Detecting Insecure Code Patterns in Industrial Robot Programs

Marcello Pogliani
Politecnico di Milano
marcello.pogliani@polimi.it

Federico Maggi
Trend Micro Research
federico_maggi@trendmicro.com

Marco Balduzzi
Trend Micro Research
marco_balduzzi@trendmicro.com

Davide Quarta
EURECOM
davide.quarta@eurecom.fr

Stefano Zanero
Politecnico di Milano
stefano.zanero@polimi.it

Abstract

Industrial robots are complex and customizable machines that can be programmed with proprietary domain-specific languages. These languages provide not only movement instructions, but also access to low-level system resources such as the network or the file system. Although useful, these features can lead to taint-style vulnerabilities and can be misused to implement malware—on par with general-purpose programming languages. In this paper, we analyze the languages of 8 leading industrial robot vendors, systematize their technical features, and discuss cases of vulnerable and malicious uses. We then describe a static source-code analyzer that we created to analyze robotic programs and discover insecure or potentially malicious code paths. We focused our proof-of-concept implementation on two popular languages, namely ABB's RAPID and KUKA's KRL. By evaluating our tool on a set of publicly available programs, we show that insecure patterns are found in real-world code; therefore, static source-code analysis is an effective security screening mechanism, for example to prevent commissioning insecure or malicious industrial task programs. Finally, we discuss remediation steps that developers and vendors can adopt to mitigate such issues.

CCS Concepts

• **Computer systems organization** → *Robotics*; • **Security and privacy** → **Software security engineering**;

Keywords

industrial robotics; security vulnerabilities; robot programming

ACM Reference Format:
Marcello Pogliani, Federico Maggi, Marco Balduzzi, Davide Quarta, and Stefano Zanero. 2020. Detecting Insecure Code Patterns in Industrial Robot Programs. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3320269.3384735

1 Introduction

Industrial robots are complex manufacturing machines at the center of modern factories. Robots are widely interconnected—through various protocols and technologies—to programmable logic controllers (PLCs), manufacturing execution systems (MESs), vision systems, and IT and OT networks in the factory floor. Industrial robots can be programmed online, using the "teach by showing" method, or offline, using purpose-built, domain-specific programming languages. These *industrial robot programming languages (IRPLs)* include special instructions to move the robot's arm(s), as well as common control-flow instructions and APIs to access low-level resources. Writing *task programs* (i.e., the programs that define the task to execute) in IRPLs is useful to implement custom tasks and integrate external systems in the production process. IRPLs provide access—in an almost unconstrained way—to several robot's resources like its mechanical arm(s), file-system, network, various fieldbus protocols, and serial communication.

Recent research looked into the security of industrial machinery, such as robots. In our previous research [19], we focused on the security properties of industrial robots, and in a follow-up paper [18], we mentioned how task programs are part of the attack surface, showing an example of an application written in a IRPL and vulnerable to a "path traversal" issue. Despite this, currently, there are neither security analysis tools for programs written in IRPLs, nor security mechanisms to implement resource isolation in common robotic operating systems (e.g., privilege separation). Furthermore, the security awareness within the industrial-automation community does not seem fully developed, yet. Indeed, from an analysis on 11 popular online industrial automation forums[1] totalling 294,680 users, we estimated that as low as 2.31% pages (10,868 out of 469,658) mention security-related keywords (e.g., security, vulnerability, and attack), and we discovered vulnerable code snippets[2].

As trends show an increased IT-OT convergence and a streamlined industrial software development with ample use of third party code [2, 9, 22], we advocate for a more systematic approach to *secure programs written in IRPLs*, on par with common general-purpose programming languages. As a first step, we propose a static source code analyzer that can pinpoint relevant code paths using dataflow analysis on the interprocedural control-flow graph, to detect vulnerable or malicious uses of security-sensitive primitives. For example, in Section 4 we show how our tool can tell whether and where a task program receives data from the network and uses that data to derive a file name, open such file and return its content over the network. We evaluated our analyzer on publicly available pro-

[1] https://forum.adamcommunity.com/index.php, https://dof.robotiq.com, https://automationforum.in, https://www.robot-forum.com/robotforum, https://control.com, https://solisplc.com/forum, http://forums.mrplc.com, https://www.reddit.com/r/robotics, http://plc.myforum.ro, https://forum.universal-robots.com, https://forums.robotstudio.com

[2] https://forums.robotstudio.com/discussion/11662/how-to-continue-cycle-in-automatic-mode/p1. This code snippet receives coordinates from a network socket (without authentication and boundary checks), and uses them to control the robot.

```
MODULE example
  VAR robtarget point0 := [
                [500,500,500],[1,0,0,0],[0,0,0,0],
                [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
  VAR robtarget point1 := [
                [700,500,500],[1,0,0,0],[0,0,0,0],
                [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
  VAR zonedata zone := z100;

  PROC main()
    FOR i FROM 1 TO 10 DO
      MoveJ point0, v100, zone, tool0, \WObj:=wobj0;
      WaitTime 4;
      MoveL point1, v100, zone, tool0, \WObj:=wobj0;
      WaitTime 5;
    ENDFOR
  ENDPROC
ENDMODULE
```

```
DEF example()

    DECL POS pos1
    DECL POS pos2

    pos1 := {X 500, Y 500, Z 500, A 0, B 0, C 0}
    pos2 := {X 700, Y 500, Z 500, A 0, B 0, C 0}

    FOR I=1 TO 10

        PTP pos1
        WAIT SEC 4
        PTP pos2
        WAIT SEC 5

    ENDFOR

END
```

Figure 1: Examples of programs written in two IRPLs: ABB's RAPID (left) and KUKA's KRL (right).

grams written in two of the most well-known IRPLs—by ABB and KUKA [20], both of which operate world-wide since decades and have hundreds of thousands of employees. Although source-code vetting can help detecting vulnerabilities early during development, in the longer term, we argue that vendors should implement mitigations such as resource isolation, higher-level communication primitives (e.g., with built-in type systems and/or authentication).

In summary, we propose the following contributions:

- We analyze the programming languages of 8 leading industrial robot vendors, detail the presence of complex and rich features, and discuss their vulnerable and malicious usages, which we deem collectively as "insecure" (Section 2, 3);
- We propose a static code analyzer for IRPLs to analyze task programs and find security issues in the use of sensitive primitives (Section 4);
- We use our tool to analyze a corpus of publicly available programs and show they contain vulnerabilities (Section 5).

We conclude by discussing potential remediation steps that can be adopted in the medium and long term (Section 6).

2 Programming Industrial Robots

IRPLs allow programmers to write repeatable, deterministic, and complex tasks, and to interconnect robots with external systems [6]. Like any software artifact, task programs also can contain flaws.

Proprietary Languages. Most IRPLs are similar to BASIC or ALGOL, augmented with robot-specific instructions, constructs, and data types. Task programs written in IRPLs are either interpreted or compiled to a proprietary binary format before execution. IRPLs provide access to system resources—including, for example, the robot's movement—through specific instructions and functions. Figure 1 shows two examples. Unlike general-purpose programming languages, IRPLs are proprietary and vendor specific. The vendors' different design choices result in different trade-offs between features and complexity. Indeed, despite there exist development and off-line simulation environments that support multiple languages[3], there is no true, standard cross-vendor solution or "lingua franca."

[3]For example, https://robodk.com.

Resource Abstraction. IRPLs are used to write complex applications that interact with external resources and systems (e.g., with multiple robots, smart end effectors, or external vision systems). Thus, many languages allow access to non-motion system resources: file systems, network, system configuration and communication with external devices (e.g., fieldbuses, serial communication). External systems are accessed via low-level APIs such as raw network sockets to exchange data to and from the network. According to our analysis, there is very little or no high-level API abstraction: Developers can only use low-level system resources. While this means more flexibility, it also implies more room for mistakes.

There are notable exceptions, which we hope will inspire future improvements. For example, Mitsubishi's MELFA BASIC provides a high-level network-based protocol to remotely control a robot in real time through dedicated instructions (e.g., `Mxt`, shorthand for "move external"). KUKA's KRL offers high-level network sockets that exchange XML-serialized data through a language extension (KUKA.Ethernet). Similarly, Universal Robot offers high-level XML RPC functionalities [25]. These exceptions are limited to network communication, although we argue that a similar abstraction approach should be applied to other security-sensitive functionalities, as it will be clear in the remainder of this section.

2.1 Access to System Resources

Driven by an increasing market demand, IRPLs offer great flexibility and capabilities. As a result, the programming features needed to interconnect with heterogeneous external devices and networks may in turn create the venue for well-known vulnerabilities, like input-validation and logic errors, on par with general-purpose languages. Moreover, IRPLs are expressive enough that can be used to write malicious programs that, when uploaded and executed on an industrial robot, have full and unmediated access to hardware resources. This would permit the attacker to implement complex malicious functionalities (e.g., the dropper-like behavior of Listing 4).

We surveyed the languages used by 8 popular industrial robot vendors, examining the available language reference documents, and manually looking at the implementation of real programs. In Table 1 we summarize the *sensitive* primitives that we identified. These primitives, when misused, may have an impact on the robot's security, the safety of its operators, or the connected systems.

Table 1: Sensitive primitives supported by the surveyed IRPLs. Section 2.3 provides exemplifying attack scenarios that exploit these primitives.

		File system operations and directory listing.		Load and execute code at runtime, including dynamically-defined code.		Receive from or send data to external systems.
Language	Vendor	File System	Directory Listing	Load Module From File	Call by Name	Communication
RAPID	ABB	✓	✓	✓	✓	✓
KRL	KUKA	✓				✓
MELFA BASIC	Mitsubishi	✓				✓
AS	Kawasaki					✓
PDL2	COMAU	✓	Indirect	✓	✓	✓
PacScript	DENSO			✓	✓	✓
URScript	Universal-Robot					✓
KAREL	FANUC	✓	✓	✓	✓	✓

We focus on three, broad categories of programming primitives: **file-system** access, dynamic or external **program loading**, and **communication** functionalities.

File-system Access. Opening, reading and writing files via IRPLs is a necessary mean to access configuration parameters, writing log information, storing the state of a program, or reading movement coordinates from a file written by another program. The first column ("file system") in Table 1 indicates whether the language supports programmatic access to files and directories (open, read, write), while the "directory listing" column indicates whether there is a programmatic way to list the available files and directories. The complexity of the file system varies widely across vendor: While some vendors (e.g., ABB, COMAU) provide a structured file system, others (e.g., MELFA) provide a flat, simple implementation, or no programmatic access to files whatsoever (e.g., Denso). In the case of COMAU, we indicated the capability of performing directory listing as "indirect", as it is possible to use the `SYS_CALL` instruction to execute system commands, including the command to list files in a directory, and subsequently parse the command's output.

Dynamic or External Program Loading. The ability to call procedures dynamically in IRPLs allow developers to write compact and modular programs. Hence, some industrial robots include:

- The ability to resolve a function reference in a loaded module at runtime, calling it through the function's *name*. This is achieved by passing a string with the function name to a routine like `CallByName`, or by using special language constructs (e.g., the `%"functionName"%` syntax for late binding in ABB RAPID). This functionality can be used to call a function where the function's name is composed of a common prefix concatenated with a parameter available at runtime.
- A wat to dynamically load a module from a file containing the task program code and execute it. This functionality allows to develop modular programs by loading modules based on input received at runtime. Note that, similarly to the directory listing functionality, dynamic module loading in COMAU is achieved executing a system command using `SYS_CALL`, rather than by a dedicated function or instruction.

These functionalities allow calling programs and procedures, being thus a way to dynamically change a program's execution flow.

Communication Functionalities. Industrial robots require communication functionalities to interface with external networks and systems. Some examples include receiving real-time position coordinates by an external program, interacting with a vision system that provides geometrical information on the position of the work piece, and sending feedback to external systems for logging.

All the surveyed languages provide some form of networking capabilities, either out of the box or as language extension (e.g., for RAPID, sockets are included in a optional package; for KRL, they are available in the KUKA.Ethernet KRL add-on; for KAREL, they are a language extension and not part of the core language).

Although sockets in IRPLs often work as in general-purpose programming languages, some IRPLs have noticeable differences. In KUKA's KRL, sockets define a "typed" interface such that the program can only exchange XML or typed binary data; in MELFA's BASIC and Kawasaki AS, the definition of the socket parameters (e.g., IP address, port) is performed out of band, by manually configuring the robot's parameters, and cannot be changed by a program.

All the surveyed vendors provide at least one way to exchange data over serial ports and fieldbuses, as it is a basic means of factory floor integration. Although our technical analysis does not focus on these non-IP networking systems, these are still direct channels connected to external devices, which are not necessarily trusted: Therefore, this attack vector should be the subject of future scrutiny.

2.2 Research Challenges and Goals

The primitives of Table 1 are not, by themselves, a security issue, and we do not aim to call out vendors for providing them: There are ways to use them securely, and their availability is a signal of the complexity and maturity reached by industrial robotics.

However, the *security impact* of these primitives has not been studied yet. Our previous research [18] found a path traversal vulnerability in a real-world program: A web server written in ABB's RAPID. This vulnerability allows attackers to access (sensitive) files outside of the root directory served by the application. This case motivated us to investigate potential insecure uses of such primitives, and to develop a IRPL-specific program analysis tool to support vulnerability discovery, as well as detection of malicious functionalities. Our goal is to bring this to the attention of the security and robotics communities, and raise awareness to the vendors.

2.3 Attack Scenarios

With the sole purpose of providing context for the remainder of the paper, we hereby provide three attack scenarios, in which an adversary may abuse the security-sensitive primitives of Table 1. We remark that these scenarios are *fictitious*, although in our opinion useful to showcase the risks of a misuse of IRPL primitives.

Industrial software—including task programs for robots—do not always provide authentication or encryption, because they assume a closed and trusted environment. However, with the increased integration of the factory floor with external services, such assumption is becoming less realistic, and arguably not future proof. Moreover, recent advanced attacks have already shown capability to propagate down to the factory floor, sometimes even up to the safety system (e.g., see Trisis or Xenotime in [24]).

Scenario 1: Unauthorized Data Access. Let's consider a task program that writes in a log file the coordinates of the paths followed by the robotic arm during its operation. Such log file is used for auditing, calibration, and quality assurance (e.g., root-cause analysis of defective products), and may contain sensitive information like intellectual property (i.e., how a product is built). Also, the task program opens a network socket because an external agent needs to retrieve the log files for post-processing and archival purposes.

Let's now assume that an attacker has compromised a machine within the same network of the robot. As a first step, the attacker may try to impersonate the agent (e.g. through ARP spoofing), connect to the socket, and exfiltrate the log.

Then, the attacker may want to move laterally, by planting malware in the robot's machine and remaining persistent. However, the login console (e.g., telnet) is password protected and does not contain known vulnerabilities. However, the attacker understands that the task program that keeps the socket open is affected by a path-traversal vulnerability: The application trusts that any request coming from the agent will contain a legitimate file path relative to the directory where the log files are stored. The attacker may be able to exploit this vulnerability to access the file containing authentication secrets, and use that to finally access the target machine via login console and compromise the robot's machine.

The first program listed in Table 5 contains an instance of such vulnerability, which our analyzer was able to automatically detect.

Scenario 2: Task Flow Alteration. Let's consider a task program that receives a stream of coordinates via a socket. This is often the case for real-time external control task programs, which allow robots to be controlled by other endpoints.

Like in **Scenario 1**, let's assume an attacker within the same network of the robot's machine, yet with no access to it. The attacker wants to disrupt the robot's operation, to alter its execution flow, causing damage and impacting on the safety of the manufacturing station. There is proper network-level protection (IP and MAC filtering), which ensures that the robot receives coordinates only from the designated controller. However, the task program is affected by an input-validation vulnerability: Any received coordinate value is automatically trusted. Therefore, an attacker able to impersonate the controller can send arbitrary coordinates, and the robot will just act accordingly and potentially cause damage.

Table 2: Sources of untrustworthy data (i.e., sensitive sources) – Section 3.1

Type	Intended Use Case	Attacker Model
File	Static data from configuration files	Contractor
Inbound Communication (network, serial, fieldbus)	Dynamic real-time data	Untrusted networks or endpoints
Teach Pendant (i.e., UI forms)	Operator-supplied data	Insider

All the programs marked as "External move", "Remote control," and "Integration server" in Table 5[4] contain instances of such vulnerability, which would allow an attacker to arbitrarily control the robot's movements.

Scenario 3: Persistent Information Stealing. Let's assume that a robot runs a task program written by a system integrator. The system integrator is either compromised, or the task program is fetched from a misconfigured or vulnerable storage (e.g., FTP). At a first glance, we may say that in such scenario an attacker could straightforwardly replace the task program to change the automation of the robot. However such a drastic approach may be noticed.

If the task program is written in any of the IRPLs (listed in Table 1) that supports dynamic code loading and networking primitives, the adversary has the opportunity to run a stealthier attack. They may slightly alter the source code of the task program to include a network communication routine that fetches code from outside (or from a hidden file), and then use dynamic loading to run it *as part of the normal automation loop*.

Although there are different strategies to implement such attack, in Table 4 we show how our analyzer can be configured to detect the code patterns of one example implementation.

3 Unsafe Programming Patterns

We now detail more technically how some IRPL primitives become sources of untrusted data, while some others can result into vulnerable uses of such untrusted data. We also show how some primitives can be abused for malicious purposes, e.g., to develop malware written in IRPLs. In our threat model, the attacker can plant a malicious task program into the robot. As exemplified in Section 2.3, this can happen in various ways and locations of the software supply chain. A detailed investigation on the initial entry points is beyond the scope of this work: We focus exclusively on the IRPLs' capabilities.

3.1 Sources of Untrustworthy Data

We consider a source of untrustworthy data any avenue where a task program written in a IRPL receives and processes data coming from the external world. Particularly, as summarized in Table 2, task programs receive inputs from files, communication mechanisms (e.g., network, serial communication, fieldbuses), and the teach

[4]Excluding those with 0 (zero) patterns.

Table 3: Vulnerable uses of untrustworthy data (i.e., in sensitive sinks) – Section 3.2

Type	Intended Use Case	Attacker Goal (Example)
Movement	Programmatically manoeuvre the robot	Unintended robot movement
File Handling	Read arbitrary files	Data exfiltration
File Modification	Write configuration	Implant a backdoor
Call by Name	Write parametric and generic code	Divert the control flow

pendant's user interface. Such (untrusted) input, if not correctly handled by the program, can be abused by an attacker. For example, data files could be tampered with by malicious third-parties like contractors; inbound communication data could originate from compromised networks and endpoints; and user interfaces could be manipulated through the physical attack surface by an insider.

3.2 Vulnerable Uses of Untrustworthy Data

Untrustworthy data can be used in various patterns that result into software vulnerabilities. Particularly, we distinguish four broad categories of "sensitive sinks." These functions, when called with parameters derived from (tainted with) untrustworthy data, lead to taint-style vulnerabilities, as summarized in Table 3.

3.2.1 Movement Commands The tainted data (received from a sensitive source) is used to control the robot's trajectory. This pattern is widely used as a way to control or influence the robot's movement from an external program, even in near real-time: Indeed, it is supported out of the box by some vendors. For example, Mitsubishi's MELFA robots provide the `Mxt` (move external) instruction, which automatically listens for UDP packets containing information about the robot position, and which is intended as a way to perform real-time control of the robot. Similarly, ABB provides the Robotware Externally Guided Motion option, which allows an external device to perform direct motion control.

Example. A popular implementation of this pattern are adapters the for third party middleware such as ROS[5], in particular the project ROS-Industrial[6], which extends ROS to support industrial robots, and became the reference open-source platform for industrial robotics [3]. To use ROS with an industrial robot, one has to rely to its interface; in many cases, the robot-side interface is a task program which listens on a network port, accepting commands that will be directly translated to the robot's movement and returning information about the robot's status over the network.

In all of the task programs that we examined, we found no authentication nor validation on the movement coordinates: An attacker who can send data on the network is able to issue any movement command, possibly moving the robot outside its normal operating range. Listing 1 shows an example written in KUKA's KRL.

[5]Robot Operating System, a popular robotic research platform: https://ros.org
[6]https://rosindustrial.org

3.2.2 File and Configuration Handling Tainted data received from a sensitive source such as a network socket is used as part of the filename parameter of a "file open" or "configuration open" instruction, without validation. In this case, a network attacker can control the name of the (configuration) file to be opened and read, leading to the disclosure of sensitive information (e.g., secret files, intellectual property) or to the overwrite of sensitive configuration files. If the robot controller implements a structured file system (e.g., COMAU, ABB, KUKA) rather than a flat one, this issue may lead to the classic directory traversal vulnerability. Note that the extent of file-system access granted to the attacker may be limited by OS-level isolation functionalities, similarly to the "chroot" mechanism in Unix.

Example. In previous research [18], we found a real case of vulnerable application for industrial robots, which was made available through ABB's RobotApps platform[7]. The application runs as a web server providing static and dynamic pages. This application is affected by a directory traversal vulnerability allowing access to arbitrary files, including the system's configuration: An attacker is able to reach secret targeted files, including sensitive configuraton, via a well-formed HTTP request (e.g., `../../secret.txt`). Listing Listing 2 presents an example of vulnerable webserver in RAPID with a directory traversal vulnerability.

3.2.3 File and Configuration Modification Functions Orthogonal to file handling functions, untrustworthy data may be used as the

[7]As a result of a disclosure to ABB, this program is no longer publicly available.

```
DEF external_movement()
    DECL axis pos_cmd

    eki_init("ExiHwInterface")
    eki_open("EkiHwInterface")

    LOOP
        eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A1", pos_cmd.a1)
        eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A2", pos_cmd.a2)
        eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A3", pos_cmd.a3)
        eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A4", pos_cmd.a4)
        eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A5", pos_cmd.a5)
        eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A6", pos_cmd.a6)

        PTP joint_pos_cmd
    ENDLOOP
END
```

Listing 1: Vulnerable socket-controlled movement (KRL).

```
MODULE VulnWebServer
  PROC main()

    SocketCreate server;
    SocketBind server, '0.0.0.0', 1234;
    SocketListen server;

    SocketAccept server, sock;

    WHILE true DO
        SocketReceive sock, \RawData:=data;
        fileName := ParseCommand(data);
        Open fileName, res;
        ReadAndSendFile(\file:=res, \socket:=sock);
    ENDWHILE
  ENDPROC
ENDMODULE
```

Listing 2: Path traversal vulnerability. (RAPID)

```
MODULE VulnCodeLoader
    PROC main()

        SocketCreate server_socket;
        SocketBind server_socket,"0.0.0.0", 1234;
        SocketListen server_socket;

        WHILE loop DO
            SocketAccept server_socket, client_socket;

            SocketReceive client_socket \Str:=data;
            function_name:=ParseFunction(data);

            %function_name%;    ! call procedure by name

            WaitRob\ZeroSpeed;
            SocketSend client_socket\Str:="R move completed";

            SocketClose client_socket;
        ENDWHILE
    ENDPROC
ENDMODULE
```

Listing 3: Vulnerable code loader. (RAPID)

content to be written in configuration files, or passed as parameter to configuration setting functions. If the data is not sanitized (e.g., checked against a white list, or against an acceptable range), an attacker may overwrite configuration values in an unexpected and potentially unsafe way.

3.2.4 *Call Procedure By Name*

Some IRPLs have the capability of resolving at runtime and programmatically the names of the routines to be called ("late binding"). For example,a developer may use a construct like `CallByName(fun_name)` in order to call a function, where `fun_name` is a string variable containing the function to be called. If this variable originates from an untrusted source and there is no input validation, the program is vulnerable: An attacker may subvert the control flow of the program, with varying effects according to the semantics of the loaded module(s).

Example. We found an instance of this programming pattern in an ABB RAPID program we found online (Listing 3 presents a simplified version). This program implements a server with multiple functionalities; to select the functionality to be called, instead of using a chain of `if` constructs, the name of the functionality is received from the socket and then passed as a parameter to a "late binding" construct. An attacker can exploit this instance of vulnerability to call any other function in the same task program.

3.3 Malicious Patterns

Enumerating all potential abuses of primitives for malicious patterns is an endless game and only limited by the creativity of the malware author, and it is intrinsically harder to compile an exhaustive list of malicious behaviors. In Table 4, we limit our focus to two examples of classic behaviors commonly found in modern malware, first to confirm that they can be implemented in IRPLs, and secondly to show how they appear in terms of code patterns.

The information stealer pattern is particularly relevant in industrial settings because both the configuration parameters and the programs residing on the robot controller are considered high valuable intellectual property, and therefore attractive assets for attackerss. A malicious IRPL making use of the information stealer patter will, for example, exfilitrate confidential information from local files through an outbound connection. Note that our attack model assumes that the attacker has *not* straight read-access to the file system, but rather infects an existing task program with an information-stealing malicious routine like in the case of a malicious, or compromised system integrator.

```
MODULE Dropper
    PROC main_loop()

        ! ... variable declaration
        ! ... socket creation and initialization

        WHILE TRUE DO
            SocketReceive clientsock, \Str:=data;
            name := ParseName(data)
            Open diskhome + "/" + name + ".mod", f;
            WHILE data DO
                SocketReceive clientsock, \Str:=rec;
                Write f, rec;
            ENDWHILE
            Load \Dynamic, diskhome \File:=name + ".mod";
            %name + ":main"%; ! call function by name
        ENDWHILE
    ENDPROC
ENDMODULE
```

Listing 4: Example of dropper malware. (RAPID)

In the dropper case, the attacker is able to download and execute any second-stage malware like we capture in Listing 4. Contrary to previous research on PLC malware [10, 16, 17], we purposely decided not to focus on the functionalities of the downloaded malware (e.g., network target enumeration, file harvesting), but rather representing this as a generic pattern.

4 A Source Code Static Analyzer for IRPLs

To quantify and analyze the extent of unsafe programming patterns in task programs, we conceived and implemented a prototype static source code analyzer for IRPLs. On one hand, we used our tool to perform an analysis of publicly available IRPL code and show which, and to what extent, unsafe and vulnerable patterns are found. As described thoroughly in section 5, none of the programs that we analyzed implemented proper input sanitization. On the other hand, we propose the use of static analysis to verify task programs before upload on robots, in order to anticipate the security (and safety) impact of the use of such programs in production.

Our proposed tool processes task programs and uses taint analysis to detect data flowing from one or more sensitive sources (e.g., data from the network) to one or more sensitive sinks (e.g. movement, file open, late binding). The analyzer is modular with respect to the supported IRPLs, and the searched code patterns are configurable by means of queries on the data flow. Our proof-of-concept implementation supports ABB's RAPID and KUKA's KRL.

Table 4: Examples of malicious patterns.

Case	Feature	Source	Sink
Information stealer	Exfiltration	File	→ Outbound Network
	Exfiltration	Config	→ Outbound Network
	Harvesting	Dir. list	→ File
Dropper	Download	Communication	→ File (code)
	Execute	File (code)	→ Call by name

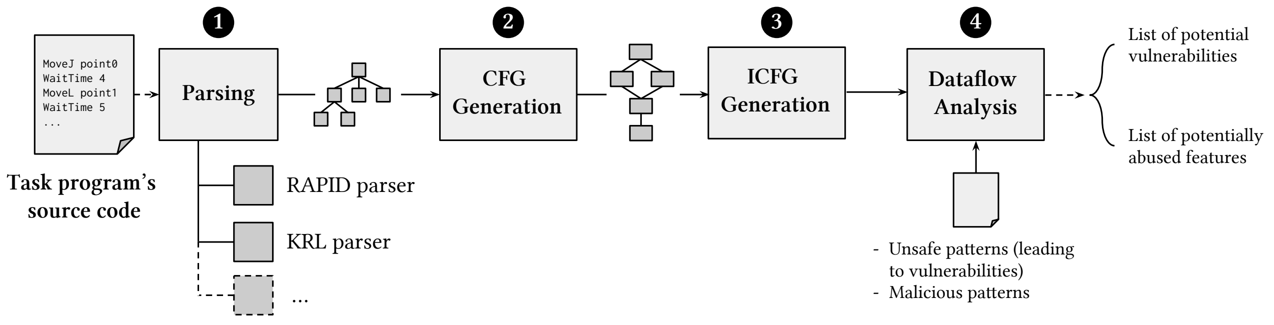

Figure 2: High-level workflow of the proposed source code analyzer.

Taint analysis techniques have been successfully applied to finding vulnerabilities in general-purpose programming languages, such as C and web development languages, but, to the best of our knowledge, there are no security-oriented applications to IRPLs. We show that taint analysis is efficient in detecting vulnerable uses of untrustworthy data and malicious patterns as outlined in Section 3, as they all can be expressed by means of source–sink paths.

4.1 System Overview and Workflow

Figure 2 shows a high-level workflow of our analyzer, which consists of the following steps: parsing, control-flow graph (CFG) generation, interprocedural control-flow graph (ICFG) generation, and data-flow analysis.

We start by parsing a task program's source code, and walking through the parse tree to generate the CFG of each function. We then produce an ICFG with the goal of representing the control flow between functions. We do this by linking the functions' CFG at function call statements. Finally, we perform a data-flow analysis on the constructed ICFG to detect potentially sensitive data flows from sources to sinks.

The final output of our analyzer consists of the list of sensitive data flows (source–sink pairs), along with some context useful to the analyst, including, for example, code lines information and relevant function names. An example is given in Listing 5, in which our tool detected a vulnerable pattern from the code of `eki_hw_iface_get` to the code of `kuka_eki_hw_interface`. In particular, it found a path from `eki_getreal` (i.e., parse a numerical value from incoming network data) to `joint_pos_tgt` (i.e., move joint to position).

4.2 Implementation Details

We implemented our prototype analyzer in Python 3, using the `networkx`[8] library for graph-manipulation tasks.

Steps 1–2: Parsing and CFG Generation. We implement the parser by using Antlr[9] to generate both the lexical analyzer and the LL(*) parser from a specification of the IRPL grammar. We developed the grammars from the information available in the reference manuals of the robot languages, and by looking at existing programs. In general, writing the grammar for a new language is not necessarily a hard task: As an example, the official language reference for ABB's RAPID [1] includes portions of the EBNF grammar, which we ported to Antlr.

Once an IRPL program is parsed, we visit its parse tree and build the CFG in memory. Each node of the CFG (also known as *basic block*, in program-analysis terminology) contains a list of instructions. These instructions are expressed in a language-independent, simplified, intermediate representation[10]. We adopted a modular approach to make our tool easily extensible to different robot languages. In particular, the only analyzer's components that are IRPL-specific are the grammar specification and the CFG generator, implemented through Antlr's visitor pattern.

Once the CFG is built, we run a set of (language-agnostic) simplification passes: For example, we add CFG edges at `goto` statements, we enforce a single exit point (return) for the CFG of each function, and we eliminate dead code blocks.

Step 3: ICFG Generation. The second step of our analysis consists in generating the ICFG. To this extent, we visit the CFG of each function, and substitute all those nodes with calls to functions defined in the same module (i.e., functions where the CFG is available) with two CFG edges:

[8]https://networkx.github.io/
[9]https://www.antlr.org/

[10]Our intermediate representation does not preserve the complete semantics of the instructions, but only their data flow. This is all we need for taint analysis.

```
{
    "sources": [
        {
            "src_var": "joint_pos_cmd",
            "source": "eki_getreal",
            "source_fn": "eki_hw_iface_get",
            "source_line_no": 180,
            "source_filename": "kuka_eki_hw_interface.src"
        }
    ],
    "sink_var": "joint_pos_tgt",
    "sink": "ptp",
    "sink_fn": "kuka_eki_hw_interface",
    "sink_line_no": 73,
    "sink_filename": "kuka_eki_hw_interface.src"
}
```

Listing 5: Output of the analysis of a task program that implements externally controlled movements (Listing 1 shows an excerpt of that).

- an edge from the instruction immediately preceding the call *to* the entry basic block of the callee's CFG. To properly model the data flow from the function calls' actual parameters to the function's formal parameters, we add additional assignment nodes along this edge.
- an edge from the exit basic block of the callee's CFG *to* the instruction following the call. We also add further nodes to correctly propagate the returned value to the caller, as well as to propagate the value of any output parameter declared as such in the function prototype.

With this procedure, we build an "exploded super graph" of all the functions in the program under analysis.

Step 4: Dataflow Analysis. We follow a forward-only dataflow analysis for taint tracking, which propagates taint information from the functions defined as sources (e.g., inbound network data) towards all the basic blocks (nodes) in the program. Afterwards, we check whether any input parameter of the functions defined as sink (e.g., coordinate passed to robot-movement functions) was tainted, and by which source. To do so, the analysis algorithm keeps track of the set of "taints" (i.e., the set of sources that influenced the value of the variable) for wach variable in every ICFG node.

To compute the result of our analysis, we use a work-list based iterative algorithm. In its essence, the dataflow analysis is defined by a *carrier lattice*, which represents the information computed for each node of the ICFG, and a *transfer function*, which defines how the information is propagated according to the semantics of each instruction. Elements in the carrier lattice are the set of sources that taint each variable. The transfer function forward-propagates the taint information from the variables used by the instruction to the variables defined by the instruction. For example, the transfer function for a binary operation adds to the taint information of the result the union of the taint information of its two operands.

Some function calls refer to functions whose implementation is not present in the analyzed programs: They may be calls to library functions or to functions defined in a file not available to the analyzer. As the analyzer doesn't have the function's CFG, we approximate the behavior of such functions assuming that the function uses all the parameters to compute the return value, if any. Hence, the default transfer function for function calls adds—to the taint information of the return value—the union of the taint information of all the parameters. However, many library functions do not work this way: They may have output parameters, as well as accepting parameters that do not influence the result in a security-sensitive way. Ignoring this fact would lead to an imprecise analysis. Hence, we model calls to library functions in a language-specific fashion: For each supported language, and for each library function, we specify which parameters are considered inputs and which parameters are considered outputs for taint propagation purposes.

Finally, our transfer function supports the concept of *sanitization*, that is an operation that removes the taint from a variable. This reflects the behavior of functions that are used for input sanitization, or that change the handled resource. For example, to track whether some data is written (exfiltration) to a user-contorlled file, we can consider the `Close` instruction as a sanitizer, as further uses of the same (closed) file descriptor would necessarily refer to a different file. Our tool supports a configuration-defined set of sanitizers.

4.3 Source and Sink Configuration

The searchable code patterns are configurable by means of source–sink pairs, so that our tool is generic with respect to them. For our evaluation, we used source–sink pairs that express the *vulnerable* patterns described in Section 3.

KUKA's KRL Configuration. As sources, we consider those functions receiving data from network via the KUKA.Ethernet KRL extension: functions starting with `eki_get` such as `eki_getreal`, and functions belonging to the KUKA.Ethernet KRL XML package, (e.g., `EKX_GetIntegerElement`). As sinks, we consider instruction movements such as `ptp`, `lin`, and `circ`.

ABB's RAPID Configuration. To detect vulnerable uses of sensitive primitives, our sources are the parameters of the function `SocketReceive` (i.e., `Str` and `RawData`). Our sinks include movement, file- and configuration-handling functions and late binding: `Move`, `Open`, `OpenDir`, `SaveCfgData`, `WriteCfgData`, `Load`, and `CallByVar`.

Malicious Behavior Detection. When detecting potentially malicious behavior, it is possible to configure our tool to use sources and sinks that reflect the patterns proposed in Table 4. For example, to detect exfiltration in ABB's RAPID, we monitor taints from `ReadRawBytes` (and other device read functions) to `SocketSend`. Since there is no universal definition of "malicious behavior," this list is not exhaustive and other patterns can be used.

5 Evaluation

Our analysis tool can detect both security-sensitive code patterns that could lead to vulnerabilities in task programs as well as malicious patterns that could lead to malware. First, using our tool we were able to confirm the path-traversal vulnerability that the authors of [18] found manually. Such vulnerability resulted in ABB removing the vulnerable application from the online repository. We found the very same vulnerability automatically, with no guidance. We also discovered several instances of task programs that handled unsanitized data received from sensitive sources, and used it to control the movements of the robot. Notable examples of this case are the various ROS-Industrial adapters, which consist of IRPL code that interface the communication protocol of the major robot vendors with that of ROS-Industrial. An attacker on the network could exploit such vulnerability to influence the movements of a robot's arm. Secondly, we ran our analyzer against a proof-of-concept of malware that we implemented, as described in Section 3.3. Our tool detected the malicious code patterns, showing that it is helpful to implement code-vetting systems.

5.1 Dataset

Our dataset[11] consists of publicly-available IRPL programs that we collected via open resources. Specifically, we used the search functionality of popular source-code repositories (like GitHub and BitBucket) and Google's advanced-search operators, searching for files with RAPID and KRL's extensions, plus some language keywords (e.g. `MoveJ` or `MoveL` for RAPID) to filter false positives. Out of the found programs, we filter only ones that use at least one of

[11]The dataset is available from: https://robosec.org/data/asiaccs2020

Table 5: Summary of findings including the number of detected insecure patterns per analyzed program. Only the programs having at least a sensitive source are considered in the analysis. Note that the label "false positive" refers to the presence of `at least` one case of false positive among the detected patterns. Note that `ptp`, `lin` and `circ` are movement instructions in KRL.

#Files	#LOC	Purpose	#Patterns	Language	Details	Errors
Vulnerability Class: network ⟶ code execution						
1	130	Demonstrator	2	RAPID	2 `%late binding instruction%`	
1	123	Demonstrator	2	RAPID	2 `%late binding instruction%`	
Vulnerability Class: network ⟶ filesystem						
4	974	Web server	5	RAPID	2 `Open`, 2 `FileSize`, 1 `OpenDir`	
Vulnerability Class: network ⟶ movement						
10	1,878	External move	51	RAPID	18 `MoveC`, 12 `MoveAbsJ`, 21 `MoveL`	9 False Positives
9	672	Experimental code	9	KRL	5 `lin`, 4 `ptp`	
1	634	Integration server	52	RAPID	25 `MoveL`, 16 `MoveAbsJ`, 5 `MoveJ`, 6 `MoveC`	
1	628	Integration server	52	RAPID	25 `MoveL`, 16 `MoveAbsJ`, 5 `MoveJ`, 6 `MoveC`	
1	537	Remote control	6	RAPID	4 `MoveL`, 2 `MoveAbsJ`	
2	460	External move	26	RAPID	10 `MoveL`, 10 `MoveAbsJ`, 6 `MoveC`	
3	453	External move	26	RAPID	10 `MoveL`, 10 `MoveAbsJ`, 6 `MoveC`	
2	397	Integration server	4	KRL	2 `lin`, 1 `ptp`, 1 `lin_rel`	
1	338	Integration server	9	KRL	2 `circ`, 5 `ptp`, 2 `lin`	
1	106	Integration server	5	KRL	2 `ptp`, 1 `lin`, 1 `ptp_rel`, 1 `lin_rel`	
1	111	Educational code	5	KRL	2 `ptp`, 1 `lin`, 1 `ptp_rel`, 1 `lin_rel`	
1	76	Integration server	1	KRL	1 `ptp`	
1	60	Code snippet	1	RAPID	1 `MoveJ`	
No vulnerability found						
32	7,165	Palletizer	0	KRL	-	
5	1,038	Integration server	0	RAPID	-	
6	337	Integration server	0	RAPID	-	3 False Negatives
2	199	Integration server	0	RAPID	-	
5	165	Example code	0	KRL	-	
1	70	Code snippet	0	RAPID	-	

the sensitive sources we consider in our analysis—other programs cannot contain any vulnerable patterns according to our definition. Overall, we collected 91 task-program files using at least a source, divided into 14 RAPID projects (39 files) and 8 KRL projects (52 files), totaling 16, 551 lines of source code excluding comments.

Representativeness. We are aware that it is very hard to find production code among public resources, because it contains intellectual property, developers are likely bound to non-disclosure agreements, and so are not allowed to share all of their artifacts with the community. However, being our work the first step in this research direction, this dataset is the only publicly-available resource. In terms of reproducibility, we argue that it is the only available research dataset to assess the correctness and the performance of tools such as ours. Alarmingly, in this humble dataset we found *zero cases* of properly implemented input validation: In other words, if a sensitive primitive was used, it would always be used insecurely. We believe that this result is useful to make our point and raise security awareness in the automation and robotics communities, which we believe would benefit from our findings.

Vulnerable Code Samples. We highlight that our dataset includes adapter code for ROS-Industrial—the reference open-source middleware for industrial robotics [3]. Although we cannot conclude about the actual adoption of such adapters, anyone searching online for sample implementations would unavoidable find these resources. Interestingly, recent research [26] showed that publicly available, vulnerable code snippets end up in real-world, popular code bases—even with modifications, yet still vulnerable. In other words, it is undeniable that developers use public resources to learn and, unfortunately, when lacking the necessary security awareness, they tend to propagate vulnerable code.

5.2 Detection Capabilities

According to the results of the automated analysis that we performed with our tool, 45.4% and 12.2% of programs (RAPID and KRL respectively) present sensitive patterns. If we count in only those programs with sensitive sources, 71.4% of the RAPID programs and 75.0% of the KRL programs have sensitive patterns that may lead to vulnerabilities.

Summary of Findings. A summary of our findings is in Table 5. The table shows the number of patterns identified for each program[12]. By *pattern* we mean an instance of a variable that "flows" into a sink and is tainted by data flowing from a sensitive source. Multiple tainted variables passed as parameters to the same sink are

[12]We consider only those programs with sensitive sources

counted as different patterns. For example, if a movement command is issued with both the target position and the velocity parameters tainted by a sensitive source, we count two patterns.

ABB's RAPID Programs. Out of the 14 RAPID programs reported in Table 5, 8 had a flow from the string parameter (`Str`) of the `SocketReceive` function towards the parameters of one of the robot movement functions e.g. `Move`; 1 program had a flow from `SocketReceive` to the late binding construct; 1 program (the web server mentioned in Section 3.2) had a flow from `SocketReceive` to the filename of multiple file handling functions, among which `Open`), leading to a path-traversal vulnerability.

KUKA's KRL Programs. Out of the 8 KRL programs with sensitive sources, 6 had at least one sensitive pattern that lead to arbitrary movement control (5 programs from the KUKA.Ethernet KRL extension and 1 from the KUKA.Ethernet KRL XML package). We did not find instances of KRL program with different sinks, such as file handling; late binding is not supported by KRL.

True Negatives. We also manually analyzed the 41 remaining KRL programs that did not have sensitive sources. Interestingly, 15 and 13 of them use either the RSI and the FRI interface, respectively. RSI and FRI allow external sources (e.g., sensors) to influence the movements of the robot. Unlike the KRL-based interfaces, the KRL program does not explicitly read data from the network and move the robot according to it; instead, the robot's controller automatically receives data—via UDP—and moves the robot accordingly. Hence, there is no *explicit* (i.e., visible) information flow in the KRL program: the KRL program just needs to set up the RSI and/or FRI interface and issue a start command, much like Mitsubishi's move external (Mxt) command. Using these interfaces allows a precise and fine-grained robot control with real-time requirements. Instead, the RAPID programs without sensitive sources did not implement any other technique to acquire external input.

5.3 Discussion and Limitations

According to our analysis, unsafe patterns are unfortunately a recurring case in robotic applications, especially for socket-controlled robot movement—not to mention the KRL programs using real-time external control interfaces. Our results justify the needs of more secure development platforms and languages, as well as major awareness on the security implications of the sensitive patterns we reported. We hope that our work will contribute in this direction.

The Case of Network Adapters. The majority of the detected patterns are for network-controlled robot movements. Network adapters are the most widely found category of publicly-available task programs as they represent an important component for platform integration. All the network adapters we found were vulnerable: The developers put no constraints on the allowed trajectories to be commanded, and they implemented no authentication. To exploit this, a network attacker needs only to send unexpected coordinates, resulting in the robot arbitrarily following the instructions.

5.3.1 False Positives and False Negatives Our tool reported two incorrect results, both related to ABB's RAPID programs and due to limitations of the current implementation. We manually analyzed both cases and discuss them in the remainder of this section

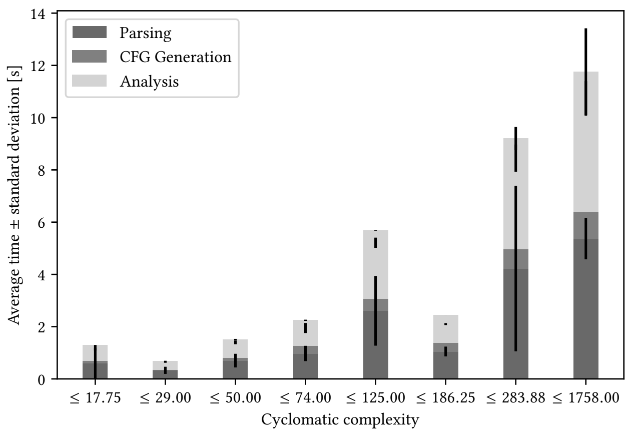

Figure 3: Performance of our tool (executed on a laptop with an Intel i5-6287U processor and 16GB RAM). Bins are equally sized, obtained with quantile-based quantization.

False Positives due to Context Insensitivity. In one program, our tool wrongly reported a flow from a network socket to a movement instruction. This false positive is due to the lack of context sensitivity in the data flow analysis that we implemented. More precisely, when propagating the taint across ICFG's edges of different functions (i.e., across an edge representing a call or a return), the analyzer fails in "linking" the call edge with the correct return edge; hence, it computes an over-approximation by propagating the data flow towards all the return edges of a function. This problem arises particularly in the case of small functions heavily used throughout the program (e.g., utility functions such as logging functions).

Future work could overcome this limitation by context-sensitive dataflow analysis, such as by framing the problem as an interprocedural, finite, distributive subset problem and solving it through state-of-the-art graph reachability algorithms suggested by the code optimization community [21].

False Negatives due to Indirect Flows. The second error consists of a false negative in the ROS adapter for ABB's RAPID. This adapter is organized as two concurrently running tasks (similar to threads in conventional programming languages) and synchronized using shared memory and interrupts. A task manages the network communication, stores the received trajectory into a shared variable, and signals the event by asserting a boolean flag that triggers an interrupt; the second task handles the interrupt and manages the robot's movement. When the interrupt is triggered, the program reads the new trajectory from the shared variable. Here, the data flow between the two threads is not reflected in the two distinct control flow graphs, but by the fact that the two threads run concurrently and use the same shared variables; hence, a classic dataflow analysis cannot handle this case, as the taint is not propagated through the control flow. We leave this as future work.

5.4 Performance

Figure 3 shows the performance of our analysis tool. Despite our analyzer is implemented in an interpreted language (Python) and is not necessarily optimized for performance, it is able to perform the analysis of thousands of lines of code in few seconds, proving to be an clear improvement with respect to performing a manual analysis. This also shows that this approach can be use to automatically scan for vulnerable or malicious programs before they are uploaded to the robot, or to software repositories.

6 Remediation and Future Research

The security issues that we identified and discussed can be mitigated. In the long term, for example, potential actions include the redesign of IRPLs, security-aware runtime environments (e.g., to support privilege separation or permission systems), and hardened operating systems for robots. However, these changes are foreseeable in the long term only. In the short term, we recommend a program-level mitigation in which tools like that one we proposed can aid developers in improving their software, and vendors and system integrators to verify the programs that they deploy.

Input Validation. As for general-purpose programming languages, an effective way to mitigate taint-style vulnerabilities is proper input validation. For instance, when parsing untrusted data to obtain motion commands or coordinates, programmers should verify that the requested values fall within application-specific boundaries, so to avoid unsafe conditions. When untrusted data is used as filename to open files, programmers should disallow path separator characters or implement whitelisting mechanisms. When using user input to compose the name of a late-bound function to call, programmers should whitelist the allowed function names (partially hindering the programming convenience of using late binding). We believe that, like in general-purpose programming languages, developers of IRPL programs should adopt these well-known best-practices; most of the surveyed IRPLs include basic functionalities (e.g., string manipulation) needed to build input-validation procedures.

Secure Communication. Network communication between robot programs typically occur between trusted parties. Because of this, IRPLs do not to support authenticated or encrypted communication. To the best of our knowledge, we are not aware of IRPL libraries that provide strong network encryption schemes (like TLS). Therefore, it's hard for developers to implement secure communication protocols, and custom solutions are often failing strategies. We believe that, in the long term, robot vendors should offer SDKs backed by OS-level support for authentication and encryption.

Secure External-move Commands. Several of the analyzed programs leverage external movement functionalities to remotely control industrial robots. Given the popularity of this use case, vendors should provide the programmers with high-level external movement functionalities that are authenticated and secure.

Privilege Separation and Permission Systems. Another important system-level recommendation is privilege separation, and the implementation of permission systems in general: The patterns listed in Table 1 and 3 may lead to security issues primarily because they require access to low-level, privileged resources. Ideally, like in mobile development, IRPL programs that access privileged resources must declare so in a "manifest". This would allow to design privilege separation or fine-grained permission systems, such that to prompt for resource access at runtime. This is a challenging path, because each vendor has complex sets of primitives, with various degrees of "impact" on the underlying resources.

Code Signing. Unwanted or malicious code patterns can be mitigated through tools like that one we propose. This should be complemented with code-signing mechanisms, like in mobile software-distribution ecosystems, which guarantee integrity and authenticity of the code running on each device, preventing, for example, backdoored code to run—under the assumption that the attacker who places the backdoor hasn't compromised the private key.

7 Related Work

This work relates to (a) the security implications of IRPLs and (b) IRPL program analysis to infer security properties. Although static analysis techniques are common to detect vulnerabilities in general-purpose languages, to the best of our knowledge, we are the first to extend them to find vulnerabilities and malicious code in IRPLs.

Security of Industrial Domain-specific Languages. The use of domain-specific languages as an attack vector for industrial control systems has been actively investigated, especially for PLCs. Govil et al. [10] present a series of malware for PLCs (written in IEC61131-3 ladder logic) and built to conceal the malicious code from human analysis; McLaughlin [16, 17] automatically analyzes a compromised target to dynamically generate malicious PLC programs; Klick et al. [13] misuse the network programming features of programming languages for PLCs to build a network proxy, thus showcasing how it is possible to perform lateral movement by leveraging the primitives exposed by PLC languages. Contrary to these works that mainly focus on the abuse of programming languages for PLCs, our work addresses the problem of robot-specific industrial languages and how robotics programs developed in such languages can be potentially analyzed for vulnerabilities and abuses.

Taint Analysis. The use of static program-analysis techniques to find vulnerabilities is a well-studied research area. In particular, we apply dataflow analysis, a robust program analysis technique originated in the 1970s [12] and widely applied to several security problems (e.g., to automatically find vulnerabilities in web applications [11], privacy leaks in mobile applications [8], or detect mobile ransomware [4]). General-purpose programming languages rely on improved and advanced data-flow analysis tools like FlowDroid, for Android [5], and Phasar [23], for C/C++ applications analysis. A recent, notable example is Joern [27], which generates a so-called property graph, stored in a graph-oriented database for efficient mining of sensitive patterns, potentially leading to vulnerabilities.

Program Analysis for Industrial Languages. Analysis tools for ICS logic have been developed mainly for quality-assurance purpose or to extract safety-related properties. Cortesi et al. [7] focus on program verification applied to robotic software, offering an overview of various static analysis techniques (e.g., model checking, data flow analysis). Zhang et al. [28] analyze the logic of PLCs and FANUC robot code, to extract invariants for software-testing purposes. Here, the authors needed to extract only the points of interaction between the IRPL code and the PLCs, in order to find safety-critical conditions. Instead, we aim at characterizing the entire dataflow of a task program to find insecure patterns. Similarly to our tool, Mandal et al. [14, 15] use static code analysis techniques, including dataflow analysis on the CFG, to analyze task programs in a multi-language fashion (including, among the others, ABB RAPID and IEC 61131-3 PLC languages). However, their focus is on checking the conformance to coding standards and detecting common programming mistakes (e.g., infinite loops, division by zero) that can result in safety violations or generic faults, and they do not consider resource access, which is crucial for security.

In short, none of the related work consider the security risk brought forth by the fact that (1) task programs are and will be no longer isolated, (2) they include powerful security-sensitive primitives such as those that offer connectivity between robots and outside world, and (3) have no resource isolation. Hence, to the best of our knowledge, we believe we are the first to apply security-oriented static program-analysis techniques to IRPLs, and successfully detect vulnerabilities in real-world robotics programs.

8 Conclusion

In this paper, we investigated potential security risks introduced by robotic programs developed in IRPLs. We analyzed the languages of 8 leading industrial vendors, outlining the presence of sensitive primitives that can be misused or lead to vulnerabilities, showing concrete examples. Then, we introduced a prototype source code static analyzer. With our analyzer, we showed that unsafe patterns exist in publicly available programs to different extents and, on the other hand, we proposed such a tool in analyzing and vetting task programs (e.g., before commissioning or distribution). In the future, we aim to extend the capabilities of our analyzer on two aspects. We plan to support other programming languages, possibly beyond the scope of robotics, for example to program computer numerical control machines. Also, we aim at introducing context-sensitive data flow analysis in order to support additional peculiar features offered by IRPLs, such as interrupt-driven control flow.

Acknowledgments

The authors would like to thank Martino Vittone, who contributed to an early prototype of the analyzer. This research has been partially funded by a research grant Politecnico di Milano received from Trend Micro Inc., and has also received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No.: 690972.

References

[1] ABB. 2010. Technical reference manual. RAPID Instructions, Functions and Data types.
[2] ABB. 2019. RobotStudio Apps. https://robotapps.robotstudio.com
[3] Tanya M. Anandan. 2019. ROS-Industrial for Real-World Solutions. https://www.robotics.org/content-detail.cfm/Industrial-Robotics-Industry-Insights/ROS-Industrial-for-Real-World-Solutions/content_id/7919
[4] Nicoló Andronio, Stefano Zanero, and Federico Maggi. 2015. Heldroid: Dissecting and detecting mobile ransomware. In *International Symposium on Recent Advances in Intrusion Detection*. Springer, Cham, 382–404.
[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299
[6] Geoffrey Biggs and Bruce MacDonald. 2003. A survey of robot programming systems. In *Proc. Australasian conference on robotics and automation*. 1–3.
[7] Agostino Cortesi, Pietro Ferrara, and Nabendu Chaki. 2013. Static analysis techniques for robotics software verification. In *IEEE ISR 2013*. IEEE, IEEE, Seoul, 1–6.
[8] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications.. In *NDSS*. 177–183.
[9] Sam Francis. 2016. Universal Robots launches app store and developer program. https://roboticsandautomationnews.com/2016/06/21/universal-robots-launches-app-store-and-developer-program/5856/
[10] Naman Govil, Anand Agrawal, and Nils Ole Tippenhauer. 2017. On ladder logic bombs in industrial control systems. In *Computer Security*. Springer, 110–126.
[11] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.
[12] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proc. 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. ACM, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945
[13] Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, and Volker Roth. 2015. Internet-facing PLCs-a new back orifice. *Blackhat USA* (2015), 22–26.
[14] Avijit Mandal, Raoul Jetley, Meenakshi D'Souza, and Sreeja Nair. 2017. A static analyzer for Industrial robotic applications. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 24–27.
[15] Avijit Mandal, Devina Mohan, Raoul Jetley, Sreeja Nair, and Meenakshi D'Souza. 2018. A generic static analysis framework for domain-specific languages. In *2018 IEEE 23rd Intl. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. IEEE, 27–34.
[16] Stephen McLaughlin. 2011. On Dynamic Malware Payloads Aimed at Programmable Logic Controllers. In *Proc. 6th USENIX Conference on Hot Topics in Security* (San Francisco, CA) *(HotSec'11)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=2028040.2028050
[17] Stephen McLaughlin and Patrick McDaniel. 2012. SABOT: specification-based payload generation for programmable logic controllers. In *Proc. 2012 ACM conference on Computer and communications security*. ACM, 439–449.
[18] Marcello Pogliani, Davide Quarta, Mario Polino, Martino Vittone, Federico Maggi, and Stefano Zanero. 2019. Security of controlled manufacturing systems in the connected factory: the case of industrial robots. *Journal of Computer Virology and Hacking Techniques* 15, 3 (01 Sep 2019), 161–175. https://doi.org/10.1007/s11416-019-00329-8
[19] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. 2017. An Experimental Security Analysis of an Industrial Robot Controller. In *2017 IEEE Symposium on Security and Privacy (SP)*. 268–286. https://doi.org/10.1109/SP.2017.20
[20] Market Research Reports. 2019. World's Top 10 Industrial Robot Manufacturers. https://www.marketresearchreports.com/blog/2019/05/08/world%E2%80%99s-top-10-industrial-robot-manufacturers
[21] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proc. 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 49–61.
[22] RobotShop. 2019. RobotShop App Store. https://www.robotshop.com/en/robot-app-store.html
[23] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 393–410.
[24] Joseph Slowik. 2019. Evolution of ICS Attacks and the Prospects for Future Disruptive Events. https://dragos.com/wp-content/uploads/Evolution-of-ICS-Attacks-and-the-Prospects-for-Future-Disruptive-Events-Joseph-Slowik-1.pdf
[25] Universal Robots Support. [n.d.]. XML-RPC communication - 16326. https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/xml-rpc-communication-16326/
[26] Tommi Unruh, Bhargava Shastry, Malte Skoruppa, Federico Maggi, Konrad Rieck, Jean-Pierre Seifert, and Fabian Yamaguchi. 2017. Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC.
[27] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.
[28] Mu Zhang, James Moyne, Z Morley Mao, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, et al. 2019. Towards Automated Safety Vetting of PLC Code in Real-World Plants. In *Towards Automated Safety Vetting of PLC Code in Real-World Plants*. IEEE, IEEE, 0.

A Example Task Programs

This appendix reports an excerpt of two vulnerable programs. Both programs allow to command the robot's movement from the network, thus presenting a "network → movement" vulnerability class.

```
MODULE GBS (SYSMODULE)
        ! -- network adapter: https://git.cs.lth.se/mathias/GBS_Robotstudio/
        VAR socketdev server_socket;
        VAR socketdev client_socket;
        VAR num IN_CMD_ID;
        ! ...
        VAR intnum intnumNailer;
        VAR intnum intSensorYOK;
        VAR intnum intSensorXOK;
        !...

PROC GBSReadCommand() ! read command from network
        VAR num counter := 1;
        IN_PARAMS_NUM := readNumberFromSocket();
        IN_CMD_ID := readNumberFromSocket();

        WHILE counter <= IN_PARAMS_NUM DO
                IN_PARAMS{counter} := readNumberFromSocket();
                counter := counter + 1;
        ENDWHILE

        !set the command status to error and no params
        OUT_STATUS := STATUS_OK;
        OUT_PARAMS_NUM := 0;
ENDPROC

PROC GBSExecCommand() ! execute movement command
        TEST IN_CMD_ID
                CASE CMD_READ_PTP: READ_PTP;
                CASE CMD_MOVE_PTP: ! ...
        ENDTEST
        OUT_STATUS := STATUS_OK;
ENDPROC

! ...

PROC rRunGBS() ! main loop
        WHILE TRUE DO
                TEST GBS_STATUS
                        CASE 0:
                                IF (GBSNeedConnection()) THEN
                                        GBSConnectSocket;
                                ENDIF
                    ! ...
                        CASE 1:
                                GBS_STATUS := 2;
                                GBSExecCommand;
                        ! ...
                        DEFAULT:
                                GBSDisconnectSocket;
                                ! ...
                        ENDTEST
        ENDWHILE
        ERROR
                GBS_STATUS:=0;
                GBSDisconnectSocket;
ENDPROC

FUNC num readNumberFromSocket() ! parse number from raw socket
        VAR num nval;
        VAR rawbytes raw_data;
        SocketReceive client_socket \RawData :=
            raw_data  ReadNoOfBytes:=4\Time:=WAIT_MAX;
        UnpackRawBytes raw_data \Network, 1, nval \Float4;
        return nval;
        ! ...
ENDFUNC

PROC writeNumberToSocket(num nval) ! write number to raw socket
        ! ...
ENDPROC
! ...
ENDMODULE
```

```
; ROS-I KUKA adapter: https://github.com/ros-industrial/kuka_experimental
def kuka_eki_hw_interface() ; main function
   decl axis joint_pos_tgt
   decl int elements_read

   bas(#initmov, 0)
   eki_hw_iface_init()
   joint_pos_tgt = $axis_act_meas
   ptp joint_pos_tgt

   $advance = 5
   loop ; main loop
      elements_read = eki_hw_iface_get(joint_pos_tgt)
      ptp joint_pos_tgt c_ptp
   endloop
end

def eki_hw_iface_init() ; initialize network interface
   decl eki_status eki_ret

   global interrupt decl 15 when
        $flag[1]==false do eki_hw_iface_reset()
   interrupt on 15
   global interrupt decl 16
        when $timer_flag[1]==true do eki_hw_iface_send()
   interrupt on 16
   wait sec 0.012
   $timer[1] = -200
   $timer_stop[1] = false

   eki_ret = eki_init("EkiHwInterface")
   eki_ret = eki_open("EkiHwInterface")
end

def eki_hw_iface_send() ; write data to network socket
   decl eki_status eki_ret
   decl real vel_percent

   if $flag[1] then
      eki_ret = eki_setreal(
                      "EkiHwInterface",
                      "RobotState/Pos/@A1",
                      $axis_act_meas.a1)
      eki_ret = eki_setreal(
                      "EkiHwInterface",
                      "RobotState/Pos/@A2",
                      $axis_act_meas.a2)
      ; ...
      if $flag[1] then
         eki_ret = eki_send(
                 "EkiHwInterface",
                 "RobotState")
      endif
   endif
   ; ...
end

deffct int eki_hw_iface_available() ; check if there is data form network
   decl eki_status eki_ret

   ; ...
   eki_ret = eki_checkbuffer(
        "EkiHwInterface", "RobotCommand/Pos/@A1")
   return eki_ret.buff
endfct

deffct int eki_hw_iface_get(joint_pos_cmd :out) ; read data from network
   decl eki_status eki_ret
   decl axis joint_pos_cmd
   ;...
   eki_ret = eki_checkbuffer(
        "EkiHwInterface", "RobotCommand/Pos/@A1")
   if eki_ret.buff <= 0 then
     return 0
   endif

   ; ...
   return 1
endfct
```