BASTION-SGX: Bluetooth and Architectural Support for Trusted I/O on SGX

Travis Peters
Dartmouth College

Reshma Lal
Intel Corporation

Srikanth Varadarajan
Intel Corporation

Pradeep Pappachan
Intel Corporation

David Kotz
Dartmouth College

ABSTRACT

This paper presents work towards realizing architectural support for Bluetooth Trusted I/O on SGX-enabled platforms, with the goal of providing I/O data protection that does not rely on system software security. Indeed, we are primarily concerned with protecting I/O from all software adversaries, including privileged software. In this paper we describe the challenges in designing and implementing *Trusted I/O* at the architectural level for Bluetooth. We propose solutions to these challenges. In addition, we describe our proof-of-concept work that extends existing over-the-air Bluetooth security all the way to an SGX enclave by securing user data between the Bluetooth Controller and an SGX enclave.

CCS CONCEPTS

• **Security and privacy** → *Access control*; *Hardware-based security protocols*; *Trusted computing*; • **Hardware** → *Networking hardware*;

KEYWORDS

Bluetooth, Trusted I/O, SGX, IoT

ACM Reference Format:
Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. 2018. BASTION-SGX: Bluetooth and Architectural Support for Trusted I/O on SGX. In *HASP '18: Hardware and Architectural Support for Security and Privacy, June 2, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3214292.3214295

1 INTRODUCTION

Trusted Execution Environments (TEEs), such as Intel's SGX, have generated considerable interest as a means to protect application code and data from unauthorized access (e.g., [6, 7]). TEEs are especially promising in light of the ubiquity of malware that threatens to compromise applications and steal or modify security- and privacy-sensitive data such as personally identifiable information (PII), passwords, credit card numbers, and health data.

For many security applications, input/output (I/O) data has the same level of sensitivity as the data protected inside a TEE-protected application, creating a need to protect I/O data against theft or tampering from a malicious actor. Therefore, many applications that need to use a TEE also need a mechanism to protect user I/O data; this is especially so where sensitive user data is frequently communicated between client devices and I/O devices such as keyboards, medical devices, and – increasingly – Internet of Things (IoT) devices.

To address this need, past work has proposed to construct *trusted paths* – secure channels between applications and a user's I/O devices. A common approach to construct such a path is to introduce a combination of trusted drivers, middleware, operating systems (OSes), virtual machines (VMs), and hypervisors. This approach, however, is not effective for SGX applications since the software outside of an SGX enclave does not have a way to attest itself to the enclave. As a result, the application's security is reduced to the security of the drivers, middleware, OS, VM, and hypervisor. We describe this problem in greater detail next.

The Trusted Path Problem. In the canonical trusted path problem, an application wants to send/receive data securely to/from peripheral devices, including Human Interface Devices (HID) and sensors. More precisely, there exists a collection of applications (***apps***) that run on a client device (***client***); a client is, for example, a desktop, laptop, tablet, or smartphone. Among the client's apps is one or more apps that handles sensitive data (e.g., banking app, health app, messaging app). These apps want to securely communicate with peripheral devices connected with the client, even in light of software-based adversaries (***malware***) that may have compromised drivers, OSes, VMs, or hypervisors.

Today's solution (Figure 1(a)) assumes that any apps, drivers, middleware, OSes, VMs, hypervisors, and hardware are capable and trustworthy with respect to handling sensitive I/O data. Not all of these components of a system are equally worthy of user trust, however. Figure 1(b) illustrates the general idea behind past work towards addressing this problem (e.g., [12, 13]). Specifically, given a client with TEE technology (e.g., SGX), TEEs are used as a mechanism to protect the execution of an app (parts of its code and data) by partitioning it into trusted and untrusted parts; this leads to the notion of a ***trusted app*** [8], which is distinct from other user apps. A combination of new, trusted components – trusted drivers, middleware, OSes, VMs, and hypervisors – are then used to construct trusted paths between the trusted app and user I/O device through untrusted software. This approach, however, is not effective for SGX-enabled apps since the software outside of the enclave does not have a way to attest itself to the enclave. Furthermore, these approaches include complex and error-prone software within the trusted computing base (TCB). As a result, the

HASP '18, June 2, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-6500-0/18/06…$15.00
https://doi.org/10.1145/3214292.3214295

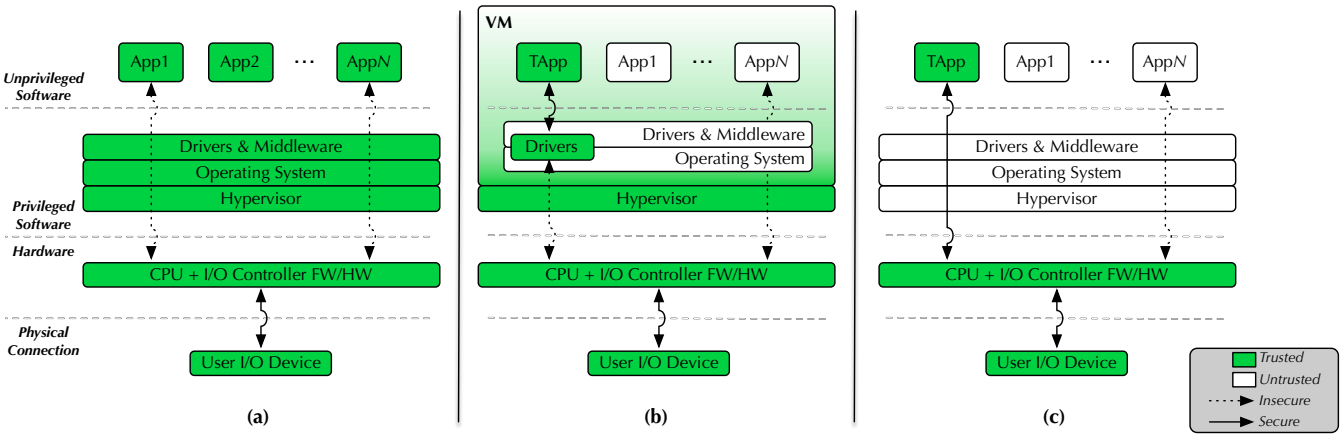

Figure 1: Overview of the trusted path problem. Today's solution is illustrated in (a). Variations of proposed solutions are illustrated in (b), all of which rely on a combination of trusted drivers, VMs, and hypervisors. Our approach, which depends on none of these trust assumptions, is illustrated in (c).

security of the trusted app is reduced to the security of the trusted components that are relied upon in their solutions.

We propose new platform features to equip SGX-enabled platforms with Trusted I/O capabilities for specific I/O paths (Figure 1(c)). These features are primarily concerned with ensuring I/O protection from all software adversaries, including privileged software. We posit that users already trust that system hardware (e.g., CPU, chipset, peripherals) operates correctly and that it is capable of protecting user data. Rather than introducing non-standard drivers or hypervisors, as past work does, we propose lightweight extensions to the platform itself. By enabling Trusted I/O in the platform – removing all system software from the TCB – we can enhance I/O security significantly.

Scope. In this paper we consider the trusted path problem for a specific wireless I/O technology: Bluetooth. Furthermore, we specifically consider the challenges around securing only the sensitive ***user data*** sent to/from Bluetooth devices (e.g., keyboard key presses, health sensor data) on SGX-enabled platforms. All references to I/O security throughout this paper are specifically aimed at securing Bluetooth I/O. While we focus our attention on Bluetooth and SGX, we postulate that similar notions apply to other wireless (e.g., Wi-Fi, NFC) and TEE technologies.

Today, wireless I/O technologies – including Bluetooth – define protection at the hardware link level and leave the client-side security up to the OS. Specifically, all I/O between the client's Bluetooth Controller and its connected Bluetooth devices is secured with existing Bluetooth security, such as over-the-air (OTA) encryption. I/O *within* the client, however, is transferred in plaintext between apps and the Bluetooth Controller. Our work addresses client-side security by securing I/O data between the Bluetooth Controller and trusted apps; our work requires no modifications to the Bluetooth protocol, the client's hardware, or the Bluetooth devices.

Challenges. Today, client devices run a vast number of apps and connect with a multitude of Bluetooth devices that enable human users to interact with these apps. It is imperative that the client's Bluetooth Controller be capable of reliably managing multiple connections with other Bluetooth devices. Therefore, in light of the trusted path problem, we need a Trusted I/O mechanism that can selectively secure only user data, and only between trusted apps and designated devices, while allowing all other apps and devices not requiring Trusted I/O to communicate as usual.

In this paper, we describe how it is possible to secure Bluetooth I/O by creating a secure tunnel between an SGX enclave and Bluetooth hardware, and show how it possible to selectively protect I/O for one or more Bluetooth devices connected to a client, all without hindering the use of other Bluetooth devices by other (non-trusted) apps, or requiring modifications to the existing Bluetooth protocol.

Contributions. The contributions of this work are:

(1) We identify and solve several challenges in realizing Trusted I/O for Bluetooth. Namely, we describe an approach to connection monitoring that can be applied within a client's Bluetooth Controller, allowing it to unobtrusively collect Bluetooth device and channel metadata.
(2) We propose BASTION-SGX, an architecture for realizing Trusted I/O specifically for Bluetooth on SGX-enabled platforms. Our architecture proposes lightweight extensions to existing Bluetooth I/O firmware to enable Bluetooth Trusted I/O.
(3) We present our proof-of-concept (PoC) work in a case study that secures sensitive Bluetooth I/O between Human Interface Devices (keyboard/mouse) and a trusted app. Specifically, our work shows how it is possible to extend existing over-the-air Bluetooth security all the way to a trusted app with our new security features that secure Bluetooth I/O between the Bluetooth Controller and the trusted app.

We emphasize that all of our contributions enhance I/O security significantly compared to today's solution in terms of protecting user I/O data from software-based adversaries on the client. Furthermore, our work offers this security all without requiring modifications to SGX, the Bluetooth devices that connect with the client device, or the client's system software; we only require modifications to the firmware of the client's Bluetooth Controller and the trusted apps.

2 BACKGROUND

In this section we present background information on the prominent technologies featured in this paper: Bluetooth and Intel's SGX.

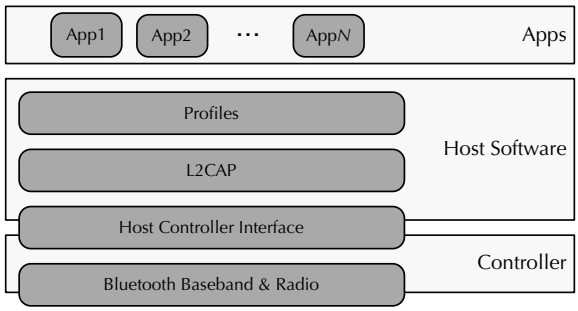

Figure 2: A simplified view of the Bluetooth stack.

2.1 Bluetooth

Although a detailed description of the Bluetooth architecture and its protocols is beyond the scope of this paper (a detailed description can be found in the Bluetooth Specification [1]), a basic understanding is required to appreciate the challenges and solutions we discuss. Note that this paper describes specific insights into our work with Bluetooth Classic. The main ideas carry over to Bluetooth Low Energy (BLE) as well. Therefore, this paper refers simply to *Bluetooth*, with the understanding that it applies to both Bluetooth Classic and BLE.

A typical deployment of Bluetooth (within a client) consists of a Host and one or more Controllers. The ***Host Controller Interface (HCI)*** is a command interface between the Host and Controllers. A ***Host*** is a logical entity made up of all the layers between Bluetooth's core profiles (i.e., Bluetooth applications and services) and the HCI. A ***Controller*** is a logical entity made up of all of the layers below the HCI, and enables the client to communicate with other Bluetooth devices. In most client devices, the Host is implemented in software, whereas the Controller is implemented with a combination of firmware and hardware. A simplified view of this deployment is shown in Figure 2.

During normal operation of Bluetooth, a physical radio channel is shared by two or more devices. Within the context of a shared physical channel, there is a complex layering of links and channels and associated control protocols that enables coordination amongst the devices as well as data to be transferred between devices. A detailed description of each of these channels and links is beyond the scope of this paper (see the Bluetooth Specification [1], Volume 1, Part A). Worthy of note in our work is, however, the ***Logical Link Control and Adaptation Protocol (L2CAP)*** channel. L2CAP channels provide a channel abstraction to applications and services. The L2CAP layer of the Bluetooth protocol carries out many functions, including segmentation and reassembly of application data, and multiplexing and de-multiplexing of multiple channels over a shared logical link. Application data submitted to the L2CAP protocol may be carried on any logical link that supports the L2CAP protocol.

Summary. In our work we focus primarily on two aspects of the Bluetooth protocol: the HCI and the L2CAP. The HCI is the interface between software (i.e., the Bluetooth Host software) and Bluetooth I/O hardware (i.e., the Bluetooth controller firmware and hardware); by extending this interface, we can enable trusted software to create secure channels for Bluetooth data within the client. The L2CAP protocol is the primary protocol for enabling applications and services; by applying Trusted I/O security at the L2CAP layer, we can offer fine-grained, channel-based protection for user data.

2.2 Intel SGX

Intel Software Guard Extensions (SGX) is a set of new instructions and mechanisms that can be used by app developers to protect selected code and data from disclosure or modification by partitioning apps into CPU-enforced containers known as ***enclaves***. Enclaves offer protected areas of execution in memory that increase security even on compromised platforms. Specifically, an enclave provides confidentiality, integrity, and replay-protection guarantees, even without trusting drivers, middleware, OSes, VMs, hypervisors, firmware, or the BIOS. SGX also provides remote and local attestation capabilities, allowing enclaves to be measured and verified – or in other words, a means for an enclave to provide proof of its authenticity. More information on SGX is available through Intel's official SGX documentation [2] as well as past academic research (e.g., [10, 12]). Today, however, Intel's SGX does not support Trusted I/O features.

3 SECURITY MODEL & CHALLENGES

In this paper, we address the trusted path problem for Bluetooth I/O (Figure 4). We break the path between the trusted software and the user (device) into two subpaths: (1) the path between trusted software and the trusted Bluetooth Controller (Figure 4, E1-E2), and (2) the path between the trusted Bluetooth Controller and a trusted Bluetooth device (Figure 4, E3-E4). As in related work [12, 13], we assume that the latter path is secure today (e.g., through standard Bluetooth OTA security). Our work focuses on the former path and how trusted software and the trusted Bluetooth Controller can secure I/O channels through untrusted Host software *within* the client. Towards this end, this section describes our goals and security model, as well as the challenges of this work.

3.1 Security Model

Threats & Adversaries. While there are certainly many types of adversaries and threats that one can imagine, in this paper we are primarily concerned with software-based adversaries, including privileged software. Specifically, we concentrate on preventing two types of software attacks: (1) unauthorized access attacks, which aim to access sensitive user data transported via Bluetooth, and (2) data injection or replay attacks, which aim to inject data that is not authentic, or replay authentic data for malicious purposes.

To this end, we consider an adversary (malware) that has various capabilities and employs various tactics to successfully perform such attacks. Specifically, adversaries ***can*** read or write the code and data of system software and untrusted apps on the client. Adversaries can create their own trusted/untrusted apps. Adversaries can also interpose on communication (i.e., intercept, insert, alter, deny messages), be it between trusted software and Trusted I/O hardware *within* the client, or between the client and device. Adversaries ***cannot*** physically access the client device or any of its connected Bluetooth devices. Adversaries cannot read or write the data or code of trusted software that is protected within a TEE, nor

the data or code protected by the trusted hardware. Adversaries cannot break encryption primitives or protocols known to be secure today. Denial-of-Service (DoS) and side-channel attacks are out of scope for our work.

Assumptions & Trust Model. In our work we assume the human user is not an adversary. We assume that the Bluetooth I/O device is implemented correctly and is trusted by the user to faithfully handle I/O on their behalf; furthermore, we assume the device accurately identifies itself to the client. We assume the client has SGX-enabled hardware. We assume that system software (e.g., drivers, OS) and other apps are *not* trusted. We assume all trusted software and trusted hardware (including firmware) is implemented and authenticated/loaded/initialized/booted correctly. We assume trusted apps trust that the I/O Controller will comply with security policies (Section 3.2), not disclose user I/O data to other devices or apps, and implement OTA protection between the client and any Bluetooth devices. We assume the physical channel between the client and device (Figure 4, E3-E4) is protected with existing Bluetooth OTA security.

Goals. In light of these threats and assumptions, we seek to achieve the following four goals: **(G1)** We aim to design a client-side architecture that is not dependent on trusted hypervisors, trusted OSes, or trusted drivers for security. **(G2)** We aim to provide confidentiality, integrity, and replay protection guarantees over select user I/O data between trusted software and hardware. **(G3)** We aim to provide protection against impersonation of trusted software or hardware. And last, **(G4)** we aim to achieve these goals in a way that does not interfere with existing I/O protocols (namely, Bluetooth) or break existing routing mechanisms.

3.2 Bluetooth Trusted I/O Security Policies

Bluetooth applications and services rely on L2CAP channels to transport user data. Thus, we apply Bluetooth Trusted I/O security to select L2CAP channels; i.e., channels that carry user data.

A ***Bluetooth Trusted I/O security policy*** specifies what Bluetooth I/O data needs to be protected between a trusted app and the Bluetooth Controller, as well as information used to secure the data. Specifically, a policy is made up of two elements: (1) information to identify the specific channel(s) that should be secured (e.g., user data from a Bluetooth keyboard), and (2) a symmetric key; the symmetric key is used to apply cryptographic protection over the specified channel's data. Here, a ***secure*** channel is one with the properties noted in goals G2 and G3 described above. We discuss what information is needed to identify specific channels that carry user I/O data in Section 3.3, and how security policies can be *programmed into* the Controller in Section 4.1.2.

Because our architecture aims to provide confidentiality of user I/O data (G2), a trusted app that programs a security policy has exclusive access to the channel(s) defined in the policy. This ensures that no other software can access the Bluetooth I/O data that the trusted app aims to secure. The OS enforces exclusive use of some devices today (e.g., keyboards, cameras) for security. For example, by default, the OS ensures keyboard input is only sent to the app that has focus to ensure that no other app can observe passwords or other sensitive data entered by the user. Since the OS and other system software is not within the TCB of our architecture, however, we aim to provide similar guarantees without relying on this untrusted software.

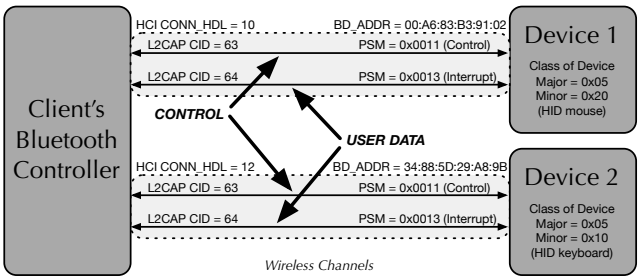

Figure 3: An example of two Bluetooth devices connected with a client. Trusted I/O requires fine-grained channel selection. For each device and channel, a Trusted I/O-enabled Bluetooth Controller maintains information about: the physical connection (HCI Connection Handle), logical channels (L2CAP CIDs) and their respective protocol/service multiplexor (PSM; control vs. data), and Class of Device (COD; composed of Major and Minor numbers).

3.3 Challenges

Today, the client's Bluetooth Controller is the gateway for all packets transported between the client's Host software and all connected Bluetooth devices. An implication of the Controller's current design and role as gateway, however, is that all ingress Bluetooth packets (device-to-host), from all connected devices, are multiplexed within the Bluetooth Controller into a single stream and delivered to Host software via the HCI. Similarly, all egress packets (host-to-device) must be demultiplexed within the Controller and sent to the correct Bluetooth device.

In our work we seek to not interfere with the Bluetooth protocol or break existing routing within the client's software (G4). Therefore, our Trusted I/O solution aims to selectively secure only user data, and only between trusted apps and designated devices, while allowing all other apps and devices not requiring secure I/O to communicate as usual. Realizing Trusted I/O for Bluetooth has raised three main issues: associating multiplexed and interleaved packets to their respective devices and channels; isolating and protecting only user data (without interferring with control channels or packet headers); and overcoming tension between enforcing high-level security policies given only low-level context. We discuss these issues next and refer to aspects of Figure 3 and Figure 6.

3.3.1 Packet Multiplexing & Interleaving. Today, a Bluetooth Controller must maintain basic metadata about connected devices and their channels so that it can route packets to Host software. Conceivably, a Bluetooth Controller need only maintain a mapping between an HCI connection handle and the corresponding device's Bluetooth address, enabling the Controller to know which packets belong to which devices. Knowing how to differentiate packets by device, while necessary, is not sufficient for securing user I/O data. To elaborate, actual user data is transported between Host software and Bluetooth devices using L2CAP channels. All L2CAP packets (Figure 6) have a channel identifier (CID), and according to the Bluetooth specification, the L2CAP CID of a packet enables

routing software to associate packets with specific L2CAP channels. Unfortunately, the CID used in L2CAP packets is guaranteed to be unique only *per device*. This can give rise to ambiguity. For example, Figure 3 illustrates a client connected with two Bluetooth devices, each with one channel used to exchange user data with the client. Per the Bluetooth specification, it is actually reasonable for the channels to have the same CID. The disambiguating attribute in this case is the identifier for the physical link (i.e., the HCI connection handle or Bluetooth Device Address, which identifies a unique Bluetooth device), which is not part of the L2CAP packet. While all of this information is not located in a single packet, we note that together, an HCI connection handle *and* an L2CAP CID can be used to uniquely identify channels.

3.3.2 Isolating & Securing User Data Only. Another issue that arises is that L2CAP packets actually come in two different types: those that carry control information and those that carry data. To ensure all user I/O data is secure, it may initially seem like a good idea to secure *all* packets – regardless of type – belonging to communication with a specific device. This approach, however, can have atrocious effects on existing Bluetooth functionality. For example, as depicted in Figure 3, Device 1 has one channel (L2CAP CID = 63) dedicated to handling signaling (e.g., enable device options, determine current device state) between the client and device; as a consequence of securing *all* packets, Trusted I/O security would obfuscate (encrypt) this signaling channel and "break" application-level functionality. For other, primary signaling channels, this can be even more destructive, breaking functionality that controls how packets are routed, and how logical channels are created, maintained, and destroyed. In general, essential Bluetooth functionality relies on access to HCI connection handles, L2CAP channel identifiers, and in some cases, even information in the L2CAP packet payloads (e.g., control parameters). Thus, in order to not break existing Bluetooth functionality, packets that contain *control information* should not be secured, and packets containing *data* should only be secured if a relevant security policy exists.

The underlying issue here is that L2CAP packets have no type descriptor in the packet that can be used to disambiguate control packets from data packets. Such a descriptor is, however, present at the time that new L2CAP channels are created. During L2CAP channel creation, a Protocol and Service Multiplexor (PSM) value is exchanged. PSM values are useful for securing channels as they provide higher-level insight into the purpose of the channel and the type of information that will be exchanged over the channel. Furthermore, some data and control channels are defined in the specification and are allocated reserved channel identifiers that indicate their purpose (data vs. control). Thus, while channel type information is not contained within all packets, such information is either standardized (and therefore need not be directly observed) or available during the creation of channels (and is therefore observable within the Controller).

3.3.3 Understanding Device Types. To enforce high-level security policies, we seek a mechanism to map security policies that are meaningful to humans and apps, to any connected devices. For instance, a Trusted I/O security policy may require that all I/O from/to a particular device, or particular *class* of device (such as all keyboard devices; see Device 2 in Figure 3), should be secure. Fortunately,

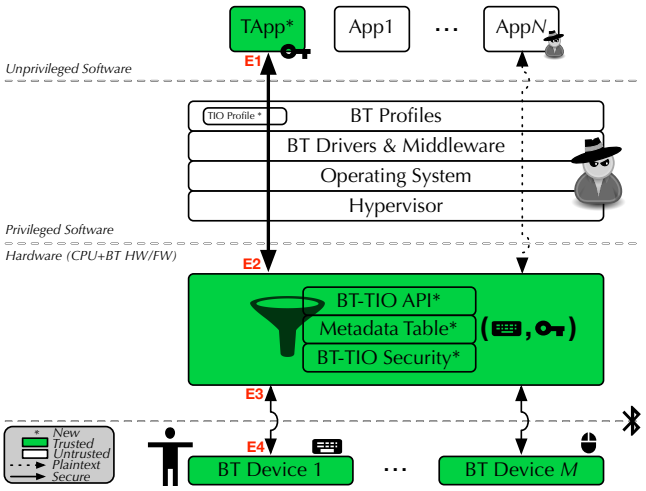

Figure 4: Overview of our Bluetooth Trusted I/O architecture. Our goal is to ensure data is secured in its transportation between two endpoints: trusted software (E1) and a Bluetooth device (E4). We assume the path between the client's Bluetooth Controller and the device (E3-E4) is secure via Bluetooth OTA security that exists today. Our work shows how I/O channels between trusted software and the trusted Bluetooth Controller (E1-E2) can be secured. Together, these paths achieve our goal.

Bluetooth defines the notion of Class of Device (COD), which provides higher-level information about the purpose and function of a device. Each Bluetooth device belongs to some class, represented by major and minor class information. Bluetooth currently defines 32 major classes (e.g., Computer, Phone, Peripheral, Health). There are a multitude of minor classes that describe subclasses of a particular major class; for example, given a major class of Peripheral, minor class information further describes if that Peripheral is a keyboard, mouse, or something else. According to the Bluetooth specification, "The major device class segment is the highest level of granularity for defining a Bluetooth device. A device's main function determines its Major Class assignment." Using Bluetooth Class of Device information to represent devices in security policies likely maps well to the high-level understanding of devices that humans and apps have.

4 PROPOSED ARCHITECTURE: BASTION-SGX

In this section we present BASTION-SGX: our Trusted I/O architecture for Bluetooth on SGX (Figure 4). BASTION-SGX is comprised of the following components: the client's trusted software, the client's Trusted I/O hardware, and some number of wirelessly connected devices. Specifically, the trusted software consists of standard SGX software and a trusted app. The trusted hardware is an SGX-enabled CPU and a Bluetooth Controller with our Trusted I/O extensions. Wireless devices connect with the client and communicate with apps.

4.1 Bluetooth Trusted I/O Controller

BASTION-SGX is centered around a Trusted I/O-enabled Bluetooth Controller, which implements the following features: (1) monitor

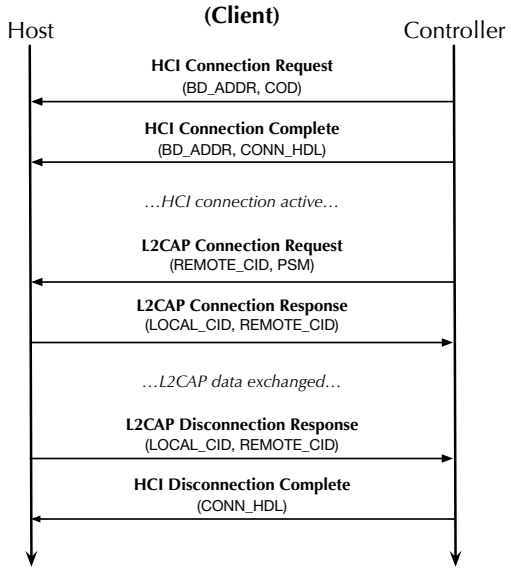

Figure 5: An example flow of a client-device connection with a summary of the relevant HCI and L2CAP connection-related events. These events are standard in Bluetooth's HCI and L2CAP protocols today. We propose to monitor and capture device- and channel-specific metadata during connection/disconnection events in our Trusted I/O-enabled Bluetooth Controller. The relevant metadata is shown in parenthesis.

connection events and maintain a Metadata Table to store information for connected Bluetooth devices and their respective channels; (2) expose an API to support Trusted I/O-related interactions with Host software; (3) filter packets in accordance with the Metadata Table; and (4) apply cryptography to provide the desired security properties over a channel. We discuss these components in greater detail next (see Figure 4).

4.1.1 Connection Event Monitoring & the Bluetooth Trusted I/O Metadata Table. The solutions we envision for the various challenges described in Section 3.3 rest in our ability to obtain metadata about devices, and their respective HCI and L2CAP channels. We assert that it is possible to obtain all of the necessary information simply by extending the Bluetooth Controller's firmware. Specifically, BASTION-SGX proposes new features to monitor HCI and L2CAP connection/disconnection events, and maintain a Metadata Table that contains the information alluded to in Section 3.3. Figure 5 provides a summary of the relevant HCI and L2CAP events, and the metadata that the Controller captures.

When a Bluetooth device first connects with a client device, the Controller creates an ***HCI Connection Handle*** (`CONN_HDL`) that the client's Host software can use to communicate with that specific device. As an example, when a device connects, the Bluetooth Controller generates an HCI Connection Request event to inform Host software that a device wishes to connect. The device is described initially by its ***Bluetooth Device Address*** (`BD_ADDR`) and ***Class of Device*** (`COD`). Upon completion of the physical connection between the client and device, the Controller generates an HCI Connection Complete event that provides Host software with the `CONN_HDL` that can be used for future communication with the device. At this point, the Host and Controller have an active HCI connection that can be used for subsequent communications between the Host software and device.

Once connected, these entities can exchange L2CAP connection requests and responses to create logical links for exchanging control and data packets. For example, a device that sends a request to the Host to create a new L2CAP channel sends two pieces of information: a `REMOTE_CID` and `PSM`. The ***Protocol/Service Multiplexor*** (`PSM`) indicates the purpose of the channel; i.e., what protocol or service will operate over the new L2CAP channel. The L2CAP channel has two endpoints: one in the Host (`LOCAL_CID`) and one in the device (`REMOTE_CID`). While the Host is free to assign any `CID` for its local endpoint, it does not control the `CID` that the device uses for its endpoint. Thus, when new L2CAP channels are formed, the Host software and device carry out an acknowledgement of the `CID` to be used for their respective endpoints. Events related to HCI and L2CAP disconnections can be similarly observed.

By monitoring these HCI and L2CAP events, the Controller can be made to capture and maintain fine-grained information about each connected device (`CONN_HDL`, `BD_ADDR`), its type (`COD`), its logical channels (`LOCAL_CID`, `REMOTE_CID`), and the protocols or services (`PSM`) operating over each channel. This information can then be used in accordance with security policies (Section 4.1.2) to filter (Section 4.1.3) and secure packets (Section 4.1.4).

4.1.2 Bluetooth Trusted I/O API. Trusted I/O features are aimed at giving trusted software the ability to create secure channels to protect specific I/O data channels. Thus, in BASTION-SGX, security policies (Section 3.2) are driven by the requirements of trusted software. There are two issues worth considering here: first, how trusted software can program security policies into the Controller, and second, how trusted software can describe security policies, based on metadata the Controller independently maintains.

Programming Security Policies. The HCI can be used to address the first issue. The HCI is already used for communication between the Host and Controller. Furthermore, the HCI protocol supports an extensible interface, often referred to as the ***Vendor Specific Debug Command (VSDC)*** interface. This interface enables vendors to add non-standardized features to Bluetooth Controllers and to enable apps to use those features. Thus, we can use this interface to support new Bluetooth Trusted I/O APIs – such as policy specification APIs for *adding* and *removing* security policies – in Trusted I/O-enabled Bluetooth Controllers.

Class-of-Device Policy Specification. One approach to specifying security policies is to identify a class of devices that should be secured along with a key (`COD`, `KEY`), and rely on the Controller to determine which channels carry sensitive data versus those that do not. As noted in Section 4.1.1, upon connecting, `COD` information is exchanged; therefore the Controller can know the `COD` of each of its connected devices. Furthermore, because `PSM` information is exchanged during the creation of any L2CAP channel, the Controller can know the purpose of each L2CAP channel, enabling it to identify (and subsequently secure) channels that carry user data (using the policy's `KEY`). This approach for defining security policies is conservative in that it allows trusted software to secure I/O between it and *any* device matching the `COD` in its policy.

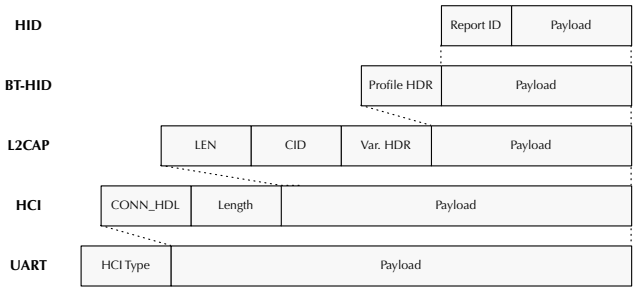

Figure 6: An example of the Bluetooth packet hierarchy. In our PoC we discuss how input data from HID devices can be secured. User data (e.g., key presses) is transported in HID packets (top level). These packets are nested in multiple layers of the Bluetooth protocol for transportation from a device to a client's Host software.

4.1.3 Bluetooth Trusted I/O Filtering. The Bluetooth Trusted I/O Filter is responsible for (1) identifying packets containing sensitive user data based on known devices (Section 4.1.1) and security policies (Section 4.1.2), and (2) securing these packets (Section 4.1.4) using the policy's KEY. Thus, as L2CAP packets are transported between the Host and Bluetooth devices, the filter examines *each* packet to determine if some security policy applies to that packet. Essentially, given an L2CAP packet, PACKET, an HCI connection handle, CONN_HDL, and information about the *direction* of the packet (host-to-device or device-to-host), DIR, the filter must decide to either apply Trusted I/O security to PACKET or allow it to pass through unaffected.

Each L2CAP PACKET contains a CID and packet length (LEN); for reference, Figure 6 illustrates the L2CAP packet structure. Also, the Controller knows the CONN_HDL to which the PACKET belongs, and which direction the packet is being transported. Therefore, the filter can check the Metadata Table to see if the PACKET belongs to a secure channel and apply the appropriate security operations (encrypt, decrypt, etc.) over the PACKET's payload based on DIR. Note that the boundary of the PACKET's payload can be determined from the LEN field in the header of PACKET; all Trusted I/O security is applied to the payload. We discuss Bluetooth this step next.

4.1.4 Bluetooth Trusted I/O Security. The details of the security applied to user data (e.g., encryption and decryption algorithms, key sizes, MACs) are implementation and security-model specific. As noted in Section 3.1, BASTION-SGX aims to provide confidentiality, integrity, replay protection, and mutual authentication guarantees (G2 and G3). In BASTION-SGX, any authenticated encryption algorithm can be used to secure the channel. Such an encryption algorithm is applied to user I/O data within trusted software and the trusted Controller so that the data can be routed in the normal way (G4) via untrusted software; by securing the I/O data before moving it out of the trusted software or Controller, we ensure the data is opaque to untrusted software (G1). Furthermore, in doing so, we protect user I/O data against the malware attacks defined in Section 3.1 – which raises the bar significantly from today's solution. Thus, assuming trusted software has a secure mechanism to share (program) keys with the Controller, BASTION-SGX can achieve its security goals. We discuss two approaches for how IA platforms can accomplish this key sharing next. We note that this key programming step is critical to Trusted I/O security, yet distinct from the contributions we describe in this paper. In fact, there is nothing especially novel behind how this key programming can be done, and as such, is not an emphasis of this paper. To convince the reader that this step is not an issue, we briefly describe two approaches next.

Dynamic Key Provisioning. This approach requires further extensions to the Bluetooth Controller that enable it to attest itself to an enclave, enabling a Controller to prove to an enclave that it is authentic Bluetooth hardware, executing authentic Bluetooth firmware; the attestation we envision is similar to what is proposed in the USB Type-C Authentication Specification [5] and the PCIe Device Security Enhancements Specification [4].[1] These specifications define nearly identical authentication architectures that allow USB and PCIe devices, respectively, to have their identity and capability cryptographically verified. These architectures provide a specific example where cryptographic verification can be used to subsequently exchange secrets to set up a secure channel between software and a device; such a channel, in our work, would be used to program security policies (Section 3.2) from trusted software into the Controller.

At a high level, these authentication architectures adapt common industry paradigms (i.e., PKI) for identity and capability verification. Specifically, a trusted root certificate authority (CA) generates a root certificate; the root certificate is used by an authentication initiator (verifier) to verify the validity of signatures generated by a device (prover) during authentication. The root certificate is also used to endorse vendor certificates, which are then used to endorse some combination of intermediate certificates and model certificates; these certificates are ultimately used to endorse per-device certificates.

Authentication happens in two steps: (1) Authentication Provisioning, and (2) Runtime Authentication. In the authentication provisioning step, the root certificate is provisioned to the authentication verifier (in our case, trusted software) to enable the verifier to verify the validity of signatures generated by a device (in our case, a Bluetooth Controller) during the runtime authentication step. Furthermore, a public/private key pair is generated for each device; the private key is provisioned into the device at the time of manufacturing along with a certificate that contains its corresponding public key, along with a signature that can be verified using the root CA's public key in the root certificate. In the runtime authentication step, the verifier queries the device to obtain its certificate, and sends a unique challenge (nonce) to the device; the device can authenticate its identity and capability by signing the challenge along with other authentication data (e.g., a measurement of its firmware) with its private key. The verifier can verify the device's response/signature using the device's public key and the root CA's public key (as well as any intermediate public key), and use the result of its verification to make a trust decision.

Assuming the device (again, in our case this is the Controller) successfully attests to its authenticity, standard protocols such as

[1] At the hardware level (i.e., within the client), components connected with the CPU via PCIe, USB, UART, etc., are commonly referred to as "parts" or "devices." Thus, references to "devices" in this context are distinct from how we use the word "device" throughout the rest of the paper, which is to refer to Bluetooth devices.

DAA-SIGMA [11] can be used to share a secret and establish a secure channel between an enclave and the Controller; the enclave can then use the secure channel to program Trusted I/O keys into the Controller.

New Platform Capability. An alternative approach envisions a new platform capability, similar to previously-envisioned capabilities. For example, the authors of the Secure Input/Output Device Management patent [9] describe a scenario similar to ours: a processor has secure execution environment support (e.g., SGX) and wishes to establish a secure connection to an I/O controller. The I/O controller includes an integrated Trusted I/O component that can receive (unencrypted) requests to configure the Trusted I/O component. In the patent, the authors provide details for how a USB controller can be equipped with trusted I/O capabilities and how encryption keys can be established between an enclave and the USB controller. Our work can use similar features for a Trusted I/O-enabled Bluetooth Controller. This new capability would make secure key programming a feature of the platform (via ISA extensions), and allow enclaves to send Trusted I/O keys securely to an authentic Bluetooth Controller.

4.2 Trusted I/O Host Software

In BASTION-SGX, trusted apps are implemented as SGX enclaves. Trusted apps are therefore subject to the same security model as SGX, and benefit from existing work towards resources for enclave software development [8, 10]. Trusted software that uses enclaves to protect select code and data protects sensitive I/O data within the enclave, and uses our Trusted I/O features to secure I/O data between itself and the Bluetooth Controller.

In theory, a trusted app can be quite simple. To create secure Bluetooth I/O channels, a trusted app needs to program security policies into the Controller using the Bluetooth Trusted I/O API (Section 4.1.2). For each new secure I/O channel, a trusted app should use a new symmetric key, which can be generated using third-party libraries such as mbedTLS [3], for example. The trusted app can then use one of the secure programming mechanisms described in Section 4.1.4 to securely send the key to the Controller. A trusted app also needs to perform cryptographic operations on incoming/outgoing data; the SGX SDK [2] offers various functions to help developers with these sorts of operations, though again, third-party libraries (e.g., [3]) can also be used.

In some cases there may be a need for additional trusted software (e.g., Trusted Bluetooth Profiles) to support trusted apps that use Trusted I/O features. (See Section 5 for an example.) In today's solution, the OS and various drivers are trusted, so it is not a problem to have them process and interpret the contents of I/O packets to make them useful for apps. Our security model rules the OS and these subsystems out of the TCB; our work is therefore unable to rely on them for these services. Alternatively, a trusted app can opt to implement any data processing/interpretation that is needed (as we do in our work). Both of these options are viable.

Ideally, we envision extending the existing SGX SDK [2] to include our proposed Bluetooth Trusted I/O extensions for SGX. Specifically, SGX SDK extensions would implement common Bluetooth Trusted I/O operations such as key generation and encryption/decryption for securing I/O data, a security policy API, and so forth, alleviating the need for developers to implement these features in their apps.

5 PROOF OF CONCEPT

Here, we aim to validate the Trusted I/O Controller and its role in our architecture (Section 4.1). Specifically, we seek to validate: (1) that our proposed metadata table can be built and enables unique channel selection for Trusted I/O security; (2) that security policies can be added/removed to/from the Controller by creating new VSDCs; (3) that packet filtering can isolate data-carrying packets and encrypt only packet data; and (4) that only the trusted app can recover (decrypt) I/O data over the secure channel it establishes with the Controller. In our current implementation, we modified Bluetooth firmware that runs within an Intel Bluetooth Controller, adding the features we describe in Section 4.1.

On initialization of the Controller, we allocate space for the metadata table. We added hooks into the existing firmware to monitor HCI and L2CAP connection/disconnection events, and update the metadata table accordingly. We also extended the Controller to support two new VSDCs for adding/removing security policies: `TIO_SET_KEY` and `TIO_CLEAR_KEY`. As packets arrive in the Controller, the Controller looks up information about the packet (i.e., to which channel it belongs) to determine whether further action is necessary (Section 4.1.3). We use the `KEY` programmed via `TIO_SET_KEY` to secure the relevant channel between trusted software and the Controller. Because the metadata table maintains information about each connected device and all of their logical channels, our current implementation uses the presence of a `KEY` for a particular channel as a flag to indicate whether or not that channel is currently a secure channel. Together, these steps enabled us to validate the above-mentioned objectives.

We also implemented a trusted app (TA). We specifically considered a TA that prompts the user for a password and wants to ensure that password entry is secure. We installed a privileged keylogger on the client to verify that the channel is in fact secure during password entry; the keylogger monitored all transactions over the HCI and logged all HID data.

At the time a user enters the password field context, the TA generates a symmetric key and sends it to the Controller, indicating that it wants to secure input from the connected keyboard device (Section 4.1.2). As a user types her password, the Bluetooth device generates packets containing the key presses. Because the device and client were previously paired, they share a symmetric key and use it to protect user data in the OTA segment of the I/O path.[2] As L2CAP packets arrive in the Controller, the Controller uses the OTA key to decrypt them. Without Trusted I/O, the Controller need only map the link identifier to the HCI connection handle, and transport packets to Host software. With Trusted I/O, however, the Controller first checks to see whether the packet belongs to a Trusted I/O channel (Section 4.1.3). If the packet belongs to a Trusted I/O channel, channel security is applied (Section 4.1.4) using the `KEY` programmed by the TA previously. If the packet does not belong to a Trusted I/O channel, no channel security is applied. In either case, the packet is encapsulated within an HCI packet and

[2]The OTA symmetric key is negotiated between the client's Controller and device's Controller. Host software (trusted or untrusted) does not have access to the OTA key.

sent to Host software via the normal transport (e.g., UART). When the TA receives the packet, it decrypts and verifies the contents.

One technical challenge that arises in our PoC work is the handling of Human Interface Device (HID) input. Because HID devices are an important part of modern computers, there are drivers and other middleware that help to process and interpret HID data. For example, keyboard input is sent through a HID subsystem that translates scan codes into text characters. In reality, this translation is fairly simple, and the TA can implement it in its own code – indeed, the TA in our PoC handles the translation itself. Alternatively, one can envision trusted middleware that handles these sorts of standard operations. In light of this, we need to prevent Host software from trying to interpret certain (HID) packets which have been secured as part of a Trusted I/O channel.

In Bluetooth, HID packets are encapsulated within Bluetooth HID packets, which are then encapsulated within L2CAP packets (Figure 6). The Bluetooth HID layer serves as a lightweight wrapper of the HID protocol defined for USB; this enables the re-use of Host software that already exists to support USB-based HID. By default, when a Bluetooth HID device connects, Host software routes its HID packets through the relevant HID subsystems, processes the packet contents, and then makes the data available to apps. To prevent the Host software from routing Trusted I/O HID packets through these HID subsystems – and erroneously interpreting packet contents – we installed a new Bluetooth profile: the ***Trusted I/O HID Profile***. This profile is functional software that exists solely to prevent premature interpretation of data, and instead, passes data to the intended trusted software for handling. We emphasize that this "glue" software is not part of our TCB: it is untrusted *functional* software that is needed only to prevent Host software from incorrectly handling certain packets.

6 SUMMARY & RELATED WORK

Addressing the trusted path problem for Bluetooth I/O raises a number of challenges that we confront in this work. Our approach bears some resemblance to the trusted path work by Zhou et al. [13]. They propose to build a trusted path between a program endpoint (trusted app) and a device endpoint (I/O hardware); they rely on a non-standard hypervisor to offer trusted path isolation from untrusted software. In our work, we eliminate any need to rely on trusted drivers, OSes, hypervisors, and so forth, for security; all data is secured within the Bluetooth Controller and the trusted app, and is therefore opaque while in transit through untrusted software. In another related work, researchers present SGXIO [12]. In SGXIO, the trusted path must be built from a user app (enclave) to a Trusted I/O driver, and from the driver to the respective I/O device. Again, this work relies on a hypervisor to realize a secure binding between the Trusted I/O driver and the actual I/O hardware. In our architecture, a specific I/O Controller (Bluetooth) is modified, enabling an SGX app to create a secure binding with the Controller directly.

In this paper, we provide an in-depth analysis of Bluetooth and various challenges in realizing Trusted I/O for Bluetooth. As a result, we present BASTION-SGX: a Trusted I/O architecture for Bluetooth on SGX. We also discuss our proof-of-concept implementation of BASTION-SGX, which adds new, lightweight features to the Bluetooth Controller and demonstrates its utility in securing user data input from keyboard devices. In future work we plan to explore extensions of this work to address other I/O paths (e.g., Wi-Fi, NFC), and evaluate the performance cost of our Trusted I/O solution. While we did not present performance measurements in this paper, we anticipate that the performance cost of our solution for securing Bluetooth I/O will be acceptable. The maximum bandwidth supported by Bluetooth is no more than a few megabytes, thus we believe that the Trusted I/O-related cryptographic operations will not introduce any perceptible latency and will certainly not have any impact on the throughput. In our PoC we implemented the cryptographic operations in firmware; to reduce performance costs, we recommend implementing some (or all) of these operations in hardware.

ACKNOWLEDGEMENTS

We thank Magnus Eriksson from Intel Corporation who provided expert consultation on the Bluetooth stack and supported our efforts in developing a proof of concept that involved changes to Intel's Bluetooth Controller firmware and the Host-side software. We also thank Sougata Sen for feedback on early drafts of this paper. This research results from a research program at the Institute for Security, Technology, and Society at Dartmouth College, supported by the NSF under award numbers CNS-1329686. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors.

REFERENCES

[1] Bluetooth Specifications. Online at https://www.bluetooth.com/specifications.
[2] Intel Software Guard Extensions. Online at https://software.intel.com/sgx/.
[3] mbed TLS. Online at https://tls.mbed.org/.
[4] PCIe* Device Security Enhancements (Draft) Specification. Online at https://www.intel.com/content/www/us/en/io/pci-express/pcie-device-security-enhancements-spec.
[5] USB Specification. Online at http://www.usb.org/developers/docs/.
[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L. Stillwell, Christof Fetzer, David Goltzsche, David Eyers, Rüdiger Kapitza, and Peter Pietzuch. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
[7] Helena Brekalo, Raoul Strackx, and Frank Piessens. Mitigating Password Database Breaches with Intel SGX. In *Proceedings of the Workshop on System Software for Trusted Execution (SysTEX)*. ACM Press, 2016. DOI 10.1145/3007788.3007789.
[8] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
[9] Steven B. McGowan. Secure Input/Output Device Management. Online at http://www.sumobrain.com/patents/wipo/Secure-device-management/WO2017023434A1.html.
[10] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
[11] Jesse Walker and Jiangtao Li. Key Exchange with Anonymous Authentication Using DAA-SIGMA Protocol. In *Proceedings of the International Conference on Trusted Systems (INTRUST)*. Springer-Verlag, 2010. DOI 10.1007/978-3-642-25283-9_8.
[12] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. *arXiv preprint arXiv:1701.01061*, 2017.
[13] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *IEEE Symposium on Security and Privacy (S&P)*, 2012. DOI 10.1109/SP.2012.42.