Chapter 10

RUNTIME INTEGRITY FOR CYBER-PHYSICAL INFRASTRUCTURES

Jonathan Jenkins and Mike Burmester

Abstract Cyber-physical systems integrate cyber capabilities (e.g., communications and computing) with physical devices (e.g., sensors, actuators and control processing units). Many of these systems support safety-critical applications such as electric power grids, water distribution systems and transportation systems. Failures of these systems can cause irreparable damage to equipment and injury or death to humans. While most of the efforts to protect the systems have focused on reliability, there are urgent concerns regarding malicious attacks. Trusted computing is a security paradigm that enables platforms to enforce the integrity of execution targets (code and data). However, protection under this paradigm is restricted to static threats.

This chapter proposes a dynamic framework that addresses runtime integrity threats that target software programs in cyber-physical systems. It is well known that the attack surface of a multi-functional program (Swiss-army knife) can be much larger than the sum of the surfaces of its single-function components (e.g., the composition of programs that are secure in isolation is not necessarily secure). The proposed framework addresses this issue using calibration techniques that constrain the functionality of programs to the strict specifications of the cyber-physical application, thus steering execution flow away from the attack surface. Integrity is assured by verifying the calibration, while the burden of validation rests with system designers. The effectiveness of the approach is demonstrated by presenting a prototype for call integrity.

Keywords: Cyber-physical systems, integrity threats, runtime

1. Introduction

Attacks on the executable code of cyber-physical systems exploit cyber vulnerabilities or physical component vulnerabilities. Attacks of the first kind focus on badly-designed software (e.g., leading to a buffer overflow) or com-

© IFIP International Federation for Information Processing 2015
M. Rice, S. Shenoi (Eds.): Critical Infrastructure Protection IX, IFIP AICT 466, pp. 153–167, 2015.
DOI: 10.1007/978-3-319-26567-4_10

promised code (e.g., malware). Attacks of the second kind exploit outputs from faulty or compromised sensors. This chapter focuses on badly-designed software because attacks on compromised code or compromised sensors can be thwarted via well-understood techniques such as static integrity engines in trusted computing architectures [30] and risk management architectures.

Attacks that exploit software vulnerabilities have been studied extensively and several protection techniques have been proposed (see, e.g., [3, 8, 11, 14, 15, 19, 20, 22, 23, 25, 26]). However, these techniques target specific system vulnerabilities, meaning that they do not transfer to other types of attacks. This chapter proposes a trust framework that addresses the dynamic (real-time) integrity of executable code using calibration techniques. The framework is intended to extend the capabilities of trusted computing architectures to support static as well as dynamic integrity verification.

2. Related Work

Several techniques exist for the dynamic protection of the integrity of software execution as opposed to the protection of data residing in system memory. Although these techniques do not offer complete dynamic integrity protection of software at runtime, they address important aspects of dynamic integrity and are a major contributor to trusted software execution. Nevertheless, the application of these techniques to secure cyber-physical systems requires adjustments in order to meet the stringent requirements imposed by the systems.

Taint tracking is a technique for tracking untrusted data (e.g., network sources) as they propagate during execution [9, 24]. Copying, arithmetic functions and control flow logic enable the spread of "taint" among data and to subsequent executions. The tracking proceeds by marking sources of suspect data in memory and causes a non-trivial slowdown factor ranging from 5.5 to 30. Taint tracking has been successfully implemented to detect attacks such as buffer overflows, format string attacks and control data overwrites [9].

Control flow integrity is a technique for constraining the control flow of software to its control flow graph, which is a representation of the control flow edges that program code can traverse. A control flow graph is produced by static analysis or by the application of analysis programs. The technique is readily implemented using software instrumentation via rewriting to insert checks of control flow jumps against the control flow graph [2].

The performance cost of full control flow integrity enforcement as implemented in software has been shown to vary from negligible to approximately 45% with some variance based on the application [2, 32]. One limitation is that attacks that obey the control flow graph (e.g., format string attacks) are not addressed. Furthermore, the high performance penalty of software-based, fine-grained (full) control flow integrity protection has resulted in weaker, coarse-grained versions (e.g., reducing the frequency or types of checks) that are subject to attack [17, 32].

Trusted computing is an architecture that supports the measurement and preservation of system integrity, as well as secure network connectivity [28, 29].

Trusted computing is distinct from other integrity protection methodologies due to its provision of an implementation-independent set of abstract engines for creating, maintaining and using robust static integrity information about software in platforms. Trusted computing defines a trust structure in which the trust in a computing platform is created by checking the static integrity of each boot stage program before execution.

Trusted computing provides static integrity protection, but it does not address the dynamic integrity of executing software. However, it can be an important factor when considering dynamic integrity protection for a cyber-physical infrastructure because employing static measurements of integrity such as digests may help. Beyond the integrity of the literal memory state of program code, techniques are required for protecting software execution.

3. Dynamic Integrity

If the designers of future critical infrastructure systems were to adopt the security practices of conventional systems, they might, for example, design an electric power grid whose data processing devices undergo regular security scans, software updates and enforce access control policies that protect resources. Such an approach cannot protect the system from every security violation. Implementing reactive security policies that address new exploits requires the deployment of resources that analyze and defeat the exploits. The financial investment required for endless attack-specific protection of critical infrastructure assets as adversaries discover new exploits would be substantial and ultimately futile. What is needed is a general protection framework of policies and mechanisms that identify and mitigate violations that appear fundamentally as deviations from correct operation.

A promising solution is to shift the burden of providing integrity for executable code from the security analyst to the software designer. The runtime integrity of code must be guaranteed via techniques that constrain code execution to the envelope of proper operations as intended by the designers. Such an approach is necessary because adding patches or fixes to badly-designed code will not prevent adversaries from bypassing the protection mechanisms by leveraging inherent code vulnerabilities. Another advantage is that the protection framework is not dependent on specific protection mechanisms (e.g., against ROP [25]) that a determined adversary can easily bypass. Note that the integrity of the code content by itself does not guarantee the security of execution – it only guarantees that the state of the code matches that intended by the designers. Thus, measures beyond the verification of code content are necessary to protect software execution and create trusted critical infrastructure devices.

The objective of this research is to extend the integrity protection of a trusted platform module to capture dynamic integrity. In the proposed approach, the software designers are ultimately responsible for code integrity provided that the code is executed within the strict confines of calibrations as specified by the designers. Cyber-physical system security has three traditional requirements:

- **Availability:** Cyber and physical components obey timing and response requirements.

- **Integrity:** Cyber and physical components have static and dynamic integrity. Dynamic integrity protection mechanisms address runtime attacks.

- **Confidentiality:** Observations of data streams/events do not cross user-user, device-device or user-device boundaries unless the observing system node holds the appropriate security level. For real-time confidentiality, forward device secrecy is required [18].

The availability of state information in real-time is crucial in critical infrastructure environments. This is because the failure of system components or other faults can lead to cascading failures and large-scale system shutdown. To prevent this from occurring, it is necessary to enforce "need-to-get-now" policies [6] because quality of service or best effort policies may be inadequate.

This work uses the Byzantine threat model [21] in which the adversary can act arbitrarily and is not restricted by any constraints other than being computationally bounded (modeled by a probabilistic polynomial-time Turing machine). The focus is mainly on dynamic integrity threats.

4. Trusted Infrastructures

Critical infrastructure systems require appropriate references, measurement and enforcement in order to guarantee their integrity. Software integrity is defined by static references for the literal conditions of memory content and by conditions for safe execution by virtue of safe memory interactions. Software designers and creators are best positioned to provide such references because they express the system functionality as well as the non-functional conditions for safe memory interactions. Thus, in order to protect critical infrastructure assets, it is essential that software designers specify the appropriate reference constraints.

In the case of cyber-physical systems, the addition of physical components means that protection methods must consider a new type of interaction between software, memory and physical components, and must also represent the integrity of the physical components. The integrity of the interface between cyber and physical objects is also a critical part of any protection scheme, and it may introduce security violations when it fails to preserve information as it is transmitted between cyber and physical devices.

A physical data source conceptually provides an event stream. No explicit program logic is presented by physical nodes to their communicating counterparts. Therefore, the integrity of the physical data source can be directly evaluated only by a physical reference value or a physical condition for its safe operation (e.g., voltage level range). Secure interactions between cyber and physical components require the integrity of cyber component execution, the adherence of physical components to references as verified by inspection and the

integrity of the cyber-physical interface components. The interactions between the cyber and physical components of a cyber-physical system must be monitored for conditions and failure modes appropriate to the potentially serious threats arising from system failure via the cyber-physical interface.

5. Dynamic Software Integrity

For runtime security, the operation of software components of cyber-physical systems must exhibit integrity, apart from the (static) intactness of the program code and data on devices that can be protected with architectures such as trusted computing [30]. This task is approached by applying constraints to operate at two stages of deployment in two fundamentally different ways.

First, a form of integrity protection is achieved by reducing the space of vulnerabilities exposed to attack at runtime. Functional constraints on the software designed for critical infrastructure assets is one way to reduce the extent of protection needed before software is executed (e.g., no non-essential functionality is included). Second, it is necessary to constrain software execution by enforcing dynamic integrity conditions at runtime. These conditions are derived from the analysis of programs, but also require input from program creators for application-dependent conditions. Program creators express the intent of algorithms in code and are, therefore, best able to characterize the envelope within which executions should remain. However, as will be shown later, the evaluation of execution with respect to such conditions must be carried out by an external engine to ensure the integrity of the measurement process itself as a general property of systematic infrastructure protection.

The proposed efforts for constraining software execution to benefit integrity can be summarized as follows:

- Software program creators must supply dynamic integrity conditions with respect to:
  - The degree of trust conferred on data sources and the degree to which they are permitted to propagate when used by programs (taint).
  - The control flow of the program (e.g., with respect to a control flow graph).
  - The procedure calls of the program.
- Software program creators must provide attestation for the calibration constraints.
- A dynamic integrity monitoring engine with system memory access must be deployed to measure the dynamic integrity of executing programs with respect to conditions. The engine can be implemented in hardware to significant advantage in a cyber-physical system.
- Appropriate responses to integrity violations must be set.

6. Constraining Software by Calibration

This section describes the design and dynamic constraints, specifically, taint tracking, control flow integrity and call integrity protection.

6.1 Design Constraints

The conventional software production process must undergo adjustments in order to comply with the stringent requirements imposed by cyber-physical systems, especially with regard to design practices and functional constraints. Non-essential functionalities may interfere with the performance of critical, required functionalities. When a software program implements additional, non-essential functionalities, some degree of resource sharing and interaction (e.g., memory) across functionalities will occur. Thus, program functionalities are subject to conflict in terms of time and resource use, potentially impacting performance and availability.

In addition to resource conflicts, non-essential program functionalities become a threat vector for compound security vulnerabilities that are constructed from conditions created by instructions for the separate functionalities. For example, consider a program that implements a search function and a maintenance function that accesses working memory. The exploitation of flaws during the execution of maintenance instructions may expose a memory address used by the search function to launch an attack; this violates the operational integrity of the entire program. Note that a program may not remain secure if arbitrarily composed with other programs or with instances of itself (universal composability) [7].

Cyber-physical system software, with its stringent demands on availability and failure consequences, must be functionally constrained to avoid introducing resource conflicts and threat vectors. The failure to implement such measures for critical infrastructure assets is unacceptable.

6.2 Dynamic Constraints

To protect the operation of cyber-physical system software, measures are necessary to preserve the dynamic integrity of software execution with regard to memory use and other properties. The enormity of this task is made more manageable by the availability of protection mechanisms for some aspects of dynamic integrity. To address this issue, a hardware dynamic integrity monitor extension is proposed to potentially achieve complete dynamic integrity. Indeed, the combination of taint tracking, control flow integrity and call integrity protection enables the comprehensive dynamic integrity monitoring of cyber-physical system software.

Taint Tracking. Recent research has demonstrated the effectiveness of taint tracking systems [9, 24]. Critical infrastructure systems justify the imple-

mentation of taint tracking using specialized hardware as detailed in [10, 12, 27], allowing for a significant reduction in performance impact.

Suh et al. [27] add tracking logic to processors (including tag tracking hardware components). Their technique involves dynamic information flow tracking, which analyzes information flows between memory areas due to program execution. They evaluate two security policies by simulating a set of information flows; the more robust policy tracks flows based on the use of tainted values in computations. The measured performance overhead for both policies is low – less than 1% for the lighter policy and at most 6% for the more robust policy.

Hardware-based tracking is also a necessary part of a dynamic integrity monitor for a cyber-physical system. This is the result of isolation from monitored executions and their resources, a requirement for a systematic cyber-physical system dynamic integrity protection mechanism that cannot itself be attacked.

Control Flow Integrity. Software-based techniques can effectively protect the control flow integrity of software, albeit with a potentially significant performance impact that hampers the adoption of pure fine-grained protection [2, 32]. Weaker versions of control flow integrity protection are inadequate to combat threats to cyber-physical infrastructures [13]. Clearly, the imposition of additional protection delays on cyber-physical systems is also unacceptable due to stringent availability demands, not to mention integrity requirements.

For the unique, critical usage scenarios of cyber-physical devices, hardware-based control flow integrity protection is justified and necessary for the isolation, functional constraints and robustness conferred on the protection functionality as well as the reduced performance burden. Nevertheless, in terms of complexity and difficulty, the hardware-based implementation of full control flow integrity protection is feasible and straightforward. Budiu et al. [4] report that simulations of coarse-grained control flow integrity introduce an average runtime overhead of 3.75% and a maximum of overhead of 7.12%. Further improvements in performance are possible via scheduling.

Call Integrity Protection. The integrity of procedure calls within programs is a relatively unrecognized, but critical, aspect of the dynamic integrity of executing software. It is possible to monitor runtime call integrity to detect sequences of integrity violations and the potentially damaging security attacks they may comprise.

Computer memory is the space in which software integrity is demonstrated by the soundness of literal memory as well as the compliance of program execution with developer intent. Thus, call integrity is measured by examining the memory associated with procedure calls, typically the stack.

Implementing call integrity protection in cyber-physical system software involves three processes: (i) instrumentation; (ii) instruction analysis or introspection; and (iii) specification creation and enforcement. Since the integrity of procedure calls and the integrity of software execution, in general, are not exposed to direct measurement as an inherent property of programs, the substrate

of call integrity must be made to expose itself for measurement. This exposure is readily achieved via software instrumentation, which involves inserting instructions into a target program to enable logic or analysis distinct from that of the target application. Such exposure is a fundamental requirement due to the lack of access by external evaluators to the syntactic memory-to-object information that is available in source code. Essentially, instrumentation for call integrity protection inserts source code or binary instructions for the purpose of measurement, exposure of integrity and detection of events.

The use of instrumentation for the call integrity measurement engine as proposed here requires the runtime analysis of instructions in order to access integrity-relevant data exposed by the instrumentation. Instruction analysis (introspection) is the inspection of target program instructions (i.e., opcodes and operands) by an analysis program. This analysis needs a runtime access interface to the signaling data exposed by the instrumentation process. At the level of instruction introspection, aggregate analysis and detection of integrity relevant events can be performed across instructions (e.g., analyzing integrity properties of arguments to a runtime call).

Finally, dynamic call integrity constraint conditions must be specified and enforced. The specification of call integrity conditions requires input from the program creators for some requirements, but in reality, many broadly applicable conditions can be constructed from application-independent relations that apply to categories of calls.

With dynamic call integrity exposed via software instrumentation and constraints on procedure call memory interactions available, dynamic call integrity evaluation and enforcement must be carried out. Clearly, if call integrity is measured and exposed in memory via instrumentation, evaluations can be performed as an additional instrumentation step. The ideal evaluation engine is robust, resource-independent and external to the program being analyzed to mitigate attacks on the protection functionality.

The particular combination of the techniques mentioned above demonstrates two critical facts concerning software integrity protection and cyber-physical infrastructure protection. First, the inclusion of taint, control flow and call integrity produces a comprehensive coverage of software execution integrity via the revelation of key memory interactions that implement execution and exhibit integrity. Second, the combination demonstrates one crucial property of workable systematic software integrity protection for cyber-physical infrastructures – protection is general via the application to integrity itself instead of particular attack profiles or effects. Approaches that apply a patchwork of techniques directed at particular attacks achieve no better protection than conventional security software and do not respect the intolerance of cyber-physical infrastructures to regular violations or conventional remediation strategies. In general, a protection strategy analyzes the integrity of critical systems as it transitions instead of reactively developing protections for each new attack. Reactive, attack-focused integrity strategies impose a cost and maintenance

burden that are incompatible with large, heavily interconnected cyber-physical infrastructures.

7. Dynamic Call Integrity via Calibration

In order to protect cyber-physical system software execution in a general manner, the hardware-based taint and control flow constraints proposed in this work must be combined with call integrity constraint enforcement, thus extending the static protection offered by trusted computing to dynamic execution protection. Dynamic call integrity information must be measured, transmitted to a location accessible to a measurement entity and evaluated before enforcing the appropriate policy.

7.1 Experimental Setup

A prototype was implemented on Linux with C source code using executable and linkable format binaries and the x86 instruction set. Source code instrumentation was implemented using a script that inserts protection-related instructions near procedure calls. The DynamoRIO runtime instruction manipulation library [16] was used to instrument binaries. The prototype was run on a laptop with an Intel Core i5-2520m 2.5 GHz dual-core processor with 3 MB cache.

7.2 Evaluation and Calibration

The goal and scope of the evaluation was to demonstrate the principles and techniques proposed: that the runtime integrity of a call can be measured and that integrity violations can be detected by combining several forms of instrumentation with introspection without requiring the original protected program to be modified by developers. The evaluation code can easily be adapted to other environments that use a working memory stack and registers by making adjustments to system parameters.

Source instrumentation was performed automatically using a script that processes C source code calls. A preamble of instrumentation instructions (e.g., source code and assembly code) was added to all procedure calls residing within a range of code specified by special start and end tags. Figure 1 shows the preamble of function `foo(a,b,c,d,e)` (i.e., the instructions above the function).

The preamble includes a set of k split calls where k is the number of analyzed arguments contained in the source code call. Each split call accepts only its individual argument and induces a compiler to produce executable instructions that manipulate the stack layout to cause the argument to be isolated at the end of the stack for integrity measurement.

Each split call has a set of instrumentation instructions inserted before and after the call. In particular, each call is marked with a special variable allocation to present a marker to the instruction introspection layer that indicates the

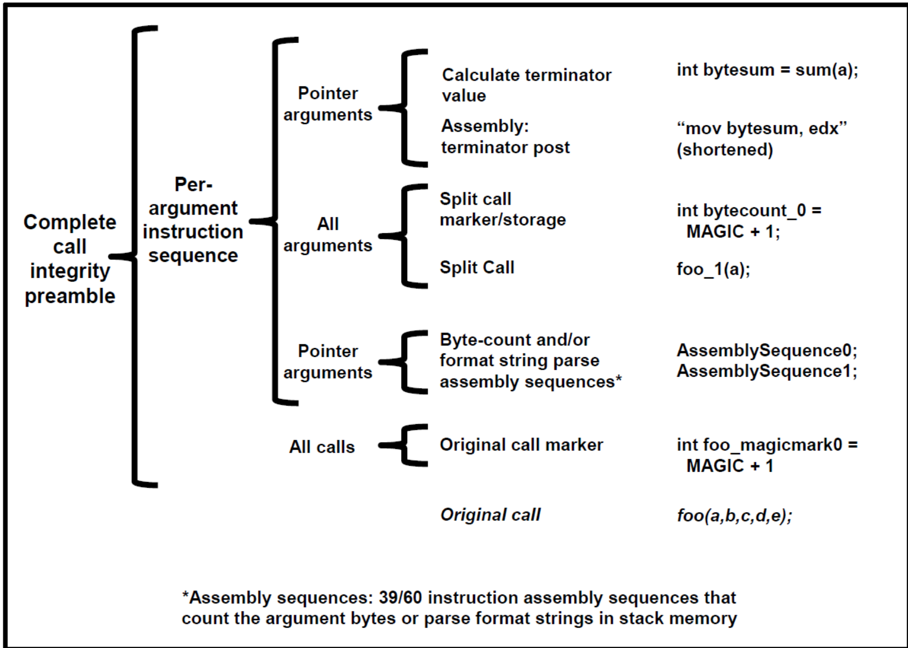

Figure 1. C code procedure call instrumentation with preamble.

presence of a split call instead of an existing call instruction (`int bytecount_0=MAGIC+1` in Figure 1). An instrumented function call (`int bytesum=sum(a)`) is inserted to compute a terminating condition value for the enumeration of parameter bytes in memory. In this case, a call to an instrumented function is incorporated to calculate the sum of the bytes as integers. Finally, inline assembly instructions are inserted to copy the terminator value to a register for the byte-enumerating measurements (abbreviated as `mov bytesum, edx`).

Assembly instructions (`AssemblySequence0; AssemblySequence1`) are inserted immediately following each split call. The instructions implement two types of primitive integrity measurements on calls that are the basis for more complex call integrity measurements. They assume that the parameter bytes can be located at a predictable location at the end of the stack in working memory before the assembly byte measurement sequence is reached in the program source code during execution.

7.3 Parameter Length

One critical primitive component of call integrity is the relationships between the lengths of parameters in working memory (especially pointer types) that may be known only during execution. Source instrumentation was applied to insert instructions to measure the parameter lengths in runtime memory using the byte-enumerating assembly sequence detailed earlier, based on the

Integrity performance measurements		Argument run-time length measurement	Argument count + format string count extraction (2 specifier, 2 arguments)	Argument count + run-time format string count extraction (2 specifier, 2 arguments)
Source instrumentation	Average relative slowdown	4.1812, 0.19/byte	0.9984,0.3328042027 per argument	1.05866696, 0.3528889878 per argument
	Absolute additional time (ms)	0.00015558, 0.00000707/byte	negligible	negligible

Figure 2. C source code call integrity measurement performance.

termination sum condition prestored in a register. Counting loop termination is based on the encountered byte sum exceeding the prestored sum. Typically, the number of additional source instructions inserted by this method for length counting is $5 + 48\ k$, where k is the number of arguments whose length is to be measured in memory.

Figure 2 shows that the average relative slowdown for length measurement is 4.1812 or 0.19 per byte measured. The memory space requirement per argument for runtime argument length measurement includes space for two integers to store a bytesum and the count result, and the use of the general purpose x86 registers.

7.4 Call Parameter Parity

Another critical aspect of procedure calls that holds valuable integrity information is the parity of call parameters. The number of parameters required by format string parameters in conversion procedure calls (such as format functions `scanf` and `printf` in C) is an important integrity information primitive that measures dangerous mismatches between the requirements of call parameters. In the case of C format functions, an entire major category of attacks is based on this disparity.

Measurements of the interactions between format string and regular parameters are implemented using the assembly sequence byte-enumerating technique as with parameter length. If the format string is visible in the source code, it is parsed to determine the number of format specifiers. The number of non-format string arguments is extracted via source code analysis during the instrumentation pass. The two values are compared during execution by inserting instructions for simple numeric relational comparison, a mismatch being indicative of a call integrity violation and a potentially dangerous call.

If the format string is known only at runtime, then the methods for isolation and bounding of parameters used for argument length measurement are reapplied to allow the format string to be analyzed in working memory. Inline assembly instructions were inserted to parse the format string in memory during execution to determine the number of format specifiers, each of which is identified by a "%" byte in C format strings. The combination of format string

parsing and argument count measurement resulted in the small performance impact shown in Figure 2. The average relative slowdown for measurement was timed at 0.9984 or approximately 0.33 per argument.

Exposure of Integrity Primitives to Evaluation. In the experiments, the exposure of integrity information to an external evaluator was carried out by adding source code instructions that perform initialization assignments with marking identifier values. Such assignments induce a compiler to emit instructions that expose to memory-accessing integrity evaluating entities the location of integrity information (e.g., stack memory offset) or meta-information about instrumented target program code that facilitates protection (e.g., a signal to the evaluation layer about the availability of an integrity measurement during execution). These capabilities allow the general exposure of measurements to a memory-accessing evaluation engine. In the prototype, DynamoRIO [16] intercepts integrity information exposed in memory via source instrumentation using its instruction introspection ability.

Evaluation of Call Integrity. The final process required to demonstrate call integrity is the evaluation of integrity information with respect to a set of integrity conditions. Appropriate enforcement of policy can be carried out based on the result of the evaluation. The conditions demonstrated in the research were:

- The relationships between the lengths of parameters in memory must comply with given relations (e.g., source does not exceed destination). A useful universal example length condition is "no length may exceed another."

- The number of format string specifiers must match the number of non-format string parameters.

Although the number of instantiations of runtime procedure calls is large, conditions for protecting categories of procedure calls can be created based on the primitives (length and parity) demonstrated with the help of developer-assistive tools.

The evaluator engine was demonstrated with DynamoRIO [16] via the analysis of the measurement results held within the instructions inserted by source instrumentation. The instruction introspection feature of DynamoRIO was applied to locate the exposed measurements during runtime, demonstrating the ability to intercept integrity information necessary for enforcing policy (i.e., simply exit upon a violation). Since DynamoRIO is unable to access working stack memory (only instructions), the evaluation was simulated (first, in the source code for demonstration purposes) via the ability to insert new executable instructions to evaluate arbitrary relational expressions on the intercepted numerical integrity measurements (as opposed to analyzing the measurements in memory as an ideal hardware evaluator would).

During the experiments, DynamoRIO was able to illustrate the characteristics and levels of component access necessary to analyze and evaluate integrity information exposed by other techniques (e.g., tool-assisted source code instrumentation). Also, it was able to demonstrate the validity of the relocatability of integrity information, a fundamental requirement of general dynamic integrity protection architectures as proposed in this work.

8. Conclusions

The principal contribution of this research is a dynamic framework that addresses runtime integrity threats to software programs in cyber-physical systems. The framework constrains the design and execution of software so that its functionality adheres to the strict requirements of cyber-physical applications. Constraints are imposed on both software design and execution and are implemented via calibration. The framework is intended to extend the protection offered by trusted computing architectures to address dynamic threats during software execution.

Acknowledgement

This research was partially supported by the National Science Foundation under Grant Nos. DUE 1241525, CNS 1347113 and DGE 1538850.

References

[1] M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, Control-flow integrity, *Proceedings of the Twelfth ACM Conference on Computer and Communications Security*, pp. 340–353, 2005.

[2] M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, Control-flow integrity principles, implementations and applications, *ACM Transactions on Information and System Security*, vol. 13(1), article no. 4, 2009.

[3] J. Bridges, T. Sartorius and S. Millendorf, Methods and Systems for Checking Run-Time Integrity of Secure Code Cross-Reference to Related Applications, United States Patent 8,639,943, 2014.

[4] M. Budiu, U. Erlingsson and M. Abadi, Architectural support for software-based protection, *Proceedings of the First Workshop on Architectural and System Support for Improving Software Dependability*, pp. 42–51, 2006.

[5] M. Burmester and B. de Medeiros, On the security of route discovery in MANETs, *IEEE Transactions on Mobile Computing*, vol. 8(9), pp. 1180–1188, 2009.

[6] M. Burmester, E. Magkos and V. Chrissikopoulos, T-ABAC: An attribute-based access control model for real-time availability in highly dynamic systems, *Proceedings of the IEEE Symposium on Computers and Communications*, pp. 143–148, 2013.

[7] R. Canetti, Universally composable security: A new paradigm for cryptographic protocols, *Proceedings of the Forty-Second IEEE Symposium on Foundations of Computer Science*, pp. 136–145, 2001.

[8] M. Castro, M. Costa and T. Harris, Securing software by enforcing dataflow integrity, *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, pp. 147–160, 2006.

[9] W. Cheng, Q. Zhao, B. Yu and S. Hiroshige, TaintTrace: Efficient flow tracing with dynamic binary rewriting, *Proceedings of the Eleventh IEEE Symposium on Computers and Communications*, pp. 749–754, 2006.

[10] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher and M. Rosenblum, Understanding data lifetime via whole system simulation, *Proceedings of the Thirteenth USENIX Security Symposium*, 2004.

[11] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, *Proceedings of the Seventh USENIX Security Symposium*, 1998.

[12] J. Crandall and F. Chong, Minos: Control data attack prevention orthogonal to memory model, *Proceedings of the Thirty-Seventh International Symposium on Microarchitecture*, pp. 221–232, 2004.

[13] L. Davi, A. Sadeghi, D. Lehmann and F. Monrose, Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection, *Proceedings of the Twenty-Third USENIX Security Symposium*, pp. 401–416, 2014.

[14] L. Davi, A. Sadeghi and M. Winandy, Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks, *Proceedings of the ACM Workshop on Scalable Trusted Computing*, pp. 49–54, 2009.

[15] L. Davi, A. Sadeghi and M. Winandy, ROPdefender: A detection tool to defend against return-oriented programming attacks, *Proceedings of the Sixth ACM Symposium on Information, Computer and Communications Security*, pp. 40–51, 2011.

[16] DynamoRIO, Dynamic Instrumentation Tool Platform (`dynamorio.org`), 2009.

[17] E. Goktas, E. Athanasopoulos, H. Bos and G. Portokalidis, Out of control: Overcoming control-flow integrity, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 575–589, 2014.

[18] C. Gunther, An identity-based key-exchange protocol, in *Advances in Cryptology – EUROCRYPT '89*, J. Quisquater and J. Vandewalle (Eds.), Springer, Heidelberg, Germany, pp. 29–37, 1990.

[19] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall and J. Davidson, ILR: Where did my gadgets go? *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 571–585, 2012.

[20] V. Kiriansky, D. Bruening and S. Amarasinghe, Secure execution via program shepherding, *Proceedings of the Eleventh USENIX Security Symposium*, pp. 191–206, 2002.

[21] L. Lamport, R. Shostak and M. Pease, The Byzantine generals problem, *ACM Transactions on Programming Languages and Systems*, vol. 4(3), pp. 382–401, 1982.

[22] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker and W. Drewry, Minibox: A two-way sandbox for x86 native code, *Proceedings of the Annual USENIX Technical Conference*, pp. 409–420, 2014.

[23] D. Malan, CS50 sandbox: Secure execution of untrusted code, *Proceedings of the Forty-Fourth ACM Technical Symposium on Computer Science Education*, pp. 141–146, 2013.

[24] J. Newsome and D. Song, Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software, *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[25] R. Roemer, E. Buchanan, H. Shacham and S. Savage, Return-oriented programming: Systems, languages and applications, *ACM Transactions on Information and System Security*, vol. 15(1), article no. 2, 2012.

[26] H. Shacham, The geometry of innocent flesh on the bone: `return-into-libc` without function calls (on the x86), *Proceedings of the Fourteenth ACM Conference on Computer and Communications Security*, pp. 552–561, 2007.

[27] G. Suh, J. Lee, D. Zhang and S. Devadas, Secure program execution via dynamic information flow tracking, *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–96, 2004.

[28] Trusted Computing Group, TCG Specification Architecture Overview, Revision 1.4, Beaverton, Oregon, 2007.

[29] Trusted Computing Group, TNC Architecture for Interoperability, Specification 1.3, Revision 6, Beaverton, Oregon, 2008.

[30] Trusted Computing Group, TPM Main Specification, Level 2, Version 1.2, Revision 116, Beaverton, Oregon, 2011.

[31] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani and D. August, RIFLE: An architectural framework for user-centric information-flow security, *Proceedings of the Thirty-Seventh International Symposium on Microarchitecture*, pp. 243–254, 2004.

[32] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song and W. Zou, Practical control flow integrity and randomization for binary executables, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 559–573, 2013.