information

Article

Defending IoT Devices against Bluetooth Worms with Bluetooth OBEX Proxy

Fu-Hau Hsu [1], Min-Hao Wu [2,*], Yan-Ling Hwang [3], Jian-Xin Chen [1], Jian-Hong Huang [1], Hao-Jyun Wang [1] and Yi-Wen Lai [1]

[1] Department of Computer Science and Information Engineering, National Central University, Taoyuan 32001, Taiwan; hsufh@csie.ncu.edu.tw (F.-H.H.); opp556687@gmail.com (J.-X.C.); gordon636798@gmail.com (J.-H.H.); alan.wang388@gmail.com (H.-J.W.); ncu106201516@gmail.com (Y.-W.L.)
[2] Department of Electronic and Information Engineering, Xiamen City University, Xiamen 361000, China
[3] School of Applied Foreign Languages, Chung Shan Medical University, Taichung 40201, Taiwan; yanling@csmu.edu.tw
* Correspondence: mhwu@csie.ncu.edu.tw

Abstract: The number of Internet of Things (IoT) devices has increased dramatically in recent years, and Bluetooth technology is critical for communication between IoT devices. It is possible to protect electronic communications, the Internet of Things (IoT), and big data from malware and data theft with BlueZ's Bluetooth File Transfer Filter (BTF). It can use a configurable filter to block unauthorized Bluetooth file transfers. The BTF is available for various Linux distributions and can protect many Bluetooth-enabled devices, including smartphones, tablets, laptops, and the Internet of Things. However, the increased number and density of Bluetooth devices have also created a serious problem—the Bluetooth worm. It poses a severe threat to the security of Bluetooth devices. In this paper, we propose a Bluetooth OBEX Proxy (BOP) to filter malicious files transferred to devices via the OBEX system service in BlueZ. The method described in this article prevents illegal Bluetooth file transfers, defending big data, the Internet of Things (IoT), and electronic communications from malware and data theft. It also protects numerous Bluetooth devices, including smartphones, tablets, laptops, and the Internet of Things, with many Linux distributions. Overall, the detection findings were entirely accurate, with zero false positives and 2.29% misses.

Keywords: Internet of Things (IoT); Bluetooth technology; IoT device security; IoT malware; BlueZ's OBEX service daemon

check for updates

Citation: Hsu, F.-H.; Wu, M.-H.; Hwang, Y.-L.; Chen, J.-X.; Huang, J.-H.; Wang, H.-J.; Lai, Y.-W. Defending IoT Devices against Bluetooth Worms with Bluetooth OBEX Proxy. *Information* **2023**, *14*, 525. https://doi.org/10.3390/info14100525

Academic Editors: Krzysztof Szczypiorski and Daniel Paczesny

Received: 20 August 2023
Revised: 24 September 2023
Accepted: 25 September 2023
Published: 27 September 2023

1. Introduction

Bluetooth is a wireless communication protocol for short-range data transfer using the 2.4 GHz radio frequency. Today, smartphones, laptops, and portable game consoles have built-in Bluetooth connectivity. Many studies have shown the spread of Bluetooth worms [1–4]. With the increasing number of Bluetooth-enabled devices around us, Bluetooth worms have become a severe problem.

Users store private information on Bluetooth-connected devices, including banking data (credit card numbers, bank account numbers), private photos or videos, text messages, health information, calendar appointments, emails, and contact information [5,6]. It makes Bluetooth security an even more significant concern. Given the information we provide through Bluetooth, a hacker may steal money, eavesdrop on conversations, seize complete control of a device, monitor or influence their victim's behavior, or even infect a device's network with malware [7]. It is critical to comprehend users' familiarity with the mitigation techniques that are accessible to protect against such vulnerabilities and security risks.

Since Bluetooth technology has existed for a while, several studies have documented the different vulnerabilities and attack methods related to Bluetooth devices. In a report published in 2010 by John Dunning, all the dangers and weaknesses related to Bluetooth technology were investigated and explained [8]. Many writers have researched the Bluetooth man-in-the-middle (MITM) attack scenario to better comprehend these types of assaults. In keeping with this, these writers have offered a variety of risk-reduction techniques that may be used by both individuals and organizations [9–11]. A thorough description of the many Bluetooth attack vectors, including worms, Trojans, DDoS assaults, MITM attacks, and more, is given [7,12,13].

Although the worm is well-known online [14,15], the mobile Internet has yet to gain much attention regarding worms. The existing methods may be briefly divided into two groups. Su et al.'s research [16] demonstrated that Bluetooth is a crucial interface for worm spreading and short-range worm containment. This finding was supported by research conducted by Yan and Eidenbenz [17], Mickens and Noble [18], and Morris-King and Cam [19], which examined the kinetics of worm transmission over the Bluetooth interface. A distributed response system created by Ziba et al. [20] uses worm signatures to eliminate nearby worms. However, the algorithm's temporal complexity makes it too challenging to handle massive networks. A graphical color-based sensor worm-coping approach was put forth by Yang et al. [21]. The approach's fundamental tenet is to broaden the variety of software versions in the network. By restricting communication between susceptible nodes, Li et al. [22] established a technique to determine a node's level of susceptibility and manage worms. By rejecting connection requests from strangers, Miklas et al. [23] exploited social interactions to increase the security of Bluetooth interfaces and lessen the transmission of malware. In order to safeguard massively dynamic mobile networks, Gao and Liu [24] devised a two-layer network model that centered on the influence of human behavior on worm propagation.

Fleizach et al. [25] validated the variations in the propagation traits of Internet and mobile Internet worms. They assessed the efficiency of MMS worms on cellular networks for long-range worm containment. Meng et al.'s [26] analysis of track data in mobile networks allowed them to look at the reliability of SMS communications. The isolation strategy was utilized by Bose et al. [27] to restrict communication between the MMS network's susceptible nodes. The core nodes in a social network should be protected initially, according to Zhu et al.'s research [28]. However, this strategy disregards the worm's method of transmission through the Bluetooth interface, leaving room for the worm to spread swiftly.

Social networks cannot be used to predict the number of clusters that the algorithm requires. Zhao et al. [29] merged the centralized and decentralized patch distribution mechanisms by creating a novel network layer model. Yang and Yang put forth an assessment approach [28] to gauge how effective patch distribution is. Identifying high-velocity infection locations is essential for distant worm containment. Community discovery [30,31] and social impact analysis [32,33] have been frequently used in recent research to tackle this problem. Bluetooth vulnerabilities can have a significant impact on actual business. A Bluetooth vulnerability could be used to steal sensitive data such as corporate secrets, customer data, or personal health information. It could cause significant damage to an organization and undermine customer trust. Bluetooth vulnerabilities can be used to plant ransomware, which could lead to business downtime and data loss. Bluetooth vulnerabilities can be used to attack industrial control systems, which could lead to production disruptions or human casualties. Specific examples of Bluetooth vulnerabilities include hackers using Bluetooth to attack a significant retailer, stealing the credit card information of more than 40 million customers, and using Bluetooth to attack a major healthcare company, stealing the medical information of more than 15 million patients. Organizations should also raise employee awareness of Bluetooth security risks and educate them on identifying and avoiding them. A Bluetooth worm will replicate itself to vulnerable devices that it can reach, and all infected devices will try to repeat themselves to other vulnerable devices. The vulnerabilities lie in the protocol itself and the implementation of the protocol. Early versions of the Bluetooth protocol were not designed with security in mind. Although later versions of the Bluetooth protocol made some changes to the pairing process for security purposes, the sheer complexity of the protocol itself makes auditing the protocol's implementation complex [34].

IoT device security procedures include safeguards such as creating secure hardware or methods to identify and stop IoT malware. By considering these variables and protecting against device hacking, several researchers in both industry and academia are tackling IoT security. However, most current research focuses on machine learning or deep learning-based detection strategies, rather than how IoT malware builds such an extensive network of infected devices [35–37]. As a result, a review is required to assess the current body of knowledge. Meanwhile, specific works—like [38–42]—discuss IoT malware, its many assaults, and its characteristics.

Malware's functional complexity has increased with time, substantially affecting practically all gadgets. Malware may be divided into many categories according to its operation, including worms, trojan horses, viruses, spyware, ransomware, rootkits, and backdoors. IoT malware is interesting because, although it falls into several categories, it is all bot-based, meaning it uses botnets for command and control, execution, and distribution. Bluetooth is a well-liked wireless technology that enables close-range communication between devices. However, the BlueBorne attack is a significant security flaw that may be used to access Bluetooth devices without authorization—a fresh approach to forecasting and stopping BlueBorne assaults. It can build the model on the application layer and has demonstrated greater efficacy than earlier models. Studies that have already been carried out [43,44] concentrate on screening malicious Bluetooth packets that might lead to remote code execution. Miretskiy et al. [45] sought to filter all files in the system while they tried to halt Bluetooth worms during the assault phase. The second method invariably resulted in a system overhead, whereas the former demands new rules for each CVE that can be identified in the future.

This study attempts to stop Bluetooth worms in the replication phase. Although this approach cannot prevent the system from being hacked, it can prevent unknown Bluetooth worms from spreading further.

There are several possible mechanisms for filtering malicious worm files in the system or the file transfer process. These can be categorized as the following:

1. System call level: The goal is to hook system calls that are related to files, such as open, close, read, write, etc. These can be accomplished by (1) modifying the kernel's system call implementation or (2) using the eBPF filter to intercept the system calls. For this approach to work, the file system must be in mandatory locking mode, which ensures that a time-of-check to time-of-use attack cannot happen [45].
2. File system level: Avfs [37] is an on-access antivirus file system that uses Linux's virtual file system (VFS) to stack on top of existing file systems and hooks the filtering mechanisms into the read/write operation of the VFS. The on-access design minimizes the performance impact compared to onopen, onclosed, or on-exec scanning.
3. Protocol level: Unlike the previous two categories, this approach focuses on files that are transferred from external devices over network or wireless protocols. Depending on the implementation and usage of the protocol stack, it may be challenging to determine at which layer(s) in the protocol stack the hook points should be placed. There are several challenges with this approach. For example, it may implement some layers in the protocol stack inside the kernel; we need to be extra careful when hooking them. Also, hooking inside the protocol layer involves manually inspecting and modifying raw packets. For protocols that live inside another protocol, the packets may fragment, and it may be challenging to reconstruct them, especially if the protocol has to deal with retransmission packets.
4. The service level, based on where the mechanism hooks into the system: When multiple applications need Bluetooth to interact with other devices, they may want to implement some Bluetooth stacks. These can typically be performed through libraries, sockets, or by sending the request to a service agent that does the job. This service

approach makes managing connected Bluetooth devices and using standard Bluetooth protocols easier.

The first section of this paper is the introduction, which describes the basic concepts and inspirations of this study. The second section is the background, which introduces the background knowledge. The third section is related research, which focuses on the BlueBorne vulnerability provided by the current malware; IoT malware attacks are not limited to any part of a device. Section 4 presents the system structure and implementation details of BlueZ's OBEX service, while Section 5 reports the evaluation results of the Bluetooth file transfer filtering mechanism. The limitations of the Bluetooth file transfer filtering mechanism are discussed, as well as the research challenges, conclusion, and future, in Section 6.

2. Background

In this section, we take a service-level approach to Linux-based Bluetooth file transfer filtering that can be easily used by modern Linux operating systems that use BlueZ as their Bluetooth implementation. We created a proxy service called the Bluetooth OBEX Proxy (BOP), which can intercept and modify all messages to and from the original Bluetooth service. It allows us to filter out malicious files transmitted via the Bluetooth service.

2.1. Bluetooth Worm

Gonzalez and Larraga et al. [46] suggested a CA-based two-dimensional model based on an epidemiological compartmentalized model to examine the transmission of Bluetooth worms. According to related research, there are seven different smartphone prevalence states: vulnerable, exposed, infected, diagnosed, carried, interrupted, and recovered. When considering homogenous smartphones, a set of local rules was created to mimic the dynamics of the model. The model has recently expanded to assess the potential for recovering and running antivirus updates and their effect on worms. To evaluate how Bluetooth malware spreads under various circumstances, the presented model does not consider additional aspects that may affect worm propagation, such as user interactions, human behavior, and malware transmission characteristics. This study also evaluated transmission ranges of 1, 10, or 100 m caused by various Bluetooth antenna types, transmission acceptance, and discoverable patterns resulting from direct user engagement. Bluetooth worms use Bluetooth as their propagation method. This short-range communication method heavily influences propagation behavior. It is proximity-based, bandwidth-limited, and can be in constant motion [4]. Figure 1 illustrates the cycle of a Bluetooth worm during infection.

2.2. BlueZ

There is a generic Bluetooth stack called BlueZ [47–49] for implementing the Bluetooth Hosting Protocol stack on Linux. It is the official Bluetooth stack for Linux and is a free-source software. BlueZ is a Linux Bluetooth stack that maps the Bluetooth protocol layer to user space daemons, kernel modules, kernel threads, configuration tools, utilities, and libraries. These components work together to provide a complete Bluetooth stack for Linux. The Bluetooth protocol stack comprises a controller chip and the corresponding host machine. A USB or UART interface connects a Bluetooth hardware device to the host. The Linux kernel communicates with the device through the host controller interface (HCI) by sending HCI packets to and from the hardware controller, as shown in Figure 2. The Bluetooth controller then creates a radio signal generated using an Asynchronous Connectionless Link (ACL) or Synchronous Connection-Oriented Link (SCO) in the baseband. SCOs are used for latency-sensitive applications that do not require a lossless data transfer, such as hands-free audio.

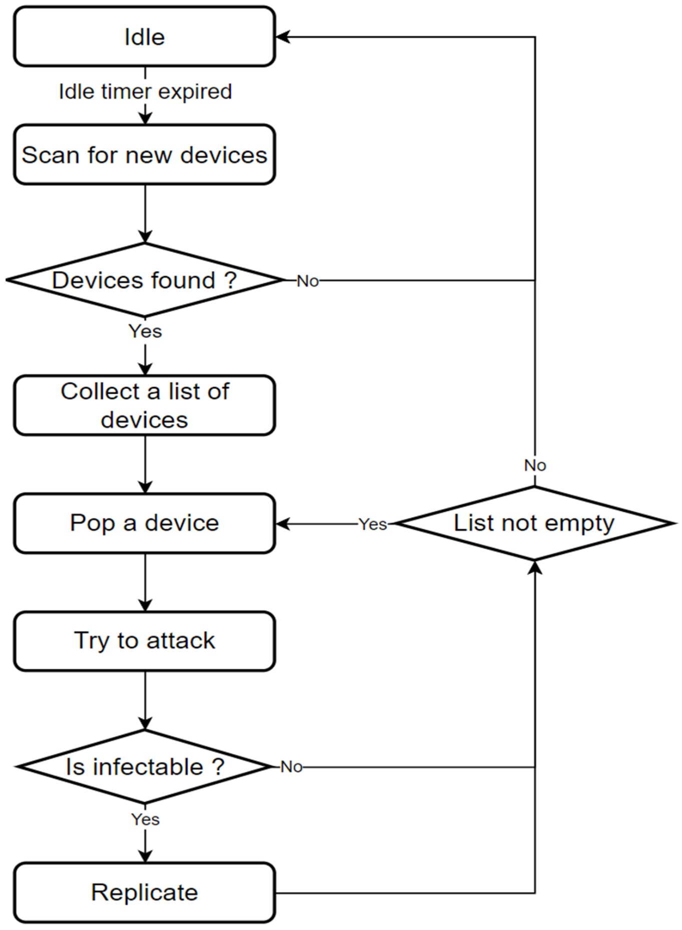


Figure 1. Bluetooth worm infection cycle.

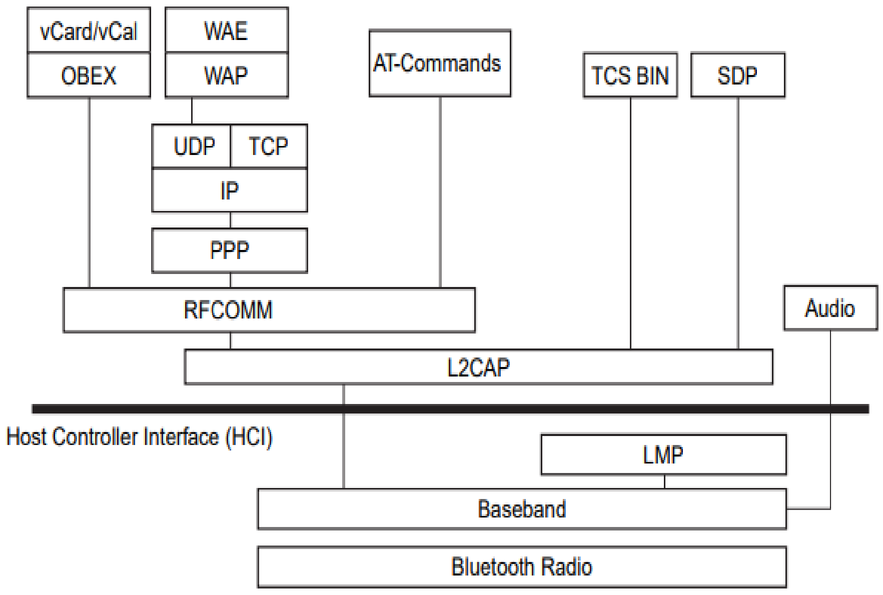


Figure 2. Bluetooth protocol stack.

While the Bluetooth controller handles the radio, baseband, and some of the HCI layers, BlueZ runs the HCI and higher layers in the kernel or user space, as shown in Table 1. Some lower-layer protocols are implemented in the kernel, and the Bluetooth core exposes the protocols through the Linux socket, simplifying the work needed to use Bluetooth in applications.

Table 1. Bluetooth protocol layers in kernel space and user space.

Kernel Space	User Space
L2CAP	SDP
SMP	OBEX
SCO audio	TCS
RFCOMM	
BNEP	
CMTP	
HIDP	

BlueZ provides service daemons for managing Bluetooth devices without knowing the level of HCI commands, command-line tools, testing, and debugging. The service daemon also allows for easy integration into the graphical user interfaces. For this service approach to work, there must be a service manager and a way to communicate with the service daemon. There is where systemd and D-Bus come in.

Bluetooth is a system service daemon for managing, pairing, and handling service discovery protocols (SDPs) with other Bluetooth devices. Applications can register Bluetooth profiles with bluetoothd, and the shapes will add to the SDP records. When a connection request is made to a Bluetooth profile, bluetoothd creates the Bluetooth socket for the requesting device and passes the socket inode to the application that registers the shape, as shown in Figure 3.

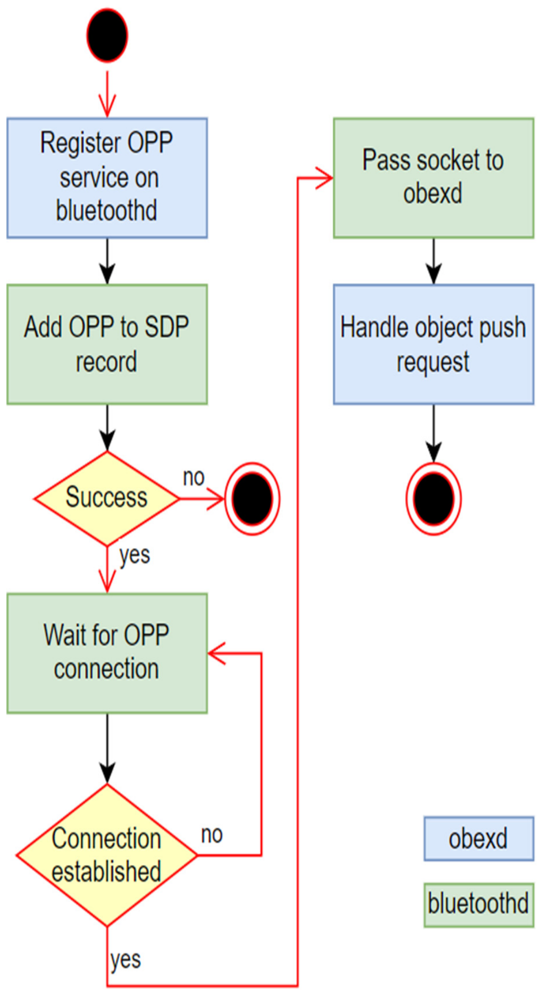


Figure 3. bluetoothd and Bluetooth service profiles.

2.3. Systemd

In Linux distributions, systemd is a frequently used scheduling system [50]. It has often drawn both acclaim and condemnation. It is considered to be highly centralized, since it plays a crucial role in managing system operations, including logging, scheduling, service monitoring, and system setup, which runs counter to the Unix tenet that each program should focus on one task properly. Systemd's supporters will point out that it is just a group of individual binary files, such as systemctl and journald, that, when combined, form a more extensive system. Whatever your opinion of systemd, it has spread so widely that it is practically impossible to ignore if you use a major Linux distribution. Red Hat first developed it in 2010 to replace older init systems, particularly SysV-style ones. By 2015, systemd had replaced SysV init and other init systems in the most popular distributions, including CentOS, RHEL, Debian, Ubuntu, and SUSE.

Systemd is one of the most popular init systems in the world. An init system is the first process, with process ID 1, to start after the kernel is loaded. It initializes other functions, services, etc., from preconfigured startup scripts or configuration files. In the case of systemd, these are called the "unit" and "unit file". A unit can be a service, socket, device, target, or slice managed by systemd, with an associated init-style unit file describing it. Systemd is responsible for calculating all unit dependencies and marks, so it can start units in parallel in the correct order, avoiding unnecessary delays in the boot process. Systemd also has command-line tools such as systemctl and journalctl for managing and logging units. And this mechanism uses D-Bus for inter-process communication, making D-Bus a mandatory requirement.

The unit files follow the UNIX-inspired file hierarchy, which is the standard for all systemd systems. Typically, the system-specific unit files for design and user services are stored in /etc/systemd/system and /etc/systemd/user, respectively. They can be modified or installed by an administrator. Vendor-supplied unit files are stored in "/usr/lib/systemd/", with a "/lib" symbolic link pointing to them for backward compatibility. Package managers such as apt, and npm install these vendor-supplied unit files and should not be modified manually. Also, the unit files in /etc/systemd/ override the unit files in /usr/lib/systemd, so you can change the behavior without modifying the default unit files. The type name is used as the file extension for each type of unit, such as ".Service" for service units and ".socket" for socket units.

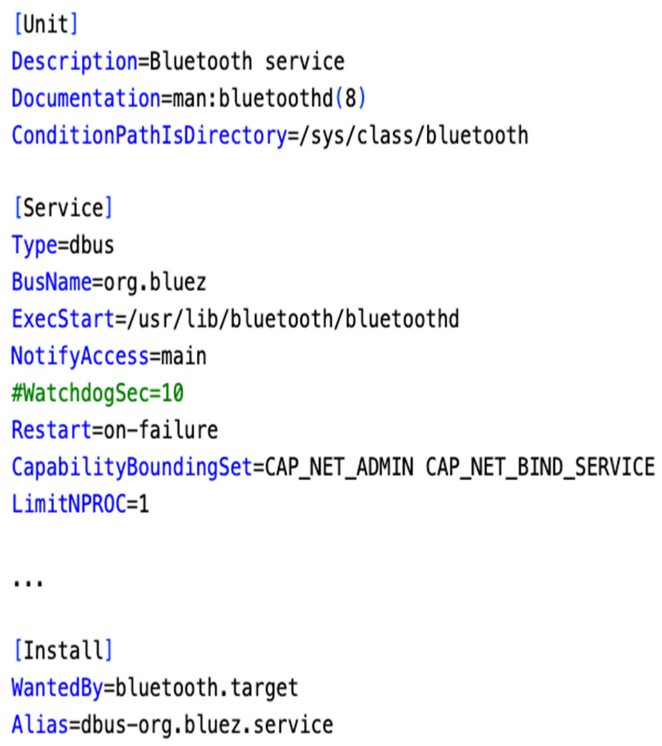
Service unit

A service unit is a configuration file that describes a service in the Linux operating system. It consists of three sections: [Unit], [Service], and [Install]. The [Unit] section contains the description and documentation information that will be displayed using systemctl. It can also specify conditions under which this unit should be run or not and identify the dependencies, triggers, and conflicts when running this service using forward properties. The [Service] section describes the type of service; how to start, stop, and reload the service; and the environment, including environment variables, capabilities, and system resources in which the service process should run. The [Install] section is used when the command-line tool, systemctl, turns the device on or off. It can specify reverse properties instead of forward ones in the [unit] section. It can also create aliases for the unit.

The service type can be simple, exec, forking, oneshot, dbus, notify, or idle; this is needed because systemd needs to track the PIDs to know if the service is running. For example, if an executable does a double fork to make itself a daemon process, but the type for that service is set to simple, then systemd will consider that service to have exited.

A service unit of type dbus must specify the bus name that it will register on the dbus, so that systemd can check for bus name collisions before starting it. The running state of the service is determined by whether the bus name is captured or released.

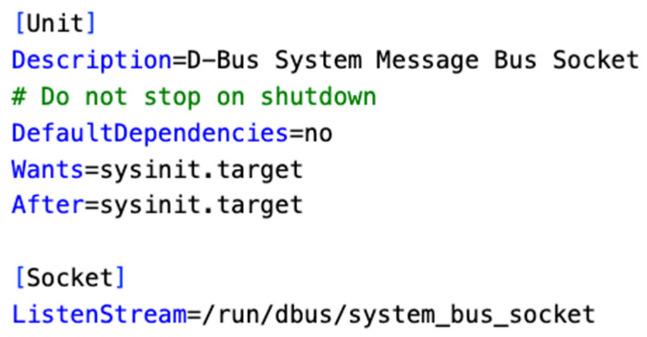
Figure 4 is a simplified system service unit file for the Bluetooth core. It is a dbus-type service that will register the "org.bluez" bus name. This service will only run if sysfs for Bluetooth are present. The service is started by running the "/usr/lib/bluetooth/bluetoothd" executable with additional Linux kernel functions such as CAP_NET_BIND_SERVICE, and CAP_NET_ADMIN.Services can use sockets for inter-process communication. Systemd provides socket-based activation for these services. Systemd will create a listening socket that waits for incoming connections and starts the associated service when a connection is made, but this does not mean that the service cannot run without the socket. The socket unit file is typically set to the same name as the service unit file. The socket is passed to the service process via sd_listen_fds using the systemd library.



```
[Unit]
Description=Bluetooth service
Documentation=man:bluetoothd(8)
ConditionPathIsDirectory=/sys/class/bluetooth

[Service]
Type=dbus
BusName=org.bluez
ExecStart=/usr/lib/bluetooth/bluetoothd
NotifyAccess=main
#WatchdogSec=10
Restart=on-failure
CapabilityBoundingSet=CAP_NET_ADMIN CAP_NET_BIND_SERVICE
LimitNPROC=1

...

[Install]
WantedBy=bluetooth.target
Alias=dbus-org.bluez.service
```

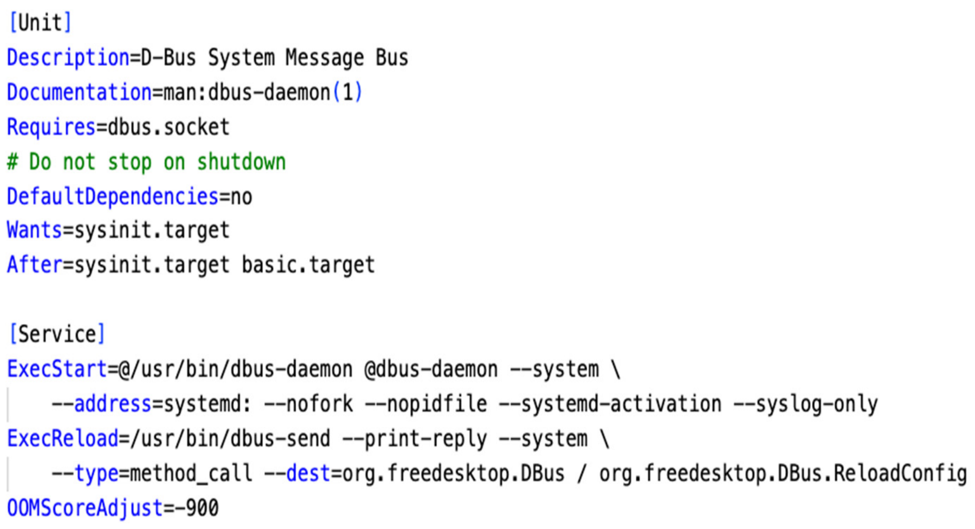
Figure 4. Simplified bluetooth.service unit file. In this figure, the syntax is shown in blue, while the comments are shown in green.

Figures 5 and 6 are the socket and service unit files for the D-Bus system message daemon. The socket unit creates a socket file placed in /run/dbus/system_bus_socket using the ListenStream variable. Systemd system services heavily use the D-Bus system message bus, so it depends on the sysinit.target. Note that the service type is not specified. Its service type is “simple”, not dbus, because it is the one that manages the D-Bus. The same is true for the session bus. The dbus.service depends on dbus.socket, and dbus-daemon contains the dbus, acting as a middleman to pass dbus messages from one to another. The out-of-memory killer’s tweak level is set to −900, which makes it very unlikely that OOM will kill the system bus dbus-daemon.



```
[Unit]
Description=D-Bus System Message Bus Socket
# Do not stop on shutdown
DefaultDependencies=no
Wants=sysinit.target
After=sysinit.target

[Socket]
ListenStream=/run/dbus/system_bus_socket
```

Figure 5. dbus.socket for system bus. In this figure, the syntax is shown in blue, while the comments are shown in green.



```
[Unit]
Description=D-Bus System Message Bus
Documentation=man:dbus-daemon(1)
Requires=dbus.socket
# Do not stop on shutdown
DefaultDependencies=no
Wants=sysinit.target
After=sysinit.target basic.target

[Service]
ExecStart=@/usr/bin/dbus-daemon @dbus-daemon --system \
    --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
ExecReload=/usr/bin/dbus-send --print-reply --system \
    --type=method_call --dest=org.freedesktop.DBus / org.freedesktop.DBus.ReloadConfig
OOMScoreAdjust=-900
```

Figure 6. dbus.service for system bus. In this figure, the syntax is shown in blue, while the comments are shown in green.

2.4. D-Bus

D-Bus is a message-oriented middleware (MOM) for client–server inter-process communication between multiple applications developed by freedesktop.org [51,52]. The freedesktop.org project, formerly the X Desktop Group (XDG), focuses on developing standards and specifications for desktop environments for the X Window System, such as GNOME and KDE. D-Bus eliminates the need to manage multiple one-to-one IPCs between processes by creating a virtual bus system. Each process sends and receives messages via the dbus-daemon, which routes these messages to the correct procedures.

There are two common types of buses: the system bus and the session bus. The system bus is for system services and is available to all users and processes. Session buses are for user services; there is a separate session bus for each user to have their desktop environment and user services. Although D-Bus is for desktop GUI components to communicate with the service daemon, some command-line tools, such as systemctl, journalctl, and bluetoothctl, also use the D-Bus interface to communicate with the service daemon.

BlueZ uses both the system bus and the session bus; the former, as shown in Figure 5, is for exposing the interface for managing Bluetooth adapters, pairing Bluetooth devices, running SDP, etc. The latter is for Bluetooth services such as OBEX; each user has a dedicated one.

1. Bus Name

Each connection to the dbus-daemon is assigned a unique connection name by the dbus-daemon, starting with a colon. For example, 1.13 is a valid, unique connection name. The server can register known names on the bus for the client to refer to, and the dbus-daemon will resolve the known names for the client when sending messages to it. An available name is concatenated by dots, such as com.example.Service1.

2. Object and Interface

A D-Bus object can have one or more interfaces, with each interface having methods, properties, and signals. An object is identified by its object path. An object path is in the form of "/path/to/this/object", where a single slash ("/") is also acceptable, and it is referred to as the "root" object. D-Bus has its own type system that supports some basic types, such as byte, boolean, int, uint, double, string, object path, array, dictionary, and Unix fd. There is also a variant type that acts like a struct. Each message contains a signature field that uses the D-Bus type code to describe the data it carries. freedesktop.org has specified four standard interfaces that objects can implement.

- org.freedesktop.DBus.Introspectable

This interface has an introspect method that returns an XML describing the object. A D-Bus Introspection XML contains the interfaces and corresponding messages that it has.

- org.freedesktop.DBus.Properties

This interface can have custom properties and three methods to obtain and set them, namely "Get", "Set", "GetAll", and a "PropertiesChanged" signal to notify other processes.

- org.freedesktop.DBus.ObjectManager

A server can implement this interface to list all the objects it exports to the bus, and it is usually implemented on the root object. This interface has a GetManagedObjects method and two signals, InterfacesAdded and InterfacesRemoved. This design is for easier client implementation. A client can subscribe to the InterfacesAdded signal, and all the property information can be abstracted from that signal message, eliminating the need to call Introspect.

- org.freedesktop.DBus.Peer

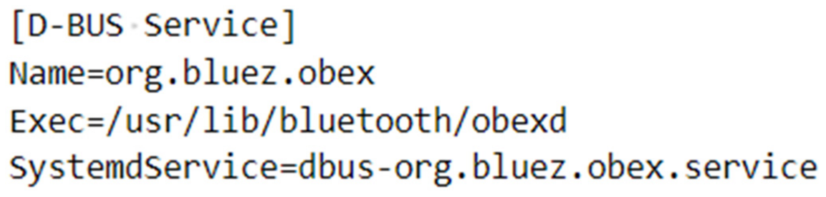
This interface has two methods, Ping and GetMachineId. "Ping" is used to check connections to the server. "GetMachineId" is used primarily when D-Bus runs over TCP/IP on different machines.

3. Message

There are five message types: INVALID, METHOD_CALL, METHOD_RETURN, ERROR, and SIGNAL. The properties interface handles the properties of these message types. A message contains sender, destination, path, interface, member, serial, and reply_serial to identify the message's source, destination, and purpose. The dbus-daemon controls the sender field, so it cannot be spoofed, while the sender contains the destination field and can be empty for sending messages such as signals. Method and signal names are specified in the member field.

4. D-Bus Service

Services that register a bus name on message buses can be run automatically by the dbus-daemon using a D-Bus service file placed in the "/usr/share/dbus-1/services/" folder, as shown in Figure 7. The binary is executed by the dbus-daemon when a process sends a message to a bus name specified in a service file, allowing systemd to manage the service through systemd's D-Bus interface.



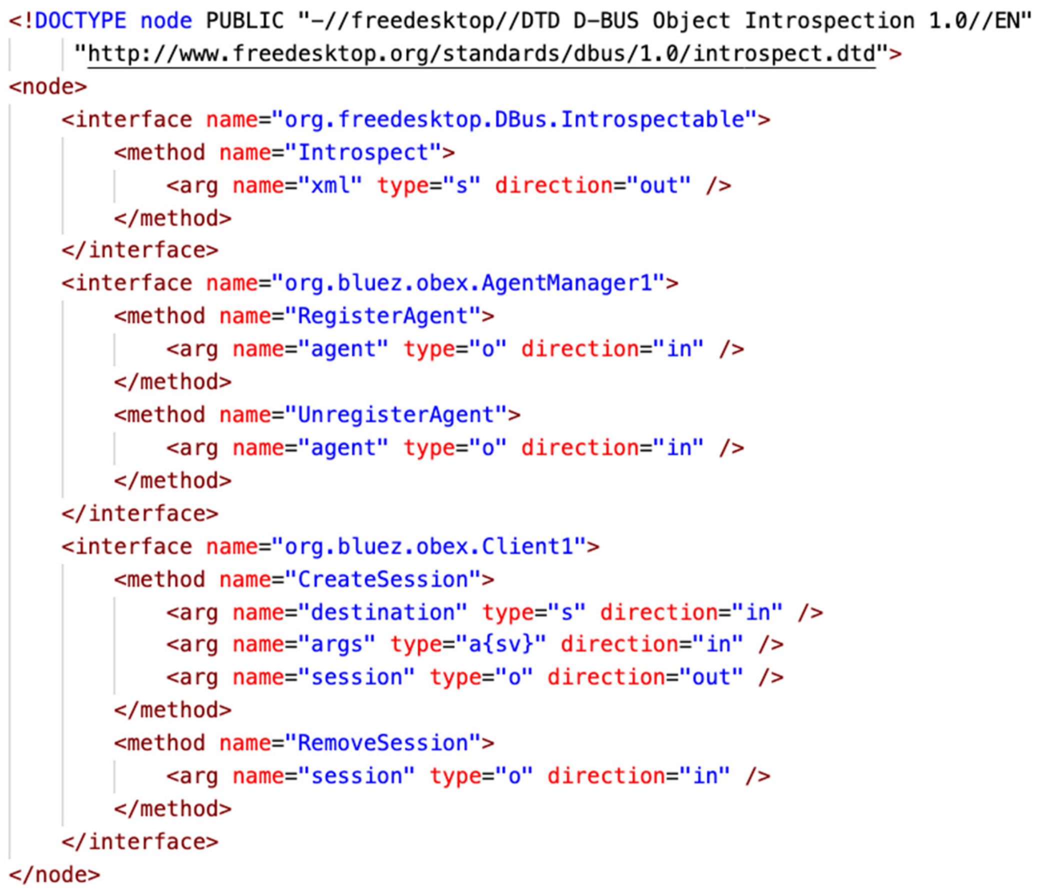
```
[D-BUS Service]
Name=org.bluez.obex
Exec=/usr/lib/bluetooth/obexd
SystemdService=dbus-org.bluez.obex.service
```

Figure 7. D-Bus service file for OBEX service.

2.5. OBEX and Object Push Profile

OBject EXchange (OBEX) [53–55] is a communication protocol for exchanging binary objects between devices. It uses the type–length–value binary format, which is easier for resource-constrained machines to parse. Bluetooth uses OBEX to implement the Object Push profile to send vCards (electronic business cards) and files to another device.

The BlueZ implementation uses an executable “obexd” that runs “obex.service” using the systemd user service. It registers a “com.bluez.obex” bus name on the session bus. It exports a “/com/bluez/obex” object that provides the interfaces for registering OBEX agents and the client interface that can create OBEX sessions in Figure 8.

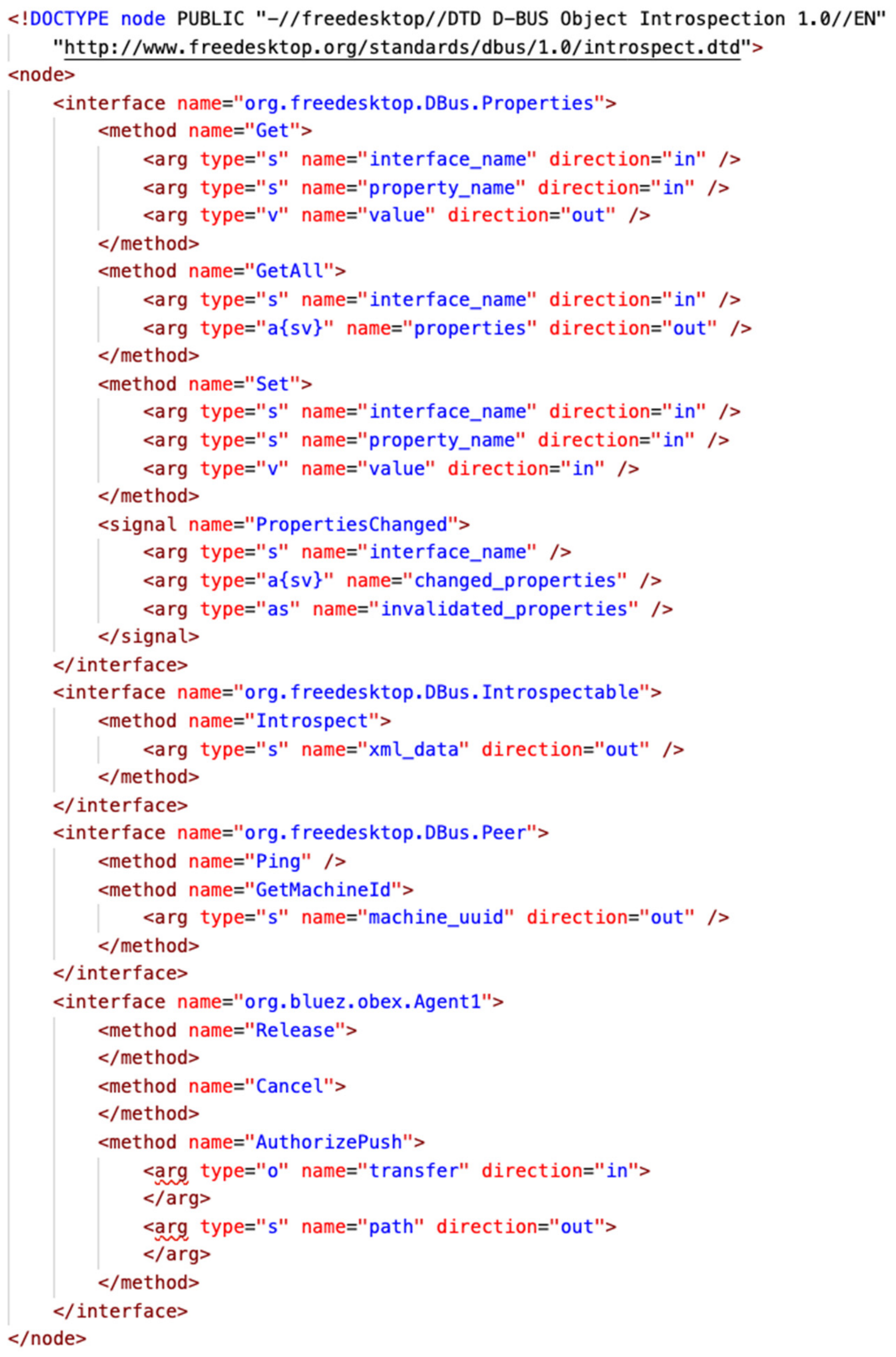


```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
    "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node>
    <interface name="org.freedesktop.DBus.Introspectable">
        <method name="Introspect">
            <arg name="xml" type="s" direction="out" />
        </method>
    </interface>
    <interface name="org.bluez.obex.AgentManager1">
        <method name="RegisterAgent">
            <arg name="agent" type="o" direction="in" />
        </method>
        <method name="UnregisterAgent">
            <arg name="agent" type="o" direction="in" />
        </method>
    </interface>
    <interface name="org.bluez.obex.Client1">
        <method name="CreateSession">
            <arg name="destination" type="s" direction="in" />
            <arg name="args" type="a{sv}" direction="in" />
            <arg name="session" type="o" direction="out" />
        </method>
        <method name="RemoveSession">
            <arg name="session" type="o" direction="in" />
        </method>
    </interface>
</node>
```

Figure 8. Interfaces of the D-Bus object “/com/bluez/obex”. The HTML tag is written in a dark red typeface; the attribute name and value are in red and blue, respectively.

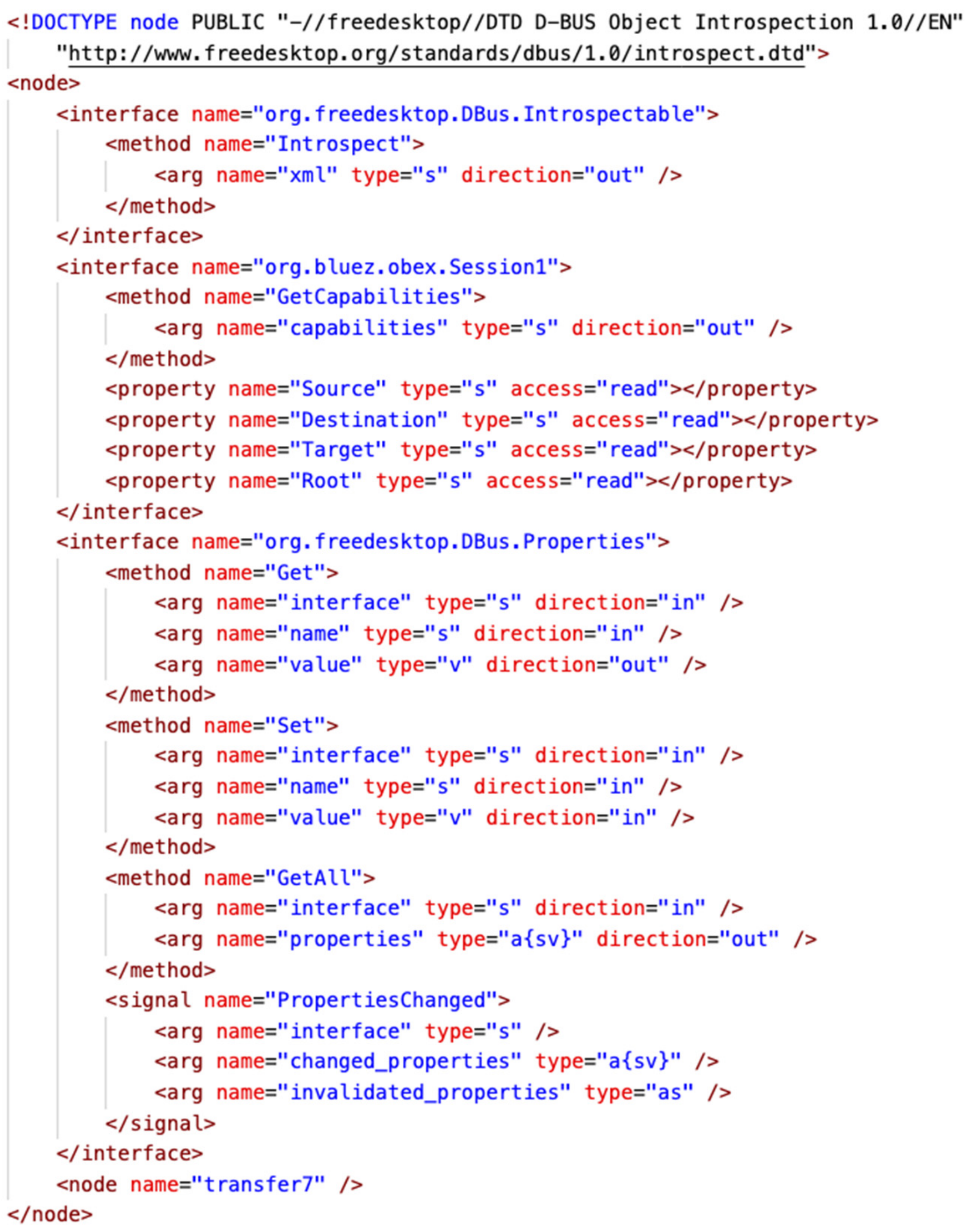
An application initiates an OBEX object push session by calling the CreateSession method on the org.bluez.obex.Client1 interface. Another application on a separate device registers an OBEX agent object that implements the org.bluez.obex.Agent1 interface in Figure 9 by calling the RegisterAgent method on the org.bluez.AgentManager1 interface in Figure 8.

When obexd receives a push request for an OBEX object, it first creates a session object, as shown in Figure 10, and a transfer object, as shown in Figure 11, then calls the AuthorizePush method of an existing OBEX agent to confirm whether to accept or reject the request. If the agent agrees with the recommendation, it returns a path where obexd will store the file. The object push starts the transfer of the file and simultaneously updates the properties of the transfer object to reflect the transfer status. On each update, the transfer object will fire a PropertiesChanged signal. The OBEX agent listens for this signal, waits for the file transfer to complete, and retrieves the file.



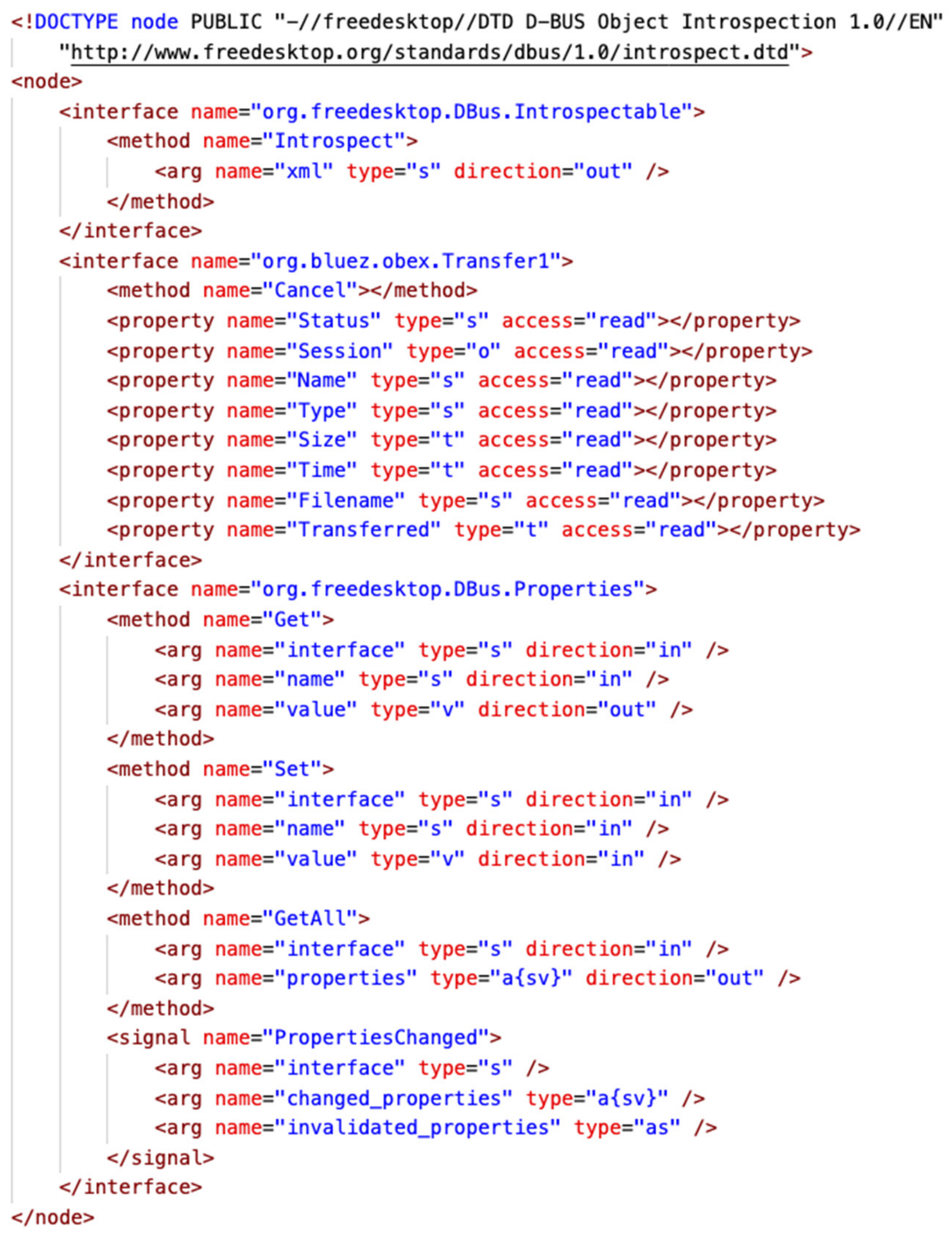
```xml
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
    "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node>
    <interface name="org.freedesktop.DBus.Properties">
        <method name="Get">
            <arg type="s" name="interface_name" direction="in" />
            <arg type="s" name="property_name" direction="in" />
            <arg type="v" name="value" direction="out" />
        </method>
        <method name="GetAll">
            <arg type="s" name="interface_name" direction="in" />
            <arg type="a{sv}" name="properties" direction="out" />
        </method>
        <method name="Set">
            <arg type="s" name="interface_name" direction="in" />
            <arg type="s" name="property_name" direction="in" />
            <arg type="v" name="value" direction="in" />
        </method>
        <signal name="PropertiesChanged">
            <arg type="s" name="interface_name" />
            <arg type="a{sv}" name="changed_properties" />
            <arg type="as" name="invalidated_properties" />
        </signal>
    </interface>
    <interface name="org.freedesktop.DBus.Introspectable">
        <method name="Introspect">
            <arg type="s" name="xml_data" direction="out" />
        </method>
    </interface>
    <interface name="org.freedesktop.DBus.Peer">
        <method name="Ping" />
        <method name="GetMachineId">
            <arg type="s" name="machine_uuid" direction="out" />
        </method>
    </interface>
    <interface name="org.bluez.obex.Agent1">
        <method name="Release">
        </method>
        <method name="Cancel">
        </method>
        <method name="AuthorizePush">
            <arg type="o" name="transfer" direction="in">
            </arg>
            <arg type="s" name="path" direction="out">
            </arg>
        </method>
    </interface>
</node>
```

Figure 9. Interfaces of a D-Bus OBEX agent object. The HTML tag is written in a dark red typeface; the attribute name and value are in red and blue, respectively.



```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
    "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node>
    <interface name="org.freedesktop.DBus.Introspectable">
        <method name="Introspect">
            <arg name="xml" type="s" direction="out" />
        </method>
    </interface>
    <interface name="org.bluez.obex.Session1">
        <method name="GetCapabilities">
            <arg name="capabilities" type="s" direction="out" />
        </method>
        <property name="Source" type="s" access="read"></property>
        <property name="Destination" type="s" access="read"></property>
        <property name="Target" type="s" access="read"></property>
        <property name="Root" type="s" access="read"></property>
    </interface>
    <interface name="org.freedesktop.DBus.Properties">
        <method name="Get">
            <arg name="interface" type="s" direction="in" />
            <arg name="name" type="s" direction="in" />
            <arg name="value" type="v" direction="out" />
        </method>
        <method name="Set">
            <arg name="interface" type="s" direction="in" />
            <arg name="name" type="s" direction="in" />
            <arg name="value" type="v" direction="in" />
        </method>
        <method name="GetAll">
            <arg name="interface" type="s" direction="in" />
            <arg name="properties" type="a{sv}" direction="out" />
        </method>
        <signal name="PropertiesChanged">
            <arg name="interface" type="s" />
            <arg name="changed_properties" type="a{sv}" />
            <arg name="invalidated_properties" type="as" />
        </signal>
    </interface>
    <node name="transfer7" />
</node>
```

Figure 10. Interfaces of a D-Bus OBEX session object. The HTML tag is written in a dark red typeface; the attribute name and value are in red and blue, respectively.



```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
    "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node>
    <interface name="org.freedesktop.DBus.Introspectable">
        <method name="Introspect">
            <arg name="xml" type="s" direction="out" />
        </method>
    </interface>
    <interface name="org.bluez.obex.Transfer1">
        <method name="Cancel"></method>
        <property name="Status" type="s" access="read"></property>
        <property name="Session" type="o" access="read"></property>
        <property name="Name" type="s" access="read"></property>
        <property name="Type" type="s" access="read"></property>
        <property name="Size" type="t" access="read"></property>
        <property name="Time" type="t" access="read"></property>
        <property name="Filename" type="s" access="read"></property>
        <property name="Transferred" type="t" access="read"></property>
    </interface>
    <interface name="org.freedesktop.DBus.Properties">
        <method name="Get">
            <arg name="interface" type="s" direction="in" />
            <arg name="name" type="s" direction="in" />
            <arg name="value" type="v" direction="out" />
        </method>
        <method name="Set">
            <arg name="interface" type="s" direction="in" />
            <arg name="name" type="s" direction="in" />
            <arg name="value" type="v" direction="in" />
        </method>
        <method name="GetAll">
            <arg name="interface" type="s" direction="in" />
            <arg name="properties" type="a{sv}" direction="out" />
        </method>
        <signal name="PropertiesChanged">
            <arg name="interface" type="s" />
            <arg name="changed_properties" type="a{sv}" />
            <arg name="invalidated_properties" type="as" />
        </signal>
    </interface>
</node>
```

Figure 11. Interfaces of a D-Bus OBEX transfer object. The HTML tag is written in a dark red typeface; the attribute name and value are in red and blue, respectively.

3. Related Work

Attack surfaces are exposed areas of a system or exploitable vulnerabilities that put a device at risk. The cyber-attack surface, human attack surface, and software attack surface are the three categories that make up the standard classification of attack surfaces. IoT malware attacks are not region-specific like other types of malwares. Therefore, in contrast to the wide variety, the attack surfaces are divided into network and network device level, service level, firmware level, and device level attack surfaces.

3.1. BlueBorne

BlueBorne [43,56] studies the Bluetooth attack surface, including Android, iOS, Windows, and Linux. BLE has several distinct locations where it is vulnerable. The security of each protocol layer varies, and a single flaw can put the entire system to a halt. The BlueBorne attack, which enables hackers to hijack a user's device without the owner's knowledge, has recently gained attention. A comprehensive and complex security model must be developed to mitigate these vulnerabilities. The foundation of the BlueBorne attack [43] is the exploitation of several vulnerabilities at various levels. Attackers can covertly take control of IoT devices, leak information, and remotely execute malicious code. Millions of devices are now under the control of hackers thanks to the BlueBorne assault, who may use these resources to create DDoS attacks that will be difficult to counter for some time. It uses eight CVEs across multiple platforms, including three remote code executions (RCEs), three information leaks, and two logical flaws, and turns them into exploits that can be turned into Bluetooth worms. In a follow-up presentation at Black Hat Europe, another information leak CVE was found that can be used with other CVEs to bypass KASLR and cause more damage to the system.

The army's lab team pointed out that the complexity of the Bluetooth specification makes it difficult to audit implementations. The major problem is multiple fragmentation mechanisms across various layers of the stack. Packages must be fragmented and reassembled numerous times to reach their destination.

3.2. Packet Filtering for BlueBorne

Seri and Vishnepolsky [43,57] check for malformed Bluetooth packets within three protocol layers of the Bluetooth protocol stack that BlueBorne uses to achieve remote code execution. It can achieve the goal of stopping BlueBorne, but this approach needs to make rules for each CVE. Manually creating these rules takes a lot of work in the long run.

With regard to BlueBorne flaws discovered in different Bluetooth stack levels, numerous vulnerabilities were discovered as Armis [43] Labs researchers examined how the Bluetooth layer is implemented in other operating systems. Under the term BlueBorne, all of these vulnerabilities were made public in 2017. L2CAP, SDP, SMP, and BNEP are the impacted Bluetooth layers [58]. Combining the BlueBorne flaws into a single assault, a hacker may take complete control of any Bluetooth-enabled device. The Bluetooth protocol specification's complexity may contribute to the high number of vulnerabilities. The vulnerable operating systems are Linux, Android, iOS, and Windows. However, fixes have already been released for these issues. Only devices with out-of-date OS versions are vulnerable now. Internet of Things (IoT) devices, desktop computers, and smartphones utilize the same operating systems. There are around 8.2 billion devices, including smartphones running Android or iOS, laptops running Linux or Windows, and Internet of Things (IoT) devices running Linux-based operating systems like the Tizen operating system. Attackers can penetrate air-gap networks utilized in locations where security is a top priority due to the broad spectral range of susceptible equipment.

3.3. LBM

Seri and Vishnepolsky [34,56,59] proposed a system for filtering malicious peripherals by applying firewall rules to the USB packets they send and receive. The system also includes a subsystem for filtering Bluetooth packets in several Bluetooth protocol layers, since Bluetooth peripherals can connect to the system via the HID interface. The system can inspect Bluetooth HCI layer packets because many Bluetooth controllers use USBs to connect to the system. Filtering malformed Bluetooth packets to protect the kernel from CVE vulnerabilities is one of the goals of this paper. For each peripheral subsystem, the LBM only needs to install a hook for incoming and outgoing peripheral data, after which modules can be created to filter specific peripheral packet types (such as USB request blocks or Bluetooth socket buffers). We use the Extended BSD Packet Filter (eBPF) mechanism, which allows filtering applications to load from user space, which is crucial for performance and scalability. Unlike previous solutions, LBM can be designed to provide a universal foundation for any peripheral protocol. Therefore, using LBM, it is easy to integrate existing solutions such as USBFILTER and USBFirewall. In addition, by incorporating modifications into the LBM core structure, it can easily handle new peripherals. We have developed hooks for the Bluetooth host control interface (HCI) and Logical Link and Adaptation Protocol (L2CAP) layers, and demonstrated the hook mechanism for the Bluetooth host control interface (HCI) and Logical Link and Adaptation Protocol (L2CAP) layers to show the adaptability and flexibility of the LBM framework. A near-field communication (NFC) protocol hook mechanism is a way to intercept and modify NFC messages as they are being transmitted between two devices. This can be used for various purposes, such as blocking malicious NFC messages, logging NFC traffic, or injecting custom NFC messages.

3.4. Avfs

Avfs [43,45] is a high-performance, portable, on-access stackable antivirus virtual file system. On-access scanning is an enhancement to onopen, onclose and on-exec scanning. The on-access scanner can prevent viruses from being written to the disk by scanning for viruses as the program reads or writes data. The user can avoid unexpected delays because the scan can only be performed when the data can be read, not when the file is opened. We have created an actual per-access virus scanning system called Avfs, a stacking file system. Avfs can be used with any other unmodified file system (such as Ext2 or NFS) and does not require the operating system to be modified because it is a stacking file system. For example, Windows clients can be transparently protected by Avfs when deployed over SMB. Avfs is a tool that can be used for general pattern matching and virus detection. It uses a modified version of ClamAV called "Oyster" for file scanning, which improves RAM usage over the original ClamAV and is embedded in the kernel for performance gains. Avfs takes a state-oriented approach to file scanning, allowing it to partially scan a file and resume scanning later until it is completely scanned. Scanned files are tagged to avoid unnecessary rescanning and are rescanned if modified.

4. Proposed Model

The proposed solution minimally perturbs the system. It does not require using a custom kernel or rebuilding and installing a modified BlueZ user space daemon. Instead, it inspects files transferred to devices through BlueZ's service daemon. Malicious files are removed from the system if transmitted through BlueZ's OBEX service.

4.1. System Design Principles

The intermediary entity, the BOP (broker of objects and protocols), occupies a central position within the intricate network of clients and the original OBEX (Object Exchange) service. Its purpose is to facilitate the seamless exchange of objects between these two entities, serving as a conduit for their communication. To accomplish this, the BOP performs the critical task of exporting objects from clients and the OBEX service. The creation and destruction of session and transfer objects are dynamic, with their instantiation and destruction triggered by receiving signals from the object managers. In addition to its intermediary role, the BOP also assumes responsibility for caching and modifying the temporary file location derived from the return of the AuthorizePush method before its transmission to the OBEX service. It performs this task with the utmost care and precision, ensuring the integrity and security of the exchanged files. To further enhance its functionality, the BOP uses a filtering mechanism activated by the occurrence of the "PropertiesChanged" signal within the transfer object. Before being passed to the standard session bus, this signal acts as a trigger for applying the filtering mechanism, ensuring that only the relevant and desired data are passed. The BOP file check handler, which carries the weight of critical responsibility, calculates the file's hash value. This estimated value is then sent to the renowned VirusTotal platform using the VirusTotal API, where a meticulous evaluation is performed to determine the presence of any malicious intent within the file being scanned. By employing this intricate and comprehensive framework, the BOP assumes its role as a crucial communication arbiter, ensuring the smooth and secure exchange of objects between clients and the OBEX service while maintaining the highest standards of reliability and effectiveness.

4.2. Bluetooth OBEX Proxy (BOP)

Without the user's knowledge, an attacker can connect to a mobile device and take control of personal information like a phonebook or calendar file. For instance, by connecting directly to a particular file containing a phonebook, the connection is established using Object Exchange (OBEX). The OBEX protocol creates an FTP connection and grants complete interactive access to the destination device's file system.

The Helemoto assault is comparable to the Bluebugging attack. It makes use of some phones' subpar "trusted device" management. Like the Bluebugging attack, the perpetrator poses as the sender of a virtual contact file (vCard) to a Bluetooth Object Exchange (OBEX) push profile that has not been authenticated on the victim's device. A Bluetooth device may send and receive objects (files) to and from other Bluetooth devices using the OBEX profile, which is part of the Bluetooth standard. The transfer procedure is interrupted once the attack starts, and the victim adds the attacker's phone to their trusted device list.

The BOP sits between the applications and the OBEX daemon via two session buses [10,60], as shown in Figure 12. Another dbus-daemon systemd service unit creates the obex-bus session bus and a socket unit described in Section 4.2.

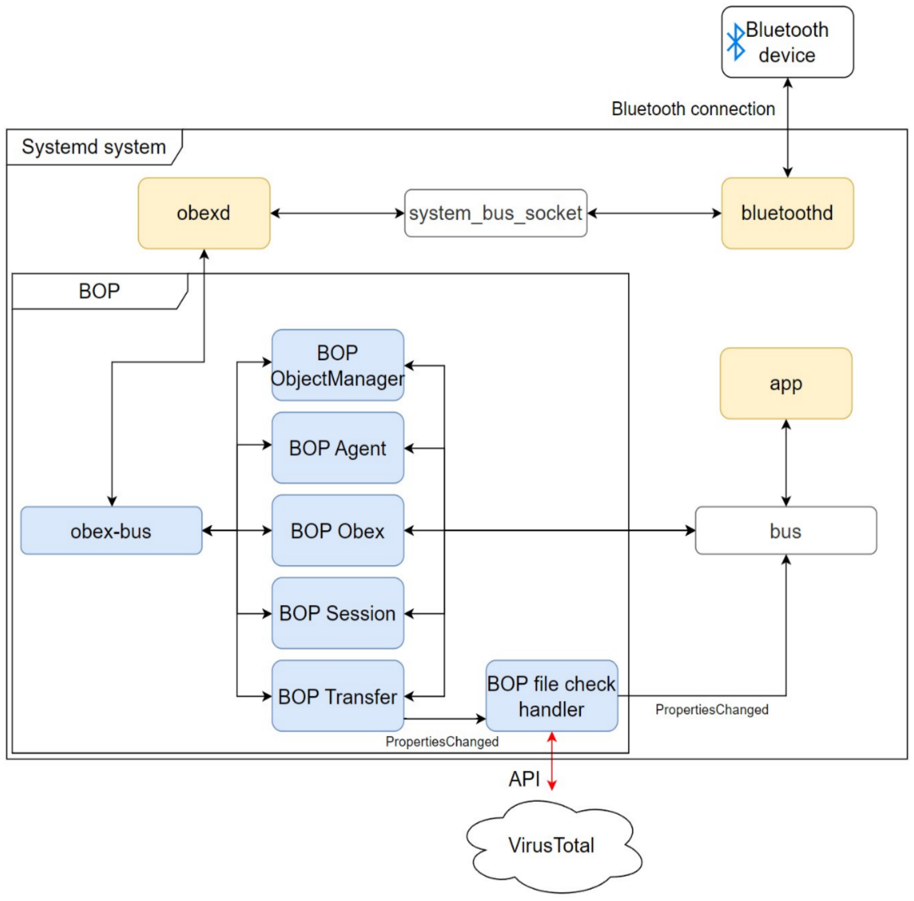


Figure 12. BOP system architecture.

1. Asynchronous Message Handling

The BOP implementation uses a new D-Bus Python package called dbus-next [61], which supports the Python async and Glib main loop. The first attempt to implement the proxy service using the legacy Python dbus package failed at the very last moment, when all other components were almost ready. The reason for the failure is that while proxying the "AuthorizePush" method call to the obex agent, the agent calls back to it for additional file transfer information, as shown in Figure 13. The handler is still in the event loop waiting for the "AuthorizePush" response from the OBEX agent, so the event loop cannot pick up the next task and cannot proxy the method call from the OBEX agent. Finally, the two method calls are timed out, and the object push is aborted.

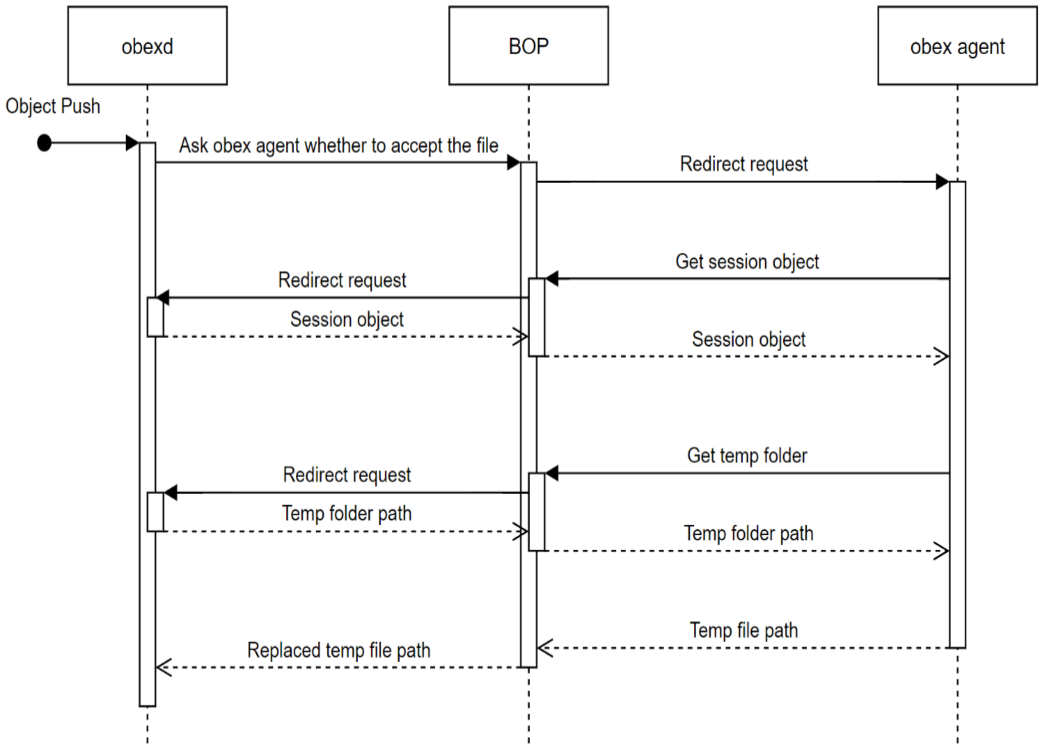


Figure 13. Sequence diagram for handling the AuthorizePush method call.

2. Message Redirection

The proxying action is carried out by changing the messages' sender, destination, serial, and reply_serial and forwarding them to the other bus, as shown in Figure 14. It allows the BOP to work without implementing all the methods and signals for all the objects. We only need to implement the object handlers for the ways we want to hook and modify the response messages. When hooking a method call, a callback function is created to intercept the response from obexd. The callback function can export new objects on a message bus or manipulate the response before passing it to the other side.

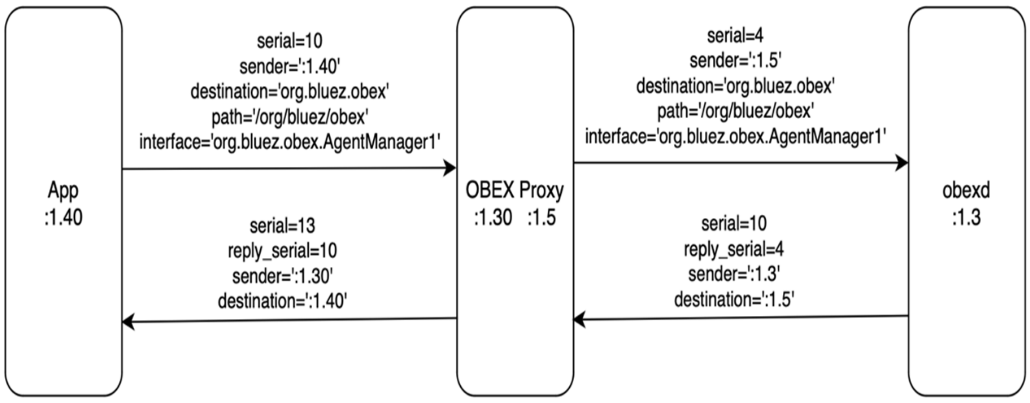


Figure 14. Proxying messages between two message buses.

3. Object Creation

An application's OBEX agent object must be proxied to the OBEX bus session bus, while the session object and transfer object must be proxied to the standard session bus. These objects are created as needed and removed when they have served their purpose. A new OBEX agent object is exported or released on the obex-bus session bus by the BOP AgentManager when the RegisterAgent or UnregisterAgent method call to obexd returns success. As for session objects and transfer objects, they are exported or removed on the default session bus by the BOP ObjectManager when it receives an InterfacesAdded or InterfacesRemoved signal before forwarding the call.

4. Filter Mechanism

If a PropertiesChanged signal indicates that the transfer is competitive, either by transfer size or transfer status, it calls the BOP file check handler and does not pass the signal to the default session bus. The file check handler always passes the signal to the default session bus when the check is complete. However, the file is copied to the original temp file location only if it passes the file check. Otherwise, the file is deleted.

5. Custom Systemd Service and D-Bus Service

When an application makes a method call to org.bluez.obex, the dbus-daemon first checks to see if a process already registers the name. If a process does not own the name, the dbus-daemon checks for D-Bus service files that specify that name. The obexd is then run by the dbus-daemon and passed to the systemd daemon, which will then manage the systemd user service.

(a) obex-bus.service and obex-bus.socket

The two files create a new session bus placed in the XDG_RUNTIME_DIR. It allows the original obex.service to talk to the BOP service.

(b) obex.service

This service unit file overrides the original service file. It changes the DBUS_SESSION_BUS_ADDRESS environment variable to point to the newly created obex-bus socket file so that obexd connects to this message bus instead of the default session bus. This service also removed the alias in the [Install] section used by the OBEX D-BUS service.

(c) dbus-org.bluez.obex.service

The original obex.service creates an alias service called dbus-org.bluez.obex.service, which is used by the OBEX D-Bus service to associate the process with systemd. This service file runs the BOP service and requires both obex.service and obex-bus.service.

5. Evaluation

The experiment setup uses a server running the proxy service and a client using the OBEX object push profile to send a file to the server. The two virtual machines (VMs) run inside a Proxmox VE server, as shown in Tables 2 and 3.

Table 2. Server running OBEX proxy.

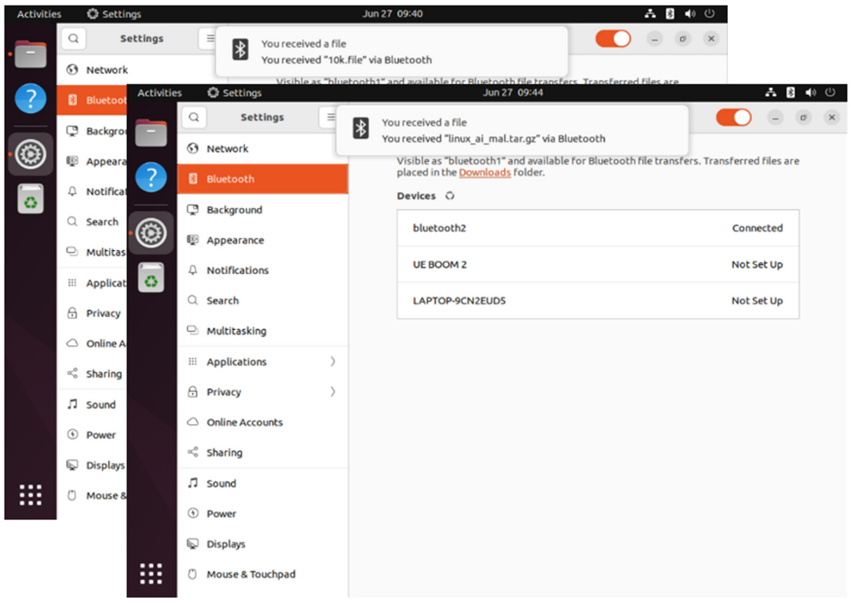
Server VM	
vCPU	2 cores
RAM	4 GB
Bluetooth	MediaTek Inc. mt7921e
OS	Ubuntu 22.04.2 LTS
Kernel	5.15.0-75-generic

Table 3. Client.

Client VM	
vCPU	2 cores
RAM	4 GB
Bluetooth	Cambridge Silicon Radio, Ltd. Bluetooth Dongle (USB)
OS	Ubuntu 22.04.2 LTS
Kernel	5.15.0-75-generic

5.1. Functional Testing

The system is tested with and without the proxy service using a normal file (10k.file) and a malicious file (linux_ai_mal.tar.gz). A normal file should be successfully transferred without any problems, while a malicious file will result in an empty file of size zero. Figures 15 and 16 show successful transfers with and without the BOP. Figure 17 shows that the file is empty when a malicious file is transferred through the BOP.


Jun 27 09:40

You received a file
You received "10k.file" via Bluetooth

Jun 27 09:44

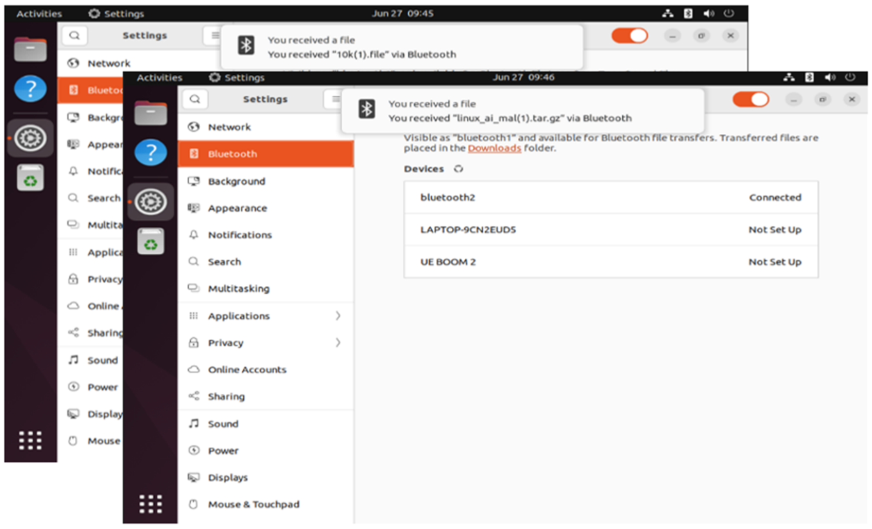
You received a file
You received "linux_ai_mal.tar.gz" via Bluetooth

Visible as "bluetooth1" and available for Bluetooth file transfers. Transferred files are placed in the Downloads folder.

Devices

bluetooth2	Connected
UE BOOM 2	Not Set Up
LAPTOP-9CN2EUD5	Not Set Up

Figure 15. Object pushes without BOP.


Jun 27 09:45

You received a file
You received "10k(1).file" via Bluetooth

Jun 27 09:46

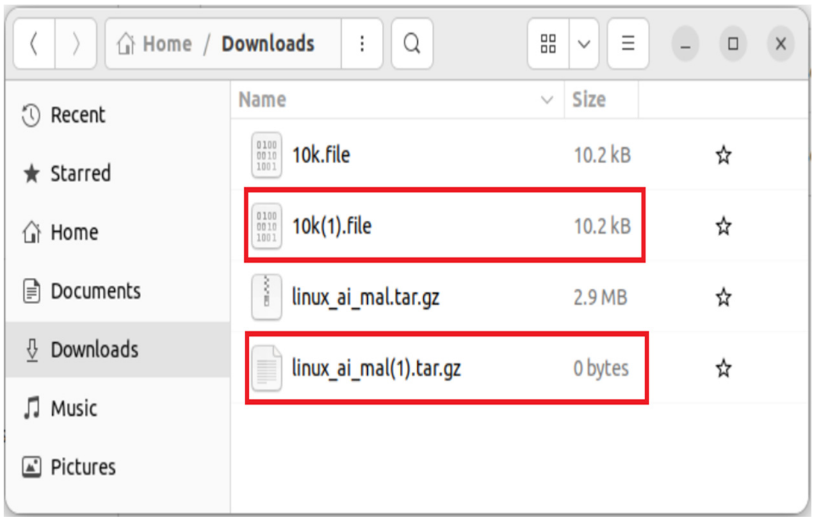
You received a file
You received "linux_ai_mal(1).tar.gz" via Bluetooth

Visible as "bluetooth1" and available for Bluetooth file transfers. Transferred files are placed in the Downloads folder.

Devices

bluetooth2	Connected
LAPTOP-9CN2EUD5	Not Set Up
UE BOOM 2	Not Set Up

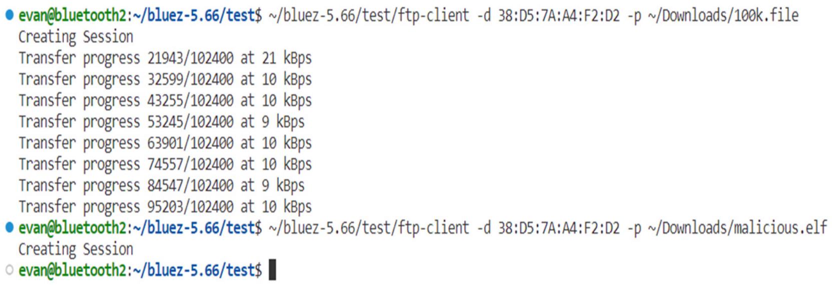
Figure 16. Object pushes with BOP.



Name	Size
10k.file	10.2 kB
10k(1).file	10.2 kB
linux_ai_mal.tar.gz	2.9 MB
linux_ai_mal(1).tar.gz	0 bytes

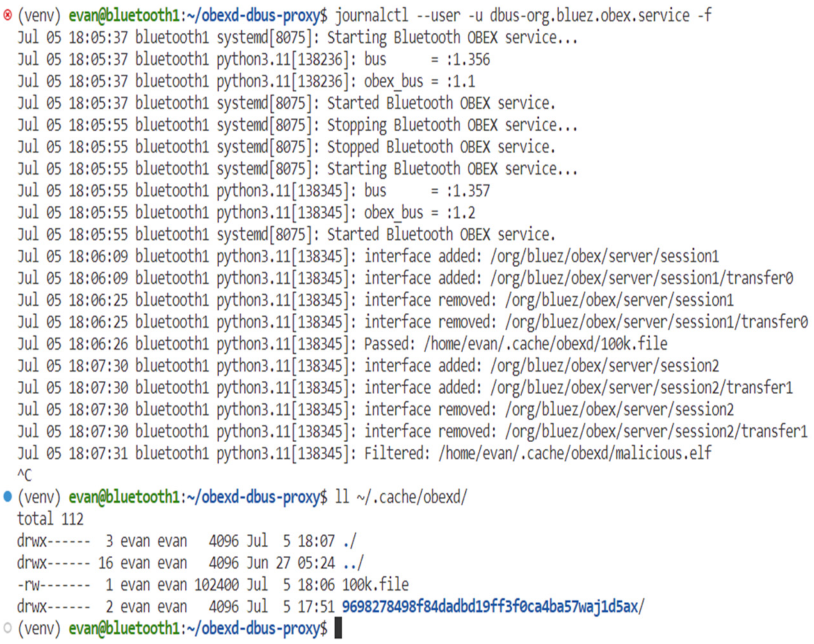
Figure 17. OPP transfer results with and without BOP. The red square are with BOP.

FTP is tested using the FTP client Python script provided by BlueZ. A standard file (100k.file) and a malicious file (malicious.elf) are sent to another Bluetooth device in Figure 18. Figure 19 shows the service log of the BOP, which shows that 100k.file passed the filter while malicious.elf did not. Below the service log is a "ls" command output showing that only 100k.file exists.



```
● evan@bluetooth2:~/bluez-5.66/test$ ~/bluez-5.66/test/ftp-client -d 38:D5:7A:A4:F2:D2 -p ~/Downloads/100k.file
  Creating Session
  Transfer progress 21943/102400 at 21 kBps
  Transfer progress 32599/102400 at 10 kBps
  Transfer progress 43255/102400 at 10 kBps
  Transfer progress 53245/102400 at 9 kBps
  Transfer progress 63901/102400 at 10 kBps
  Transfer progress 74557/102400 at 10 kBps
  Transfer progress 84547/102400 at 9 kBps
  Transfer progress 95203/102400 at 10 kBps
● evan@bluetooth2:~/bluez-5.66/test$ ~/bluez-5.66/test/ftp-client -d 38:D5:7A:A4:F2:D2 -p ~/Downloads/malicious.elf
  Creating Session
○ evan@bluetooth2:~/bluez-5.66/test$ █
```

Figure 18. File Transfer with and without BOP.

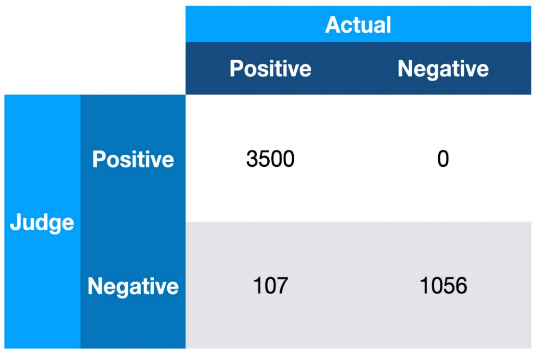


```
⊗ (venv) evan@bluetooth1:~/obexd-dbus-proxy$ journalctl --user -u dbus-org.bluez.obex.service -f
  Jul 05 18:05:37 bluetooth1 systemd[8075]: Starting Bluetooth OBEX service...
  Jul 05 18:05:37 bluetooth1 python3.11[138236]: bus      = :1.356
  Jul 05 18:05:37 bluetooth1 python3.11[138236]: obex_bus = :1.1
  Jul 05 18:05:37 bluetooth1 systemd[8075]: Started Bluetooth OBEX service.
  Jul 05 18:05:55 bluetooth1 systemd[8075]: Stopping Bluetooth OBEX service...
  Jul 05 18:05:55 bluetooth1 systemd[8075]: Stopped Bluetooth OBEX service.
  Jul 05 18:05:55 bluetooth1 systemd[8075]: Starting Bluetooth OBEX service...
  Jul 05 18:05:55 bluetooth1 python3.11[138345]: bus      = :1.357
  Jul 05 18:05:55 bluetooth1 python3.11[138345]: obex_bus = :1.2
  Jul 05 18:05:55 bluetooth1 systemd[8075]: Started Bluetooth OBEX service.
  Jul 05 18:06:09 bluetooth1 python3.11[138345]: interface added: /org/bluez/obex/server/session1
  Jul 05 18:06:09 bluetooth1 python3.11[138345]: interface added: /org/bluez/obex/server/session1/transfer0
  Jul 05 18:06:25 bluetooth1 python3.11[138345]: interface removed: /org/bluez/obex/server/session1
  Jul 05 18:06:25 bluetooth1 python3.11[138345]: interface removed: /org/bluez/obex/server/session1/transfer0
  Jul 05 18:06:26 bluetooth1 python3.11[138345]: Passed: /home/evan/.cache/obexd/100k.file
  Jul 05 18:07:30 bluetooth1 python3.11[138345]: interface added: /org/bluez/obex/server/session2
  Jul 05 18:07:30 bluetooth1 python3.11[138345]: interface added: /org/bluez/obex/server/session2/transfer1
  Jul 05 18:07:30 bluetooth1 python3.11[138345]: interface removed: /org/bluez/obex/server/session2
  Jul 05 18:07:30 bluetooth1 python3.11[138345]: interface removed: /org/bluez/obex/server/session2/transfer1
  Jul 05 18:07:31 bluetooth1 python3.11[138345]: Filtered: /home/evan/.cache/obexd/malicious.elf
  ^C
● (venv) evan@bluetooth1:~/obexd-dbus-proxy$ ll ~/.cache/obexd/
  total 112
  drwx------  3 evan evan   4096 Jul  5 18:07 ./
  drwx------ 16 evan evan   4096 Jun 27 05:24 ../
  -rw-------  1 evan evan 102400 Jul  5 18:06 100k.file
  drwx------  2 evan evan   4096 Jul  5 17:51 9698278498f84dadbd19ff3f0ca4ba57waj1d5ax/
○ (venv) evan@bluetooth1:~/obexd-dbus-proxy$ █
```

Figure 19. FTP transfer results with and without BOP.

5.2. Accuracy

The accuracy of the BOP was tested with 3607 malicious files downloaded from Bazaar's daily submissions [62] that are smaller than 100 KB in size, as well as 1056 files from BlueZ's GitHub repository source files [63]. Files with any "malicious" or "suspicious" records from VirusTotal are counted as malicious, while others are non-malicious. One hundred and seven malicious files passed the BOP filter, while the BOP blocked none of the normal files in Figure 20. Hence, the false positive rate of the BOP is zero, and the false negative rate of the BOP is 2.29%. The 107 false negatives are discussed in the last section. In the accuracy section, we say that there are 107 false positives. Later, we uploaded the 107 files to VirusTotal to see if the result changed. Two files were marked as malicious, and the remaining 105 files are still marked as clean by VirusTotal. Among the 105 files, 58 have community comments in the VirusTotal database. Fifty of them are from Nextron Systems' THOR APT scanner. Most of them are PE files with malformed or corrupted headers. One file from FileScan.IO was marked as clean, as well as two from Malshare, two from Joe Sandbox Analysis, one with suspicious comments, one with malicious words, and one with size 0.

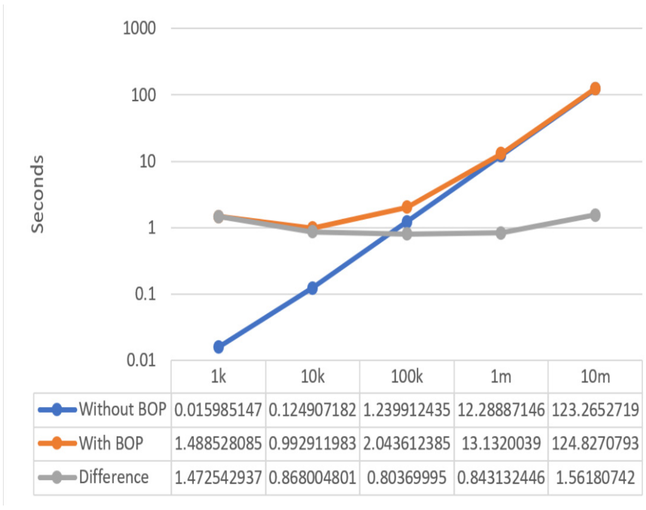


		Actual	
		Positive	Negative
Judge	Positive	3500	0
	Negative	107	1056

Figure 20. BOP accuracy.

5.3. Performance

The performance of the BOP was tested using five different files of 1 KB, 10 KB, 100 KB, 1 MB, and 10 MB in size, created using the truncate command in Figure 21. Each file was tested ten times to calculate the average transfer time. The Bluetooth connection of the two test VMs was already established to avoid measuring the time to develop and disconnect the Bluetooth connection before and after the OBEX protocol was actually executed. The time was calculated from when an application received the AuthorizePush method call to when it received the signal indicating that the transfer was complete.



	1k	10k	100k	1m	10m
Without BOP	0.015985147	0.124907182	1.239912435	12.28887146	123.2652719
With BOP	1.488528085	0.992911983	2.043612385	13.1320039	124.8270793
Difference	1.472542937	0.868004801	0.80369995	0.843132446	1.56180742

Figure 21. BOP performance test with different file sizes.

The performance difference comes from the time it takes to receive a response from the VirusTotal API after making an API call. The request time has a significant impact on the performance of the BOP.

5.4. Functionality Comparisons

The BOP was implemented in the Linux operating system kernel. One of the most well-known antivirus programs for Linux is ClamAV. Linux machines cannot have their file systems automatically scanned by ClamAV. Therefore, the BOP can identify malware as soon as it is downloaded to a Linux computer over Bluetooth. However, ClamAV will find the infection in the subsequent scanning cycle. And it can take several hours to scan the entire file system. Users of ClamAV must specify to ClamAV which files in which directories they should repeatedly check to avoid the issue mentioned above. Benign files that have already been examined in the guides must be scanned again by ClamAV. Finally, VirusTotal, which has access to more than 60 viral signature databases, is used by the BOP. However, ClamAV only has one database of signatures. Therefore, the BOP is better than ClamAV at identifying Bluetooth malware (Table 4).

Table 4. Functionality comparisons between the BOP and ClamAV. In this table, V means YES. X represents NO, and # is "the number of".

	Automatically Scan	# of Signature Databases Used	Need to Configure Directory
BOP	X	60+	X
ClamAV	V	1	V

6. Conclusions

The novelty of the BOP approach lies in the fact that it is the first proposed centralized proxy to filter malicious files transferred through BlueZ's OBEX system service. Traditional approaches usually require each device to run its security software, which makes the device's security dependent on the device manufacturer's security capabilities. The BOP approach improves security and reliability by centralizing security checks in a single agent. The BOP approach can improve the security of Bluetooth devices by impacting the following areas, reducing the risk of Bluetooth devices being attacked by malicious files. We created a proxy service called the BOP on the D-Bus message bus to intercept all messages between the server and client processes of the BlueZ OBEX service. The messages proxied by the BOP are then used to filter malicious files transferred to the device using the VirusTotal APIs. To further restrict access to temporary files, create a dedicated user account to run a centralized OBEX service for each user on the system. It will allow for more granular control over who has access to temporary files and will help mitigate the risk of unauthorized access. A proxy can be also created to route requests between two message buses. It can be used to improve performance and security by ensuring that requests are routed through a secure channel. It is a proxy that can be also created between Bluetooth and obexd daemons. It can be used to intercept the sockets passed to obexd, and to filter packets based on their content. It can help to prevent malicious packets from being sent to obexd, and can help to protect the system from attack. Finally, a proxy can be used to deny or allow access to services for which users originally have or do not have permissions. It can implement a more granular access control policy and help protect the system from unauthorized access.

Author Contributions: Conceptualization, F.-H.H. and J.-X.C.; Methodology, F.-H.H.; Software, J.-H.H. and Y.-W.L.; Validation, H.-J.W.; Formal analysis, H.-J.W. and Y.-W.L.; Data curation, J.-H.H.; Writing—original draft, M.-H.W.; Writing—review & editing, M.-H.W. and Y.-L.H.; Visualization, J.-X.C.; Project administration, F.-H.H. and M.-H.W. All authors have read and agreed to the published version of the manuscript.

Funding: Ministry of Science and Technology of Taiwan under no. MOST 111-2221-E-008-080-MY3.

Data Availability Statement: The data supporting this study's findings are available from the corresponding author upon reasonable request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. González, G.; Lárraga, M.E.; Alvarez-Icaza, L.; Gomez, J. Bluetooth worm propagation in smartphones: Modeling and analyzing spatio-temporal dynamics. *IEEE Access* **2021**, *9*, 75265–75282. [CrossRef]
2. Nallusamy, T.; Ravi, R. Investigation on cybernetic worm propagation in Bluetooth enabled devices. *Caribb. J. Sci.* **2022**, *52*, 1450–1460.
3. Ghillani, D.; Gillani, D.H. A perspective study on Malware detection and protection, A review. *Authorea* **2022**. *preprints*. [CrossRef]
4. Mahboubi, A.; Camtepe, S.; Ansari, K. Stochastic modeling of IoT botnet spread: A short survey on mobile malware spread modeling. *IEEE Access* **2020**, *8*, 228818–228830. [CrossRef]
5. Carettoni, L.; Merloni, C.; Zanero, S. Studying bluetooth malware propagation: The bluebag project. *IEEE Secur. Priv.* **2007**, *5*, 17–25. [CrossRef]
6. Podhradsky, A.L.; Casey, C.; Ceretti, P. The Bluetooth honeypot project. In Proceedings of the Wireless Telecommunications Symposium 2012, London, UK, 18–20 April 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 1–10.
7. Hassan, S.S.; Bibon, S.D.; Hossain, M.S.; Atiquzzaman, M. Security threats in Bluetooth technology. *Comput. Secur.* **2018**, *74*, 308–322. [CrossRef]
8. Dunning, J. Taming the blue beast: A survey of bluetooth based threats. *IEEE Secur. Priv.* **2010**, *8*, 20–27. [CrossRef]
9. Albahar, M.A.; Haataja, K.; Toivanen, P. Bluetooth MITM vulnerabilities: A literature review, novel attack scenarios, novel countermeasures, and lessons learned. *Int. J. Inf. Technol. Secur.* **2016**, *8*, 25–49.
10. Haataja, K.; Hyppönen, K.; Pasanen, S.; Toivanen, P. MITM attacks on Bluetooth. In *Bluetooth Security Attacks: Comparative Analysis, Attacks, and Countermeasures*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 61–70.
11. Sandhya, S.; Devi, K.S. Contention for man-in-the-middle attacks in Bluetooth networks. In Proceedings of the 2012 Fourth International Conference on Computational Intelligence and Communication Networks, Mathura, India, 3–5 November 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 700–703.
12. Haataja, K.; Hypponen, K.; Toivanen, P. Ten years of bluetooth security attacks: Lessons learned. In *Computer Science I Like*; University of Eastern Finland: Kuopio, Finland, 2011; p. 45.
13. Minar, N.B.-N.I.; Tarique, M. Bluetooth security threats and solutions: A survey. *Int. J. Distrib. Parallel Syst.* **2012**, *3*, 127. [CrossRef]
14. Wang, Y.; Wen, S.; Xiang, Y.; Zhou, W. Modeling the propagation of worms in networks: A survey. *IEEE Commun. Surv. Tutor.* **2013**, *16*, 942–960. [CrossRef]
15. Zou, C.C.; Towsley, D.; Gong, W. Modeling and simulation study of the propagation and defense of internet e-mail worms. *IEEE Trans. Dependable Secur. Comput.* **2007**, *4*, 105–118. [CrossRef]
16. Su, J.; Chan, K.K.W.; Miklas, A.G.; Po, K.; Akhavan, A.; Saroiu, S.; de Lara, E.; Goel, A. A preliminary investigation of worm infections in a bluetooth environment. In Proceedings of the 4th ACM Workshop on Recurring Malcode, Alexandria, VA, USA, 3 November 2006; pp. 9–16.
17. Yan, G.; Eidenbenz, S. Modeling propagation dynamics of bluetooth worms (extended version). *IEEE Trans. Mob. Comput.* **2008**, *8*, 353–368. [CrossRef]
18. Mickens, J.W.; Noble, B.D. Modeling epidemic spreading in mobile environments. In Proceedings of the 4th ACM Workshop on Wireless Security, Cologne, Germany, 2 September 2005; pp. 77–86.
19. Morris-King, J.R.; Cam, H. Controlling proximity-malware infection in diverse tactical mobile networks using K-distance pruning. In Proceedings of the MILCOM 2016—2016 IEEE Military Communications Conference, Baltimore, MD, USA, 1–3 November 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 503–508.
20. Zyba, G.; Voelker, G.M.; Liljenstam, M.; Méhes, A.; Johansson, P. Defending mobile phones from proximity malware. In Proceedings of the IEEE INFOCOM 2009, Rio de Janeiro, Brazil, 19–25 April 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 1503–1511.
21. Yang, Y.; Zhu, S.; Cao, G. Improving sensor network immunity under worm attacks: A software diversity approach. In Proceedings of the 9th ACM International Symposium on Mobile ad hoc Networking and Computing, Hong Kong, China, 26–30 May 2008; pp. 149–158.
22. Li, F.; Yang, Y.; Wu, J. CPMC: An efficient proximity malware coping scheme in smartphone-based mobile networks. In Proceedings of the 2010 Proceedings IEEE INFOCOM, San Diego, CA, USA, 14–19 March 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 1–9.
23. Miklas, A.G.; Gollu, K.K.; Chan, K.K.; Saroiu, S.; Gummadi, K.P.; De Lara, E. Exploiting social interactions in mobile systems. In Proceedings of the International Conference on Ubiquitous Computing, Tyrol, Austria, 16–19 September 2007; Springer: Berlin/Heidelberg, Germany, 2007; pp. 409–428.
24. Gao, C.; Liu, J. Modeling and restraining mobile virus propagation. *IEEE Trans. Mob. Comput.* **2012**, *12*, 529–541. [CrossRef]

25. Fleizach, C.; Liljenstam, M.; Johansson, P.; Voelker, G.M.; Mehes, A. Can you infect me now? Malware propagation in mobile phone networks. In Proceedings of the 2007 ACM Workshop on Recurring Malcode, Alexandria, VA, USA, 2 November 2007; pp. 61–68.
26. Meng, X.; Zerfos, P.; Samanta, V.; Wong, S.H.; Lu, S. Analysis of the reliability of a nationwide short message service. In Proceedings of the IEEE INFOCOM 2007—26th IEEE International Conference on Computer Communications, Anchorage, AK, USA, 6–12 May 2007; IEEE: Piscataway, NJ, USA, 2007; pp. 1811–1819.
27. Bose, A.; Hu, X.; Shin, K.G.; Park, T. Behavioral detection of malware on mobile handsets. In Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services, Breckenridge, CO, USA, 17–20 June 2008; pp. 225–238.
28. Zhu, Z.; Cao, G.; Zhu, S.; Ranjan, S.; Nucci, A. A social network based patching scheme for worm containment in cellular networks. In *Handbook of Optimization in Complex Networks: Communication and Social Networks*; Springer: New York, NY, USA, 2012; pp. 505–533.
29. Zhao, D.; Wang, L.; Wang, Z.; Xiao, G. Virus propagation and patch distribution in multiplex networks: Modeling, analysis, and optimal allocation. *IEEE Trans. Inf. Forensics Secur.* **2018**, *14*, 1755–1767. [CrossRef]
30. Zhang, X.; Cao, G. Transient community detection and its application to data forwarding in delay tolerant networks. *IEEE/ACM Trans. Netw.* **2017**, *25*, 2829–2843. [CrossRef]
31. Lu, Z.; Sun, X.; Wen, Y.; Cao, G.; La Porta, T. Algorithms and applications for community detection in weighted networks. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *26*, 2916–2926. [CrossRef]
32. Peng, S.; Wu, M.; Wang, G.; Yu, S. Containing smartphone worm propagation with an influence maximization algorithm. *Comput. Netw.* **2014**, *74*, 103–113. [CrossRef]
33. Yang, W.; Wang, H.; Yao, Y. An immunization strategy for social network worms based on network vertex influence. *China Commun.* **2015**, *12*, 154–166. [CrossRef]
34. Wu, J.; Wu, R.; Antonioli, D.; Payer, M.; Tippenhauer, N.O.; Xu, D.; Tian, D.; Bianchi, A. {LIGHTBLUE}: Automatic {Profile-Aware} Debloating of Bluetooth Stacks. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; pp. 339–356.
35. Vasan, D.; Alazab, M.; Venkatraman, S.; Akram, J.; Qin, Z. MTHAEL: Cross-architecture IoT malware detection based on neural network advanced ensemble learning. *IEEE Trans. Comput.* **2020**, *69*, 1654–1667. [CrossRef]
36. Huda, S.; Miah, S.; Yearwood, J.; Alyahya, S.; Al-Dossari, H.; Doss, R. A malicious threat detection model for cloud assisted internet of things (CoT) based industrial control system (ICS) networks using deep belief network. *J. Parallel Distrib. Comput.* **2018**, *120*, 23–31. [CrossRef]
37. Parra, G.D.L.T.; Rad, P.; Choo, K.-K.R.; Beebe, N. Detecting Internet of Things attacks using distributed deep learning. *J. Netw. Comput. Appl.* **2020**, *163*, 102662. [CrossRef]
38. De Donno, M.; Dragoni, N.; Giaretta, A.; Spognardi, A. Analysis of DDoS-capable IoT malwares. In Proceedings of the 2017 Federated Conference on Computer Science and Information Systems (FedCSIS), Prague, Czech Republic, 3–6 September 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 807–816.
39. Hallman, R.; Bryan, J.; Palavicini, G.; Divita, J.; Romero-Mariona, J. IoDDoS-the internet of distributed denial of sevice attacks. In Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security, Porto, Portugal, 24–26 April 2017; Scitepress: Setúbal, Portugal, 2017; pp. 47–58.
40. Shobana, M.; Rathi, S. Iot malware: An analysis of iot device hijacking. *Int. J. Sci. Res. Comput. Sci. Comput. Eng. Inf. Technol.* **2018**, *3*, 2456–3307.
41. Vignau, B.; Khoury, R.; Hallé, S. 10 years of IoT malware: A feature-based taxonomy. In Proceedings of the 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Sofia, Bulgaria, 22–26 July 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 458–465.
42. Vignau, B.; Khoury, R.; Hallé, S.; Hamou-Lhadj, A. The evolution of IoT Malwares, from 2008 to 2019: Survey, taxonomy, process simulator and perspectives. *J. Syst. Archit.* **2021**, *116*, 102143. [CrossRef]
43. Almiani, M.; Razaque, A.; Yimu, L.; Minjie, T.; Alweshah, M.; Atiewi, S. Bluetooth application-layer packet-filtering for blueborne attack defending. In Proceedings of the 2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC), Rome, Italy, 10–13 June 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 142–148.
44. Tian, D.J.; Hernandez, G.; Choi, J.I.; Frost, V.; Johnson, P.C.; Butler, K.R. LBM: A security framework for peripherals within the linux kernel. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 967–984.
45. Miretskiy, Y.; Das, A.; Wright, C.P.; Zadok, E. Avfs: An On-Access Anti-Virus File System. In Proceedings of the 13th USENIX Security Symposium, San Diego, CA, USA, 9–13 August 2004; pp. 73–88.
46. García, G.G.; Ramirez, M.E.L. Modeling the spatio-temporal dynamics of worm propagation in smartphones based on cellular automata. In Proceedings of the 2016 European Modelling Symposium (EMS), Pisa, Italy, 28–30 November 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 196–201.
47. Cäsar, M.; Pawelke, T.; Steffan, J.; Terhorst, G. A survey on Bluetooth Low Energy security and privacy. *Comput. Netw.* **2022**, *205*, 108712. [CrossRef]

48. Wang, H.; Xi, M.; Liu, J.; Chen, C. Transmitting IPv6 packets over Bluetooth low energy based on BlueZ. In Proceedings of the 2013 15th International Conference on Advanced Communications Technology (ICACT), PyeongChang, Republic of Korea, 27–30 January 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 72–77.
49. Beutel, J.; Krasnyanskiy, M. Linux BlueZ Howto. Available online: http://www.grc.upv.es/localdocs/bluezhowto.pdf (accessed on 23 September 2023).
50. Kirkbride, P. *Basic Linux Terminal Tips and Tricks*; Springer: Berlin/Heidelberg, Germany, 2020.
51. Basig, L.; Lazzaretti, F. *Reliable Messaging Using the CloudEvents Router*; OST Ostschweizer Fachhochschule: Rapperswil, Switzerland, 2021.
52. Celesti, A.; Fazio, M.; Galletta, A.; Carnevale, L.; Wan, J.; Villari, M. An approach for the secure management of hybrid cloud–edge environments. *Future Gener. Comput. Syst.* **2019**, *90*, 1–19. [CrossRef]
53. Groza, B.; Andreica, T.; Berdich, A.; Murvay, P.-S.; Gurban, E.H. Prestvo: Privacy enabled smartphone based access to vehicle on-board units. *IEEE Access* **2020**, *8*, 119105–119122. [CrossRef]
54. Zeadally, S.; Siddiqui, F.; Baig, Z. 25 years of bluetooth technology. *Future Internet* **2019**, *11*, 194. [CrossRef]
55. Kiourtis, A.; Mavrogiorgou, A.; Kyriazis, D. A comparative study of bluetooth spp, pan and goep for efficient exchange of healthcare data. *Emerg. Sci. J.* **2021**, *5*, 279–293. [CrossRef]
56. Seri, B.; Livne, A. *Exploiting Blueborne in Linux-Based IoT Devices*; Armis: Palo Alto, CA, USA, 2019.
57. Seri, B.; Vishnepolsky, G. *The Dangers of Bluetooth Implementations: Unveiling Zero Day Vulnerabilities and Security Flaws in Modern Bluetooth Stacks*; ArmisLabs: Palo Alto, CA, USA, 2017; pp. 1–38.
58. Seri, B.; Vishnepolsky, G. *BlueBorne-Technical Report*; Technical Report; Armis: Palo Alto, CA, USA, 2017; 41p. Available online: https://www.scribd.com/document/360135609/BlueBorne-Technical-White-Paper (accessed on 21 September 2023).
59. Godwin, S.; Glendenning, B.; Gagneja, K. Future security of smart speaker and IoT smart home devices. In Proceedings of the 2019 Fifth Conference on Mobile and Secure Services (MobiSecServ), Miami Beach, FL, USA, 2–3 March 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–6.
60. Caldwell, L.; Ekerfelt, S.; Hornung, A.; Wu, J.Y. *The Art of Bluedentistry: Current Security and Privacy Issues with Bluetooth Devices*; Semantic Scholar; University of Washington: Seattle, WA, USA, 2006.
61. freedesktop.org. File-Hierarchy—File System Hierarchy Overview. Available online: https://www.freedesktop.org/software/systemd/man/file-hierarchy.html (accessed on 21 September 2023).
62. Bazaar. Malware-Bazaar. Available online: https://datalake.abuse.ch/malware-bazaar/daily/ (accessed on 21 September 2023).
63. O. L. B. p. Stack. BlueZ. Available online: https://github.com/bluez/bluez/archive/refs/heads/master.zip (accessed on 21 September 2023).