MANTECH Publications
Journal of Computational Intelligence Theory and Its Applications
Volume 2 Issue 1

Embedded Monitoring and Control

Pooja Sutar [1], S. G. Pawar[2]
Department of Computer Engineering
Sinhgad Institute of Technology and Science (Narhe)
Corresponding Authors: *poojasutar424@gmail.com[1], sgpawar_sits@sinhgad.edu[2]*

Abstract

Modern web applications make frequent use of third-party scripts, often in ways that allow scripts loaded from external servers to make unrestricted changes to the embedding page and access critical resources including private user information. This paper introduces tools to assist site administrators in understanding, monitoring, and restricting the behavior of third-party scripts embedded in their site. We developed Script Inspector, a modified browser that can intercept, record, and check third-party script accesses to critical resources against security policies, along with a Visualize tool that allows users to conveniently view recorded script behaviors and candidate policies and a Policy Generator tool that aids script providers and site administrators in writing policies. Site administrators can manually refine these policies with minimal effort to produce policies that effectively and robustly limit the behavior of embedded scripts.

Keywords: *Microcontroller, Security, Runtime requirements monitoring.*

I. INTRODUCTION

Defining and verifying system requirements are a critical challenge in embedded systems. During development, system verification often includes test-based and formal verification methods. Overall, test and verification comprise as much as 75% of the design effort. Even with the significant effort, exhaustively verifying software and hardware to be bug free before deployment is prohibitive. This implies that software and hardware bugs may have

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

unintended consequences, particularly in safety-critical applications. Additionally, accounting for all operating conditions of embedded systems at the design stage is infeasible, as all possible conditions that may affect the system performance cannot be evaluated a priori.

As evidenced from numerous examples, testing and verification solely during development is insufficient. Embedded systems also present unique challenges in monitoring correctness. Tight integration of hardware and software means monitoring must consider both hardware and software, which increases the monitoring complexity. This tight integration prevents the use of traditional methods to observe, monitor, or control the state of these components at Runtime monitoring should be nonintrusive, as intrusive monitoring methods can perturb a system's behavior and result.

Hardware-based runtime monitoring can be utilized to dynamically check that the system implementation adheres to required properties. Several hardware-based, software monitoring approaches have been proposed, summarized in [5]. TA and LTL are used in many approaches (e.g., P2V), but capture complex temporal ordering of events and not precise time constraints. Other approaches (e.g., TxMon, SecMon, HARD, and AVIO-H) use nonprogrammable hardware or require processor modifications, which restricts software updates and increases design complexity.

Some approaches (e.g., iWatcher and ReEnact) use hardware to accelerate the analysis of monitored events. While less intrusive than pure software-based approaches, the limited use of software is still intrusive.

Hardware utilizes the RMG to monitor the system execution behavior and detect execution states that differ from the correct system behaviors. For observed failures, the hardware provides detailed information on the current system state that can be used to diagnose the root cause.

In contrast to TA and LTL, UML sequence diagram are utilized to provide a closer and more direct representation of the system's software, results in more efficient monitoring hardware, and avoids state explosion common in those approaches.

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

II. RUNTIME REQIUREMENTS MONITORING METHODOLOGY

The NIRM methodology for in-situ requirements monitoring consists of:

1) a design time methodology for system requirements specification and analysis and

2) in-situ, nonintrusive hardware for monitoring and checking requirements at runtime within the deployed system.

3) initial software development and UML sequence diagrams;

4) requirements extraction;

5) RMG generation;

6) RMG analysis and reduction;

7) RMG configuration generation and binary integration; and

8) in-situ, nonintrusive monitoring of system execution.

A. Real-Time UML sequence Diagram

Sequence diagrams capture the execution flow of programs for individual tasks, communication between tasks, and interfaces with hardware components for different use case scenarios. These sequence diagrams are a standard approach used within the design and analysis of software. Real-time UML [3] and the UML profile for modeling and analysis of real-time and embedded systems [7] provide extensions to UML sequence diagrams specifying real-time scheduling constraints. However, these UML models primarily focus on facilitating schedulability analysis and performance analysis, but not in-situ requirements monitoring.

To support the specification of timing requirements, the NIRM approach utilizes extensions to real-time UML sequence diagrams that incorporate timing annotations to specify timing constraints for individual operations and sequence of operations, such that existing tools for creating sequence diagrams can be directly utilized.

Timing annotations capture constraints for the total elapsed time between events, which typically represents the worst-case execution time (WCET) determined using existing real-time analysis methods.

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

Fig. 1(a) presents an illustrative example of a single UML sequence diagram with timing annotations for sensor acquisition within the navigation, speed, and steering control operations of an autonomous vehicle.

The sequence diagram specifies the order of interactions between those components, the names of messages that are passed between components, and the timing requirements. Designers can specify numerous sequence diagrams to capture different execution scenarios, which can be integrated together. Events and timing annotations can be specified at different granularity, ranging from fine-grained monitoring of specific conditions or loops to coarse-grained monitoring of an entire function or task.

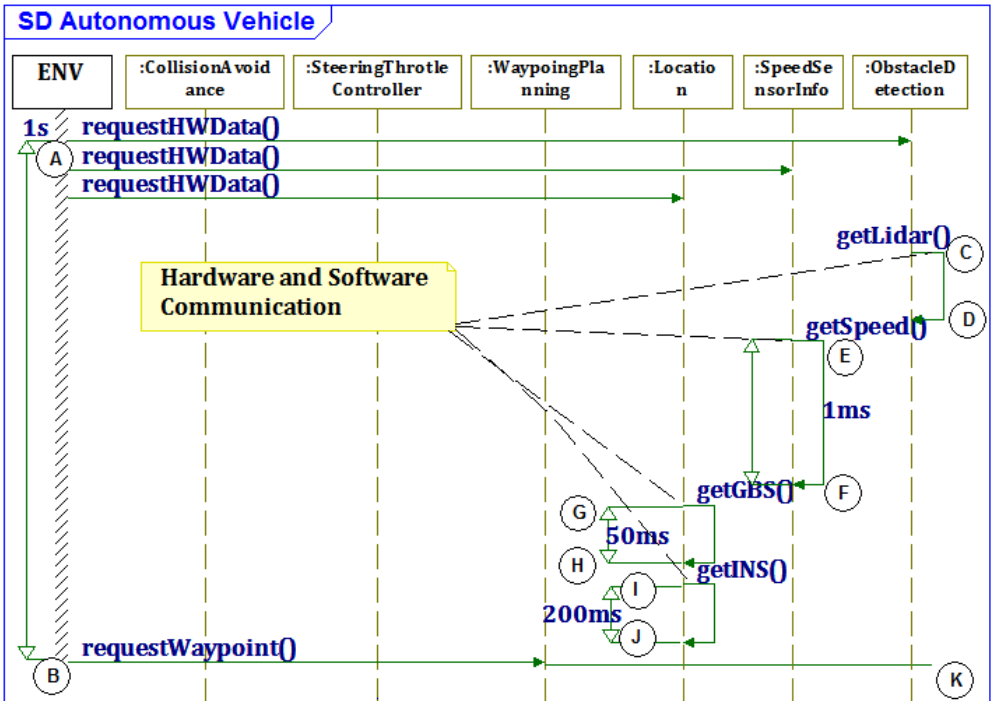


Fig. 1 (a) Definition of events superimposed on UML sequence diagram

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

B. Runtime Monitoring graph

We define an RMG, extracted from the sequence diagrams, that captures all requirements that can be monitored. The RMG defines the set of events (or probes) that must be observed at runtime, the correct execution order for those events, and timing constraints between events, where applicable. All messages and causality of messages (i.e., sender and receiver) within the sequence diagrams are considered system events.

Fig. 2(a) illustrates the extraction of events within the UML sequence diagram for the sensor data acquisition of the autonomous vehicle, which represents an initial 1:1 mapping between a single sequence diagram and RMG. The RMG enables the extraction and integration of events and timing information from multiple sequence diagrams into a single model, which can leverage hierarchy to efficiently represent system requirements, although the automated identification of hierarchy is left as future work.

The equations are an exception to the prescribed specifications of this template. You will need to determine whether or not your equation should be typed using either the Times New Roman or the Symbol font (please no other font). To create multileveled equations, it may be necessary to treat the equation as a graphic and insert it into the text after your paper is styled.

Number equations consecutively. Equation numbers, within parentheses, are to position flush right, as in (1), using a right tab stop. To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use a long dash rather than a hyphen for a minus sign. Punctuate equations with commas or periods when they are part of a sentence, as in

$$a + b = \gamma \qquad (1)$$

Note that the equation is centered using a center tab stop. Be sure that the symbols in your equation have been defined before or immediately following the equation. Use "(1)," not "Eq. (1)" or "equation (1)," except at the beginning of a sentence: "Equation (1) is ..."

C. Runtime memory attacks

Adversaries exploit vulnerabilities present in the software to perform runtime attacks on memory, such as buffer overflow based stack smashing and CRAs. Buffer overflow based attacks are prevalent in embedded

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

systems, because the software running on them mainly uses native languages C/C++ and assembly—which lack bounds checking for the variables. In addition to this, the increasing complexity and connectivity in these systems result in increased vulnerabilities and a large attack surface, consequently providing ample opportunities for performing attacks.

In general, the attackers exploit the vulnerabilities in the victim software by gaining the access to the dynamic data segment, such as the stack or heap to overflow the allocated data space of the variables. In the stack smashing, first, the attackers inject their own code by providing malicious inputs to the variables and then overwrite the return address in the stack to redirect the control flow to their injected code. While in the case of CRAs—ROP and JOP—the attackers use small instruction sequences, called gadgets, present The adversaries first create the payloads, consisting of gadgets or the code pointers that target the gadgets in the library or victim program.

Then, they execute these gadgets in an orchestrated order to perform malicious operations. Though, DEP or W⊕X protection [3] (which implements non executable memory) can thwart the code injection attack, CRAs can still be accomplished, as they use existing code sequences instead of the injected code.

III. RELATED WORKS

We discuss the related work in two parts. The first part summarizes the software-based protection approaches, while the second part discusses the more relevant hardware-based architectural techniques. We grouped the relevant defense approaches based on their techniques and implementation platform (software or hardware), as shown in Table I.

A. Software Based Approaches

The runtime attacks on memory mainly use the buffer overflow technique [16], which still remains the most prevalent attacks until now, and CRA. Stack smashing is a common type of buffer overflow technique used by attackers to inject their own code in the stack. Several software techniques have been proposed to prevent buffer overflow attacks. Stack Guard [16] uses a marker between a function pointer and a return address in the stack, whereas RAD [17] uses protection code into the prologs and epilogs of function calls to protect the overflow of the return address.

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

To protect from the CRA, Davi et al. [18] proposed a technique to monitor the repeated small sequences of instructions ending in return statements. Li et al. [19] proposed binary rewriting techniques to replace the return statements. The major limitations of software techniques are the high performance overhead and an increase in the code size resulted from the source code or binary modification. Besides, the software approaches have limited coverage of security vulnerabilities, which lead to unknown attacks.

In addition, the software solutions are mainly based on the known instructions at the compilation time, which can be bypassed by the unintended branch instructions. The unintended branch instructions are formulated by pointing the PC to the middle of the multibyte instructions. For the ISA, such as Intel, which has multibyte instructions, pointing to the middle of instructions leads to the formation of different instructions that are not intended during the compilation time.

B. Hardware Based Approaches

Of greater relevance to our approach are the micro architectural techniques. DEP was implemented as a hardware memory protection technique [3], [20] to prevent code injection attacks. DEP does not allow a memory page to be writable and executable at the same time. Several techniques have been proposed to protect from runtime attacks on control flow.

Execute-Only-Memory (XOM) architecture uses the compartmentalized storage of program code in order to isolate programs running on the same CPU [21]. It requires the program to be encrypted by the vendor, and the program is first decrypted when loaded from memory. However, XOM protects only encrypted programs, whereas most of programs and library code available are in the unencrypted form.

Arora et al. [22] proposed a hardware-assisted architecture to monitor control flow at three granularity levels: inter procedural level with program's static function call graph, intra procedural level for each branch and jump within a function, and integrity of instructions. However, this technique does not consider indirect branches, which can lead to attacks, such as code reuse. In addition, it has high-performance overhead due to the integrity checking at an instruction level.

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

C. Figures and Tables

Table I presents an overview of the types of failures we utilized to evaluate our approach. For each failure type, we inserted the modeled errors within the autonomous vehicle system at numerous locations. For each RRMG and each failure type, we executed the system 100 times, randomly activated one error during each execution, and measured the error detection rate achieved by the NIRM approach.

Fig. 1 shows the program memory layout for common ROPand JOP attacks. ROP uses gadgets ending in a return instruction [4], while JOP uses gadgets ending in an indirect jump instruction [5]. First, the control structure is corrupted consists of return addresses, which targets the gadgets ending in ret instruction in the library or victim code. Initially, the stack pointer is moved to the manipulated return address. The execution of ret instruction in the gadget moves the stack pointer to the next return address. The stack pointer is used to orchestrate the gadgets in sequence. While in the case of JOP, the dispatcher gadget is used, which acts as virtual program counter (PC), to advance the execution control from one gadget to another. Each gadget contributes certain computations to perform the complete attack.

TABLE I Runtime Attacks

Techniques	**Table Column Head**	
	Software based	***Hardware based***
Isolation based	Stack guard,RAD	DEP,XOM
Runtime code integrity		REM,DIC
Signature based	Davi et al	Kayaalp et al

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved

REFERENCES

1) R. Alur, "Timed automata," in Computer Aided Verification. Trento, Italy: Springer, 1999, pp. 8–22.

2) N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," IEEE Trans. Softw. Eng., vol. 30, no. 12, pp. 859–872, Dec. 2004.

3) B. Douglass and D. Harel, Real-Time UML: Developing Efficient Objects for Embedded Systems. Reading, MA, USA: Addison-Wesley, 1997.

4) T. Firley, H. Michaela, D. Karsten, G. Thomas, and G. Ursula, "Timed sequence diagrams and tool-based analysis—A case study," in UML'99—The Unified Modeling Language. Heidelberg, Germany: Springer, 1999, pp. 645–660.

5) A. Nassar, F. J. Kurdahi, and W. Elsharkasy, "NUVA: Architectural support for runtime verification of parametric specifications over multicores," in Proc. Int. Conf. Compilers Archit. Synth. Embedded Syst., Amsterdam, The Netherlands, 2015, pp. 137–146.

6) A. Pnueli, "The temporal logic of programs," in Proc. Symp. Found. Comput. Sci., Providence, RI, USA, 1977, pp. 46–57.

7) (Sep. 2014). UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. [Online]. Available: http://www.omgmarte.org

Page 20-28 © MANTECH PUBLICATIONS 2017. All Rights Reserved