SCI3D: Report

Xabier LEGASPI

July 2025

Abstract

Fused Filament Fabrication has numerous applications in Industry 5.0. It has permitted firms to perform rapid prototyping for decades and, in the recent years, researchers have been making progress to overcome technical challenges. Now is becoming adopted for low volume production of customized products on-demand. Firms would like to explore possibilities of new business models that depend heavily on digitalization and are organized in a decentralized and locally distributed manner. However, managers will have to face a wide range of security threats, such as theft of technical data.

The goal of this report is to first present an in-depth security review of FFF printers, showing manufacturers how they are failing to protect the confidentiality and integrity of 3D models of printed objects. In particular, we intend to demonstrate the impact of sniffing attacks at the level of the communication between the different hardware components of FFF printer boards. To do so, we propose a novel attack oriented to FFF printers based on Klipper firmware that permits to reconstruct a 3D model from low-level data with a extremely high precission, in contrast to the side-channel reconstructions present in the literature. We also hint the proof-of-concept of a possible deffensive technique.

Keywords. Fused Filament Fabrication, Firmware Security, Cyber-Physical Systems.

♦ Contents

♦ 1 Introduction

Additive Manufacturing (AM) – also known as 3D printing – involves fabricating a 3D object by incrementally adding material layer by layer using a 3D printer machine. Among the existing AM technologies, *Fused Filament Fabrication* (FFF) is the most prevalent [34]. FFF is positioned to provide a disruptive transformation in the business models and supply chain of industrial production [5, 9, 7, 26]. Disrupting technologies are, as a rule, developed without having security in mind. It is not before the large-scale deployment of the technology that it becomes exposed to security threats [39, Section 19.1].

The *FFF process chain* involves the creation of a digital design file, its compilation in the G-code instructions that control the FFF printer actions, and the transfer of these G-codes to the FFF printer for building the 3D object. Two major threats to this process chain are conditioning the large sale adoption of FFF: *theft of technical data*, such as design and G-code files; and *sabotage* of FFF printers and printed objects. Even though there have been no publicly disclosed security incidents so far, managers are becoming increasingly concerned about these threats.

Despite the growing body of research on security of FFF, gaps remain. First, published attacks usually involve the explotation of FFF printer firmware vulnerabilities [59, 32, 33, 43, 38, 3], and the reconstruction of design data from side channels of FFF printers [1, 14, 16, 17]. Nevertheless, prior work has not provided any systematic low-level analysis of open-source FFF printer firmware vulnerabilities, only a high-level one [42]. Moreover, even though some side-channels are able to reconstruct the geometry of the object with high accuracy, there are no indications ensuring that an object reproduced from the reconstructed geometry preserves the original quality. Second, defensive techniques usually involve encryption, blockchain technology, digital right management (DRM) [56, 26, 55], or side-channel signatures [2]. However, to the best of our knowledge, hardware-base, defensive techniques have not yet been proposed. Further, it has been pointed out by the experts that FFF-specific protection measures for FFF printers are yet to be proposed [50].

The G-code commands G1 make up about 95% of the G-codes sent to a FFF printer to build a 3D object. In this work, we study the data-flow relative to the transformation made to a G1 command by the FFF printer firmware. This transformation creates a sequence of low-level commands that are able to control directly the FFF printer hardware. In particular, we will study *Klipper* [25] firmware due to its open-source aspects, its current deployment in next-generation machines, and specially for its decoupling of the FFF printer hardware in several boards that are connected to a common control area network (CAN). As an application, we obtain a new attack that permits a complete reconstruction of the geometry of a part from low-level data. The adversary model consists on an attacker with physical access to the FFF printer and with the capability of connecting a malicious device to the CAN of the FFF printer that permits the exfiltration of the low-level data transiting among boards.

This report is structured as follows. In section 2, we give some background for FFF. In section 3, we make a state of the art of published attacks and defensive techniques relative to FFF printers. In section 4 we describe the data flow of G1 command in Klipper. In section 5 we explain the reconstruction of the G1 command, our main contribution, and we propose an attack. In section 6 we explain a proof-of-concept (PoC) of a deffensive technique permitting to protect the CAN communication. We conclude in section 7.

⧫ 2 Background

A *part* is joined material forming a functional physical object that could constitute all or a section of an intended product. *Additive Manufacturing* (AM), also known as 3D printing, is the process of designing, slicing and printing a part by building it one layer at a time with the help of a computer-aided machine, known as 3D printer. There are many different AM techniques, with each having their own benefits and limitations and each being able to build parts from different raw materials [47, 4]. This work focuses on *Fused Filament Fabrication* (FFF), which is the most widely used AM technique [34].

This section is structured as follows. In section 2.1 we describe the architecture of a FFF printer. In section 2.2 we introduce G-codes. In section 2.3 we give a detailed description of the process chain in FFF. In section 2.4 we bring in context distributed manufacturing. In section 2.5 we give a high-level overview of security threats in FFF. In section 2.6 we introduce Klipper firmware. Finally, in section 2.7 we state our research goal.

■ 2.1 FFF Printer

The *FFF build process* is the computer-aided mechanical and thermodynamic process in which a *spool* of solid *thermoplastic filament* is fed, heated, melted, and deposited on a build plataform in a predetermined path, where the filament cools, solidifies and joins to form a solid part (Figure 2). A *FFF printer* is an organised set of hardware and software resources designed to automatize the FFF build process. In this section we describe the FFF printer architecture.

Conceptually, a simple FFF printer can be abstracted to a model ressembling to Figure 1.

Print head

The *print head* comprises the *cold end* and the *hot end*. At the cold end, a stepper motor known as the *extruder* loads filament from the spool to the FFF printer and feeds it to the hot end, ensuring a smooth transition from solid to melted filament. The extruder is associated with an axis E. At the hot end, the filament first melts controlled by the *heater cartridge*, *thermistor* and *cooling fan*, and then is extruded onto the print bed through the nozzle to create the part.

Print bed

The *print bed* or *build plataform* is the surface on which the nozzle puts down the first layer of the molten filament. Most FFF printers have a heated print bed to help with the adhesion of the print. Once a layer is complete, the print bed moves up or down and the FFF build process repeats building up the part layer upon layer.

Motion system

The FFF printer moves the print head around, sliding it over the *rails* and laying down melted material at a precise position, where it cools down and solidifies. The *toolpath* is the precise path that the printer head follows to extrude material. The printer models the movement of the print head in a 3D vector space of canonical basis X, Y, and Z, which represents the *axis*. There is a stepper motor in correspondence with each of the axis X,Y, and Z. The movement of the print head depends on the *kinematics* of the printer. The most common kinematics are the cartesian, coreXY, polar, and delta. Each kinematics uses its own system of coordinates to update the position of the print head in the 3D vector space. The most simple kinematics is the cartesian one. The *feedrate* is the speed at which any of the axis are moved. The *endstops* are a sensor that enable the printer to detect its position along each axis, indicating when an axis end has been reached. Some printers integrate the endstops exclusively in the firmware.

Board

The *board* inteprets instructions and controls the actuators and sensors of the FFF printer that handle movement and temperature. This is typically a printed circuit board (PCB) with a microcontroller, stepper motor drivers, and any other required connectivity to the printer and interfaces to the user. Often the boards include serial ports to interact directly with the firmware, accepting communication protocols such as CAN bus, Ethernet, USB, or UART. The *firmware* of the printer consists on the software components of the board whose source or binary code is written to and read from the non-volatile memories of the board. Components may include bootloaders, operating systems, network interfaces and applications. The firmware interprets incoming instructions and acts according to them. The most popular open-source firmwares are Klipper, Marlin, and Rep-Rap. Many proprietary FFF printers use a version of these firmwares that is adapted by the manufacturer.

Power supply unit (PSU)

Most of the power that the PSU supplies to the FFF printer is oriented to filament melting and print head motion.

Interface

Most FFF printers come with a LCD screen that permits human interaction with the controller board; network interfaces; and they can be reached over the network with a web application integrated in the firmware.

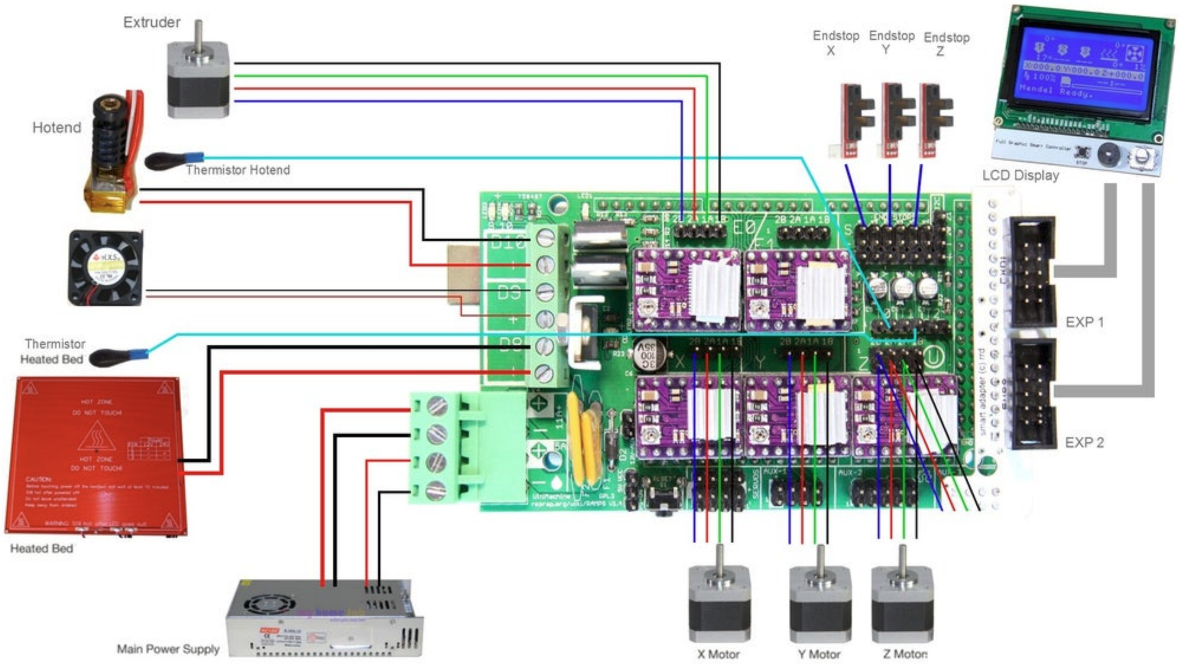

Figure 1: Schematic of a FFF printer

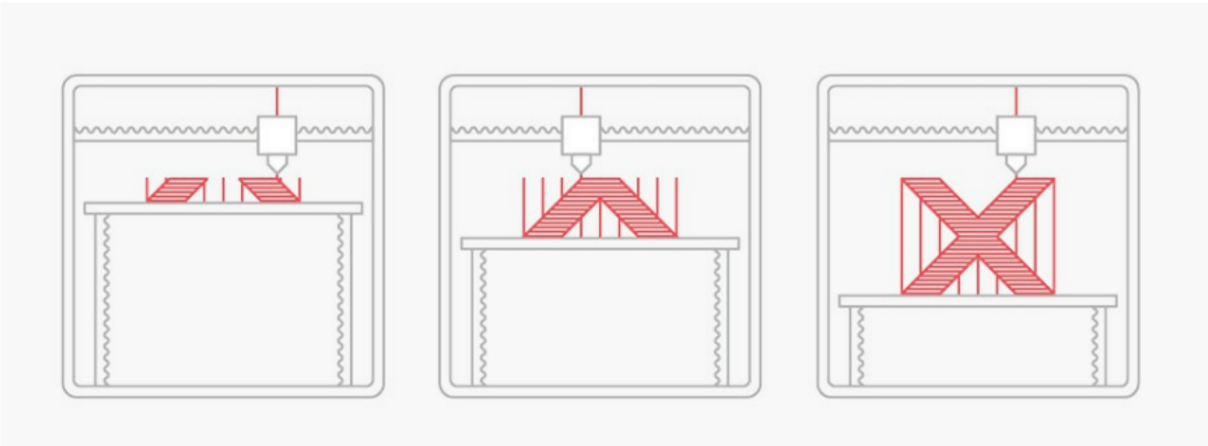

Figure 2: FFF build process [47]

2.2 G-Code [49]

In this section we describe the instruction set of a FFF printer.

A *word* is a letter followed by a number, e.g. `X1`. The letter of a word is called *address character*. A *G-code* is a sequence of words encoding an instruction for a FFF printer. The first word in a G-code is the *command*; all the remaining words are its *arguments* [60]. For example, `G1` in line 4 of Figure 3 tells the printer to move the print head in a straight line to the given coordinates `X`, `Y`, and `Z`, and to extrude the given amount of material `E`. The units are in mm. G-codes can reuse their argument values for next commands. For example, `G1` in line 3 of Figure 3 defines how fast the movement should be done indicating a feedrate `F`. The unit are in mm/min. This value is reused for all upcoming `G1` commands until it is changed again, The combination of multiple G-codes forms a program that produces a physical object; we call this a *G-code file*. The most common address characters used in G-code files are `G` and `M`. The language family of all G-codes is generally known as *G-Code*.

The G-codes supported by a FFF printer depend on the printer's firmware and the hardware. Although there is a G-Code standard, many vendors modify and extend the standard G-codes to fit the needs of FFF, so there is a patchwork of different variants. For instance, using the letter `E` to control the extruder is not part of the G-Code standard. See the Wiki of the RepRap project [48]. Other vendors also use instruction formats similar to G-Code to create additional commands for general configuration and status. For instance, in Klipper firmware, the G-code-like instruction `SET_KINEMATIC_POSITION` `X`$=1.$ `Y`$=0.$ `Z`$=1.$ forces the FFF printer to believe that the print head is at the position $(1, 0, 1)$. See the Klipper documentation [24].

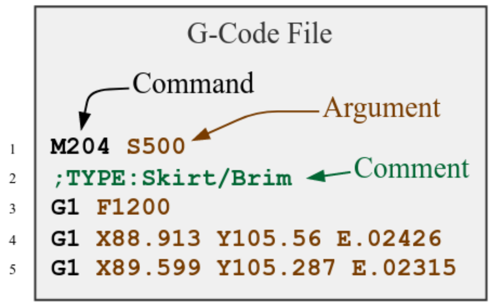

Figure 3: G-Code File Example [49]

■ 2.3 Process Chain in FFF

The *FFF process chain* divides FFF into a series of stages that transform a concept product into a part: (i) designing, (ii) slicing, (iii) printing, and (iv) post-processing (Figure 4). These stages are executed sequentially and form a pipeline such that the output of one stage is the input for the next. In this section, we analyze the FFF process chain.

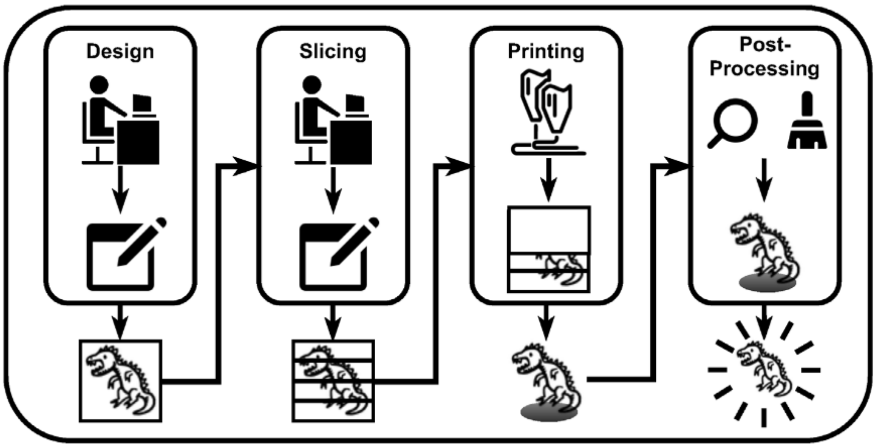

Figure 4: FFF Process Chain [54]:

▲ 2.3.1 Designing

Design engineers transform a concept product into a digital design file for a part with specific *functional* and *mechanical* properties. They use *Computer Aided Design* (CAD) software. There are two steps:

(i) **Volume Model Generation.** A *volume model* of the part is created in a parametric way. Generating initial design data with reverse-engineering techniques, such as 3D scanners, may be helpful if a concept part is physically available [39, Chapter 16]. In FFF, functional properties depend on geometry whereas mechanical properties depend on support structures, part orientation, anisotropy, infill, holes [47, Chapter 11], and optimization [37].

(ii) **Facet Model Generation.** A *facet model* describing the boundary geometry of the part is created with the conversion of the CAD file into a specific format. The *Standard Tessellation Language* (.stl) file format is the most common. Alternatively, newer file formats are the XML-based *3D Manufacturing Format* (.3mf) or the *Additive Manufacturing File* (.amf) format, which can contain metadata [39, Section 15.2.2] with additional specifications.

▲ 2.3.2 Slicing

Manufacturing engineers, manipulating the facet model, configure the FFF process parameters to calculate the optimal toolpaths for the print head to follow when depositing material for each layer. The engineers use slicing software. There are two steps:

(i) **FFF Process Parametrization.** The most common FFF process parameters are nozzle print speed, nozzle distance to part, nozzle extrusion temperature, nozzle diameter, nozzle filament supply speed and print bed temperature. [39, Section 15.2.4]. This step has a general impact on building speed, quality and mechanical properties of the part. It depends on the FFF printer and source material.

(ii) **G-Code Generation.** Recall subsection 2.2. The generated G-code file (.gcode) lists the G-codes that are deduced from the optimal toolpaths and that will control the printer. In particular, these G-codes control the amount of extruded material; heating, melting and cooling temperature of the hot end; temperature of the print bed; and the motion of the print head along the X, Y, and Z axis. Besides, in FFF, there are print parameters that appear implicitely in the G-code file, such as support location, part orientation, infill, and layer height. For instance, the layer height corresponds with the movements of the Z axis. We remark that the G-code file targets specific FFF process parameters and source material [39, Section 15.2.3].

▲ 2.3.3 Printing

The part is manufactured on the FFF printer under the supervision of a FFF printer operator. There are two steps:

(i) **FFF Printer Preparation.** In FFF, printer preparation actions include the control of quality of relevant printer components, such as the power source, the print bed and the nozzle; and at least the following activities: fill the printer with the right type and quantity of source material; and transfer the G-code file to the printer using different methods that include memory cards, USB connections, and wide or local area network connections (WAN or LAN). See [39, Section 15.2.5] and [51].

(ii) **FFF Build Process.** Recall subsection 2.1. During the build process, a FFF printer follows commands of the G-code file to print a part. At this point the FFF printer only depends on the transmitted G-code file, power supply, source material supply, and potential FFF printer operator interventions, such as part removal. See [39, Section 15.2.6] and [51].

▲ 2.3.4 Post-Processing

In FFF, quality assurance engineers perform at least (i) support removal, (ii) surface finishing, and (iii) quality control, the last one to verify that the functional and mechanical properties of the part meet design specifications, excluding possible defects. See [47, Section 2.5] and [39, Sections 15.2.8 and 15.2.9]. This step is heavily dependent on source material and printed part.

2.4 Distributed FFF

Distributed FFF [61, 7, 15, 20] refers to the division of the FFF process chain across multiple, often geographically dispersed locations to produce parts close to the point of demand. The goal is to shorten the supply chain, shorten lead times, and lower logistics costs by localizing production. Digital design files are shared with certified local facilities, which then manufacture the parts on demand, enabling rapid response to regional demand or regulatory changes.

FFF disrupts the traditional manufacturing industry by outsourcing, shortening and virtualizing the traditional manufacturing process chain at the cost of introducing cyber-physical security threats all along the process chain. Indeed, the FFF process chain creates a stream that includes software providers, cloud storage providers, design engineers, manufacturing engineers, FFF printer operators and quality-assurance engineers that could belong either to the same organization or be partners of different organizations, depending on the underlying business model [39, Chapter 5]. Each phase in the FFF processs chain involves the use of hardware or software that is connected to the internet. Moreover, printing involves physical access to FFF printers.

Conceptually, distributed FFF can be abstracted to a model ressembling to Figure 5.

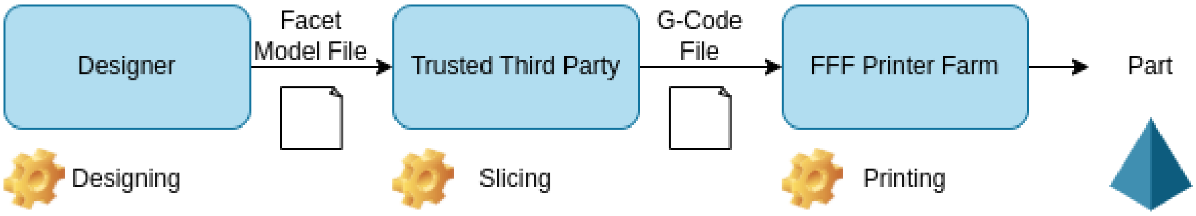

Figure 5: Schematic of Distributed FFF

In this model, there are three actors: a designer, a trusted third party and a FFF printer farm, each of them carrying out respectively, the designing, the slicing and the printing phases of the FFF process chain. We remark that there is technical data of two different nature involved during slicing that comes from the design and printing phases. Besides, there are situations where the slicing software is integrated in the CAD software (design phase) or in the FFF printer firmware (printing phase) [39, Section 15.2.3]. For these reasons, slicing is the most controversial phase in terms of cyber-physical security for outsourcing business models, specially due to the risk it poses to the intellectual property [61]. Therefore, it is reasonable to think that threats involving the reconstruction or manipulation of the G-code file will draw the attention of attackers. A trust model such that of Figure 5 reduces the risk proportionally for both the designer and the FFF printer farm.

■ 2.5 Security Threats in FFF

A *security threat* is what an adversary *would do*, considering the estimated impacts of attacks without the probability of their manifestation. From a high-level viewpoint that focus on the whole FFF process chain, the FFF security research literature shows broad consensus on the threat categories of technical data theft, sabotage, illegal part manufacturing, and data in-/exfiltration [63, 39, 42]. In this section we describe these threats (Figure 6).

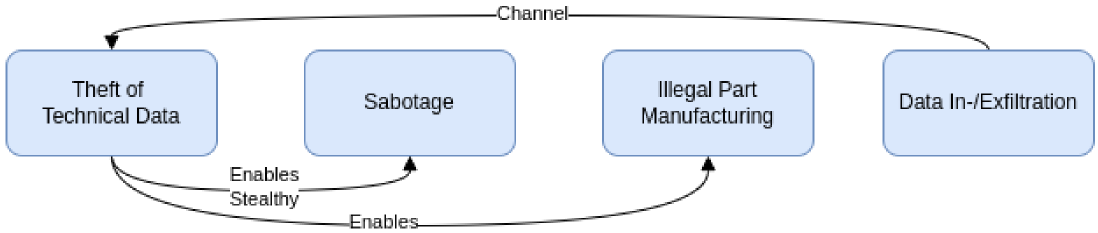

Figure 6: FFF Security Threats

▲ 2.5.1 Theft of Technical Data

Theft of technical data refers to attacks stealing intellectual property (IP); trade secrets not protected as IP; and theft-worthy data that may not be considered sensitive by the holder but could act as an enabler or enhancer of sabotage or illegal part manufacturing attacks. Theft of technical data has four attack targets: part geometry and required properties (functional, mechanical), FFF process parameters, and printing environment. For instance, competitors might be interested into the types of prototypes being printed in the research lab; and sometimes they would like to obtain information about the network or the printer to plan future attacks.

Attack methods fall in three categories. In the first category there is a benignly protected environment and an external adversary installing or inducing the installation of malware. The installation could be done using local area connection (LAN) or wide area network connection (WAN). The adversary exploits a software vulnerability of CAD software, slicing software, printer firmware, or networked device to launch a confidentiality attack and uses a covert channel to exfiltrate the obtained information (subsubsection 2.5.4). In the second category, the adversary has physical access to the printer and exploits a printer firmware vulnerability to launch a confidentiality attack that permits to exfiltrate information via a wired serial connection to the printer. In the third category, the adversary performs a side-channel attack in proximity to or with physical access to the printer. The measured physical quantities could be sound, electromagnetic emissions, temperature, or power consumption at the printer level; and time at the CAD software, slicing software and printer firmware levels.

▲ 2.5.2 Sabotage

Sabotage refers to attacks causing denial of service, deliberate physical damage or destruction. Sabotage has five attack targets: part, FFF printer, and the environment of each stage of the

FFF process chain (designing, slicing, printing).

Attack methods fall in four categories. In the first category the adversary disrupts the printing service by causing perceptible physical damage to the printer or to the part geometry; or by creating software interruptions at the level of the CAD software, slicing software or printer firmware. In the second category, the adversary conducts a traditional denial of service attack on networked devices of any of the environments. In the third category, the adversary causes imperceptible physical damage to the part to degradate its functional or mechanical properties. For example, this could cause a denial of service during the part assembly due to size unfitness, or a decrease in the lifetime of the part. In the fourth category, the adversary targets the environments installing malicious software to either launch integrity attacks on the networked devices; or to circumvent high-temperature safety-controls of the FFF printer, exploting the thermoplastic material flammability and toxicity, which could cause a fire or influence the air quality with toxic particles.

▲ 2.5.3 Illegal Part Manufacturing

Illegal part manufacturing refers to the unmonitored reproduction of parts in a sensible context. It is usually a consequence of theft of technical data. Illegal part manufacturing has three attack targets. First, reproduction of parts protected by digital rights. Second, reproduction of parts that are subject to export controls at a military scale, such as those related to nuclear programs. It is unknown whether or not FFF could be used for prototyping some of these parts. Third, reproduction of parts that are dangerous or subject to legal restrictions at a civilian scale, such as firearms. In particular, FFF is used in the production of homemade firearms and in the production of useful tools for the authorities. This threat category does not correspond to our abstraction level and is therefore beyond our scope.

▲ 2.5.4 Data In-/Exfiltration

Data in-/exfiltration refers to the misuse of the data flow in the FFF process chain as the carrier of a covert communication channel. It could act as an enabler for theft of technical data. Data In-/Exfiltration has three attack targets: design file (.stl, .3mf, etc), G-code file, and part. The atack method is to encode information on the targets. For instance, the design and G-code files may be used to download malware to a secure network, and the part may be used to exfiltrate environment information with the help of an imperceptible QR code printed in it. This threat category is beyond our scope.

2.6 Klipper

Klipper [25] is a popular open-source FFF printer firmware that distributes the workload in two boards that interact through serial connection: the *interpreter* (KI) and the *controller* (KC). This is in contrast to other popular open-source firmwares, such as Marlin and RepRap, that centralize the workload using a single board. At present, it is unkown what architecture is more performant. According to [25], the main sponsors of Klipper are the firms Bigtreetech [8], Peopoly [40], and Obico [35]. There are unpublished indications that the Chinese FFF printer manufacturers Anycubic, Bambulab, Biqu, Flsun, and Fystech may be using a modified version of Klipper in some of their FFF printers. Recently, many security issues have appeared involving some of these manufacturers [46, 45, 22]. In this section, we overview Klipper's architecture.

Conceptually, Klipper can be abstracted to a model ressembling to Figure 7.

Interpreter

KI is a PCB with a system on chip (SoC, e.g., a Raspberry Pi) that runs a Linux based operating system (e.g., Raspberry Pi OS) and a Python daemon called Klippy. The main goal of Klippy is to translate G-Code into a lower-level encoding that we will refer to as K-Code, and send the output to KC. Morally, K-Code encodes any elementary task that may be carried out by KC in order to read from or write to the pins of an actuator or sensor of the FFF printer. Klippy depends on a configuration file that permits setting control parameters for the actuators and sensors, including speed ranges, temperature ranges, or pins of the KC board; loading non-standard G-codes; or activating advanced functionalities of the kinematics. We remark that, in order to improve its performance, Klippy uses the Python module `cffi` to call functions from a shared C libary at those points of the execution that require many scientific computations. KI is usually set up to additionally run an Application Programming Interface (API) that exposes Klippy to the LAN (e.g., Moonraker), and a web service that permits to interact remotely with Klippy (e.g., Fluidd). The most common APIs are based on WebSocket, HTTP, MQQT and JSON-RPC protocols. Klippy has its own native API (the module `webhooks`).

Controller

KC is a PCB with an ARM microcontroller unit (MCU) that runs a C executable `klipper.bin`. During the build of `klipper.bin`, the users selects the architecture of the MCU (e.g., STM32) and the serial communication protocol between KI and KC (e.g., USB, UART or, CAN). The executable is usually downloaded on KC using a SD Card. The main goal of KC is to decode and execute the K-codes sent by KI in real time, without doing any additional computations. KC controls the stepper motor drivers, the thermistors, the thermocouples, and any other relevant actuators and sensors of the FFF printer. The communication between KI and KC is asyncrhonous, i.e. Klippy does not wait for a confirmation of KC to send the next K-code. KI sends K-codes to KC continously, and KC queues them based on priority before its execution.

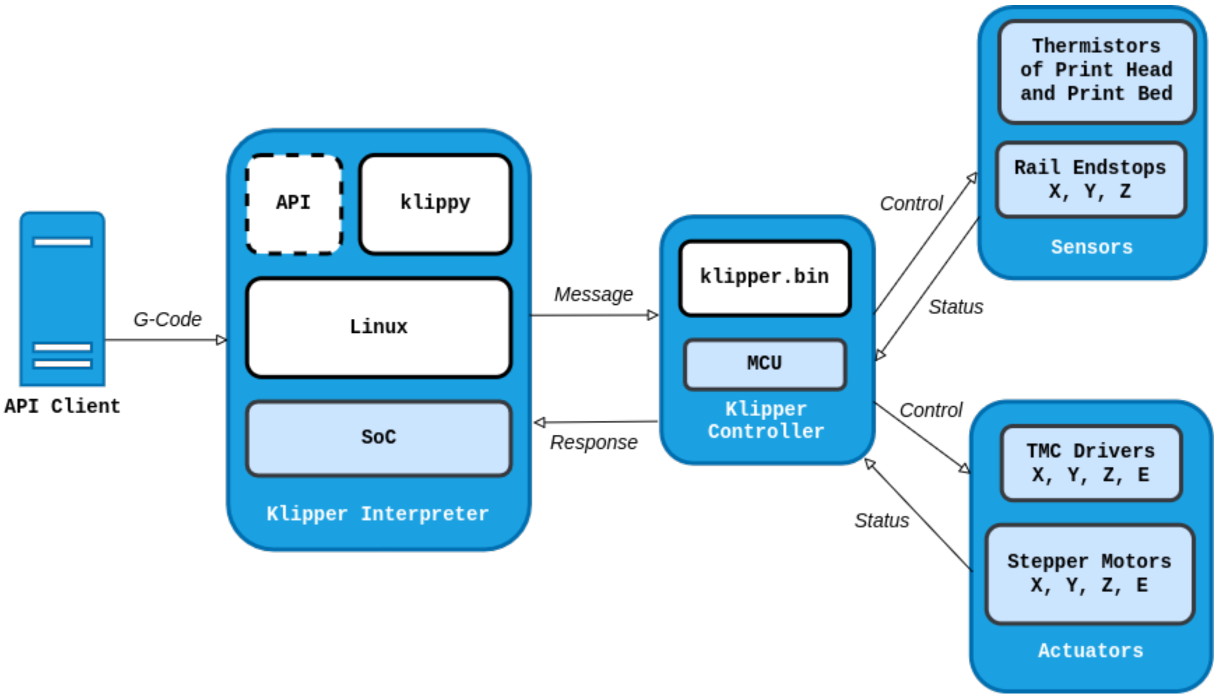

Figure 7: Schematic of Klipper

2.7 Research Goal

We intend to develop an attack in which an adversary sniffs a set of K-codes that are generated by KI in Klipper after interpreting a given G-Code file. The adversary wants to understand to what extent K-Code reflects G-Code, in order to reconstruct the G-Code file and theft information about the part, the FFF process parameters or the printing environment.

♦ 3 State of the Art

Many authors have surveyed and threat modeled the field of distributed AM security in general [62, 63, 41, 29, 64, 50, 51]. Other authors have chosen to focus exclusively on distributed FFF security, including contributions such as a threat model for the FFF process chain [54], an attack tree on FFF printer firmware [42], and a systematic analysis of malicious G-codes [49]. In this section, we present a state of the art review of published attacks exploiting FFF printer vulnerabilities and corresponding defensive techniques. The identified attacks exploit either a printer firmware vulnerability (section 3.1) or a side channel (section 3.2). We were unable to identify fault-injection attacks or attacks based on a micro-architecture vulnerability. This is reasonable, since the field of AM security is relatively young.

■ 3.1 Attacks Based on a Firmware Vulnerability

From the design phase to the deployment of the FFF printer firmware into production, there are numerous stages where security vulnerabilities can emerge. The firmware is first written by humans in the form of source code, which may contain bugs or algorithmic errors. This source code is then compiled into binary code, understandable by the board of the FFF printer, potentially introducing vulnerabilities that only become apparent during execution. The firmware requires to set various printer configurations and network connections, which, if poorly implemented, can create security flaws. The firmware needs to be updated at some point, which brings more risks to the table. Additionally, the FFF printer is exposed to the physical access of a malicious insider and vulnerable to the many existing attacks of IoT devices [11, 31]. A firmware vulnerability means the possibility of compromising security properties through the execution of the binary program and its interactions with its environment. The most common studied security properties are confidentiality, integrity, and availability. We enumerate some published attacks exploiting firmware vulnerabilities of 3D printers.

In 2013, C. Xiao [59] did one of the earliest works highlighting the cybersecurity risks inherent in AM. The author demonstrates some proof-of-concept (PoC) attacks targeting the FFF printer firmware to emphasize the importance of securing the entire FFF process chain. For instance, in one PoC the author modified the Sprinter firmware of a desktop RepRap Prusa printer to alter the temperature feedback loop, leading the printer to believe that the print head temperature was double the real temperature. In particular, the author manipulated the implementation of the G-code command M109, which sets the print head temperature and waits until it reach.

In 2016, S. Moore et al. [32] performed static and dynamic code analyses of Cura 3D, ReplicatorG, Repetier-Host and Marlin firmware that revealed several vulnerabilities in their code bases. These included buffer overflows and unencrypted host-printer communications. The security implications of these vulnerabilities include theft of technical data and part sabotage. The authors also noted that weaknesses in the G-code structure provide opportunities for printed part manipulation.

In 2017, S. Moore et al. [33] investigated the malicious modifications to FFF printer firmware, particularly focusing on Printrbot's branch of Marlin firmware. The authors developed and implemented malicious code that could be triggered by specific printer commands sent from a desktop application, enabling the firmware to ignore legitimate print jobs, substitute them with malicious print commands, and manipulate extrusion feedrates. Their work provides an initial assessment of how such firmware attacks can alter the physical properties and quality of parts, revealing a critical vulnerability in FFF. This research highlights the potential for firmware-level sabotage to degradate parts, and lays foundational groundwork for future studies on malicious activities targeting the FFF printer.

In 2020, M. McCormack et al. [30] present a comprehensive study of security vulnerabilities in networked 3D printers, analyzing 13 different devices and 5 real-world manufacturing network deployments using their tool, C3PO. The research identifies 8 major types of vulnerabilities, including susceptibility to denial-of-service (DoS) attacks, lack of encryption for data in transit, unauthenticated command execution, and exposure to known exploits like WannaCry [58]. Many printers allowed attackers to spoof devices, inject malicious commands, or crash the system, sometimes requiring a power cycle to recover. The study also found that some FFF printers were deployed on publicly accessible networks, increasing exposure to remote attacks. Additionally, the presence of numerous embedded devices (e.g., IoT cameras) on the same networks created multiple attack vectors. The authors emphasize that these security flaws exist across both low-cost desktop and industrial-grade printers, underscoring the urgent need for improved network security, authentication, and encryption measures in printing environments to protect intellectual property and ensure production integrity.

In 2022, H. Pearce et al. [38] proposed an attack that uses a stealthy bootloader Trojan, FLAW3D, targeting AVR-based Marlin-compatible FFF printers to sabotage parts. The attack leverages bootloader control over the initial Marlin installation, replacing the bootloader with FLAW3D. This bootloader Trojan is able to scan for and modify byte patterns in the Marlin executable, triggering G-code manipulations that reduces the extrusion feedrate and reorders G-code commands. The authors were able to degrade the mechanical strength of parts by up to 50%. FLAW3D operates within tight memory constraints (under 1.7 KB), evades detection by programming tools, and exploits inherent vulnerabilities in inexpensive boards commonly used in desktop FFF printers.

In 2024, H. Alkofahi et al. [3] perform a black-box analysis to the toolpath file format and communication protocol of an industrial *Stratasys Dimension Elite* FFF printer, priced $ 31,000. This toolpath file format is a proprietary version of G-Code format. Then, they study adversary-in-the-middle attacks at the level of the LAN connection of the FFF printer. They propose a sniffing attack, a replacement attack and a sabotage attack that target the confidentiality and integrity of the tool-path file data exposed in the communication channel between PC controller and 3D printer.

In 2024, Rais et al. [42] systematically examined FFF printer firmware attacks. The authors propose an attack tree centered on attack goals and introduce new nine distinct printer firmware attacks based on firmware tempering. These attacks were demonstrated on standard FFF printers compatible with Marlin firmware and evaluated through both destructive and non-destructive tests, including tensile strength and air quality assessments. Beyond firmware, the study assesses a total of 48 attacks spanning the design, slicing, and printing stages, highlighting printer firmware tampering as a critical and stealthy attack method. To quantify attack practicality, the authors introduce an Attack Feasibility Index (AFI) that ranks attacks by their likelihood at different stages of the printing process. The work underscores the risks posed by malicious firmware in complex, globalized supply chains—such as stealthy malware infections akin to Stuxnet [57]—that could sabotage large-scale printer farms. Ultimately, the study calls for intensified cybersecurity research and targeted defensive techniques to protect the FFF process chain, especially focusing on firmware security to safeguard the integrity of the parts.

▲ 3.1.1 Defensive Techniques

Defensive techniques for preventing the theft of technical data are largely identical to those used in security of IoT devices [11, 31, 65]: remote authentication, secure boot, access control, encryption, file hashes, etc. Some work has been done to adapt security of IoT devices to FFF printers. Several companies addressing the distributed and outsourced manufacturing market offer commercial options that implement DRM, encryption, or blockchain technology [23, 26, 10, 28, 56, 55] to provide "intellectual property protection, manufacturing repeatability, and traceability" and unlock the potential of distributed manufacturing. In a recent patent, Z. Oligschlager et al. [36] propose to use cryptographic keys unique to a FFF printer for end-to-end encryption designs.

Defensive techniques against sabotage encompass both prevention and detection. Prevention is largely achieved through IoT security measures measures [31] and good security policy; FFF security researchers have yet to propose any FFF-specific measures [39, Section 19.3.2]. Several methods for detecting sabotage attacks against FFF printers based on side channel data have been proposed and experimentally evaluated. Many of these rely on comparison against the side channel readings of a known-good FFF building process. This can be done by generating and comparing signatures [6, 18, 44]. A subset of publications have instead reconstructed the design (into an approximative G-code file or toolpath file) and compared the result against the original model [1, 13, 16]. In either case, this approach has the advantage of being air-gapped from the FFF printers itself and non-intrusive. A more comprehensive approach would involve the development of an anomaly detection system that analyzes both physical-operational side channel data such as sound, electric current and mangetic fields and digital domain data such as network traffic and application logs of the FFF printer [42].

■ 3.2 Passive Physical Attacks

Physical attacks exploit hardware characteristics to breach the security properties of a system. In this context, a correct implementation of a secure protocol is no longer necessarily secure. These attacks require physical access to the targeted device. This is the case for a

FFF printer. *Passive physical attacks*, also known as *side-channel attacks*, are a passive attack method based on observing measurable physical quantities, such as time, power consumption, electromagnetic emissions, sound or temperature during the operation of the target system. These measurable quantities vary depending on the instructions executed and the data being processed. We enumerate some published side-channel attacks measuring physical quantities observed in FFF printers. All the publications concern the theft of tecnical data of either the part geometry or the FFF process parameters.

Sound

In 2016, M. Al Faruque et al. [1] developed the first side-channel attack in AM using acoustic emanations from the motors of a FFF 3D Printer. The authors identified that motor noise could be correlated with operating speed, direction, and the axis of movement. Using a machine-learning approach, they reconstructed motor movements for the printing of a basic polygon. They achieved 78.35 % accuracy of axis prediction, on average, and had a 17.82 % error in movement distance estimation. Shortly afterwards, Hojjati et al. [21] and Song et al. [53] presented machine-learning-based reconstructions using the acoustic side-channel, relying on smartphone audio sensors, and improved the results.

Temperature

In 2016, S. Chhetri et al. [14] explore how thermal emissions from FFF printers can be exploited as a side-channel to reconstruct the G-code file during the FFF build process. The authors develop a novel attack model that uses thermal video recordings combined with image processing, signal processing, machine learning, and context-based post-processing to infer the movements of the print head (X,Y), extruder (E), and print-bed (Z). and translate these into reconstructed G-codes. Their experiments, conducted on a Printrbot 3D printer with an infrared camera, demonstrate the feasibility of partially reconstructing printed objects solely from thermal data.

Mobile Phone Sensors: Electromagnetic Emissions, Sound, Video, Vibrations

In 2018, Y. Gao et al. [16] made significant strides in reconstruction accuracy and metrics with their work on process monitoring. Using a multi-sensor suite of accelerometer, magnetometer, and camera, the authors reconstruct the FFF process parameters involving properties of infill toolpath, print speed, layer thickness, and fan speed. Each of the parameters required its own reconstruction approach, and the authors deploy different metrics based on their own judgement of appropriateness to validate their results. The most noteworthy is the use of the Hausdorff Distance, a distance measure that reports the deviation between two sampled curves, which the authors applied on a per-layer basis. Using their method, they propose a

PoC defensive technique that could be deployed on smartphones to protect agains FFF process parameters sabotage.

Power Consumption

In 2021, J. Gatlin et al. [17] introduce an attack method to reconstruct design files using power side-channel analysis, even when encryption is used. The method involves analyzing the power consumption of the X,Y, and Z stepper motors to reverse engineer the design file. By measuring the power signals from the stepper motors, the adversary can create a point cloud representation of the printed part, bypassing any encryption-based protection and other DRM-based protections. The point cloud does not only describes the exterior surface of the object, but also the interior points corresponding to infill, internal voids, support material, etc. The reconstructions that they produced to validate of their results are moderately distorted, but remain recognizible, and the errors could be corrected by a human engineer to achieve the original quality for simple objects. One of the challenges is to quantify the errors and determine its meaningfulness. They use axis recongition, average distance error, and Hausdorff distance. They acknowledge that it is difficult to evaluate at first sight if good estimations of these distance-based metrics correspond to the printability of a part. Their reconstruction does not bear in mind that the motors X and Y act at continuosly, i.e. , at the same time, and not discretely, i.e. one after another. This creates a drift that could impact on the printability of the part and cause a failure. One limitation is the difficulty from passing from the point cloud to the volume facet of the part. The nozzle is slightly above form the extruder position and the source material filament has a non-negligeable diameter. Another limitation is that their approach cannot distinguish between body and support material.

▲ 3.2.1 Defensive Techniques

According to M. Yampolskiy et al. [39, Section 19.3.1], defending against side-channel attacks is a difficult, FFF-specific problem. There have been attempts at technical defenses that reduce the amount of information carried by the side channels of a FFF printer [2], making reconstruction of the technical data more challenging. However, the state of the art in side channel reconstruction is continuing to advance in and outside of FFF. The degree to which information leakage can be reduced in the face of new techniques is not known.

The attack publication of J. Gatlin et al. [17] suggests that the primary defense against side channel theft must be policy-based and not encryption-based. This includes limiting physical access to FFF printers and making deliberate decisions about whether sensitive designs can be outsourced. In addition, no technical means can mitigate the threat of 3D scanning, which directly captures geometric technical data. Together, these indicate that technical data theft will be a permanent risk factor for FFF that can be reduced but not entirely eliminated.

♦ 4 Data-Flow Analysis in Klipper

The goal of this section is to describe the data flow of a G1 G-code command. In subsection 4.1 we will give an overview of the data flow. In subsection 4.2 we describe Klippy's initialization. Finally, in subsection 4.3 we describe the bells and whistles of the data flow.

■ 4.1 Data-Flow Overview

Recall that Klipper architecture (Figure 8) consists on two PCBs that we named `Klipper Intepreter` (KI) and `Klippper Controller` (KC). Both components communicate using a bus protocol that could be USB, UART, or CAN.

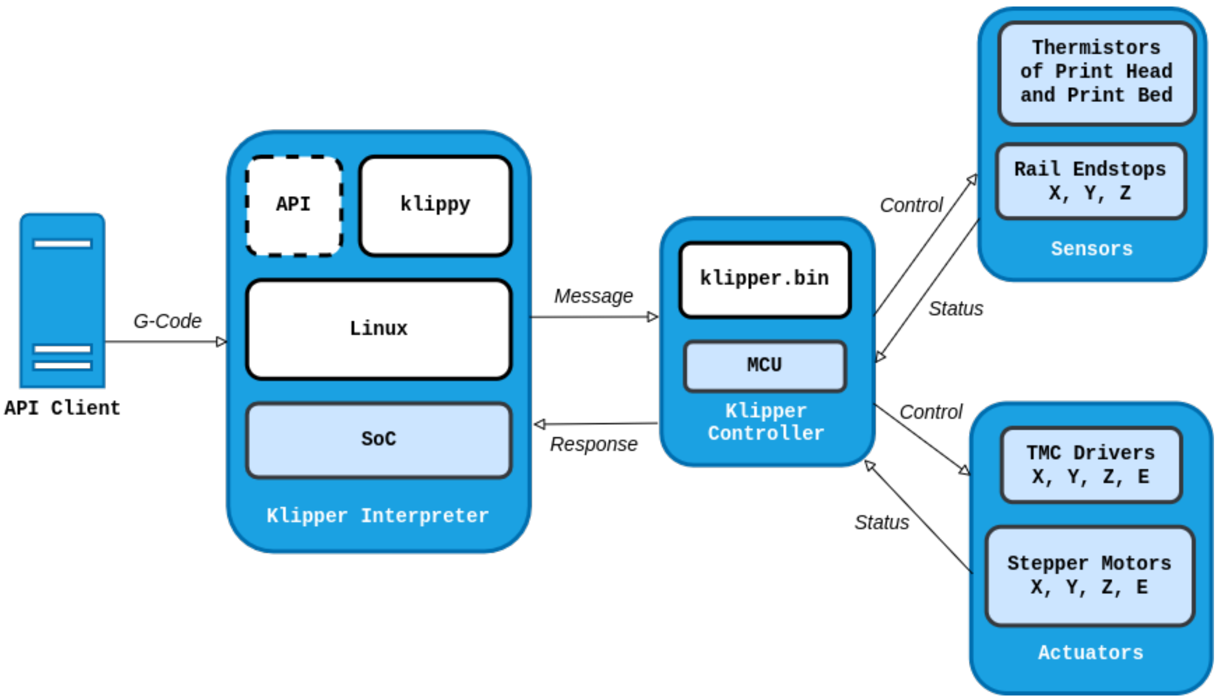

Figure 8: `Klipper` architecture

KI

`KI` is a PCB with a system on chip (SoC) that runs a Linux based operating system and a daemon based on a python script `klippy.py`. The script depends on several main and extra python modules contained in the folders `klippy/` and `klippy/extras/`. We remark that, in order to improve its general performance, one of the modules (`cffi`) permits `klippy.py` to call functions from a shared C libary (`c_helper.so`) at specific points of its execution. The script `klippy.py` also depends on a configuration file `printer.cfg` that permits to configure control parameters for the actuators and sensors, including speeds, temperatures, but also pins of the `klipper-mcu` board, G-code macros, and some extra modules. `klippy.py` is usually set up as a daemon that exposes an Application Programming Interface (API) to a network and waits for the remote transmission of a G-Code file. The most common APIs are based on WebSocket, HTTP, MQQT and JSON-RPC protocols. `klippy.py` has its own native API (`webhooks.py`).

We now describe the execution of `klippy.py`. At initialization, `klippy.py` enters an infinite loop and starts a scheduler based on a reactor design pattern (`reactor.py`) that will handle the tasks/events defined in the different modules. It subsequently creates both a background logger thread (`queuelogger.py`) that writes debug messages to the log, and a `Printer` object thread (`/klippy.py`) that first reads the configuration file printer.cfg (`configfile.py`) and then launches the main modules and extra loaded modules (`webhooks.py`, `gcode.py`, `pins.py`, `mcu.py`, `toolhead.py` `mcu.py`, etc). In particular `mcu.py` and `pins.py` load the module `serialhdl.py` and initialize the communication with `KC`. The `Printer` object thread consists on three subthreads. We will focus on the data flow of a G1 G-code command:

(S1) The main subthread (`gcode.py`, `toolhead.py`, `trapq.c`, `itersolve.c`, `steocompress.c`, `stepper.c`) handles the G-code file received at the API and processes the G-code commands of type `move` (G0, G1). These are the commands that move the stepper motors of the FFF printer corresponding to the axis X, Y ,Z , E. During this subthread, (i) the incoming G-code commands are first transformed into internal calls (`gcode.py`). Next, (ii) the internal calls corresponding to G-code commands of type move are transformed into step times for the stepper motors (`toolhead.py`, `itersolve.c`) using the appropriate kinematics of the FFF printer (`kinematics/cartesian.py`, `kinematics/delta.py, etc`). During this process, the movements are transformed to a trapezoidal profile that divides trajectories in acceleration, cruise and deceleration phases and some information about the feedrate is lost when computing the speed at the junction phases and when applying the input shaper to reduce resonance. Finally, (iii) these step pulse times are compressed to `KC` command *descriptions* (K-codes) of type `queue_step` (`stepcompress.c`). The rest of G-code commands contained in the G-Code file are handled differently at (ii) and compresed to a specific type of K-code.

(S2) The second thread (`serialqueue.c`) handles low-level I/O communication with `KC` through a serial port in terms of the low-level communication protocol (USB, USART, CAN, etc). In particular, it defines the transport protocol to encode/decode, frame/unframe and send/receive the messages containing the K-codes and reception acknowledgements. This handling can be thought as a *remote procedure call* (RPC) mechanism.

(S3) The third thread is inside `serialhdl.py` and is used to process response messages of `KC`. In particular, this thread reads time and status parameters from actuators and permits to update them according to the policy fixed by `KI`. This policy is both in the configuration file `printer.cfg` but also hard-coded in some modules (e.g, `toolhead.py`).

KC

KC is a PCB with a microcontroller unit (MCU) that runs a C program `klipper.bin`. During the build of `klipper.bin`, one selects the architecture of the MCU (STM32, atmega, etc) and the communication interface between host (USB, UART, CAN, etc), among other things.

We now describe the execution of `klipper.bin`. The binary `klipper.bin` has a scheduler (`sched.c`) that handles three or four families of declared tasks. Among these families we find the command tasks. The scheduler of the binary `klipper.bin` handles the incoming messages of KI by decapsulating the messages, extracting the K-code and executing a command task that corresponds to an id that appears in a hardcoded dictionary and depends on the given description (`command.c`). This dictionary is shared with KI at the initialization of the communication. Next, the command task is run and some low-level code corresponding to a hardware component of the MCU (`spi.c`, `i2c.c`, etc) is executed to control the associated actuator or sensor, creating electrical signals that are transformed into physical actions. We included the most common actuators and sensors in Figure 7. The actuators and sensors return some status messages that are subsequently encapsulated in acknowledgement response messages and sent back to KI.

The descrptions of KI and KC, can be summarized in Figure 9.

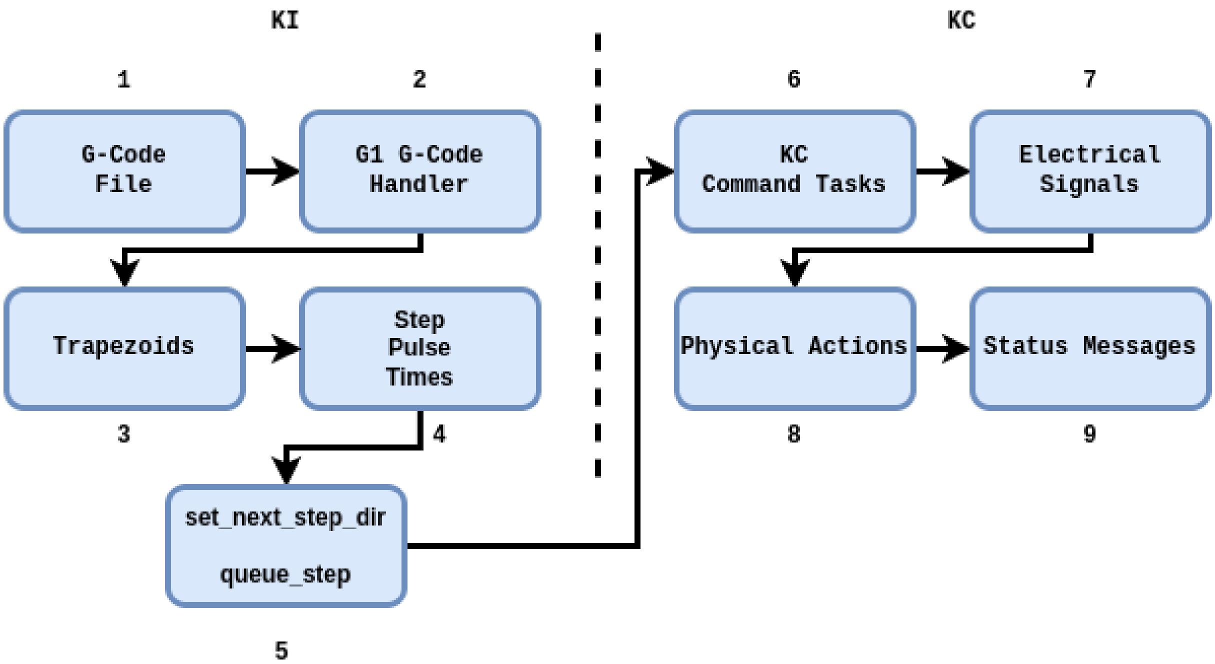

Figure 9: Processing of a G1 G-code command

4.2 Initialization

In this section we describe the call graph of Klipper initialization on the side of KI (Figure 10). We recall that this corresponds to the Python daemon klippy that runs on KI. All the files mentioned in this section are in the folder `klipper/klippy/`.

main

The entry point of klippy is the function `main` in `klippy.py`.

```
def main():
    usage = "%prog [options] <config file>"
    opts = optparse.OptionParser(usage)
    opts.add_option("-i", "--debuginput", dest="debuginput",
                    help="read commands from file instead of from tty port")
    opts.add_option("-I", "--input-tty", dest="inputtty",
                    default='/tmp/printer',
                    help="input tty name (default is /tmp/printer)")
    opts.add_option("-a", "--api-server", dest="apiserver",
                    help="api server unix domain socket filename")
    opts.add_option("-l", "--logfile", dest="logfile",
                    help="write log to file instead of stderr")
    opts.add_option("-v", action="store_true", dest="verbose",
                    help="enable debug messages")
    opts.add_option("-o", "--debugoutput", dest="debugoutput",
                    help="write output to file instead of to serial port")
    opts.add_option("-d", "--dictionary", dest="dictionary", type="string",
                    action="callback", callback=arg_dictionary,
                    help="file to read for mcu protocol dictionary")
    opts.add_option("--import-test", action="store_true",
                    help="perform an import module test")
--
snip
--
```

Listing 1: klippy.py

This part of `main` sets up and parses command-line options for the program using Python's `optparse` module. It defines how users can interact with the program when launching it from the command line. Each `add_option` call specifies a possible flag, its short and long forms (like -i and –debuginput), where its value should be stored (dest), and a help message describing its purpose. For example, –debuginput allows reading commands from a file instead of the default TTY port, –input-tty sets the TTY device name (defaulting to `/tmp/printer`), –api-server specifies a Unix socket for API communication, and –logfile lets the user direct logs to a file. The –verbose flag enables debug-level logging, while –debugoutput and –dictionary allow advanced debugging and MCU protocol customization. The –import-test flag triggers a module import test. Finally, `opts.parse_args` processes the actual command-line input, separating options into the options object and positional arguments into args, preparing them for use later in the program.

```
def main():
--
snip
--
    options, args = opts.parse_args()
    if options.import_test:
        import_test()
    if len(args) != 1:
        opts.error("Incorrect number of arguments")
    start_args = {'config_file': args[0], 'apiserver': options.apiserver,
                  'start_reason': 'startup'}
    debuglevel = logging.INFO
    if options.verbose:
        debuglevel = logging.DEBUG
    if options.debuginput:
        start_args['debuginput'] = options.debuginput
        debuginput = open(options.debuginput, 'rb')
        start_args['gcode_fd'] = debuginput.fileno()
    else:
        start_args['gcode_fd'] = util.create_pty(options.inputtty)
    if options.debugoutput:
        start_args['debugoutput'] = options.debugoutput
        start_args.update(options.dictionary)
    bglogger = None
    if options.logfile:
        start_args['log_file'] = options.logfile
        bglogger = queuelogger.setup_bg_logging(options.logfile, debuglevel)
    else:
        logging.getLogger().setLevel(debuglevel)
--
snip
--
```

Listing 2: klippy.py

This section of `main` processes the parsed command-line options and prepares the environment for Klippy's execution. It first checks if the –import-test flag is set, running a diagnostic import test if so, and then validates that exactly one positional argument (the configuration file) is provided, enforcing correct usage. Next, it assembles a start_args dictionary to hold key startup parameters such as the config file path, API server socket, and the startup reason. The code then establishes the logging level based on the verbosity option, and configures input handling: if a debug input file is specified, it opens the file and records its file descriptor for G-code commands; otherwise, it creates a pseudo-terminal for printer communication. If a debug output file is specified, it is added to the startup arguments along with any MCU protocol dictionary options. Finally, the function configures logging—either setting up background logging to a file if specified by calling `setup_bg_logging`, or adjusting the root logger's level for console output—ensuring that all diagnostic and operational information will be properly recorded as the printer process runs. This careful setup ensures that Klippy starts with the correct configuration, input/output channels, and logging tailored to the user's needs.

```python
def main():
--
snip
--
    logging.info("Starting Klippy...")
    git_info = util.get_git_version()
    git_vers = git_info["version"]
    extra_files = [fname for code, fname in git_info["file_status"]
                   if (code in ('??', '!!') and fname.endswith('.py')
                       and (fname.startswith('klippy/kinematics/')
                            or fname.startswith('klippy/extras/')))]
    modified_files = [fname for code, fname in git_info["file_status"]
                      if code == 'M']
    extra_git_desc = ""
    if extra_files:
        if not git_vers.endswith('-dirty'):
            git_vers = git_vers + '-dirty'
        if len(extra_files) > 10:
            extra_files[10:] = ["(+%d files)" % (len(extra_files) - 10,)]
        extra_git_desc += "\nUntracked files: %s" % (', '.join(extra_files),)
    if modified_files:
        if len(modified_files) > 10:
            modified_files[10:] = ["(+%d files)" % (len(modified_files) - 10,)]
        extra_git_desc += "\nModified files: %s" % (', '.join(modified_files),)
```

```
    extra_git_desc += "\nBranch: %s" % (git_info["branch"])
    extra_git_desc += "\nRemote: %s" % (git_info["remote"])
    extra_git_desc += "\nTracked URL: %s" % (git_info["url"])
    start_args['software_version'] = git_vers
    start_args['cpu_info'] = util.get_cpu_info()
    if bglogger is not None:
        versions = "\n".join([
            "Args: %s" % (sys.argv,),
            "Git version: %s%s" % (repr(start_args['software_version']),
                                   extra_git_desc),
            "CPU: %s" % (start_args['cpu_info'],),
            "Python: %s" % (repr(sys.version),)])
        logging.info(versions)
    elif not options.debugoutput:
        logging.warning("No log file specified!"
                        " Severe timing issues may result!")
--
snip
--
```

Listing 3: klippy.py

This section of `main` gathers and logs detailed version and environment information before starting the printer process. It retrieves Git data—including the current version, branch, remote, and any untracked or modified Python files—to help track changes and ensure reproducibility. This information, along with CPU and Python details, is added to the startup arguments and logged for diagnostics if background logging is enabled. If no log file is specified, a warning is issued about potential timing issues, emphasizing the importance of proper logging for troubleshooting and stable operation.

```
def main():
--
snip
--
    gc.disable()
--
snip
--
```

Listing 4: klippy.py

The line `gc.disable()` turns off Python's automatic garbage collection to prevent unpredictable pauses and ensure consistent timing, which is important for real-time firmware like Klippy. Memory is still managed by reference counting, but objects in reference cycles won't be automatically cleaned up. This approach should be used with care for a safe memory management.

```python
def main():
--
snip
--
    # Start Printer() class
    while 1:
        if bglogger is not None:
            bglogger.clear_rollover_info()
            bglogger.set_rollover_info('versions', versions)
        gc.collect()
        main_reactor = reactor.Reactor(gc_checking=True)
        printer = Printer(main_reactor, bglogger, start_args)
        res = printer.run()
        if res in ['exit', 'error_exit']:
            break
        time.sleep(1.)
        main_reactor.finalize()
        main_reactor = printer = None
        logging.info("Restarting printer")
        start_args['start_reason'] = res

    if bglogger is not None:
        bglogger.stop()

    if res == 'error_exit':
        sys.exit(-1)
```

Listing 5: klippy.py

This final part of `main` is responsible for launching and managing the main printer control loop, as well as handling restarts and cleanup. It enters an infinite loop where, on each iteration, it prepares the logging system (if background logging is enabled), manually triggers garbage collection, and then creates instances of the `Reactor` and `Printer` classes, passing in all the prepared startup arguments to `printer`.

`Reactor` is a concrete implementation of the *reactor design pattern*, which acts as a central event loop that continuously waits for events like timers or I/O readiness, demultiplexes them using Python's `select` module, and dispatches each to the appropriate registered handler. This allows Klippy to efficiently handle many concurrent events in a single thread, enabling asynchronous and non-blocking operation for tasks like hardware communication and scheduling.

`Printer` is the core class responsible for managing the overall state and operation of the FFF printer in Kl. The `Printer::run()` method starts the printer's main operation and blocks until the printer needs to shut down or restart. If `Printer::run` returns '*exit*' or '*error_exit*', the loop breaks, ending the program. Otherwise, the code waits one second, finalizes and cleans up the reactor and printer objects, logs a restart message, and updates the start reason for the next run.

After exiting the loop, the code stops the background logger if it was used, and if the exit was due to an error, it terminates the process with a nonzero exit code.

Printer `Printer` acts as the central manager for the printer's state, configuration, and core operations.

```python

class Printer:
    config_error = configfile.error
    command_error = gcode.CommandError
    def __init__(self, main_reactor, bglogger, start_args):
        self.bglogger = bglogger
        self.start_args = start_args
        self.reactor = main_reactor
        self.reactor.register_callback(self._connect)
        self.state_message = message_startup
        self.in_shutdown_state = False
        self.run_result = None
        self.event_handlers = {}
        self.objects = collections.OrderedDict()
        # Init printer components that must be setup prior to config
        for m in [gcode, webhooks]:
            m.add_early_printer_objects(self)
```

```
--
snip
--
```

Listing 6: klippy.py

The constructor `Printer::__init__` receives the main event reactor, a background logger, and startup arguments, storing them for use throughout the printer's lifecycle. It registers a connection callback (`Printer::_connect`) with the `Reactor`, which will be triggered as part of the startup sequence. The class also sets up state-tracking variables, including the current status message, shutdown state, and result of the main run loop. It maintains an ordered dictionary (`Printer::objects`) to store all instantiated printer objects (such as hardware interfaces and feature modules), and a dictionary for event handlers, supporting Klippy's event-driven architecture. Before loading the main configuration, it calls the function `add_early_printer_objects` in `gcode.py` and `webhooks.py` to call the constructors of the core classes which handle, the reception, parsing and dispatching of the G-codes (`GCodeIO`, `GCodeDispatch`), as well as the web API permitting to interact with the FFF printer (`WebHooks`, `GCodeHelper`, `QueryStatusHelper`). ensuring these essential components are available early in the startup process. `Printer` provides methods to access startup arguments, the `Reactor`, and the current state, and serves as the main point of reference for other modules to register event handlers or look up other printer objects.

```python
class Printer:
    config_error = configfile.error
    command_error = gcode.CommandError
--
snip
--
    def _connect(self, eventtime):
        try:
            self._read_config()
            self.send_event("klippy:mcu_identify")
            for cb in self.event_handlers.get("klippy:connect", []):
                if self.state_message is not message_startup:
                    return
                cb()
        except (self.config_error, pins.error) as e:
            logging.exception("Config error")
            self._set_state("%s\n%s" % (str(e), message_restart))
```

```
            return
        except msgproto.error as e:
            msg = "Protocol error"
            logging.exception(msg)
            self._set_state(msg)
            self.send_event("klippy:notify_mcu_error", msg, {"error": str(e)})
            util.dump_mcu_build()
            return
        except mcu.error as e:
            msg = "MCU error during connect"
            logging.exception(msg)
            self._set_state(msg)
            self.send_event("klippy:notify_mcu_error", msg, {"error": str(e)})
            util.dump_mcu_build()
            return
        except Exception as e:
            logging.exception("Unhandled exception during connect")
            self._set_state("Internal error during connect: %s\n%s"
                            % (str(e), message_restart,))
            return
        try:
            self._set_state(message_ready)
            for cb in self.event_handlers.get("klippy:ready", []):
                if self.state_message is not message_ready:
                    return
                cb()
        except Exception as e:
            logging.exception("Unhandled exception during ready callback")
            self.invoke_shutdown("Internal error during ready callback: %s"
                                 % (str(e),))
    def run(self):
        systime = time.time()
        monotime = self.reactor.monotonic()
        logging.info("Start printer at %s (%.1f %.1f)",
                     time.asctime(time.localtime(systime)), systime, monotime)
        # Enter main reactor loop
        try:
            self.reactor.run()
        except:
            msg = "Unhandled exception during run"
```

```
            logging.exception(msg)
            # Exception from a reactor callback - try to shutdown
            try:
                self.reactor.register_callback((lambda e:
                                                self.invoke_shutdown(msg)))
                self.reactor.run()
            except:
                logging.exception("Repeat unhandled exception during run")
                # Another exception - try to exit
                self.run_result = "error_exit"
        # Check restart flags
        run_result = self.run_result
        try:
            if run_result == 'firmware_restart':
                self.send_event("klippy:firmware_restart")
            self.send_event("klippy:disconnect")
        except:
            logging.exception("Unhandled exception during post run")
        return run_result
--
snip
--
```

Listing 7: klippy.py

Recall that `main` calls the `Printer::run` method is the main entry point for starting the printer's host-side logic. It initiates the event loop, manages the printer's lifecycle, and coordinates the startup sequence, including configuration loading and component initialization. In particular it calls the `Reactor::run` method, which is responsible for running the event loop itself; it continuously monitors registered file descriptors, timers, and callbacks, dispatching events as they occur to ensure asynchronous operations like G-code reception, hardware communication, and scheduled tasks are handled efficiently. The `Printer::connect` method is invoked as a callback during startup by the `Reactor` and tasked with establishing the printer's operational state: it loads the configuration by calling `Printer::_read_config`, instantiates all required printer components, and sets up communication with the microcontroller.

```
class Printer:
    config_error = configfile.error
    command_error = gcode.CommandError
```

```
--
snip
--
    def load_object(self, config, section, default=configfile.sentinel):
        if section in self.objects:
            return self.objects[section]
        module_parts = section.split()
        module_name = module_parts[0]
        py_name = os.path.join(os.path.dirname(__file__),
                               'extras', module_name + '.py')
        py_dirname = os.path.join(os.path.dirname(__file__),
                                  'extras', module_name, '__init__.py')
        if not os.path.exists(py_name) and not os.path.exists(py_dirname):
            if default is not configfile.sentinel:
                return default
            raise self.config_error("Unable to load module '%s'" % (section,))
        mod = importlib.import_module('extras.' + module_name)
        init_func = 'load_config'
        if len(module_parts) > 1:
            init_func = 'load_config_prefix'
        init_func = getattr(mod, init_func, None)
        if init_func is None:
            if default is not configfile.sentinel:
                return default
            raise self.config_error("Unable to load module '%s'" % (section,))
        self.objects[section] = init_func(config.getsection(section))
        return self.objects[section]
    def _read_config(self):
        self.objects['configfile'] = pconfig = configfile.PrinterConfig(self)
        config = pconfig.read_main_config()
        if self.bglogger is not None:
            pconfig.log_config(config)
        # Create printer components
        for m in [pins, mcu]:
            m.add_printer_objects(config)
        for section_config in config.get_prefix_sections(''):
            self.load_object(config, section_config.get_name(), None)
        for m in [toolhead]:
            m.add_printer_objects(config)
        # Validate that there are no undefined parameters in the config file
```

```
        pconfig.check_unused_options(config)
--
snip
--
```

Listing 8: klippy.py

The `Printer::_read_config` method is responsible for reading and parsing the printer's configuration file (typically `printer.cfg`). Together with `Printer::_load_object`, these methods load the configuration sections, validate them, and prepare the necessary data structures for use by the rest of the firmware. In general, each [section] in `printer.cfg` is in correspondence with a Python module in the folder `extras/`. After reading and parsing the configuration, the method `Printer::_load_object` proceeds to instantiate printer objects. Additionally, by calling `add_printer_objects`, this method loads the printer objects corresponding to the classes `PrinterPins` (`pins.py`), `MCU` (`mcu.py`), `ToolHead` (`toolhead.py`), and `PrinterExtruder` (`kinematics/extruder.py`).

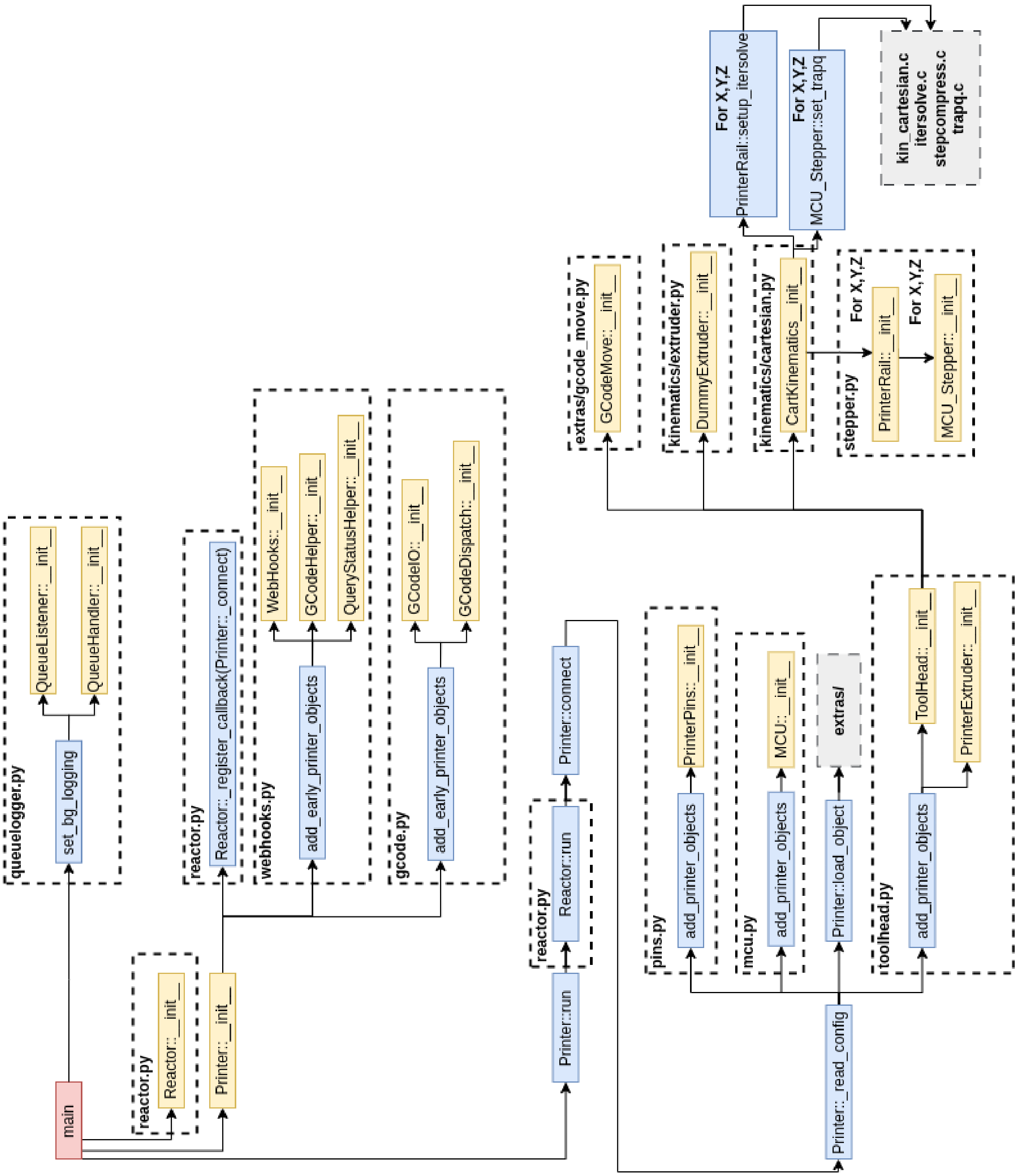

Figure 10: Call Graph of `klippy` initialization.

■ 4.3 Data-Flow in Klippy

In this section we describe Figure 11, Figure 12, Figure 13, Figure 15, and Figure 14.

▲ 4.3.1 G-code Parsing

In this section we begin the description of Figure 11.

GCodeHelper::_handle_script. The data flow for executing a G1 command in Klipper begins with the `GCodeHelper::_handle_script` method when the module `webhooks.py` is configured as the default web API in Klipper, which listens to messages formatted in JSON. This method handles G-code scripts triggered by macros or other scripted events, preparing the G-code commands for execution and forwarding them to the dispatcher `GCodeDispatch::run_script`. If `webhooks.py` is not the default, this step is bypassed.

GCodeDispatch::run_script. Following script handling, `GCodeDispatch::run_script` executes the prepared G-code script. Similar to `GCodeHelper::_handle_script`, this function is called only under the `webhooks.py` default web API setup. It processes the list of G-code commands generated by the script sequentially, dispatching each command to `GCodeDispatch::process_commands`.

GCodeIO::process_data. Alternatively, G-code commands enter the system through `GCodeIO::process_data`. This function processes raw G-code data received from web APIs external to Klipper, such as Moonraker. Unlike the G-code script-based flow in the two previous paragraphs, `GCodeIO::process_data` directly parses and forwards incoming G-code commands to `GCodeDispatch::process_commands`.

GCodeDispatch::_process_commands. The data flow for a G1 command in Klipper begins with the GCodeDispatch::_process_commands function. This function serves as the main entry point for all incoming G-code commands. It parses the G-code stream received from the host or user, identifies each command, and determines the appropriate handler function for execution. When a G1 command is encountered, the system prepares to process it by dispatching it to the specific handler responsible for linear moves.

GCodeMove::cmd_G1. Once the G1 command is recognized, control is passed to the GCodeMove::cmd_G1 handler. This function is responsible for parsing the parameters of the G1 command, such as the target coordinates (X, Y, Z, E) and the feedrate (F). After extracting these values, the handler invokes the move subsystem—typically by calling ToolHead::move—to initiate the process of planning and executing the requested movement.

ToolHead::move. The ToolHead::move function acts as the central planner for coordinated printer motion. Upon receiving the parsed move parameters, it creates a new Move object, which encapsulates all details of the movement, including start and end positions, velocities, and any extrusion information. This function is pivotal in transitioning from G-code parsing to motion planning.

Move::init. The creation of a new move is handled by the Move::__init__ constructor. This function initializes the move object by computing its geometric properties, such as distance, direction, and the required velocities. All relevant parameters, including those necessary for kinematic calculations and extrusion, are stored within the move object for subsequent processing.

CartKinematics::check_move. Once the move object is initialized, the system calls CartKinematics::check_move to validate the move against the printer's kinematic constraints. This function ensures that the move is feasible given the printer's geometry (Cartesian, CoreXY, Delta, etc.) and calculates the necessary stepper movements for each axis. This step is crucial for preventing impossible or unsafe movements.

PrinterExtruder::check_move. In parallel with kinematic validation, PrinterExtruder::check_move examines the extrusion parameters of the move. This function checks that the requested extrusion amount is within safe operational limits and that it matches the length and speed of the move. This step helps prevent issues such as over-extrusion or filament jams.

▲ 4.3.2 Trapezoid Profile Generation

In this section we continue with the description of Figure 11. ToolHead is a class that buffers movement-related G-codes and models each move as a trapezoidal velocity profile, specifying acceleration, cruising speed, and deceleration. It relies on the LookAheadQueue class, which analyzes upcoming Moves and their trapezoidal profiles to adjust speeds at corners by calculating junction speeds and timing to ensure smooth transitions between moves. It flushes moves to the step pulse generator system when enough moves have been buffered. This system is based on the stepper_kinematics class of the c_helper.so that Klippy generates at runtime. stepper_kinematics is allocated by ToolHead at initialization for each motor X,Y,Z, and E and each of them manages its own trapq and a stepcompress classes.

LookAheadQueue::add_move. After validation, the move is added to the lookahead queue using LookAheadQueue::add_move. The lookahead buffer allows Klipper to analyze multiple upcoming moves before they are executed, enabling the planner to optimize acceleration and deceleration across move boundaries. This results in smoother and more efficient printer motion.

Move::calc_junction. Next, Move::calc_junction is called to calculate the motion at the junction between the current move and the previous one. This function determines the optimal speed and acceleration for transitioning from one move to the next, taking into account jerk and cornering limits. Proper junction calculation is key to maintaining print quality and mechanical longevity.

LookAheadQueue::flush. Once lookahead optimization is complete, LookAheadQueue::flush commits the planned moves to the execution queue. This function finalizes the sequence of moves, ensuring that all optimizations are applied and that the moves are ready for execution by the motion subsystem.

Move::set_junction. With the moves finalized, Move::set_junction sets the actual entry and exit speeds and accelerations for each move. These parameters are based on the calculations performed during lookahead and junction optimization, ensuring that each move transitions smoothly to the next. **HERE there may be loss of information**.

ToolHead::process_moves The ToolHead::process_moves function is responsible for processing moves from the execution queue. It initiates the actual execution of each planned move, coordinating with both the extruder and the kinematics subsystems to ensure synchronized and accurate motion.

ToolHead::trapq_append. To manage acceleration and deceleration profiles, ToolHead::trapq_append adds the move to the trapezoidal velocity queue. This scheduling step ensures that each move is executed with the correct acceleration and deceleration, resulting in smooth and controlled movement.

PrinterExtruder::move. During execution, PrinterExtruder::move synchronizes the extruder's operation with the linear movement. This function ensures that the extruder stepper motor dispenses filament at the correct rate, matching the planned movement and maintaining consistent extrusion throughout the print.

ToolHead::_advance_move_time. The ToolHead::_advance_move_time function advances the internal move timing. It updates the system's clock to reflect the progress of the current move, ensuring that timing remains accurate as the printer executes each segment of the planned path.

ToolHead::_advance_flush_time. Similarly, ToolHead::_advance_flush_time is responsible for flushing completed moves and preparing the system for the next move in the queue. This function maintains the flow of execution, ensuring that the printer is always ready to process the next planned action.

▲ 4.3.3 Step Pulse Time Generation

In this section we describe Figure 14, and mention Figure 13 and Figure 12. Klipper uses a C-based iterative solver (itersolve.c, Figure 14) to translate the trapezoidal profiles of queued Moves ainto precise timing for stepper motor pulses, controlling motor speed and acceleration in real-time. This approach is necessary because, for many kinematic systems, there's no closed-form solution for time-to-position mapping. Stepper motors require precisely timed pulses to move smoothly and accurately. For complex kinematics (like CoreXY, Delta, etc.), the relationship between time and step position is non-linear and can't be solved with a simple formula. The iterative solver in itersolve.c "guesses" the time for each step from the positions that where calculated in `kin_cartesian.c`, checks how close it is to the desired position, and refines its guess until it's accurate enough—this is called iterative root finding of the movement equations, which are quadratic functions.

itersolve_generate_steps

```c

// Generate step times for a range of moves on the trapq
int32_t __visible itersolve_generate_steps(struct stepper_kinematics *sk,
                                           double flush_time) {
    double last_flush_time = sk->last_flush_time;
    sk->last_flush_time = flush_time;
    if (!sk->tq)
        return 0;
    trapq_check_sentinels(sk->tq);
    struct move *m = list_first_entry(&sk->tq->moves, struct move, node);
--
snip
--

```

Listing 9: itersolve.c

The function begins by printing debug information and initializing timing variables. It updates the stepper kinematics object's last_flush_time to the new flush_time and retrieves the move queue (trapq). If there is no move queue, it returns immediately. It then checks the integrity of the move queue with trapq_check_sentinels and gets the first move from the queue.

```

--
```

```
snip
--
    while (last_flush_time >= m->print_time + m->move_t)
        m = list_next_entry(m, node);
--
snip
--
```

Listing 10: itersolve.c

The code advances the move pointer to the first move that has not already been flushed. It does this by iterating through the move list until it finds a move whose end time (m->print_time + m->move_t) is after the last flush time. This ensures that step generation starts at the correct point in the move queue.

```
--
snip
--
    double force_steps_time = sk->last_move_time + sk->gen_steps_post_active;
    int skip_count = 0;
    for (;;) {
        double move_start = m->print_time, move_end = move_start + m->move_t;
        if (check_active(sk, m)) {
            if (skip_count && sk->gen_steps_pre_active) {
                // Must generate steps leading up to stepper
                // activity
                double abs_start = move_start - sk->gen_steps_pre_active;
                if (abs_start < last_flush_time)
                    abs_start = last_flush_time;
                if (abs_start < force_steps_time)
                    abs_start = force_steps_time;
                struct move *pm = list_prev_entry(m, node);
                while (--skip_count && pm->print_time > abs_start)
                    pm = list_prev_entry(pm, node);
                do {
                    int32_t ret = itersolve_gen_steps_range(sk, pm, abs_start,
                                                            flush_time);
                    if (ret)
                        return ret;
```

```
                    pm = list_next_entry(pm, node);
                } while (pm != m);
            }
--
snip
--
```

Listing 11: itersolve.c

This is the core loop of the function. For each move, it determines whether the move is "active" for the stepper using check_active. If the move is active and there were previously skipped moves (tracked by skip_count), and if gen_steps_pre_active is set, it generates steps for the time window leading up to the active move. This is done by walking backward through the move list and generating steps for each skipped move within the pre-active window, ensuring that any required pre-motion step generation (for things like optimizing resonance in kin_input_shaper.c, where **there may be loss of information**) is handled.

```
--
snip
--
    // Generate steps for this move
            int32_t ret =
                itersolve_gen_steps_range(sk, m, last_flush_time, flush_time);
            if (ret)
                return ret;
            if (move_end >= flush_time) {
                sk->last_move_time = flush_time;
                return 0;
            }
            skip_count = 0;
            sk->last_move_time = move_end;
            force_steps_time = sk->last_move_time + sk->gen_steps_post_active;
        } else {
--
snip
--
```

Listing 12: itersolve.c

If the move is active, the function generates steps for the current move from the last flush time up to the current flush time by calling itersolve_gen_steps_range. If the move ends after the flush time, it updates the last move time and returns. Otherwise, it updates the last move time to the move's end and recalculates the force_steps_time for post-active step generation. The loop then continues to the next move.

```
--
snip
--
```

Listing 13

```
--
snip
--
        if (move_start < force_steps_time) {
            // Must generates steps just past stepper activity
            double abs_end = force_steps_time;
            if (abs_end > flush_time)
                abs_end = flush_time;
            int32_t ret = itersolve_gen_steps_range(sk, m, last_flush_time, abs_end);
            if (ret)
                return ret;
            skip_count = 1;
        } else {
            // This move doesnt impact this stepper - skip it
            skip_count++;
        }
        if (flush_time + sk->gen_steps_pre_active <= move_end)
            return 0;
    }
--
snip
--
```

Listing 14: itersolve.c

If the move is not active, but its start time is less than force_steps_time (which means it falls within the post-active window), the function generates steps for the move up to the end of the post-active window. It sets skip_count to indicate that moves have been skipped. If the move is not within the post-active window, it simply increments the skip count and moves to the next move. If the flush time plus the pre-active window is less than or equal to the move's end time, it returns, as no further step generation is needed for now.

```
--
snip
--
    m = list_next_entry(m, node);
}
```

Listing 15: itersolve.c

At the end of the loop, the function advances to the next move in the list and repeats the process, continuing to generate steps as needed for both active and inactive moves, handling all necessary pre- and post-active step windows.

iterolve_gen_steps_range

```
sk_calc_callback calc_position_cb = sk->calc_position_cb;
double half_step = .5 * sk->step_dist;
double start = abs_start - m->print_time, end = abs_end - m->print_time;
if (start < 0.)
    start = 0.;
if (end > m->move_t)
    end = m->move_t;
struct timepos old_guess = {start, sk->commanded_pos}, guess = old_guess;
int sdir = stepcompress_get_step_dir(sk->sc);
int is_dir_change = 0, have_bracket = 0, check_oscillate = 0;
double target = sk->commanded_pos + (sdir ? half_step : -half_step);
double last_time = start, low_time = start,
       high_time = start + SEEK_TIME_RESET;
if (high_time > end)
    high_time = end;
```

```
--
snip
--
}
```

Listing 16: itersolve.c

This block sets up all the variables needed for step generation. It calculates the time window for step generation relative to the move's start, initializes the guess structure for the iterative search, determines the step direction, and sets the initial target position for the next step. It also establishes the search bounds for the secant/bisection method used to find the exact timing of each step.

```
--
snip
--
    double guess_dist = guess.position - target;
    double og_dist = old_guess.position - target;
    double next_time =
        ((old_guess.time * guess_dist - guess.time * og_dist) /
         (guess_dist - og_dist));

    if (!(next_time > low_time && next_time < high_time)) { // or NaN
        if (have_bracket) {
            next_time = (low_time + high_time) * .5;
            check_oscillate = 0;
        } else if (guess.time >= end) {
            break;
        } else {
            next_time = high_time;
            high_time = 2. * high_time - last_time;
            if (high_time > end)
                high_time = end;
        }
    }
    old_guess = guess;
    guess.time = next_time;
    guess.position = calc_position_cb(sk, m, next_time);
    guess_dist = guess.position - target;
--
```

```
snip
--
```

Listing 17: itersolve.c

This block is the heart of the algorithm. It uses the secant method (a root-finding algorithm) to guess the time at which the stepper will reach the next step position. If the guess is out of bounds or invalid, it falls back to bisection or exponential search. The position at the new guess time is calculated using the supplied callback, which models the move's kinematics.

```
--
snip
--
    if (fabs(guess_dist) > .000000001) {
        double rel_dist = sdir ? guess_dist : -guess_dist;
        if (rel_dist > 0.) {
            if (have_bracket && old_guess.time <= low_time) {
                if (check_oscillate)
                    old_guess = guess;
                check_oscillate = 1;
            }
            high_time = guess.time;
            have_bracket = 1;
        } else if (rel_dist < -(half_step + half_step + .000000010)) {
            sdir = !sdir;
            target = (sdir ? target + half_step + half_step
                           : target - half_step - half_step);
            low_time = last_time;
            high_time = guess.time;
            is_dir_change = have_bracket = 1;
            check_oscillate = 0;
        } else {
            low_time = guess.time;
        }
        if (!have_bracket  high_time - low_time > .000000001) {
            if (!is_dir_change && rel_dist >= -half_step) {
                stepcompress_commit(sk->sc);
            }
            continue;
```

```
        }
    }
--
snip
--
```

Listing 18: itersolve.c

This block checks if the guess is close enough to the target (the position where a step should occur). If not, it updates the search bounds depending on whether the guess overshot or undershot the target. It also handles direction changes and avoids oscillations. If the search interval is still too wide, it continues iterating.

```
--
snip
--
    int ret = stepcompress_append(sk->sc, sdir, m->print_time, guess.time);
    if (ret)
        return ret;
    target = sdir ? target + half_step + half_step
                  : target - half_step - half_step;
    double seek_time_delta = 1.5 * (guess.time - last_time);
    if (seek_time_delta < .000000001)
        seek_time_delta = .000000001;
    if (is_dir_change && seek_time_delta > SEEK_TIME_RESET)
        seek_time_delta = SEEK_TIME_RESET;
    last_time = low_time = guess.time;
    high_time = guess.time + seek_time_delta;
    if (high_time > end)
        high_time = end;
    is_dir_change = have_bracket = check_oscillate = 0;
}
--
snip
--
```

Listing 19: itersolve.c

When the correct timing for a step is found, this block commits the step using stepcompress_append. It then updates the target for the next step, recalculates the search window for the next iteration, and resets flags for direction changes and oscillation checks.

```
sk->commanded_pos = target - (sdir ? half_step : -half_step);
if (sk->post_cb)
    sk->post_cb(sk);
return 0;
```

Listing 20: itersolve.c

After all steps in the range are generated, this block updates the stepper's commanded position to reflect the end state. If a post-processing callback is defined, it is called. The function then returns success.

▲ 4.3.4 Step Pulse Time Compression

In this section we describe Figure 15. stepcompress.c is a C module in `klippy/chelper` that implements step pulse time compression. Its main job is to take a sequence of scheduled stepper motor pulse times generated in the previous seciton, and compress them into a compact format that can be efficiently transmitted from KI to KC. This optimization makes high-speed, precise motion possible by reducing the amount of data sent to KC while preserving timing accuracy for stepper motors. Instead of sending each step time, Klipper compresses a sequence of steps into a parametric form:

$$\text{step}[i] = \text{step}[0] + \text{interval} * i + \text{add} * i * (i - 1)/2$$

where:

1. interval is the base time between steps,
2. add is an incremental adjustment per step,
3. count is the number of steps in the group

The most important computations are in `compress_bisect_add`. **The compression may introduce very small timing errors, but these are kept within a configurable maximum error (see the parameter `max_error`) and do not affect print quality in practice.** The code checks for invalid sequences and ensures that the compressed steps match the intended timing within allowed error bounds.

stepcompress_append. This function is responsible for adding a new (delta_time, step, dir) tuple to the current step compression buffer. It collects the timing and direction/step changes for a segment of stepper motion, preparing them for efficient transmission to the microcontroller. This function is called repeatedly as steps are generated for a move.

stepcompress_commit. Once a set of steps has been appended and is ready to be finalized, stepcompress_commit is called. This function marks the current set of compressed steps as complete, making them ready for queuing and eventual transmission. It ensures that the buffer is in a consistent state and that the data can be safely handed off to the next stage.

stepcompress_flush. This function is used to flush the current compression buffer, forcing any remaining (possibly incomplete) step data to be finalized and prepared for queuing. It is typically called at the end of a move or when a buffer boundary is reached, ensuring that no step data is left unprocessed.

queue_append. After steps are compressed and committed, queue_append is called to add the finalized step data to the queue that will be sent to the microcontroller. This function manages the queue of stepper commands, ensuring that data is stored in the correct order and ready for transmission.

set_next_step_dir. This function sets the direction and step state for the next step to be appended. It is called internally during step generation to update the state machine that tracks the current direction and step signal for each axis.

queue_append_extend. If the current queue entry can be extended with additional step data (for example, if the next move is continuous with the previous one), queue_append_extend is used. This function appends additional compressed steps to an existing queue entry, optimizing bandwidth and reducing the number of individual messages sent to the MCU.

queue_append_far. When a move is too far in the future (for example, due to lookahead or timing constraints), queue_append_far is called. This function handles the special case where a queue entry must be scheduled for execution at a later time, ensuring proper timing and synchronization.

queue_flush. This function flushes the entire step queue, sending all queued stepper commands to the microcontroller. It is typically called when the queue reaches a certain size or when a synchronization point is needed (such as the end of a print move or during a pause).

compress_bisect_add. This function is used during the step compression process to optimize the division of step segments. It applies a bisecting algorithm to determine the best way to split step data for maximum compression efficiency, balancing timing accuracy and message size. **HERE there may be loss of information**.

add_move. add_move is a higher-level function that initiates the process of converting a the step pulse time segments computed previously on queue_step commands.

message_alloc_and_encode. At the final stage before transmission, message_alloc_and_encode is called. This function allocates a message buffer and encodes the compressed stepper data into the format required by the microcontroller protocol. It ensures that the message is properly structured and ready for low-level transmission. This function is at the interface with `serialqueue.c`.

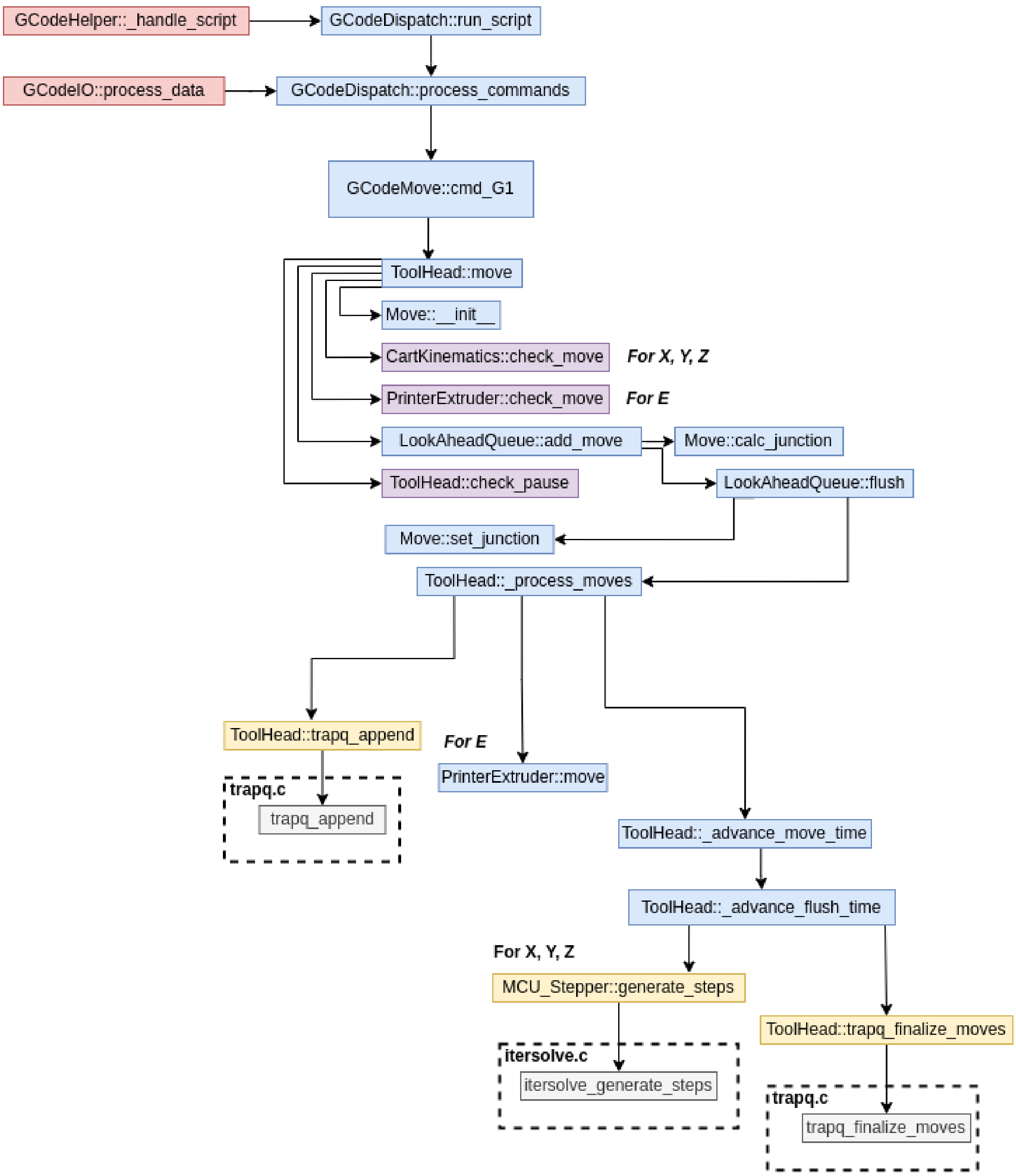

Figure 11: Data flow of a G1 linear move.

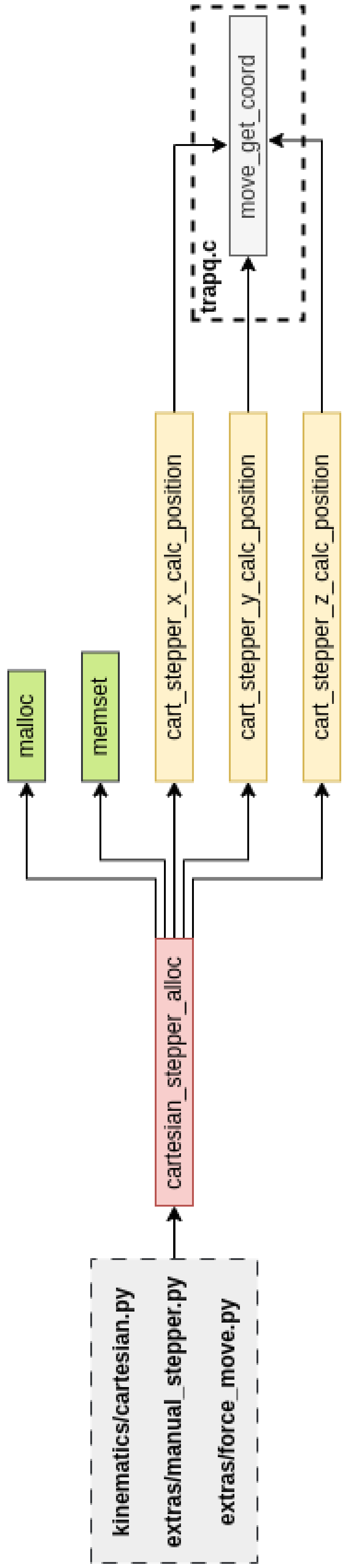

Figure 12: Call Graph of `kin_cartesian.c`.

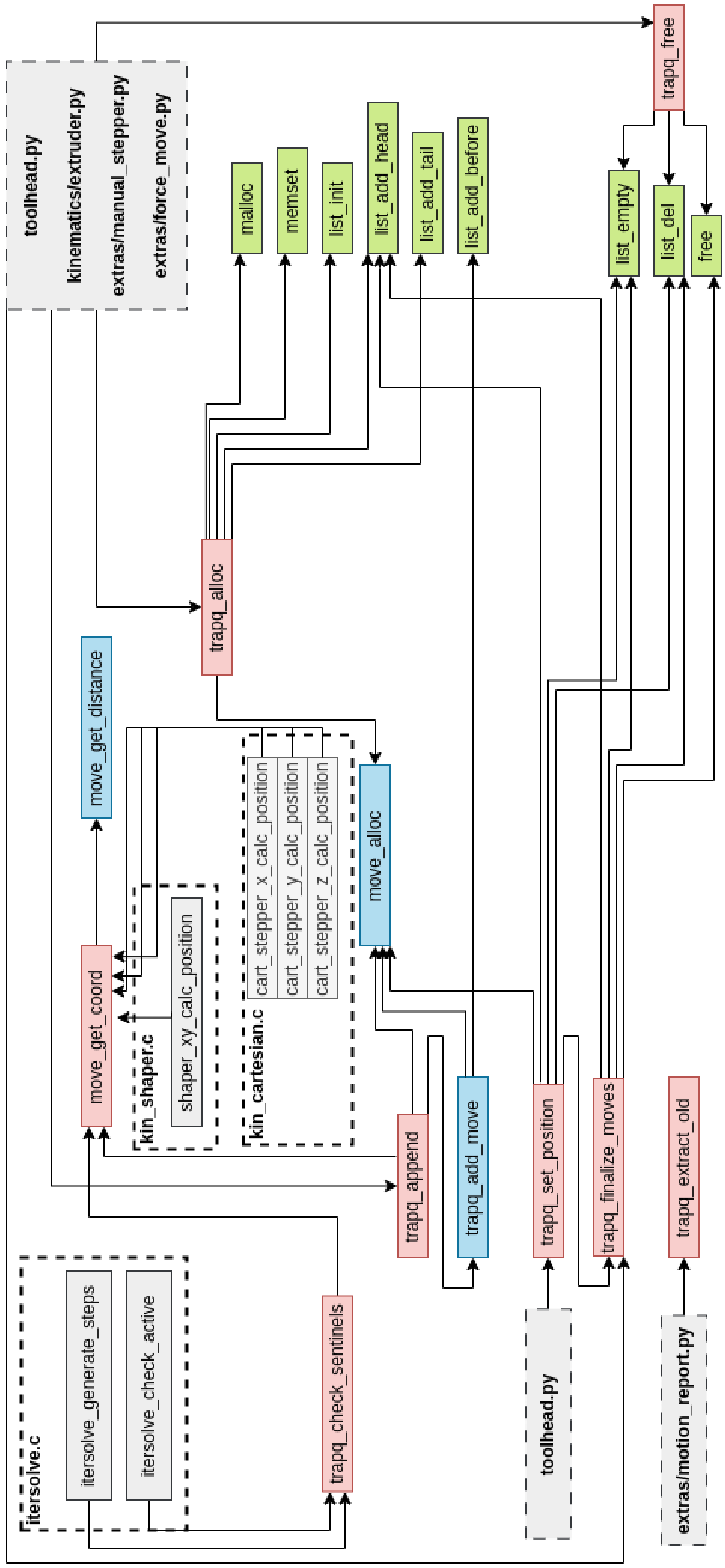

Figure 13: Call Graph of `trapq.c`.

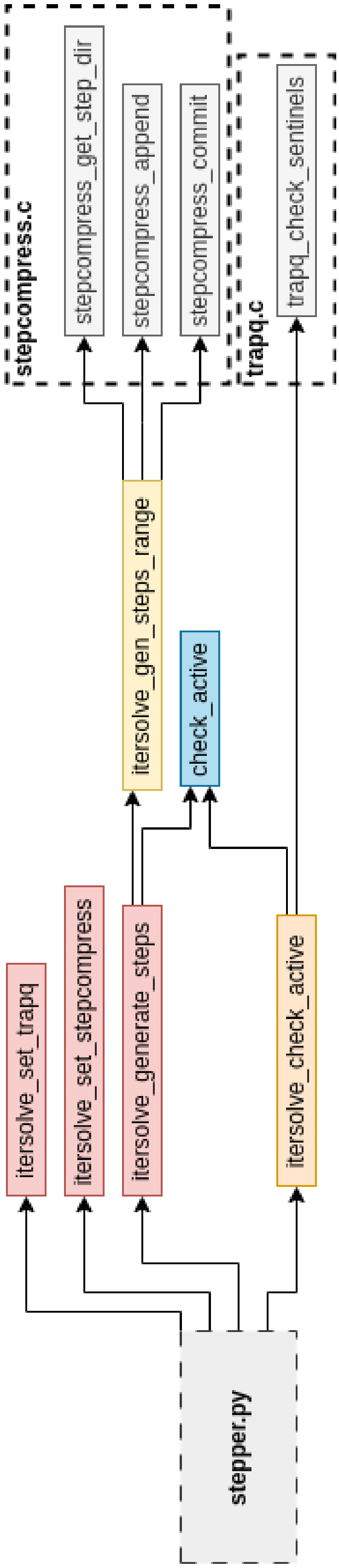

Figure 14: Call Graph of `itersolve.c`.

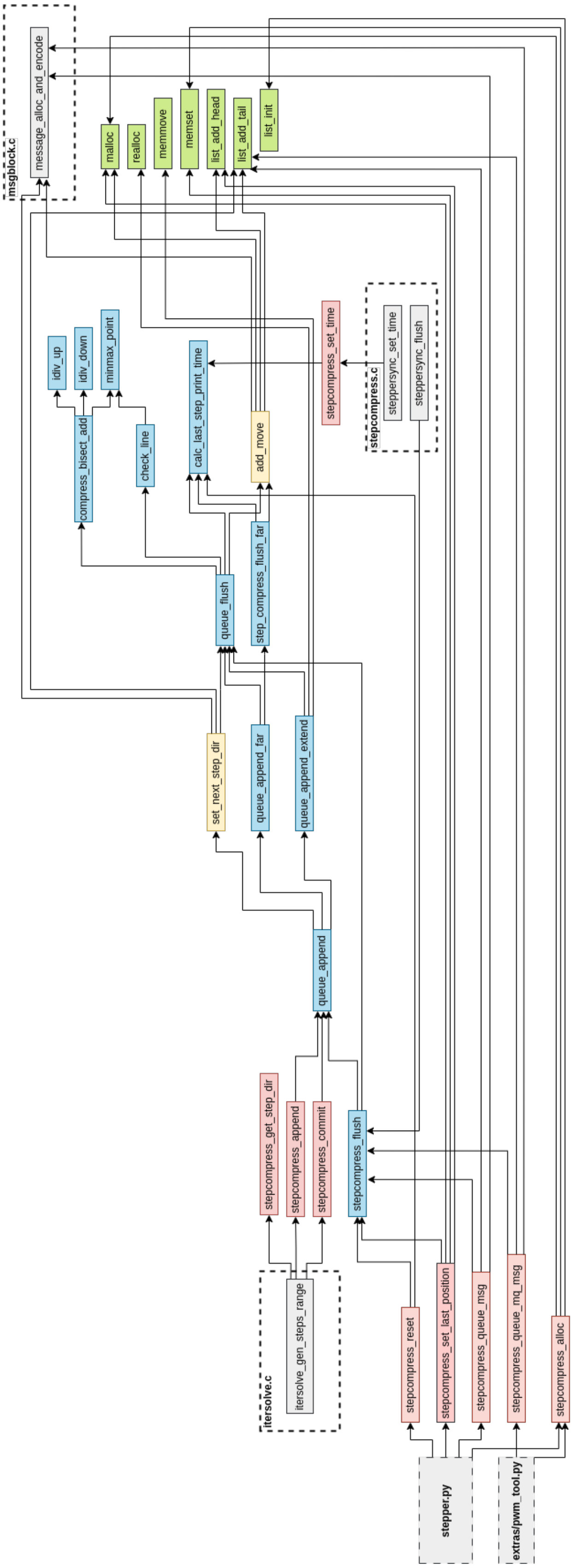

Figure 15: Call Graph of `stepcompress.c` relative to the class `steppercompress`.

⧫ 5 Reconstruction of a G1 Command

Recall that the toolpath is the sequence of movements of the FFF printer print head corresponding to G-Code commands of movement type, such as G1. The toolpath is interpreted by the printer and translated into a series of stepper motor movements. The motor controller generates current in the correct pattern to achieve these movements. The particular properties of this current and recognizing the patterns necessary for executing G1 G-code commands present several challenges to the reconstruction process. In this section we describe a prototype pool reconstrucing G1 commands from operational data used for communication between FFF printer boards.

■ 5.1 Reconstruction

We developed protoyope tool permitting to reconstruct the G1 G-codes for the axis X,Y from Klipper K-codes of type queue_step set_next_step_dir. We validated our results with the complete reconstruction of a G-code file encoding a circle in the plane XY. The code of the prototype has been attached in a separated folder, with the code commented therein.

■ 5.2 Application: Sniffing Attack

FFF printers can have many boards that control their sensors and actuators and try to keep them safe. They communicate through each other to operate the FFF printer in real-time, which means that a reliable communication channel through these booards is required. One of the most common communication solutions is the *Controller Area Network* (CAN) bus. Unfortunately, due to its nature, the CAN bus has no security, only safety measures. It was designed to be a robust communication bus that can withstand a high amount of noise while providing relatively high transfer speeds. The CAN standard is a broadcast protocol that does not provide message authentication nor encryption. This makes easier for attackers to carry out sniffing attacks. This is well-known to the experts and it has been extensively studied in the context of car hacking [27, 12, 19, 52]. To the best of our knowledge, we are not aware of any results involving the study of sniffing atacks in FFF printers for the CAN bus. The sniffing attack of [3] is done in the context of a local area network (LAN) protocol, not for the CAN bus.
An application of the G-code reconstruction could be a sniffing attack as described by Figure 16. In Figure 16 an attacker with physical access to the FFF printer connects a Linux-basedmalicious device (in red) to the CAN, sniffs data using the command line utility SocketCAN (can-utils) and recovers the geometry of the printed part using the proposed tool, thus stealing intellectual property. KI could be a RaspberryPi, and the KC boards could be the models SKR3 and EBB of BigTreeTech.

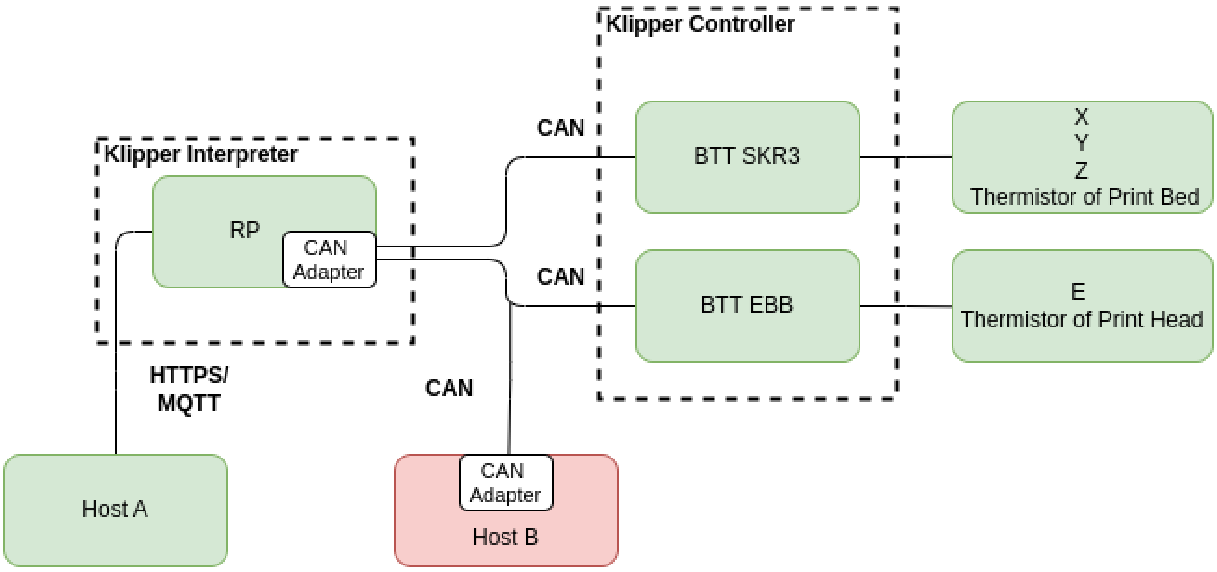

Figure 16: Sniffing Attack

♦ 6 Defensive Techniques

In this section, we give a quick proof-of-concept for securing the communication between KI and KC in Klipper. The documentation of the communication protocol is available at the website of `Klipper`:

https://www.klipper3d.org/Protocol.html

Message blocks are at most 64 bytes long, with a header, a content, and trailer:

<1 byte length><1 byte sequence><n–byte content><2 byte crc><1 byte sync>

The *header* contains the length of the total message and the sequence number. The *content* contains a `KC` command description (e.g., "update_digital_out oid=6 value=1") but encoded as an integer in *Variable Length Quantity* (VLQ). The *trailer* contains a *Cyclic Redundancy Check* (CRC) byte and another byte for syncronisation between `KI` and `KC`, used for the acknowledgements.

The entry/exit points to encrypt/decrypt the incoming/outgoing message block buffers are in `serialqueue.c` for `KI` and in `command.c` for `KC`. Looking to the data-flow, we were able to identify four points of the execution where the message block buffers pass through the encapsulation and de-encapsulation methods, two for each `KI` and `KC`. See the comments `// Encrypt Here` (Listing 21, Listing 23) and `// Decrypt Here` in (Listing 22, Listing 24). This happens at a high level, before the drivers of the communication peripherals enter in the game for `KC`. This means that the implementations of the encryption/decryption at those points would not depend neither on the architecture nor on the communication method (USB, Serial, CAN) that the user selects at compilation time. We used the tools `Ghidra` and `gdb` to analyse the call graphs of `klipper.bin` and `c_helper.so` (the C shared library used by klippy that contains `serialqueue.c`) to make things easier.

Next, we did tested the encryption/decryption with the trivial example of cryptographic primitive: a shift, i.e., transforming a byte x to $x + 1$, for every byte in the message block. We verified our technique by running the module of klippy `console.py`, which permits to see in the terminal the points of `klipper.bin` where we added some extra `printf` for debugging. The well-known tool `screen` follows a similar principle, but it did not work for unkwnown reasons. We thought abou encrypting the whole message blocks, but perhaps it would be enough to encrypt the contents. In any case, the CRC bytes should be outside the encryption.

The next step was choosing a non-trivial primitive that corresponds to the requirements of `KC`. To do so, one needs to bear in mind the security requirements, the security level and the performance. We wanted to protect the confidentiality, integrity an authenticity of message blocks. We did not consider the authenticity of the endpoints of communication, since this is out of our scope for being another kind of PoC. We had no cryptographic accelerator on the microcontroller board used for testing (STM32F1 family), no randomness peripheral to generate keys and no secure-element to storage the keys. We reviewed the most common primitives used for IoT devices in this reference [31]. It turns out that AES-GCM-AEAD suits everybody needs. We were thinking that 256 bits of security level would be a good choice, but we could not test how this would work in our board at level of performance.

Finally, the last think we could do, was the choice of cryptographic libraries. Again, we explored the most common ones used for IoT devices, in terms of secure implementation, performance and level of documentation. For `KH` we used `openssl`, open-source, and for `KC` the `STM32_Cryptographic` library, proprietary, available at:

https://www.st.com/en/embedded-software/x-cube-cryptolib.html

Together with the libraries, we obtained the examples of encryption and encryption to understand the behaviour of the libraries in practice and use the same techniques to integrate the on `klipper` later on. There are still many things to bear in mind. For instance, the most well-known vulnerability for AES-GCM-AEAD is actually the lack of randomness on the generation of the initialization vector for the GCM part, so perhaps it is a must to use a True Random Number Generator (in hardware) for `KC`.

6.1 Component KI

The code of this subsection is in the file klipper/klippy/chelper/serialqueue.c.

Encryption in function build_and_send_command

```
// Construct a block of data to be sent to the serial port
static int
build_and_send_command(struct serialqueue *sq, uint8_t *buf, int pending, double eventtime)
{
    int len = MESSAGE_HEADER_SIZE;
    while (sq->ready_bytes) {
        // Find highest priority message (message with lowest req_clock)
        uint64_t min_clock = MAX_CLOCK;
        struct command_queue *q, *cq = NULL;
        struct queue_message *qm = NULL;
        list_for_each_entry(q, &sq->pending_queues, node) {
            if (!list_empty(&q->ready_queue)) {
                struct queue_message *m = list_first_entry(
                    &q->ready_queue, struct queue_message, node);
                if (m->req_clock < min_clock) {
                    min_clock = m->req_clock;
                    cq = q;
                    qm = m;
                }
            }
        }
        // Append message to outgoing command
        if (len + qm->len > MESSAGE_MAX - MESSAGE_TRAILER_SIZE)
            break;
        list_del(&qm->node);
        if (list_empty(&cq->ready_queue) && list_empty(&cq->upcoming_queue))
            list_del(&cq->node);
        memcpy(&buf[len], qm->msg, qm->len);
        len += qm->len;
        sq->ready_bytes -= qm->len;
        if (qm->notify_id) {
            // Message requires notification - add to notify list
            qm->req_clock = sq->send_seq;
            list_add_tail(&qm->node, &sq->notify_queue);
        } else {
            message_free(qm);
        }
    }

    // Fill header / trailer
    len += MESSAGE_TRAILER_SIZE;
```

```
    buf[MESSAGE_POS_LEN] = len;
    buf[MESSAGE_POS_SEQ] = MESSAGE_DEST | (sq->send_seq & MESSAGE_SEQ_MASK);
    uint16_t crc = msgblock_crc16_ccitt(buf, len - MESSAGE_TRAILER_SIZE);
    buf[len - MESSAGE_TRAILER_CRC] = crc >> 8;
    buf[len - MESSAGE_TRAILER_CRC+1] = crc & 0xff;
    buf[len - MESSAGE_TRAILER_SYNC] = MESSAGE_SYNC;

    // ENCRYPT HERE
    printf("build_and_send_commmand:buffer\n");
    for (int i = 0; i < len; i++) {
        buf[i] += 1;
        printf("%x ", buf[i]);
    }
    printf("\n\n");

    // Store message block
    double idletime = eventtime > sq->idle_time ? eventtime : sq->idle_time;
    idletime += calculate_bittime(sq, pending + len);
    struct queue_message *out = message_alloc();
    memcpy(out->msg, buf, len);
    out->len = len;
    out->sent_time = eventtime;
    out->receive_time = idletime;
    if (list_empty(&sq->sent_queue))
        pollreactor_update_timer(sq->pr, SQPT_RETRANSMIT, idletime + sq->rto);
    if (!sq->rtt_sample_seq)
        sq->rtt_sample_seq = sq->send_seq;
    sq->send_seq++;
    sq->need_ack_bytes += len;
    list_add_tail(&out->node, &sq->sent_queue);
    return len;
}
```

Listing 21: Encryption in KI

Decryption in function `input_event`

```
// Callback for input activity on the serial fd
static void
input_event(struct serialqueue *sq, double eventtime)
{
    if (sq->serial_fd_type == SQT_CAN) {
        struct can_frame cf;
        int ret = read(sq->serial_fd, &cf, sizeof(cf));
        if (ret <= 0) {
```

```
            report_errno("can read", ret);
            pollreactor_do_exit(sq->pr);
            return;
        }
        if (cf.can_id != sq->client_id + 1)
            return;
        memcpy(&sq->input_buf[sq->input_pos], cf.data, cf.can_dlc);
        sq->input_pos += cf.can_dlc;
    } else {
        int ret = read(sq->serial_fd, &sq->input_buf[sq->input_pos]
                       , sizeof(sq->input_buf) - sq->input_pos);
        if (ret <= 0) {
            if(ret < 0)
                report_errno("read", ret);
            else
                errorf("Got EOF when reading from device");
            pollreactor_do_exit(sq->pr);
            return;
        }
        sq->input_pos += ret;
    }

    // DECRYPT HERE
    for (int i = 0; i < sq->input_pos; i++) {
        sq->input_buf[i] -= 1;
    }

    for (;;) {
        int len = msgblock_check(&sq->need_sync, sq->input_buf, sq->input_pos);
        if (!len)
            // Need more data
            return;
        if (len > 0) {
            // Received a valid message
            handle_message(sq, eventtime, len);
        } else {
            // Skip bad data at beginning of input
            len = -len;
            pthread_mutex_lock(&sq->lock);
            sq->bytes_invalid += len;
            pthread_mutex_unlock(&sq->lock);
        }
        sq->input_pos -= len;
        if (sq->input_pos)
            memmove(sq->input_buf, &sq->input_buf[len], sq->input_pos);
    }
```

```
}
```

Listing 22: Decryption in KI

6.2 Component KC

The code of this subsection is in the file `klipper/src/command.c`.

Encryption in function `command_encode_and_frame`

```
// Encode a message and then add a message block frame around it
uint_fast8_t
command_encode_and_frame(uint8_t *buf, const struct command_encoder *ce, va_list args)
{
    uint_fast8_t msglen = command_encodef(buf, ce, args);
    command_add_frame(buf, msglen);
    // ENCRYPT HERE
    for (int i = 0; i < msglen; i++) {
        buf[i] += 1;
    }
    return msglen;
}
```

Listing 23: Encryption in KC

Decryption in `command_find_and_dispatch`

```
// Find a message block and then dispatch all the commands in it
int_fast8_t
command_find_and_dispatch(uint8_t *buf, uint_fast8_t buf_len, uint_fast8_t *pop_count)
{
    // DECRYPT HERE
    for (int i = 0; i < buf_len; i++) {
        buf[i] -= 1;
    }

    int_fast8_t ret = command_find_block(buf, buf_len, pop_count);
    if (ret > 0) {
        command_dispatch(buf, *pop_count);
        command_send_ack();
    }
    return ret;
}
```

Listing 24: Decryption in KC

♦ References

[1] Mohammad Abdullah Al Faruque, Sujit Rokka Chhetri, Arquimedes Canedo, and Jiang Wan. 2016. Acoustic Side-Channel Attacks on Additive Manufacturing Systems. In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*. 1–10. https://doi.org/10.1109/ICCPS.2016.7479068

[2] Mohammad Abdullah Al Faruque, Jishen Wan, and Sujit Rokka Chhetri. 2019. Defending Side Channel Attacks in Additive Manufacturing Systems. Patent. URL: https://patents.google.com/patent/US10212185B1/en.

[3] Hamza Alkofahi, Heba Alawneh, and Anthony Skjellum. 2024. MitM Attacks on Intellectual Property and Integrity of Additive Manufacturing Systems: A Security Analysis. *Computers & Security* 140 (2024), 103810. https://doi.org/10.1016/j.cose.2024.103810

[4] ASTM International. 2021. ISO/ASTM 52900:2021. Additive Manufacturing — General Principles — Fundamentals and Vocabulary. URL: https://www.iso.org/standard/74514.html.

[5] Martin Baumers, Phill Dickens, Chris Tuck, and Richard Hague. 2016. The Cost of Additive Manufacturing: Machine Productivity, Economies of Scale and Technology-Push. *Technological Forecasting and Social Change* 102 (2016), 193–201. https://doi.org/10.1016/j.techfore.2015.02.015

[6] Sofia Belikovetsky, Yosef A. Solewicz, Mark Yampolskiy, Jinghui Toh, and Yuval Elovici. 2019. Digital Audio Signature for 3D Printing Integrity. *IEEE Transactions on Information Forensics and Security* 14, 5 (2019), 1127–1141. https://doi.org/10.1109/TIFS.2018.2851584

[7] Avner Ben-Ner and Enno Siemsen. 2017. Decentralization and Localization of Production: The Organizational and Economic Consequences of Additive Manufacturing (3D Printing). *California Management Review* 59, 2 (2017), 5–23. https://doi.org/10.1177/0008125617695284 arXiv:https://doi.org/10.1177/0008125617695284

[8] Bigtreetech. 2025. Bigtreetech Website. URL: https://bigtree-tech.com/.

[9] Marcel Bogers, Ronen Hadar, and Arne Bilberg. 2016. Additive Manufacturing for Consumer-Centric Business Models: Implications for Supply Chains in Consumer Goods Manufacturing. *Technological Forecasting and Social Change* 102 (2016), 225–239. https://doi.org/10.1016/j.techfore.2015.07.024

[10] Hing Kai Chan, Min Guo, Fangli Zeng, Ying Chen, Tian Xiao, and James Griffin. 2023. Blockchain-Enabled Authentication Platform for the Protection of 3D Printing Intellectual Property: A Conceptual Framework Study. *Enterprise Information Systems* 17, 11 (2023), 2180776. https://doi.org/10.1080/17517575.2023.2180776 arXiv:https://doi.org/10.1080/17517575.2023.2180776

[11] Fotios Chantzis, Ioannis Stais, and Paulino Calderon. 2021. *Practical IoT Hacking: The Definitive Guide to Attacking the Internet of Things*. No Starch Press.

[12] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, USA, 6.

[13] Sujit Rokka Chhetri, Arquimedes Canedo, and Mohammad Abdullah Al Faruque. 2016. KCAD: Kinetic Cyber-Attack Detection Method for Cyber-Physical Additive Manufacturing Systems. In *Proceedings of the 35th International Conference on Computer-Aided Design (Iccad '16)*. Association for Computing Machinery, Austin, Texas and New York, NY, USA, Article 74. https://doi.org/10.1145/2966986.2967050

[14] Sujit Rokka Chhetri, Mohammad Abdullah Al Faruque, Sina Faezi, and Arquimedes Canedo. 2016. *Forensics of Thermal Side-Channel in Additive Manufacturing Systems*. Technical Report CECS-TR-01-16. Center for Embedded and Cyber-Physical Systems, University of California, Irvine.

[15] Pierre-Alexis Ciavaldini. 2017. MakerNet : la fabrication distribuée. *Réalités Industrielles* (Aug. 2017).

[16] Yang Gao, Borui Li, Wei Wang, Wenyao Xu, Chi Zhou, and Zhanpeng Jin. 2018. Watching and Safeguarding Your 3D Printer: Online Process Monitoring against Cyber-Physical Attacks. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 3, Article 108 (Sept. 2018). https://doi.org/10.1145/3264918

[17] Jacob Gatlin, Sofia Belikovetsky, Yuval Elovici, Anthony Skjellum, Joshua Lubell, Paul Witherell, and Mark Yampolskiy. 2021. Encryption Is Futile: Reconstructing 3D-printed Models Using the Power Side-Channel. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (Raid '21)*. Association for Computing Machinery, New York, NY, USA, 135–147. https://doi.org/10.1145/3471621.3471850

[18] Jacob Gatlin, Sofia Belikovetsky, Samuel B. Moore, Yosef Solewicz, Yuval Elovici, and Mark Yampolskiy. 2019. Detecting Sabotage Attacks in Additive Manufacturing Using Actuator Power Signatures. *IEEE access : practical innovations, open solutions* 7 (2019), 133421–133432. https://doi.org/10.1109/ACCESS.2019.2928005

[19] László Gazdag, Károly Farkas, and Levente Buttyán. 2021. Development of a Man-in-the-Middle Attack Device for the CAN Bus. In *Proceedings of the 2021 Conference on IT in Design and Society (CITDS)*. Budapest University of Technology and Economics, 129–136.

[20] Nikhil Gupta, Akash Tiwari, Satish T. S. Bukkapatnam, and Ramesh Karri. 2020. Additive Manufacturing Cyber-Physical System: Supply Chain Cybersecurity and Risks. *IEEE access : practical innovations, open solutions* 8 (2020), 47322–47333. https://doi.org/10.1109/ACCESS.2020.2978815

[21] Avesta Hojjati, Anku Adhikari, Katarina Struckmann, Edward J. Chou, Thi Ngoc Tho Nguyen, Kushagra Madan, Marianne S. Winslett, Carl A. Gunter, and William P. King. 2016. Leave your phone at the door: Side channels that reveal factory floor secrets. In *CCS 2016 - Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communication Security*. Association for Computing Machinery, United States, 883–894. https://doi.org/10.1145/2976749.2978323

[22] Carl Hurd. 2024. *AnyCubic Chitubox AnyCubic Plugin readDatHeadVec Heap-Based Buffer Overflow Vulnerability. URL: https://talosintelligence.com/vulnerability_reports/TALOS-2021-1376*. Technical Report TALOS-2021-1376. Cisco Talos.

[23] Identify 3D. 2020. Identify3D Info Sheet. URL: https://identify3d.com/wp-content/uploads/2019/06/Identify3DInfosheet-1.pdf.

[24] Klipper. 2025. G-Codes. URL: https://www.klipper3d.org/G-Codes.html/.

[25] Klipper. 2025. Klipper Documentation. URL: https://www.klipper3d.org/.

[26] Maximilian Klöckner, Stefan Kurpjuweit, Chander Velu, and Stephan M. Wagner. 2020. Does Blockchain for 3D Printing Offer Opportunities for Business Model Innovation? *Research-Technology Management* 63, 4 (2020), 18–27. https://doi.org/10.1080/08956308.2020.1762444 arXiv:https://doi.org/10.1080/08956308.2020.1762444

[27] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*. 447–462. https://doi.org/10.1109/SP.2010.34

[28] Jiewu Leng, Yuanwei Zhong, Zisheng Lin, Kailin Xu, Dimitris Mourtzis, Xueliang Zhou, Pai Zheng, Qiang Liu, J. Leon Zhao, and Weiming Shen. 2023. Towards Resilience in Industry 5.0: A Decentralized Autonomous Manufacturing Paradigm. *Journal of Manufacturing Systems* 71 (2023), 95–114. https://doi.org/10.1016/j.jmsy.2023.08.023

[29] Priyanka Mahesh, Akash Tiwari, Chenglu Jin, Panganamala R. Kumar, A. L. Narasimha Reddy, Satish T. S. Bukkapatanam, Nikhil Gupta, and Ramesh Karri. 2021. A Survey of Cybersecurity of Digital Manufacturing. *Proc. IEEE* 109, 4 (2021), 495–516. https://doi.org/10.1109/JPROC.2020.3032074

[30] Matthew McCormack, Sanjay Chandrasekaran, Guyue Liu, Tianlong Yu, Sandra DeVincent Wolf, and Vyas Sekar. 2020. Security Analysis of Networked 3D Printers. In *2020 IEEE Security and Privacy Workshops (SPW)*. 118–125. https://doi.org/10.1109/SPW50608.2020.00035

[31] Dominik Merli. 2024. *Engineering Secure Devices: A Practical Guide for Embedded System Architects and Developers*. No Starch Press.

[32] Samuel Bennett Moore, Phillip Armstrong, Todd McDonald, and Mark Yampolskiy. 2016. Vulnerability Analysis of Desktop 3D Printer Software. In *2016 Resilience Week (RWS)*. 46–51. https://doi.org/10.1109/RWEEK.2016.7573305

[33] Samuel Bennett Moore, William Bradley Glisson, and Mark Yampolskiy. 2017. Implications of Malicious 3D Printer Firmware. In *Hawaii International Conference on System Sciences*. https://doi.org/10.24251/HICSS.2017.41899.

[34] Mordor Intelligence Research & Advisory. 2025. 3D Printing Market Size & Share Analysis - Growth Trends & Forecasts (2025 - 2030). URL: https://www.mordorintelligence.com/industry-reports/3d-printing-market.

[35] Obico. 2025. Obico Website. URL: https://www.obico.io/.

[36] Zachary Oligschlaeger, Benjamin Baltes, and Jennifer Chin. 2020. Secure 3D Printing. Patent. URL: https://www.freepatentsonline.com/y2020/0326683.html.

[37] Sumit Paul. 2021. Finite Element Analysis in Fused Deposition Modeling Research: A Literature Review. *Measurement* 178 (2021), 109320. https://doi.org/10.1016/j.measurement.2021.109320

[38] Hammond Pearce, Kaushik Yanamandra, Nikhil Gupta, and Ramesh Karri. 2022. FLAW3D: A Trojan-Based Cyber Attack on the Physical Outcomes of Additive Manufacturing. *IEEE/ASME Transactions on Mechatronics* 27, 6 (2022), 5361–5370. https://doi.org/10.1109/TMECH.2022.3179713

[39] Eujin Pei, Alain Bernard, Dongdong Gu, Christoph Klahn, Mario Monzón, Maren Petersen, and Tao Sun (Eds.). 2023. *Springer Handbook of Additive Manufacturing*. Springer International Publishing. https://doi.org/10.1007/978-3-031-20752-5

[40] Peopoly. 2025. Peopoly Website. URL: https://peopoly.net/.

[41] Jaco Prinsloo, Saurabh Sinha, and Basie von Solms. 2019. A Review of Industry 4.0 Manufacturing Process Security Risks. *Applied Sciences* 9, 5105 (2019). https://doi.org/10.3390/app9235105

[42] Muhammad Haris Rais, Muhammad Ahsan, and Irfan Ahmed. 2024. SOK: 3D Printer Firmware Attacks on Fused Filament Fabrication. In *Proceedings of the 18th USENIX*

Conference on Offensive Technologies (WOOT'24). USENIX Association, Philadelphia, PA, USA and USA, Article 18. https://doi.org/10.5555/3696933.3696951

[43] Muhammad Haris Rais, Ye Li, and Irfan Ahmed. 2021. Dynamic-Thermal and Localized Filament-Kinetic Attacks on Fused Filament Fabrication Based 3D Printing Process. *Additive Manufacturing* 46 (2021), 102200. https://doi.org/10.1016/j.addma.2021.102200

[44] Muhammad Haris Rais, Ye Li, and Irfan Ahmed. 2021. Spatiotemporal G-Code Modeling for Secure FDM-based 3D Printing. In *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems (Iccps '21)*. Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/3450267.3450545

[45] Reddit and Anycubic Community. 2024. Important Notice-Security Issue of Anycubic Cloud.

[46] Reddit and SyntheticStart. 2023. My Printer Started Itself at 4am and Caused Some Damage before I Woke Up.

[47] Ben Redwood, Filemon Schöffer, and Brian Garret. 2017. *The 3D Printing Handbook: Technologies, Design and Applications*. 3D Hubs, Amsterdam, The Netherlands.

[48] RepRap. 2025. G-Code. URL: https://reprap.org/wiki/G-code.

[49] Jost Rossel, Vladislav Mladenov, Nico Wördenweber, and Juraj Somorovsky. 2025. Security Implications of Malicious G-Codes in 3D Printing. In *Proceedings of the 34th USENIX Security Symposium (USENIX '25)*.

[50] Prithwish Basu Roy, Mudit Bhargava, Chia-Yun Chang, Ellen Hui, Nikhil Gupta, Ramesh Karri, and Hammond Pearce. 2023. A Survey of Digital Manufacturing Hardware and Software Trojans. arXiv:2301.10336

[51] Tomáš Šlajs, Ivo Klaban, and Petr Františ. 2023. Cyber-Attack Vectors Focused on Additive Manufacturing. In *2023 Communication and Information Technologies (KIT)*. 1–5. https://doi.org/10.1109/KIT59097.2023.10297115

[52] Craig Smith. 2016. *Car Hacker's Handbook: A Guide for the Penetration Tester*. No Starch Press.

[53] Chen Song, Feng Lin, Zhongjie Ba, Kui Ren, Chi Zhou, and Wenyao Xu. 2016. My Smartphone Knows What You Print: Exploring Smartphone-Based Side-Channel Attacks against 3D Printers. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (Ccs '16)*. Association for Computing Machinery, Vienna, Austria and New York, NY, USA, 895–907. https://doi.org/10.1145/2976749.2978300

[54] Alyxandra Van Stockum, Elizabeth Kurkowski, Tiffany Potok, Curtis Taylor, Joel A. Dawson, Mason Rice, and Sujeet Shenoi. 2022. Attack-Defense Modeling of Material Extrusion Additive Manufacturing Systems. In *Critical Infrastructure Protection XVI*, Jason Staggs and Sujeet Shenoi (Eds.). Springer Nature Switzerland, Cham, 121–153. https://doi.org/10.1007/978-3-031-20137-0_5

[55] Viaccess-Orca. 2025. Secure Manufacturing Platform. URL: https://www.viaccess-orca.com/secure-manufacturing-platform.

[56] Vistory. 2025. MainChain. URL: https://vistory.com/mainchain-en/.

[57] Wikipedia. 2010. Stuxnet. URL: https://en.wikipedia.org/wiki/Stuxnet.

[58] Wikipedia. 2017. WannaCry. URL: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[59] Claude Xiao. 2013. Security Attack to 3D Printing. Presentation at XCON2013, Beijing. URL: https://www.claudxiao.net/2013/08/attack-3d-printing/.

[60] Xun Xu. 2008. *Integrating Advanced Computer-Aided Design, Manufacturing, and Numerical Control: Principles and Implementations*. IGI Publishing, Hershey, PA.

[61] Mark Yampolskiy, Todd R. Andel, J. Todd McDonald, William B. Glisson, and Alec Yasinsac. 2014. Intellectual Property Protection in Additive Layer Manufacturing: Requirements for Secure Outsourcing. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW-4)*. Association for Computing Machinery, New York, NY, USA, Article 7. https://doi.org/10.1145/2689702.2689709

[62] Mark Yampolskiy, Wayne King, Gregory Pope, Sofia Belikovetsky, and Yuval Elovici. 2017. Evaluation of Additive and Subtractive Manufacturing from the Security Perspective. In *Critical Infrastructure Protection XI*, Mason Rice and Sujeet Shenoi (Eds.). Springer International Publishing, Cham, 23–44. https://doi.org/10.1007/978-3-319-70395-4_2

[63] Mark Yampolskiy, Wayne E. King, Jacob Gatlin, Sofia Belikovetsky, Adam Brown, Anthony Skjellum, and Yuval Elovici. 2018. Security of Additive Manufacturing: Attack Taxonomy and Survey. *Additive Manufacturing* 21 (2018), 431–457. https://doi.org/10.1016/j.addma.2018.03.015

[64] Jiayuan Zhang, Mengyue Feng, Yuhang Huang, Tao Feng, Wenjie Wang, He Wang, and Han Zhang. 2022. Survey of 3D Printer Security Offensive and Defensive. In *Frontiers in Cyber Security*, Chunjie Cao, Yuqing Zhang, Yuan Hong, and Ding Wang (Eds.). Springer Singapore, Singapore, 163–178. https://doi.org/10.1007/978-981-19-0523-0_11

[65] Sébastien Ziegler, Renáta Radócz, Adrian Quesada Rodriguez, and Sara Nieves Matheu Garcia (Eds.). 2024. *Springer Handbook of Internet of Things*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-031-39650-2

♦ Contact

Xabier Legaspi
Université de Rennes. 35042 Rennes, France.
xabier.legaspi@irisa.fr