Using Logical Error Detection in Software Controlling Remote-Terminal Units to Predict Critical Information Infrastructures Failures

George Stergiopoulos[1], Marianthi Theocharidou[2], and Dimitris Gritzalis[1(✉)]

[1] Information Security and Critical Infrastructure Protection (INFOSEC) Laboratory, Department of Informatics, Athens University of Economics and Business, Athens, Greece
{geostergiop,dgrit}@aueb.gr

[2] European Commission, Joint Research Center (JRC), Security Technology Assessment Unit, Institute for the Protection and the Security of the Citizen (IPSC), Via E. Fermi 2749, 21027 Ispra, Italy
marianthi.theocharidou@jrc.ec.europa.eu

Abstract. A method for predicting software failures to critical information infrastructures is presented in this paper. Software failures in critical infrastructures can stem from logical errors in the source code which manipulates controllers that handle machinery; i.e. Remote Terminal Units and Programmable Logic Controllers in SCADA systems. Since these controllers are often responsible for handling hardware in critical infrastructures, detecting such logical errors in the software controlling their functionality implies detecting possible failures in the machine itself and, consequently, predicting single or cascading infrastructure failures. Our method may also be tweaked to provide estimates of the impact and likelihood of each detected error. An existing source code analysis method is adjusted to analyze code able to send commands to SCADA systems. A practical implementation of the method is presented and discussed. Examples are given using open-source SCADA operating interfaces.

1 Introduction

An Industrial control system (ICS) is a general term that encompasses several types of control systems, including supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and others, such as Programmable Logic Controllers (PLC) and Remote Terminal Units (RTUs). ICSs are typically used in industries such as electrical, water, oil and gas, chemical and others [20].

An industrial control system usually has a central unit and a number of control locations. Data and commands are exchanged as messages. Many Critical Infrastructures (CIs) use Remote Terminal Units (RTUs) and Programmable Logic Controllers (PLCs) as control locations, in order to handle the machinery and functionality of an infrastructure (e.g. valves, sensors, breakers, etc.). Thus, a failure on any one of them may affect the operation of the entire infrastructure and start a cascading event, where multiple CIs fail due to their dependencies.

T. Tryfonas and I. Askoxylakis (Eds.): HAS 2015, LNCS 9190, pp. 672–683, 2015.
DOI: 10.1007/978-3-319-20376-8_60

A *Programmable Logic Controller* (PLC) is a microprocessor-based controller, which implements functions such as logic, sequencing, timing, counting to control machines and processes [18]. Logical rules in the form of `"if (A and B) occurs, switch on C"` are used to program this type of controllers. Similar to PLCs, *Remote Terminal Units* (RTUs) are in charge of representing the communication interface between the remote substation and the SCADA master control [2]. *Supervisory Control and Data Acquisition* (SCADA) control systems operate with coded signals over communication channels and provide control of RTU equipment [19]. RTU/PLC logical rules provide a fertile ground for logical errors, especially if business logic restrictions are imposed incorrectly when handling processes and advanced control functions.

Motivation. When industries connected SCADA systems with the Internet, these control systems were exposed to various vulnerabilities and threats [1]. Built as stand-alone, isolated control systems, they lacked the proper security measures needed to support a robust and safe functionality over the Internet. For example, an over-the-Internet, man-in-the-middle attack on a green diesel generator that led to a total melt-down has been presented in the literature [2].

Obviously, the consequences of maliciously operating SCADA systems can be devastating. Modern implementations are using the internet and allow common programming languages (such as Java) to send and receive data and commands through the central control unit to RTUs. Such implementations include, amongst others, openSCADA [4], MODBUS4 J [17] and JAMOD [5], which uses the MODBUS protocol. The added complexity coupled with open-source components resulted in an increase in software weaknesses, vulnerabilities and failures in these types of systems [6]. A number of logical threats have been registered in public databases [7]. Source code analysis of programs that remotely control RTUs opens a path to prevent software failures. Such failures may lead to malfunction in individual controllers or infrastructures, or even initiate cascading failure to dependent infrastructures. Yet, to our knowledge, no effort has been made in detecting logical flaws in the functionality of software that controls RTUs; flaws due to erroneous implementation of the intended functionality in source code.

Example 1. Consider an RTU that controls the gas supply in an infrastructure, using coils; a faulty implementation can lead to sending the same “increase flow” command more times than what the pipe system can handle, eventually making pressure reach critical levels. Automated detection of such program behavior is a relatively uncharted territory.

Contribution. In this paper, we adjust a recently developed method for detecting logical errors in software [8–11, 33], in order to detect logical errors in high-level source code controlling Critical Infrastructure field devices (e.g. temperature sensors, valves etc.) through SCADA which implements RTUs at remote locations. We aim to detect software flaws that may create malfunctions to infrastructure control systems due to erroneous implementation of functionality into source code that controls field sensors. The presented methodology is based on static and dynamic source code analysis.

Our contribution is threefold: (a) We introduce a way to detect logical flaws in software that may lead to failures in CIs. (b) Detected flaws are classified according to their severity and likelihood of causing a failure; more dangerous failures (i.e. failures that affect critical functionality) get higher Risk values than others. (c) To support the applicability and efficiency of this approach, we present proof-of-concept example tests on appropriate software, using common, open-source implementations. Tests demonstrate the ability of this method to detect software flaws in real-world scenarios, under reasonable parameters (i.e. tests were based on plausible, albeit small, software test-bed examples).

2 Related Work

Concerning control systems, research has focused mainly on securing SCADA systems through the use of alternate communication links, robust devices and added security measures [1, 2, 7]. Yet, published research suggests that software does play an important factor in cyber physical systems' stability.

An empirical analysis of software-induced failure events in the nuclear industry shows that software faults and human errors are inevitable in complex systems [31]. Recently, approaches have focused on demonstrating the effect of cyber-attacks on the physical components of cyber-physical systems. The demo presented in [32] showcases the effectiveness of attacks on terms of voltage instability to power grid and possible cascading effects to other infrastructures, as railway transport and power mark. Moreover, [14] depicts how an error in one of these systems could have a cascading and catastrophic impact on the whole infrastructure. More specifically, a software-implemented fault injection technique is used to induce errors/faults inside devices used in power grid substations. A single error in a substation device is proven able to render the operator in the control center unable to control the operation of a relay in the substation.

The method used in this paper is based on previous research [8–11, 33]. In [8], the authors describe how they used the Daikon tool [15, 16] to infer a set of behavioral specifications called likely invariants that represent the behavioral aspects during the execution of web applets. They use NASA's Java Pathfinder (JPF) [12] for model checking the application behavior over symbolic input, in order to validate whether the Daikon results are satisfied or violated. The analysis yields execution paths that, under specific conditions, can indicate the presence of certain types of logic errors that are encountered in web applications. The described method is applicable only to single-execution web applets. A variant of the method is used in [9–11], where we presented a first implementation of the APP_LogGIC tool. In [9], we specifically targeted logical errors in GUI applications. We presented a preliminary deployment of a Fuzzy Logic ranking system to address the problem of false positives and applied the method on lab test-beds. In [10], the Fuzzy Logic system was formally defined. Finally, in [11], preliminary results of a real-world application of the method were presented. The research in [13], focuses on flaws found in web applications.

The output of the method presented here can be used as partial input to risk analysis and dependency analysis methods for CIs. Research has suggested several methodologies of this type, either focusing on cascading consequences [28] or on the risk derived

from potential failures in CI dependencies [3, 22–24]. Many of these methods rely on CI operators' assessment as input. Thus, they could benefit from the analysis performed in this paper; albeit only for software failure threats.

3 Building Blocks

This section describes the three building blocks of the methodology that will be developed later in this paper: (i) the methodology for modelling the functionality of software, (ii) the testing method that detects contradictions that indicate the existence of logical errors and (iii) the fuzzy system used to classify detections. These techniques are implemented as steps when detecting logical errors in software [9, 11].

3.1 Intended Functionality Model Using Dynamic Invariants

A representation of a program's behavior can be inferred in the form of dynamic invariants, i.e. source code rules in the form of assert statements. Invariants are inferred by dynamic analysis of source code using MIT's Daikon tool [15, 16]. Dynamic invariants are logical rules for variables, such as `p!=null` or `var=="string"` that hold true at certain point(s) of a program in all monitored executions. They represent program behavior.

If executions monitored by Daikon are representative use case scenarios, based on Business Logic documentation, and cover all functionality of the software under test, then the generated dynamic invariants refer to the program's intended functionality and the extracted programmed behavior matches the programmer's intended behavior [8, 9]. An example dynamic invariant generated is:

Daikon observes the values that the program computes during execution and reports, as in Fig. 1, assertions about source code variables that hold true throughout all SUT executions (much like "laws of conduct" for correct execution [10, 11]). For example, the dynamic invariant of Fig. 1 shows that, upon invocation of method `exec()`, the value of the variable `TopLevel_Chart_count` equals‘ `2.0`’.

```
rjc.Chart.Wait_for_stable_rate_100000203_exec():::ENTER
this.TopLevel_Chart_count == 2.0
```

Fig. 1. Dynamic invariants produced by daikon dynamic analysis

3.2 Testing Invariants Against Program Executions

A possible *logical error* exists if two different versions of the same execution path are detected (i.e. same instructions executed in the same order implemented in a source code method), which differ in some state, such that one path satisfies a Daikon's dynamic invariant while the other version violates it [8].

Example 2. Consider the code in Fig. 2: Let's say that the path/state depicted in Fig. 2 shows the beginning instruction of the aforementioned method. Figure 1's invariant must hold *true* every time execution enters the `rjc.Chart.Wait_for_stable_`

`rate_100000203_exec()` method. Yet, it is clear from the memory read in Fig. 2, that the invariant is *violated* (false), since variable `TopLevel_ Chart_count = 1`. In this case, this path violates the corresponding dynamic invariant. If a different version of this path is found to have `TopLevel_ Chart_count = 2` in the same execution path, then a logical error is probably manifesting at that point, since two same executions lead to different states of a program [8–11, 33]. This contradiction, if present, signifies a logical error inside `exec()`.

```
[rjc/Chart.java:342] : if(this.TopLevel_Chart_count == 1)
        STATE VARIABLE: this.TopLevel_Chart_count -> 1
```

Fig. 2. Execution path output state: instruction executed and variable content

To do the aforementioned test, we need as many executions of a program's functionality as possible, in order to test its possible paths-states. Someone can do that by hand but, in realistic situations, that is not feasible. Instead we use NASA's Java Pathfinder (JPF) tool [12]. JPF is a static analysis tool able to verify Java programs and collect runtime information of a program and more [12]. Using custom-made extensions, our method symbolically executes a program's source code and analyzes multiple execution paths and states along these paths, over a wide range of input. JPF allows us to test multiple execution paths of a program against inferred dynamic invariants, in search for contradictions and violations in the description of a program's logic. Figure 2 depicts an example execution record from JPF.

3.3 Classification of Detections – Fuzzy Logic System

Verifying Dynamic Invariants - Logical Error Detection. Yet, not all possible detections are dangerous for causing unstable execution of software and, consequently, manifest failures in infrastructure controls. If a program error located in some execution path does not cause unstable execution of the software, it does not manifest as a failure [10, 11]. For this reason, this method adopts a notion of risk for classifying logical error detections. Risk is quantified by means of a fuzzy logic system based on two measuring functions, namely Severity and Vulnerability. These functions complement invariant verification and act as filters. Severity, with values from a [1, 5] Likert scale, quantifies the impact of a logical error with respect to how it affects execution. Vulnerability, with values from the same scale, quantifies the likelihood of a logical error to appear.

Severity. It refers to the impact of a source code point on execution. By measuring the Severity of a point in source code, we assign a Severity rating to the accessed variables. Source code points that affect the execution flow of software are considered dangerous, since logical errors tend to manifest on the variables used in them [8, 9, 11]; for example, the IF-statement `if(isAdmin == true)` represents a check on `isAdmin`:. This conditional branch is a control flow point where un-intended execution deviations may occur [10]. Thus, the involved transition is classified as important (rating 3–5 on the scale) and rated as **Medium (3)**. A variable is assigned only one rating, depending on how the variable is used in transitions (Table 1).

Table 1. Severity ranks in the Likert scale

Linguistic value	Condition	Severity level
Low	Random variable Severity	1
Low	Random variable Severity	2
Medium	Severity for variables holding data originated from **user input**	3
Medium	Severity for variables used once on an **"IF" branch**	3
High	Severity for variables used in a conditional branch twice or more on an **"IF" branch** and/or a **"SWITCH" branch**	4
High	Severity for variables used as a **data sink** and in a conditional branch on an **"IF" branch** and/or **a "SWITCH" branch**	5

Vulnerability. By measuring the Vulnerability of a source code point, we also assign the given Vulnerability rating to the accessed variables used in its transition. A variable is assigned only one overall Vulnerability rating, depending on how the variable is used. Rating conditions are presented in Table 2 below.

Table 2. Vulnerability levels in the Likert scale

Linguistic value	Condition	Vulnerability level
Low	No invariant incoherencies / No improper checks of variables.	0
Medium	Multiple propagation of input data.**No RTU functionality affected**.	2
Medium	M*ultiple propagation* to method variables with improper checks. (**Might affect other functionality indirectly**).	3
Medium	*Improper/insufficient checks* on input data – Variables also used branch conditions. **Functionality doesn't involve field devices**.	4
High	*Improper/insufficient checks* on input data – Variables also used branch conditions**. Functionality involves field devices**.	5
High	*Invariant enforcement AND invariant violation* in alternate versions of same execution path. **Functionality controls field devices**.	5

Risk. Risk represents a calculated value assigned to each source code point and its corresponding variables, by aggregating the aforementioned Severity and Vulnerability ratings. It is calculated using Fuzzy Logic's linguistic variables in the form of IF-THEN rules, e.g. "*IF Severity IS low AND Vulnerability IS low THEN Risk IS low*". For clarity, all scales (Severity, Vulnerability and Risk) share the same linguistic characterization: "*Low*", "*Medium*" and "*High*". A complete analysis of the Fuzzy Logic system is provided in [11]. Table 3 depicts the Risk matrix used.

4 A Method to Predict Failures in Infrastructure Software

The techniques presented in the previous section has been proven effective in detecting a subset of possible logical errors in both laboratory test-beds and real-world source code alike [8–11, 33]. Since logical errors violate the functionality of software as intended by its programmers, if that functionality corresponds to a RTU, then it is obvious that these logical errors *will* create failures in controls.

Table 3. Risk for each variable = Severity x Vulnerability

Vulnerability \ Severity	Low	Medium	High
Low	Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	High

The aspects presented previously in Sects. 3 and 4, can be summarized using the following sentence: If an execution path of a program controlling an RTU violates a (combination of) dynamic invariant(s) but a different version of the same path is found to enforce the same invariants, then this means that a possible logical error exists, due to the contradiction between two executions and that source code point must be analyzed to see which RTU functionality it affects. The detailed proposed method consists of the following steps to detect such logical errors in high-level code:

1. **Gather Business Logic documentation** for the *software under test* (SUT), along with any functionality restrictions and prerequisites. Since this a white box program audit, we consider that both the source code and relevant documentation is available.
   a. Model functionality into flowcharts, depicting all software interactions with hardware control systems and RTUs.
   b. Break flowchart into multiple flows, one for each intended functionality.
2. **For each flow, perform dynamic analysis** of the SUT using the Daikon tool.
   a. Sum of executions must cover the SUT's intended functionality.
   b. Gather dynamic invariants inferred in executions (invariants are rules that effectively describe the functionality analyzed during execution).
3. **Filter dynamic invariants** and keep those referring to source code methods with high risk functionality.
   a. Source code methods that manipulate control systems (such as coils). Failure in this part of the code may result in field device malfunctions.
   b. Methods imposing confidentiality, availability or integrity measures to the program communication (e.g. authentication methods etc.). Failure in this part of the source code compromises the overall security and communication between the software and the RTU.
4. **Instrument the SUT's source code with the dynamic invariants** of interest.
   a. Invariants are embedded into code as Java Assertions. An *assertion* is a statement in the Java language that enables to test assumptions about programs [27].
5. **Symbolically execute** the instrumented source code using the JPF tool.
   a. Covers a broad magnitude of possible SUT execution paths using backtracking.
   b. Tests numerous combinations of input and execution (i.e. all possible functionality; not only the intended one).
6. **Check for assertion violations**.
   a. Flag invariants detected to be both enforced and violated in a different version of the same execution path. This indicates of a logical error.

7. **Classify each detection** into Severity and Vulnerability levels. The more a source code point affect the SUT's execution, the higher it scores in the system.
   a. If detection ratings are above medium (3), classify detection as high risk.
   b. If functionality affected causes a failure to a component or the entire control system, classify this detection as a possible initiating factor for cascading failures in depended infrastructures.

The original methodology has been previously implemented in a prototype tool with successful results in [9–11]. Briefly, the tool accepts input from Daikon (step 2) and automate steps 3 to 6 using step 2's output and the source code files.

5 Experiments and Practical Use

To the best of our knowledge, there is no commercial test-bed or open-source revision of an SUT with a reported set of existing logical errors. Since, the purpose of this article is to introduce a novel method and display its applicability, our experiments are based exclusively on formal fault injection into two different open-source test-bed examples of manipulating a RTU using Java software.

5.1 Open-Source SCADA Control Implementations

As mentioned earlier, the Internet is now widely used to connect web-based SCADA systems, since it is a cheap, efficient way to deploy geographically unrestricted control systems. Software able to communicate with such systems is being developed in high-level languages, using common open-source implementations of system protocols.

For the purposes of this article, we focus our analysis a Java library able to communicate and control SCADA systems, named JAMOD [5]. JAMOD applies the MODBUS protocol to connect software interfaces with RTUs in SCADA systems, using registers that manipulate control circuits (coils, holding registers etc.). This test utilized the tools and extensions from previous research [13].

The code found in Figs. 1 and 2 can be used with real RTUSs, like the L75 RTU unit [26]. The L75 RTU unit is an RTU that provides remote control actuation of quarter-turn valves and other rotary devices. Here, this RTU is not responsible for failure realizations; the flaw manifests in the Java code which controls the RTU.

Software-to-RTU Command Code – A Realistic Example. Let Coil 1 be the valve shaft motion control circuit. A write data command sending "FF00" will either start or continue valve movement; a write data of "0000" will stop valve movement. Other data values will return an exception response [26]. Using this realistic example, we wrote a MODBUS command able to do just tha, using the function "Write Coil" on bus 5: The hexadecimal version of the command would be: `05, 05, 00, 00, FF, 00, 8D, BE`. Figure 3 depicts high-level sample instructions able to control the flow of a valve, using the aforementioned command. When variable `stopFlow` is false, an "open valve" request is sent, otherwise a "close valve".

```
// MODBUS Request to open valve in Coil 0000 (hex)
if (!stopFlow) {
  hi = Integer.parseInt("FF",16); // open valve
}else {
  hi = Integer.parseInt("00",16); // close valve
}
low = Integer.parseInt("00",16);
reges = new SimpleRegister(hi, low);
write_sreq = new WriteSingleRegisterRequest(ref , reges);
trans.setRequest( write_sreq ); trans.execute(); }
```

Fig. 3. High-level code able to manipulate a gas shaft in an L75 RTU unit

The Logical Error. The logical error utilized is a real-world error taken from [29]. It is a classified CWE-226 error in the Common Weakness Enumeration database [30].

The control system must not utilize more than a specific amount of gas pipes, otherwise high pressure will cause a rupture, leaks and, consequently, a failure to the distribution system. Each time the program sends an RTU request to open a valve, it must check if the pressure in pipes has reached a maximum or not. If true, it should stop the flow. Variable "`stopFlow`" controls this functionality. Thus, the logical error manifests when `StopFlow` is updated. The software checks a pressure sensor to see if it should allow further increase in gas pressure or not. This check should take place each time a valve shaft is ordered to open. Yet, a higher number of pipes than the allowed maximum can be opened, by bypassing the aforementioned pressure check using alternate execution routes, due to sensitive information that remains unclear in objects.

To better understand this execution deviation, refer to Fig. 4. Due to an erroneous initialization of variable checked, the check can be bypassed if two consecutive "increase flow" instructions are sent within 3–4 s (i.e. before connection resets).

```
if (choice.equalsIgnoreCase("1")) {
  StopFlow = readRegisterPressure(con);
  checked = true;
  System.out.println("---     Information:     Max     pressure
reached");}
else if (choice.equalsIgnoreCase("2")) {
  if (checked) increaseFlowBugged(size, cushion);
  else System.err.println("--- Error: Check pressure"); }
```

Fig. 4. Source code example - checks imposed to handle pressure limits

5.2 Method Execution Results on the Faulty Example

To test our method on the example above, a simulator is used to emulate an RTU, since acquiring a fully operational SCADA control was not feasible. For our needs, we used the PLCSimulator [27], originally created to allow testing of Texas Instruments 500 MODBUS RTU serial driver without the need for physical equipment. It supports MODBUS over TCP, so this makes it ideal for testing purposes. We deployed the method presented in Sect. 5. The results for each step were the following (only output concerning the flaw is presented due to space limitations):

Step 1. Functionality in this example has two flows currently available:

```
A. Exec choice(1), Exec choice(2).
B. Exec choice(2), Exec choice(1), Exec choice(2).
```

Steps 2–3. Dynamic analysis of flows yielded 40 dynamic invariants for the selected functionality. Amongst them, the next invariant refers to the hidden logical error:

```
Bug.readRegisterPressure():::ENTER
this.checked == false
```

meaning that, upon entering execution of method `readRegisterPressure()`, variable `checked` should always be FALSE.

Steps 4–6. Dynamic invariants where instrumented inside the source code in their corresponding points and the software was executed symbolically. An assertion violation was detected for method `readRegisterPressure()`: Two execution were found where checked was TRUE and FALSE respectively, implying a logical error.

Step 7. Our method classified this invariant with **Severity = 3** (*variables used in a conditional branch once*) and **Vulnerability = 5** (*Invariant enforcement AND invariant violation in alternate versions of same path - Functionality controls RTUs*), thus yielding a **total Risk value of ~4.5** for the specific dynamic invariant and its variable.

6 Conclusions and Limitations

Results show that detecting logical error in software that may lead to CI failures is indeed feasible (up to a certain complexity level) even in real-world implementations. The use of Fuzzy Logic can exclude errors in non-critical points of the source code which do not divert execution. Risk output can be used as input to existing risk analysis methods. Currently diverse methodologies rely on expert opinion to estimate the likelihood and impact of possible threats [3, 21–24]. Introducing technical results contributes to the accuracy of assessments (albeit only for software threats).

Yet, the method suffers by the same limitations as its original form. Complex invariant rules need deep semantic analysis and loops are not supported [11]. Daikon's dynamic execution must cover as much SUT functionality as possible, otherwise, dynamic invariants inferred will not correctly describe the source code's behavior, as intended by its programmer. Logical errors that are not software-based, such as programmable circuits, cannot be analyzed using the current method.

Acknowledgment. This project has received funding from the European Union's Seventh Framework Programme for Research, Technological Development and De-mon-stration, under grant agreement no. 312450. The European Commission's support is gratefully acknowledged.

References

1. Krutz, R.: Securing SCADA Systems. Wiley, Indianapolis (2005)
2. Alcaraz, C., Lopez, J., Zhou, J., Roman, R.: Secure SCADA framework for the protection of energy control systems. Concurrency Comput. Pract. Experience **23**(12), 1414–1430 (2011)
3. Theoharidou, M., Kotzanikolaou, P., Gritzalis, D.: Risk assessment methodology for interdependent critical infrastructures. Int. J. Risk Assess. Manag. Special Issue on Risk Analysis of Critical Infrastructures **15**(2/3), 128–148 (2011)
4. OpenSCADA: Open-source supervisory control and data acquisition system. http://openscada.org/. Accessed 2014
5. Wimberger, D.: Jamod - Java modbus implementation (jamod.sourceforge .net). http://jamod.sourceforge.net/ (2004). Accessed 2014
6. Cardenas, A., Amin, S., Sastry, S.: Research challenges for the security of control systems. In: 3rd USENIX Workshop on Hot Topics in Security (HotSec 2008), USA (2008)
7. Chikuni, E., Dondo, M.: Investigating the security of electrical power systems SCADA. In: AFRICON (2007)
8. Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, J.: Toward automated detection of logic vulnerabilities in web applications. In: Proceedings of the 19th USENIX Symposium, USA (2010)
9. Stergiopoulos, G., Tsoumas, B., Gritzalis, D.: Hunting application-level logical errors. In: Barthe, G., Livshits, B., Scandariato, R. (eds.) ESSoS 2012. LNCS, vol. 7159, pp. 135–142. Springer, Heidelberg (2012)
10. Stergiopoulos, G., Tsoumas, B., Gritzalis, D.: On business logic vulnerabilities hunting: the APP_LogGIC framework. In: Lopez, J., Huang, X., Sandhu, R. (eds.) NSS 2013. LNCS, vol. 7873, pp. 236–249. Springer, Heidelberg (2013)
11. Stergiopoulos, G., Katsaros, P., Gritzalis, D.: Automated detection of logical errors in programs. In: Lopez, J., Ray, I., Crispo, B. (eds.) CRiSIS 2014. LNCS, vol. 8924, pp. 35–51. Springer, Heidelberg (2015)
12. The Java PathFinder tool. NASA Ames Research Center. babelfish.arc.nasa.gov/trac/jpf/
13. Doupe, A., Boe, B., Vigna, G.: Fear the EAR: discovering and mitigating execution after redirect vulnerabilities. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 251–262, ACM (2011)
14. Kuan-Yu, T., Chen, D., Kalbarczyk, Z., Iyer, R.: Characterization of the error resiliency of power grid substation devices. In: 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 1–8, 25–28 June 2012
15. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**, 35–45 (2007)
16. The Daikon invariant detector manual. http://groups.csail.mit.edu/pag/daikon/
17. MODBUS4J. http://sourceforge.net/projects/modbus4j/. Accessed January 2014
18. Bolton, W.: Programmable Logic Controllers. Elsevier, Amsterdam (2009)
19. IEEE Standard C37 1994. Definition, Specification and analysis of systems used for supervisory control, data acquisition and automatic control

20. Stouffer, K., Falco, J., Kent, K.: Guide to supervisory control and data acquisition and industrial control systems security. NIST (2008)
21. EAR/Pilar-risk analysis environment. http://www.ar-tools.com/en/index.html
22. Kotzanikolaou, P., Theoharidou, M., Gritzalis, D.: Interdependencies between critical infrastructures: analyzing the risk of cascading effects. In: Bologna, S., Hämmerli, B., Gritzalis, D., Wolthusen, S. (eds.) CRITIS 2011. LNCS, vol. 6983, pp. 104–115. Springer, Heidelberg (2013)
23. Kotzanikolaou, P., Theoharidou, M., Gritzalis, D.: Assessing n-order dependencies between critical infrastructures. Int. J. Crit. Infrastruct. **9**(1/2), 93–110 (2013)
24. Kjølle, G., Utne, I., Gjerde, O.: Risk analysis of critical infrastructures emphasizing electricity supply and interdependencies. Reliab. Eng. Syst. Saf. **105**, 80–89 (2012)
25. Oracle Java SE documentation. http://docs.oracle.com/. Accessed 2014
26. FlowServe L75 series electric actuator. FCD LMAIM7502-00 – 07/05
27. PLCSimulator. http://www.plcsimulator.org/. Accessed 2014
28. Kotzanikolaou, P., Theoharidou, M., Gritzalis, D.: Cascading effects of common-cause failures on critical infrastructures. In: Butts, J., Shenoi, S. (eds.) Critical Infrastructure Protection VII. IFIP AICT, vol. 417, pp. 171–182. Springer, New York (2013)
29. Boland, T., Black, P.: Juliet 1.1 C/C++ and JAVA test suite. Computer **45**(10), 88–90 (2012)
30. The common weakness enumeration initiative. MITRE Corporation. cwe.mitre.org/
31. Fan, C., Yih, S., Tseng, W., Chen, W.: Empirical analysis of software-induced failure events in the nuclear industry. Saf. Sci. **57**, 118–128 (2013)
32. Soupionis, Y., Benoist, T.: Demo abstract: demonstrating cyber-attacks impact on cyber-physical simulated environment. In: ACM/IEEE International Conference on Cyber-Physical Systems, p. 222, 14–17 April 2014
33. Stergiopoulos, G., Katsaros, P., Gritzalis, D.: Source code profiling and classification for automated detection of logical errors. In: Proceedings of the 3rd International Seminar on Program Verification, Automated Debugging and Symbolic Computation, Germany (2014)