(t) = \{ e^{(p)}(t - 1), b_m \} \] (4)

2) From Abstract States to Observable Traces: We represent the abstract states of execution defined in the system assets with relevant observable signals at runtime. Because the monitoree is a black box, we can only access its interface. Therefore, as developed in section III-B1, we substitute a program execution state by a combination of hardware and software signals. To our knowledge, there is no HIDS based on the combination of both data types for a faithful state description.

3) Graph Coverage: The HIDS is based on comparing a golden model of normal execution to compare with current execution of the program at runtime. To build a reliable golden model of execution, we train a NLP model with an extended dataset, which provides high coverage on the execution graph.

V. INTRUSION DETECTION SYSTEM INTEGRATION

A. Monitoree

The HIDS perceives the monitoree as a black box, observing its interfaces to detect misuse operations. Thus, this security is non intrusive: it does not require modifications in the monitored software.

The quality of the detection relies on an exhaustive golden model to distinguish normal from malicious execution. Therefore, the HIDS requires a representative dataset of monitoree’s traces of execution; although we can easily generate many traces corresponding to normal execution, it is generally not trivial to obtain malicious traces. For this second case, the Common Vulnerability and Exposure database [10] can help as a reference baseline.

B. Separation Kernel

1) HIDS Implementation Constraints: System call events are handled at kernel level. HPC must be unwritable from user space to prevent any attempt of corruption from an attacker operating from this execution level. Thus, both system call and HPC events tracing must be performed at kernel level. Data analysis can be decorrelated from tracing time. Depending on the implementation complexity, it can require the use of dedicated Artificial Intelligence frameworks. For these two reasons and to prevent any safety impact, the detection must run from user space inside an isolated VM. Hence, we decompose the HIDS as detailed on Figure 2, between the collector kernel driver and the analyzer user-level application. The analyzer can apply recovering strategies, for instance restarting a VM detected malicious. Both services run privilege operations and therefore, they must be part of the MILS separation kernel.

2) Security of the HIDS: A worthwhile detection depends on the external observability of the monitoree. We assume the kernel layer secure by design: an attacker has no direct access to the HIDS executing at kernel level. She also can not access the user-level HIDS implementation because of MILS architecture separation properties. To limit interferences, the analyzer runs on reserved area of system hardware (i.e. dedicated memory region and CPU cores).

C. HIDS Integration in a Critical Environment

Because it is non intrusive, the HIDS does not directly affect certification processes of the monitoree. However, system safety certification must reevaluate system Worst Case Execution Times considering new delays induced by kernel add-ons.
VI. DEPLOYMENT CHALLENGES

A. Security Infrastructure Costs

Previous work [11] showed a limited impact on time performance for system calls tracing, considering different types of monitoree programs. Depending on the amount of generated system call, the rate is generally below 1% of the total monitoree execution time. We now combine HPC monitoring with system call tracing. We expect the additional overhead to be small, since these operations consist in reading and writing to few CPU registers. The spatial overhead is non negligible: the HIDS analyzer monopolizes parts of the monitored system’s computing resources such as reserved memory area and CPU cores. The use-case defines the trade-off between performance impact and optimizations to apply. For the system to react before any severe damage applies, the detection must be fast enough. Ideally, the HIDS detects and counter the attack before it reaches its goal. Trace infering is low cost compared to model training; dedicated hardware such as GPU or FPGA can still contribute to accelerate this step.

B. Quality of the Detection

For the detection to be usable, it first has to be reliable enough. This means it should minimize false alarms, while maximizing the detection of actual threats. Data is a key for accurate detection. For example, Natural Language Processing manipulates datasets gathering millions of samples. Collecting representative data from normal and malicious activities in embedded mixed-criticality is not trivial. There are indeed diverse system configurations. For example depending on the use-case, a system can run Linux, AUTOSAR, PikeOS, ARINC OS. Hardware platforms can vary, and monitoree profiles as well: the monitoree can be a network client, Programmable Logic Controller, guest OS, etc.

C. System Recovery Strategies

Once a threat has been detected, the HIDS chooses between the following recovery strategies to mitigate the attack:

- **Suspend/kill/reboot:** we interrupt the intrusion.
- **Isolate:** we limit interactions with other system components to avoid any attack propagation.
- **Migrate:** this is usually not possible in a critical environment, where every VM is configured statically.
- **Signal/log:** whenever there is high uncertainty or the risk is too important, we only notify the system administrator of a potential threat.

Two parameters define the strategy to apply. First the detection trust level given by the HIDS provides a probability of intrusion between 0 and 1. Then the monitoree profile combines a criticality level, an indicator of openness and exposure to threats, with a level of interactions with other system components.

VII. CONCLUSION AND FUTURE WORK

We presented a Host Intrusion Detection System (HIDS) to secure runtime execution of a program in a mixed-criticality embedded system. To isolate the HIDS service from potential adversary, we defined it part of the separation kernel inside a MILS system architecture. After formalizing the definition of a program’s execution, we based our HIDS on a substantial execution state description, covering both hardware and software system signals. Since literature has to our knowledge not explored this approach, we will evaluate the deployment of our HIDS framework in future work, through concrete use-cases. The embedded environment will be a major constraint in this task: firstly because of its hardware resources limitation, and then because building software golden models on these platforms is non trivial. Data is indeed a crucial factor for a reliable detection. However because of the very context dependent system definition, much effort has to be spent for collecting plentiful representative data.

REFERENCES

[1] S. Tverdyshev, H. Blasum, B. Langenstein, J. Maebe, B. De Sutter, B. Leconte, B. Triquet, K. Miller, M. Paulitsch, A. Sdng-Freiherr von Blomberg, and A. Tillequin, “MILS architecture,” Zenodo, 2013.

[2] J. Rushby, “The design and verification of secure systems,” in Eighth ACM Symposium on Operating System Principles, 1981.

[3] S. Designer. (1997) Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63

[4] S. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” in Journal of Computer Security, 1998.

[5] D. Fiser and W. G. Sanchez. (2018) Detecting attacks that exploit Meltdown and Spectre with performance counters. https://blog.trendmicro.com/trendlabs-security-intelligence/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters/

[6] P. Kocher, J. Horn, A. Fagh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in 40th IEEE Symposium on Security and Privacy, 2019.

[7] P. Krishnamurthy, R. Karri, and F. Khorrarami, “Anomaly detection in real-time multi-threaded processes using hardware performance counters,” in IEEE Transactions on Information Forensics and Security, 2020.

[8] ARMv8-A Architecture Reference Manual.

[9] G. Kim, H. Yi, J. Lee, Y. Paek, and S. Yoon, “LSTM-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems,” in Proceedings of the 28th Annual Computer Security Applications Conference, 2012.

[10] Common Vulnerabilities and Exposures. https://cve.mitre.org/index.html

[11] M. Kadar, S. Tverdyshev, and G. Fohler, “System calls instrumentation for intrusion detection in embedded mixed-criticality systems,” in 4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems, 2019.