Don't Talk Unless I Say So! Securing the Internet of Things With Default-Off Networking

James Hong*, Amit Levy*, Laurynas Riliskis†, and Philip Levis*
*Stanford University †harmony.ai
Email: {hongjn,levya,pal}@cs.stanford.edu, laurynas@harmony.ai

***Abstract*—The Internet of Things (IoT) is changing the way we interact with everyday objects. "Smart" devices will reduce energy use, keep our homes safe, and improve our health. However, as recent attacks have shown, these devices also create tremendous security vulnerabilities in our computing networks. Securing all of these devices is a daunting task.**

In this paper, we argue that IoT device communications should be *default-off* and desired network communications must be explicitly enabled. Unlike traditional networked applications or devices like a web browser or PC, IoT applications and devices serve narrowly defined purposes and do not require access to all services in the network. Our proposal, Bark, a policy language and runtime for specifying and enforcing minimal access permissions in IoT networks, exploits this fact. Bark phrases access control policies in terms of natural questions (who, what, where, when, and how) and transforms them into transparently enforceable rules for IoT application protocols. Bark can express detailed rules such as "Let the lights see the luminosity of the bedroom sensor at any time" and "Let a device at my front door, if I approve it, unlock my smart lock for 30 seconds" in a way that is presentable and explainable to users.

We implement Bark for Wi-Fi/IP and Bluetooth Low Energy (BLE) networks and evaluate its efficacy on several example applications and attacks.

***Index Terms*—Internet of Things, Network security**

I. Introduction

Dropping silicon costs and a proliferation of low-power, CMOS-based radios have allowed small, low-power computers to be embedded into a variety of everyday objects, including light bulbs, cameras, health sensors, HVAC (heating, ventilation, and air conditioning) systems, and locks. On the one hand, these "smart" devices reduce our energy use, keep our homes safe, and improve our health. On the other hand, they create tremendous security vulnerabilities in our computing networks, as demonstrated by recent attacks on HUE light bulbs [1], dolls that spy on children [2], spam from connected appliances [3], IoT worms ([4], [5]), and distributed-denial-of-service (DDoS) attacks by the Mirai botnet of compromised security cameras and DVRs ([6], [7]).

Two factors make these Internet of Things (IoT) devices notorious weak links in network security; they are easy to compromise, and their compromise goes undetected. IoT devices are often low-margin products whose primary goal is convenience. They often do not follow security best practices and, even if they do, may not be updatable when best practices change or bugs are discovered. The attack on Dyn, for example, was launched from devices with weak default passwords that could be easily commandeered [6], while the Miele attack enables an active attacker to read arbitrary files on a commercial dishwasher [8]. Other common vulnerabilities include sending passwords and sensitive data in the clear and allowing connections from arbitrary, unauthenticated hosts ([2], [9]). To make matters worse, detecting these attacks is difficult. Following Mark Weiser's vision of "calm computing" [10], IoT devices are intended to be unobtrusive and seamlessly helpful. As a result, owners interact with them infrequently and in very limited ways—*how would you know if your thermostat was part of a botnet?*

Given that IoT devices are so easy to undetectably compromise, one might conclude that they represent an undefeatable security threat. However, one property of the Internet of Things makes them easier to secure and to contain in the event of compromise. Unlike complex, application-rich end-user devices such as laptops and tablets, IoT devices have narrow, application-specific uses. A thermostat, light bulb, or webcam does not act as a platform for scores of new networked applications each year. Consequently, their communication patterns are also stable and predictable. A Nest thermostat talks only to Nest servers, while only a small number of clients connect to home webcams.

In this paper, we argue that because of their high-risk but narrow traffic patterns, communication to IoT devices should be *default-off*. By default, network connectivity to an IoT device is not allowed—a gateway or router blocks it. *Desired* communications must be explicitly and finely enabled. For example, rather than allowing any smart watch to connect to your smart lock (opening the possibility of a network attack or compromise), any device wishing to connect to your lock must be explicitly added to a whitelist.

Making communication *default-off* has tangible security benefits. It would have prevented the Dyn attack because gateways would not have allowed hacked devices to send arbitrary DNS queries to Dyn servers. Moreover, *default-off* would also prevent network attackers from exploiting common vulnerabilities in devices (e.g., open Telnet and SSH ports) to spread malware and worms [4].

If communication to IoT devices is off by default, then *how does one specify what communication is allowed?* For example, consider a smart door lock, which you want your cleaning person to be able to open when they come to clean your home every other week. Such an access control rule involves a principal (the person) who might be identified by a phone IMEI or public key, a location (at your front door), and a time (every other week at the expected time). At the same time, rules must be comprehensible and configurable by end-users so that they may grant and revoke access as their requirements and the principals involved change. An IoT access control policy specification language needs to be expressive, precise, and presentable in terms which non-technical users can understand.

This paper proposes Bark, a policy specification language for enabling communication when the network is *default-off*. Bark describes how IoT devices may communicate in terms of natural questions—*who*, *what*, *where*, *when* and *how*—and sentences, consisting of a **subject**, **object**, **action**, and **conditions**. These capture underlying structure and behaviors in the network in user understandable terms.

The main contributions of this paper are:

1) An argument for a *default-off* communication model for the IoT to defend devices from network attacks and vice-versa (§III). We justify *default-off* by analyzing recent attacks and vulnerabilities and by measuring the traffic patterns of a number of popular consumer IoT devices.
2) The Bark specification language (§IV) for whitelisting access to IoT applications. For example, Bark can express policies such as "Allow any phone inside the house to control the home lighting." and "Allow residents to control the home lighting from anywhere."
3) A practical application of two-factor authentication style schemes to express network access control policies that require a user-in-the-loop or non-network context. For example, "Let my friend's phone unlock my front door if I approve it via SMS."

To show Bark's applicability for commercially available IoT devices and networks, we demonstrate an enforcement runtime on a Wi-Fi access point (§V). We also present Bark on Beetle, a gateway architecture for Bluetooth Low Energy (BLE) [11]. Our prototypes show that Bark can express diverse application semantics and transparently defend against network threats to and from the Internet of Things.

II. Background

The Internet of Things serves a wide range of uses, including home automation, smart cities, cold chain monitoring, and precision agriculture. This paper focuses on the consumer IoT consisting of end-user devices in day-to-day life. Such devices come from a multitude of hardware vendors and are beginning to see widespread adoption as prices fall to affordable levels.

Consumer IoT devices have a diverse set of functions and security requirements. They differ from nodes in traditional wireless sensor networks in that devices are often unique (in their network) and user interactive. Common device types include convenience devices such as smart lights [12] and automatic window blinds [13]; privacy-sensitive devices such as cameras and activity trackers [14]; and dangerous devices such as locks [15], ovens [16], insulin-pumps [17], and HVAC [18]. Users purchase these off-the-shelf devices and expect them to work out of the box.

Many IoT applications reside in vertically integrated silos; the embedded device connects to the manufacturer's cloud and users configure their devices using the manufacturer's app or web management interface. The August smart door lock [15], for example, attaches to the network through a single-purpose gateway: either the August app running on a smart phone or a nearby August hub device. The gateway, in turn, connects to August servers. Vertical integration has consequences for securing IoT networks as a whole. While services like IFTTT [19] allow users to automate actions across proprietary vendor APIs, users are limited to the security measures that IoT vendors implement on devices themselves. Attacks have shown these defenses to be insufficient and useless once a device has been compromised. Moreover, new vulnerabilities can be discovered over time since even security conscious developers make mistakes [20].

Despite a fairly fragmented ecosystem of devices and gateways [21], most devices use common protocols to communicate. Many use Wi-Fi to connect to an existing home access point and communicate using TCP/IP-based web protocols like HTTP and TLS. Other devices that use low-power first-hop links such as Bluetooth Low Energy, ZigBee, or Z-Wave are bridged to the Internet through gateways that use traditional Wi-Fi and Ethernet links.

Threat Model

We focus on the network (especially gateways) as a common denominator across a wide range of consumer IoT devices. Our threat model makes the following assumptions:

1) All parties except for the intended users and services of an application are untrusted. For instance, if a HVAC application consists of a thermostat, cloud servers, and a phone app on the homeowner's phone, then only these are trusted to communicate. Arbitrary hosts in the Internet and in the same network are never trusted.
2) Cloud services that communicate with IoT devices may be trusted by end-users in many cases, but might become untrusted. For example, the cloud service may become compromised or defunct while the device is still usable. Likewise, users may be concerned about remote access due to the dangerous or sensitive nature of some devices.
3) Devices may become compromised and be used to launch network attacks (e.g., DDoS against Dyn [7]). Although we cannot prevent compromised devices from exercising their regular communication patterns, we can prevent devices from communicating in novel ways (e.g., as part of the Mirai botnet).
4) Gateways that enforce policies on IoT devices in the network are trusted. Consequently, Wi-Fi routers with weak default passwords are out of scope for this paper.
5) Devices, apps, and remote servers communicate exclusively through gateways, and there are no side-channels.
6) Transparently policing network access does not create new vulnerabilities in unmodified applications. Once communication has been allowed, all existing

application-layer authentication, encryption, and integrity measures implemented by vendors continue to function.

III. The Case for Default-Off

Enabling unfettered network for IoT devices poses a large security risk, allowing remote compromise, malicious firmware upgrades, and DDoS attacks from hacked devices. Turning off the network completely, however, is not an option as this would undo all of the benefits of connected operation. At the same time, when operating correctly, IoT applications use networks in narrow and fixed ways (§III-B).

We propose a network architecture for the Internet of Things in which communication is *default-off* and must be explicitly enabled. In this model, when an IoT device joins a network, the gateway blocks all communication to and from the device. The device owner approves rules on the gateway(s) that (akin to firewall rules but designed for IoT applications) allow for connections, messages, and access to application-level resources. The rules are specific to models of devices and explainable to end-users.

While such an access control layer in the network cannot protect devices from physical hacking or tampering, it can stop endpoints from communicating in new and dangerous ways. For instance, a smart refrigerator should not under any circumstance be able to send (spam) emails to arbitrary hosts in the Internet [3]. Similarly, while the network would allow an owner to connect their laptop to their webcams, webcams cannot open new connections or query DNS servers.

A. Three Example Attacks

Many IoT devices on the market today are vulnerable. *Default-off* can protect devices from being trivially compromised, block attacks by compromised devices, and mitigate the harms of overly trusted cloud services. We describe three categories of attacks that *default-off* would address.

Directory Traversal, Port Scanning, and IoT Worms: Device vulnerabilities often result from poor input validation, inadequate authentication, and other subpar security practices. For instance, the Hajime worm [4] targets devices with open Telnet servers and weak passwords. Poor path validation allows malicious HTTP requests to traverse the file system of a commercial dishwasher [8]. In the absence of a firewall, network attackers can discover which devices are present in a network and scan for common vulnerabilities.

Dyn Attack and Botnets of Things: IoT devices can behave arbitrarily when compromised. In the Dyn attack, the Mirai botnet performed DNS lookups from tens of millions of unique IP addresses, culminating in 1.2Tbps of traffic [7]. Besides DDoS, compromised IoT devices may be used to send spam, spread malware, and anonymize illegal activities ([4], [5]).

Over-Permissioned Cloud: IoT devices trust remote servers to perform tasks such as issuing remote commands on behalf of (or even oblivious to) users. An attacker who compromises a vendor's servers can control devices, spy on users, or coordinate large scale attacks (such as force a blackout of connected lights or overload a city's energy grid). In one such attack on Avanti Markets's internal network, hackers managed to deploy malware onto thousands of kiosks manufactured by the company [22]. On a lesser scale, anyone with knowledge of a user's online login credentials can remotely control the user's home devices through existing web management portals.

B. IoT Communication Study

The consumer IoT consists of highly specialized devices, which follow narrow and stable communication patterns. The timing of communication matters as regular communication is event-driven and sparse: e.g., a smart door lock must receive packets over the network to be locked or unlocked remotely. To understand normal traffic coming into and out of IoT devices, we measured IP traffic to and from several devices, including a Belkin WeMo Mini smart plug [23], TP-Link Kasa light bulb [24], Nest thermostat [18], and an Amazon Echo [25].

The **Belkin WeMo smart plug** communicates with two servers in the cloud (EC2). It sends STUN[1] traffic for NAT traversal as well as TLS traffic for remote control. A device on the local network, running the WeMo app, communicates with the WeMo by broadcasting a SSDP[2] discovery packet to UDP port 1900. This triggers a reply from the WeMo, announcing its presence and services. The device proceeds to initiate HTTP requests to the WeMo on HTTP paths corresponding to setting binary state, firmware version, and firmware update. This exchange is unauthenticated and happens in the clear.

The **TP-Link Kasa light bulb** also supports discovery through a similar broadcast to and response from UDP port 9999. The discovering device subsequently connects to TCP port 9999. While the WeMo uses HTTP, the Kasa uses a non-standard application layer protocol when communicating locally. Remote access, by servers in the cloud, to the WeMo plug and Kasa light bulb can be disabled inside their apps.

We observed the **Nest Learning Thermostat** communicating with two servers in the cloud whenever the thermostat is set remotely. These two TLS connections are to ports 443 and 9543. The latter connection is held open to allow the Nest cloud server to push commands to the thermostat, which occurs when the thermostat is modified in the web UI, in the mobile app, or by the Amazon Echo.

Finally, the **Amazon Echo** connects to a small number of servers when spoken to and when the daily report feature is invoked. The Echo also periodically sends UDP datagrams to a range of ports used by traceroute on an EC2 server, possibly to test connectivity. Depending on the "skills" installed, the Echo makes DNS lookups for `www.meethue.com`, `opml.radiotime.com`, a subdomain of `nest.com`, and `www.example.*` domains.

In order to function correctly, each of these devices requires the ability to communicate to a very small subset of the services and endpoints in the local network and Internet.

[1]Session Traversal Utilities for NAT

[2]Simple Service Discovery Protocol, for Universal Plug and Play (UPnP)

These communication patterns are stable and expressible by whitelisting at varying resolutions, depending on whether the traffic is encrypted or not (i.e., TCP ports for TLS and paths for HTTP). At the same time, non-standard ports, a range of domains, and other protocols such as STUN and SSDP add complexity to the network behaviors.

C. Three Example Applications

Open protocols enable interesting interactions between devices and applications of all vendors. Bluetooth Low Energy is one promising example that has seen widespread commercial adoption. To demonstrate the expressiveness and precision of Bark's policies, we present three applications on Beetle, an open gateway architecture for BLE [11].

Home Lighting: The system consists of embedded devices such as wireless light bulbs, switches, and ambient sensors, grouped by the room that they are in. Just like with physical switches, anyone physically in the home can turn lights on and off with their phone, without any pre-authorization. A resident's phone, however, can turn lights on and off from anywhere. Finally, a cloud application monitors energy consumption and automatically saves power by turning off lights.

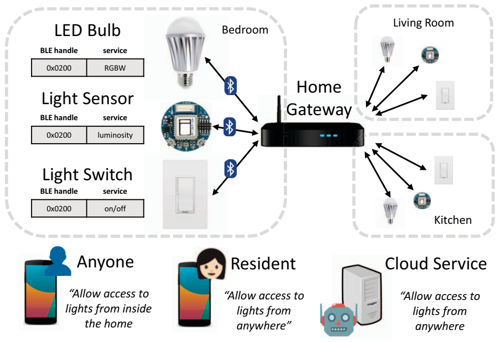


Fig. 1: Lighting application with groups and desired policies.

Cloud Monitoring: Today's devices trust the cloud with control at all times. As previously described, an attacker who can issue commands from a vendor's trusted server can wreak havoc in connected homes. We propose an alternative design where the cloud is semi-trusted. An application running in the cloud analyzes and archives data from sensors in the network in a observe-only manner. When the cloud tries to send commands to actuators, a secondary attestation mechanism proves that the cloud application acts on behalf of the user.

Smart Door Lock: A Bluetooth lock connects to a gateway and different users control it with a generic unlocking app on their phones. The homeowner's phone has full access to the lock from anywhere. A child's phone can see whether it is locked/unlocked from anywhere but cannot unlock it unless near the house (they cannot remotely let people in by accident). A guest's phone, once (remotely) authorized by the owner, can operate the lock for the few days of their stay. The guest can arrive at the home in the middle of the day

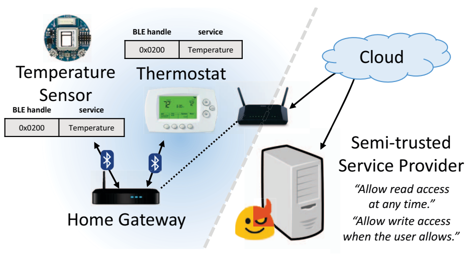


Fig. 2: Cloud monitoring and remote home control service with access to IoT devices in the home.

and receive access without requiring any preconfiguration; the owner authenticates the guest's phone when they arrive, based on a text message exchange or a short phone call. Finally, a pre-authorized house cleaner's phone can operate the lock in a certain time window every week.

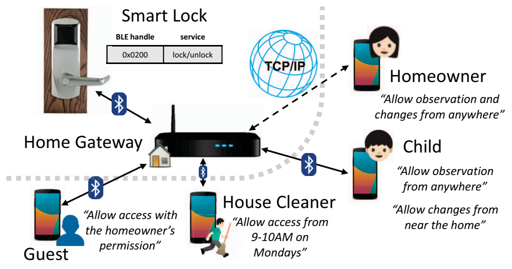

Smart Lock
BLE handle
service
0x0200
lock/unlock
TCP/IP
Homeowner
"Allow observation and changes from anywhere"
Home Gateway
Child
"Allow observation from anywhere"
"Allow changes from near the home"
House Cleaner
"Allow access from 9-10AM on Mondays"
"Allow access with the homeowner's permission"
Guest

Fig. 3: Shared door lock with multiple users, each with different access permissions.

IV. Policy Language for Default-Off

Bark enables fine-grained access for applications that require it. Intended to be understandable and mappable to natural language, Bark has a similar structure, with two levels of abstraction: types (parts of speech) and rules (sentences).

A. *Types:* who, what, where, when, how

There are five types in the Bark language, consisting of answers to the questions *who*, *what*, *where*, *when*, and *how*. These questions represent how humans naturally reason about and relate agents, actions, and resources in the physical world to principals and resources in the network.

Who refers to principals (devices and apps), and, indirectly, to users. A *who* encapsulates all of the information necessary to identify a principal to a gateway. For instance, a *who* can be a device such as `Alice's Heart Rate Monitor`, which maps to an unique device address (e.g., `A9:E7:DD:8D:BC:EF`), or a `Heart Rate App` installed on Alice's android phone identified by the tuple (`Alice's phone`, `app_id`). Likewise, local IoT devices connected to a Wi-Fi AP are identified by MAC address and servers in the cloud are named by an IP or domain name. First-hop gateways are responsible for authenticating *who*s: e.g., using pre-shared keys for Wi-Fi and association models for BLE.

Just as devices have owners in the physical world, a *who* can be registered with a human owner in Bark. For instance, Alice is the owner of `Alice's Heart Rate Monitor`, and the homeowner is a natural owner of the home lighting system.

For rules with two-factor authentication, a *who*'s owner can authorize access to the *who*'s services and authenticate the *who* when it accesses other resources in the network (see §IV-C).

What is a service that is exposed by a *who*. The format and granularity of a *what* matches the naming mechanisms of the application protocol.

Bark can provide semantically meaningful access control at a sub-*who* granularity when services themselves can be identified. BLE service and characteristic types are named by UUIDs that correspond to names such as `Heart Rate Service`, `Body Sensor Location`, and `Heart Rate Measurement`. On an IP device, services can be coarsely mapped to transport protocol and ports: e.g., TCP ports 80 (HTTP), 443 (HTTPS), 22 (SSH), 23 (Telnet). Moreover, when services are unencrypted (or the key is known to the gateway), HTTP paths and DNS names can be enumerated as strings.

Where is location both in a physical sense and in the network topology. Specifically, a *where*, for a *who*, refers to the first-hop gateway to which a device is connected.

These first-hop gateways take several forms. Some IoT devices are mobile in the physical world. Personal devices such as fitness trackers and watches associate in star-topologies around phones. Stationary devices such as locks and lights connect through fixed gateways. Remote clients connect through the home router and NAT.

How is a command or operation at the granularity of a *who*'s application layer protocol. For instance, the HTTP methods (`GET`, `POST`, `DELETE`, etc.) are the allowed operations for HTTP, while for DNS an operation can be a query. BLE supports `read`, `write`, and `subscribe` style transactions on 16-bit handles. Lastly, if only the transport protocol is known or parsable, such as for an encrypted TLS connection or a custom protocol over UDP, then the actions are just `connect` or `send_datagram/recv_response`.

When is a boolean condition such as a time restriction or the result of evaluating a boolean function.

Bark supports static time restrictions in a Cron-like format. These time restrictions are straightforward to express and are common to existing work on IoT access control systems [26]. However, static time restrictions cannot capture semantics such as "allow access for 3 minutes," so time also includes a timeout property. For example, when expressing a rule for a request with UDP transport, timeout limits for how long responses are allowed following a request.

To express more complex conditions, Bark allows users to name oracles to evaluate arbitrary boolean functions. These are evaluated over *who*, *what*, *where*, *how*, and the oracle's own knowledge and state as inputs. One way to specify an oracle is to provide a URL which Bark can query at runtime. However, in §IV-C, we propose three built-in oracle types that involve a human-in-the-loop and are sufficient to handle many common IoT patterns.

B. Rules

Humans describe access to resources using sentences. We say "let Bob use my car for a day" and likewise for IoT devices such as a light or a lock. Our understanding of sentences involves a subject, a verb (an action), an object, and modifiers (e.g., conditions). Like human language, rules in Bark define permissible network communication in terms of **subjects**, **objects**, **actions**, and **conditions**.

A **subject** is a pair consisting of a *who* and a *where*, capturing the client principal and its location in the network.

An **object** refers to the resource being accessed and is described by a 3-tuple: a *who*, a *where*, and a *what*. These are the serving entity, its location, and the service itself.

An **action** is a *how* that defines how the subject is allowed to interact with the object.

Finally, **conditions** is an algebraic expression of boolean *when*s combined with $\wedge$, $\vee$, and $\neg$ that express time restrictions, complex expressions, and non-network state.

In summary, subjects, objects, and actions are composed from Bark types, namely *who*s, *where*s, *what*s, and *how*s. As an example, the **subject** and **object** for a rule to allow a light switch to modify a light bulb when both are connected to a home gateway are `(light switch, home gateway)` and `(on/off, light bulb, home gateway)`, respectively. The **action** is `write`, and the **conditions** are `any time`. Fig. 4 generalizes this structure. Because the underlying mappings that Bark types represent are protocol and device specific, Bark can express policies for IoT devices of varying networks and functions.

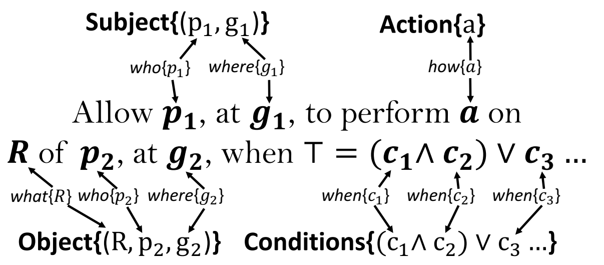

Subject{(p_1, g_1)}
Action{a}
who{p_1}
where{g_1}
how{a}
Allow p_1, at g_1, to perform a on
R of p_2, at g_2, when $\top = (c_1 \wedge c_2) \vee c_3$ ...
what{R}
who{p_2}
where{g_2}
when{c_1}
when{c_2}
when{c_3}
Object{(R, p_2, g_2)}
Conditions{$(c_1 \wedge c_2) \vee c_3$ ...}

Fig. 4: *who*, *what*, *where*, *when*, and *how* types form the **subject**, **object**, **action**, and **conditions** and allow for human interpretation of rules.

C. Two-Factor Authentication Style Oracles

IoT traffic is often sporadic and spurred on by physical events caused by some user: e.g., unlocking a door or flipping a switch. In these situations, it is reasonable to have a user or machine oracle in-the-loop when making access decisions. We propose three oracles for *when* to serve as generic second factors to "authenticate" an action, utilizing concepts that end-users are already familiar with from other computing contexts. In English, these conditions are:

User Attestation: *"the user attests to the subject's action."*

When a *who* attempts to perform an action or access a resource in the network, we determine that the *who* is acting faithfully on the user's behalf. By default, the user is the **subject** *who*'s owner. One example where this condition can be applied is in the cloud monitoring application (to ensure that the cloud cannot modify the HVAC settings at arbitrary times of day without the user's knowledge).

Owner Approval: *"the object's owner authorizes an action."*
A device owner wishes to be notified to grant access each time access is requested (e.g., for their door lock). The owner of the **object**'s *who* receives a prompt (via SMS).

Password Auth: *"the user provides pre-shared credentials."*
Embedded IoT devices commonly lack user interfaces for users to provide credentials for authentication. With this condition, the user must present a password to the gateway on behalf of their device, via a form UI. Password authentication can be used to share IoT devices (similar to using a Wi-Fi password) and they can be one-time-use and **subject** specific.

D. Groups and Wildcards

Groups simplify management of entities in the network and exist in most related access control systems. Policies sometimes need to address more than just a single subject or object. The lighting application, for example, needs rules to grant access to all of the lights in a room. Since several rules may refer to all of the lights, being able to refer to them collectively simplifies adding and removing new lights, as doing so requires no changes to the rules.

Bark allows users to group related principals (*who*), gateways (*where*), and services (*what*) and address them collectively. For instance, a group of *who*s called `family` might include Alice, Bob, and Eve's phones. Similarly, `home gateways` (a group of *where*s) can include gateways on different floors of the home.

Policies also often need to specify that anything of the matching type is allowed. This allows **subject**s and **object**s to match to devices and services whose names are not yet known. The lighting application, for example, needs to grant anyone who is physically home access to the lights. Bark supports such "any" specifications through wildcards. Wildcards enable patterns such as any *who* at a *where*, or a specific *who*, any-*where*.

Some *what*s also use wildcards internally. For instance, in the BLE service hierarchy, possible expressions include "any BLE characteristic under a particular service" or "a characteristic under any service." For HTTP and DNS, a regular expression or simple pattern can express a set of paths and domains (e.g., `/login/*` and `*.nest.com`).

Dynamic Rule Generation (All vs. One): Groups and wildcards complicate the interpretation of conditions (especially those that have a user-in-the-loop). When a condition is satisfied for a rule containing groups or wildcards, the rule needs to distinguish whether the condition has been satisfied for every element or only the one that triggered rule evaluation. For instance, if a rule says "allow any device ($*$) to connect to the front door lock when the homeowner grants access," it needs to distinguish whether access is granted to all devices or only to the specific device that attempted to connect.

Bark removes this ambiguity with `all` or `one` annotations on groups and wildcards, which indicate how $*$ binds. In the lock example, approving access for the `one` guest device means that the dynamically generated rule grants access to only that device. In contrast, using `all` in the light example gives access to every light.

Explainable Matching and Overlap Resolution: Rule matching must be understandable, predictable, and intuitive, even when groups and wildcards can cause overlap. Rules in Bark can only whitelist access, and Bark handles overlapping rules by considering their disjunction (logical-$\vee$). A whitelist-only scheme improves understandability and implies that a particular communication is permitted if and only if there is at least one rule that allows it. This contrasts with other systems that allow priority levels, negation, and chaining of rules.

Although Bark does not inherently prevent overly generic rules from being written (e.g., "allow everyone access to everything at any time"), it is easy to log which rules are in effect for each access so that overly broad rules can be identified and removed.

E. Managing Liveness with Caching and Leases

Bark policies are intended to be managed centrally, but enforcement occurs in the data plane, consisting of one or more of gateways. Liveness requires that applications must continue to operate even when the network is unreliable or in a partitioned state; a system that relies completely on the cloud or has a single point of failure would fail this requirement. Also, we require that installation of new rules and revocation of access be verifiable and happen in bounded time.

These properties are obtained with caching and leases. Gateways replicate rules for their valid lease duration. When a user installs a new rule, an error is reported if installation fails or the rule cannot be replicated. In case permissions change, access is verified continuously, not just at the start of a connection. Finally, if an expired lease cannot be renewed, then access reverts to *default-off*.

Oracles too can become partitioned from the network or inundated with requests. Results returned by *when*-oracles in conditions are cached for a time dictated by the oracle. This has the added effect of reducing the overhead of reevaluation, such as when human input is needed.

V. Implementation

We implemented a policy server along with a web-based UI for the user to list rules and manage allowed communication. The user can register new devices and gateways, as well as construct and modify rules using Bark primitives: *who*, *what*, *where*, *when*, and *how*. The user can also attach conditions such as two-factor authentication and other oracles. Currently, our server manages two forms of gateways, a Wi-Fi AP running Linux and a BLE gateway.

A. Wi-Fi Access Point

To demonstrate Bark with today's devices, we implemented a daemon in Python for a Wi-Fi AP running Linux. The daemon intercepts all IP packets to and from devices in the network by interfacing with `netfilter` and `dnsmasq`.

Naming (at the Link Layer): Devices inside the local network are identified by their MAC address and referenced with human understandable names. For instance, our Belkin WeMo smart plug has MAC address `60:38:e0:e7:b1:79` from the vendor and a name given by the user at deployment time (e.g., `bedroom-plug`). These MAC addresses map to names (the *who* in rule subjects and objects) and also to IP addresses from leases at the DHCP server.[3]

Transport Layer: Our runtime supports TCP and UDP protocols. Connection state is tracked for each TCP flow. When a host attempts to initiate a connection with a SYN, the gateway retrieves the *who* corresponding to the source and destination addresses, which are then matched with rule subjects and rule objects. Likewise, the same procedure applies for UDP. Response packets are allowed for a limited amount of time, set as a timeout when specifying the rule action. This is sufficient, for instance, to express the UPnP/SSDP discovery exchange where a host broadcasts a UDP packet and UPnP devices unicast responses. Patterns such as DNS query and response can also be expressed with timeouts.

Application Layer: Although a rule that enables on for TCP port 80 and UDP port 53 can already whitelist HTTP and DNS, this is still more permissive than needed. Instead, the gateway also filters HTTP and DNS traffic at the application layer. By specifying *how* and a finer *what*, Bark can restrict queries to only a small set of operations on domain names or specific URIs: e.g., a firmware update URI on the Belkin WeMo plug, `/upnp/event/firmwareupdate1`.

B. Bluetooth Low Energy Gateway

Multiple BLE gateways constitute the IoT data plane. Gateways communicate with the policy server and each other to build a circuit-switched network of BLE attribute handles.

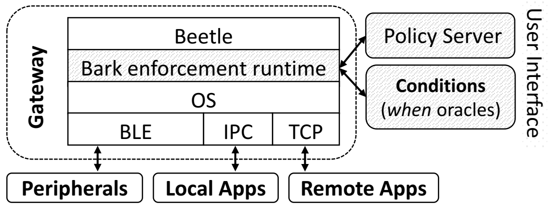


Fig. 5: Bark system architecture for BLE.

BLE Gateway: BLE devices use the Generic Attribute Profile (GATT) as their application layer protocol [28]. Transactions such as `read`, `write`, and `subscribe/notify` in GATT are performed on attributes in a 16-bit handle space. Each attribute has a type, given by its UUID. To expose a higher-level interface to applications, GATT reserves special UUIDs and adds handles that delimit services, characteristics, and descriptors in a hierarchy. The protocol is self-describing and clients can perform discovery when connecting.

Beetle [11] gateways enable multi-hop interactions between BLE peripherals and applications by circuit switching over handles. We extend Beetle to use Bark rules when forwarding.

Policy Enforcement: When a packet arrives at a BLE gateway, the enforcement runtime derives the **subject** from the source device and its gateway, **object** from the destination device, gateway, and handle. The **action** is the GATT opcode.

The enforcement runtime decides whether a transaction should be allowed to traverse the network by querying its rules for matching subjects, objects, and actions, and then evaluating conditions (*when*) to decide access (Fig. 6).

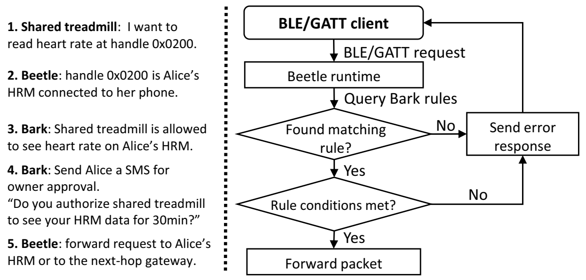

1. Shared treadmill: I want to read heart rate at handle 0x0200.
2. Beetle: handle 0x0200 is Alice's HRM connected to her phone.
3. Bark: Shared treadmill is allowed to see heart rate on Alice's HRM.
4. Bark: Send Alice a SMS for owner approval. "Do you authorize shared treadmill to see your HRM data for 30min?"
5. Beetle: forward request to Alice's HRM or to the next-hop gateway.
BLE/GATT client
BLE/GATT request
Beetle runtime
Query Bark rules
Found matching rule?
No
Send error response
Yes
Rule conditions met?
No
Yes
Forward packet

Fig. 6: Logic to determine whether a request is permitted, in this case to a BLE heart rate monitor.

VI. Evaluation

We evaluate the expressiveness of Bark and its practicality for preventing contemporary IoT attacks. In each figure, underlined identifiers represent principals, gateways, resources, and groups that are registered beforehand and known to the policy server and gateways.

A. Wi-Fi/IP: Enabling On for Existing IoT

§III-B characterized traffic from several off-the-shelf IoT devices: an Amazon Echo [25], a Nest Thermostat [18], a Belkin WeMo plug [23], and a TP-Link Kasa light bulb [24]. Bark is sufficiently expressive to enable network access for these devices in the ways that they were intended to be used.

Connecting IoT to the Cloud: IoT devices typcially initiate a small number of connections to servers in the cloud. Fig. 7 shows a rule for the Nest Thermostat to allow it to connect to the cloud with two destination ports.

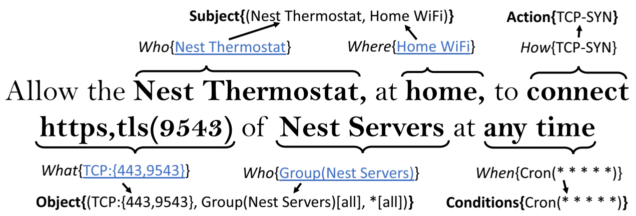

Subject{(Nest Thermostat, Home WiFi)}
Action{TCP-SYN}
Who{Nest Thermostat}
Where{Home WiFi}
How{TCP-SYN}
Allow the Nest Thermostat, at home, to connect https,tls(9543) of Nest Servers at any time
What{TCP:{443,9543}}
Who{Group(Nest Servers)}
When{Cron(* * * * *)}
Object{(TCP:{443,9543}, Group(Nest Servers)[all], *[all])}
Conditions{Cron(* * * * *)}

Fig. 7: Rule to enable connections from the Nest thermostat to servers the cloud. Nest servers is a group of domains/IPs (*whos*) that the Nest is allowed to contact.

[3]While MAC and IP addresses can be spoofed, we consider this to be out of scope for this paper. Specifically, the Wi-Fi access point is responsible for authenticating local devices using one or more PSKs. [27] proposes unique PSKs for each device to prevent spoofing of link-layer addresses.

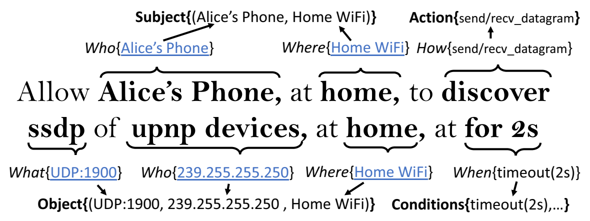

Subject{(Alice's Phone, Home WiFi)}
Action{send/recv_datagram}
Who{Alice's Phone}
Where{Home WiFi}
How{send/recv_datagram}
Allow Alice's Phone, at home, to discover ssdp of upnp devices, at home, at for 2s
What{UDP:1900}
Who{239.255.255.250}
Where{Home WiFi}
When{timeout(2s)}
Object{(UDP:1900, 239.255.255.250 , Home WiFi)}
Conditions{timeout(2s),...}

Fig. 8: Rule to allow discovery of the WeMo plug. The WeMo has 2s to reply to the phone's broadcast discovery messages.

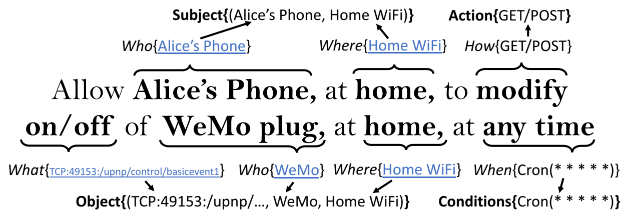

Subject{(Alice's Phone, Home WiFi)}
Action{GET/POST}
Who{Alice's Phone}
Where{Home WiFi}
How{GET/POST}
Allow Alice's Phone, at home, to modify on/off of WeMo plug, at home, at any time
What{TCP:49153:/upnp/control/basicevent1}
Who{WeMo}
Where{Home WiFi}
When{Cron(* * * * *)}
Object{(TCP:49153:/upnp/..., WeMo, Home WiFi)}
Conditions{Cron(* * * * *)}

Fig. 9: Rule to allow modification to the WeMo's state. The WeMo runs a HTTP server on port 49153 and uses the path `/upnp/control/basicevent1` to toggle state.

Supporting Local Communication: Both the Belkin WeMo smart plug and TP-Link Kasa light bulb allow hosts in the local network to control actuation. Granting a *who* (e.g., Alice's smartphone) this permission involves one rule to allow for discovery and one rule to allow for connections. Figs. 8 and 9 show these patterns for the WeMo smart plug over HTTP.

Filtering DNS Queries: In normal use, IoT devices interact with network services such as DNS in predictable ways: e.g., to resolve a small set of domains to IPs. Bark can limit the queries that an IoT device such as Amazon Echo is allowed to make. Recall that the Echo queries for `www.meethue.com`, `opml.radiotime.com`, `*.nest.com`, and `www.example.*`. Fig. 10 shows a rule that would allow a device to use DNS, but prevent denial-of-service attacks against DNS if the device were to be hacked. With user-in-the-loop *when*s, we can also install a second rule that causes the device owner to be alerted if these traffic patterns change.

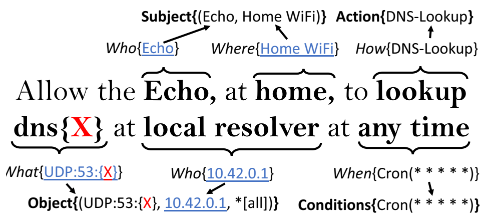

Subject{(Echo, Home WiFi)}
Action{DNS-Lookup}
Who{Echo}
Where{Home WiFi}
How{DNS-Lookup}
Allow the Echo, at home, to lookup dns{X} at local resolver at any time
What{UDP:53:{X}}
Who{10.42.0.1}
When{Cron(* * * * *)}
Object{(UDP:53:{X}, 10.42.0.1, *[all])}
Conditions{Cron(* * * * *)}

Fig. 10: Rule for an Echo to query for a set, **X**, of domain names that have been explicitly whitelisted.

B. Wi-Fi/IP: Three Example Attacks

The three attacks described in §III-A can be prevented with a *default-off* communication model and reasonable Bark rules.

Directory Traversal, Port Scanning, and IoT Worms: First, the directory traversal attack on the Miele dishwasher [8] is preventable by Bark. Bark sees the dishwasher as a *who* and its set of valid HTTP paths as *what*s. A relative path such as `/../../../../../` would not match a valid *what*.

More generally, *default-off* would block port scans at the gateway since no rule would exist to allow traffic to all destination ports. Only ports required for the device to function would be allowed; e.g., a single function IoT device should not be allowed to listen on Telnet and SSH ports or act as a DNS resolver (as some devices do [29]). Blocking these ports and traffic from unauthorized hosts eliminates the most common attack vectors that are used by IoT worms ([4], [5]) to infect new devices.

Dyn Attack and Botnets of Things: Bark would prevent IoT devices from sending DNS lookups or initiating a large number of connections to an unwhitelisted server for DDoS purposes. Likewise, IoT devices such as lightbulbs and cameras would not be allowed by default to send email with SMTP or to attack other devices in the same network.

Over-Permissioned Cloud: While Bark does not prevent hackers from gaining control of vendors' clouds or users from losing account credentials, Bark rules can impose restrictions on when remote control from the cloud is allowed or considered safe. For instance, traffic from remote servers to a connected oven may only be allowed during times of day when an adult resident is at home or able to authenticate the access. In §VI-C, we show how Bark can prevent actuation without the owner's knowledge by adding two-factor authentication.

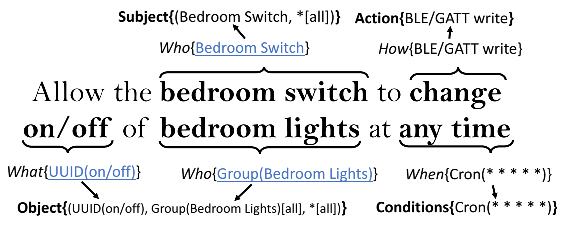
C. BLE: Three Example Applications

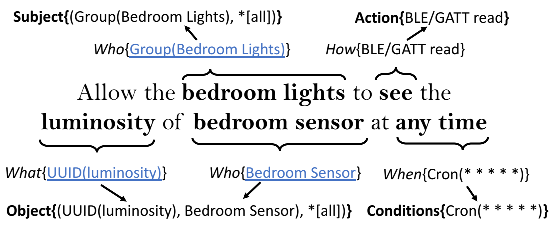
We further evaluate the expressiveness, precision, and presentablility of Bark rules using the three application examples from §III-C by showing the policies associated with each one.

Home Lighting: A home lighting system consists of fixed devices such as light-bulbs, switches, and ambient sensors. These devices are grouped according to the room that they are in and their arrangement doesn't change. To express access rules in Bark, a user first registers these devices and assigns each to the appropriate room group. The light-bulbs have an `on/off` service and a `RGBW` service that controls color and luminosity; recall that BLE uses UUIDs to indicate service types. The sensors have a service for `luminosity`.

For each room in the building, the switch requires access to the `on/off` service on each light bulb in the room. Each light bulb requires access to luminosity on the sensor. Figs. 11 and 12 show how to phrase these accesses for a particular room (bedroom) into Bark rules using **subject**, **object**, **action**, and **conditions**.

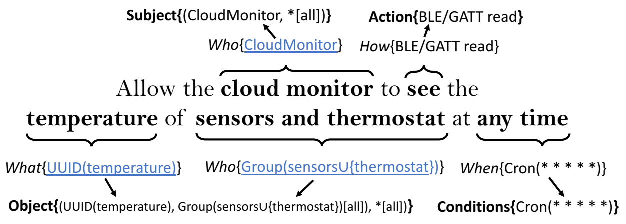
The system also grants full access to all lights to anyone physically in the home, to residents, and to a cloud service. This can be expressed in three rules. The **object** `(UUID(on/off), Group(All Lights)[all], *[all])`, **action** `(read/write)`, and **conditions** `(any time)` are the same for each. The rule **subject**s are `(*[all], Home Gateway)` (anyone connected directly to the home gateway), `(Group(Residents'`

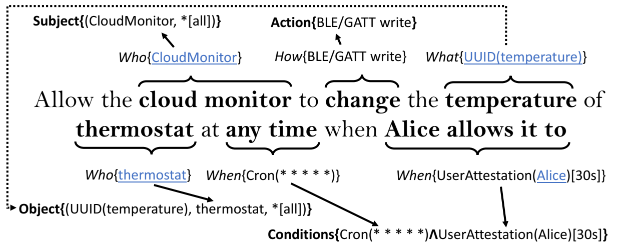

Subject{(Bedroom Switch, *[all])}
Action{BLE/GATT write}
Who{Bedroom Switch}
How{BLE/GATT write}
Allow the bedroom switch to change on/off of bedroom lights at any time
What{UUID(on/off)}
Who{Group(Bedroom Lights)}
When{Cron(* * * * *)}
Object{(UUID(on/off), Group(Bedroom Lights)[all], *[all])}
Conditions{Cron(* * * * *)}
Fig. 11: Rule for a switch and a group of lightbulbs.


Subject{(Group(Bedroom Lights), *[all])}
Action{BLE/GATT read}
Who{Group(Bedroom Lights)}
How{BLE/GATT read}
Allow the bedroom lights to see the luminosity of bedroom sensor at any time
What{UUID(luminosity)}
Who{Bedroom Sensor}
When{Cron(* * * * *)}
Object{(UUID(luminosity), Bedroom Sensor), *[all])}
Conditions{Cron(* * * * *)}
Fig. 12: Rule for a group of lightbulbs and a sensor.

`Phones)[all], *[all])` (residents anywhere), and `(Cloud Service, *[all])` (cloud service anywhere).

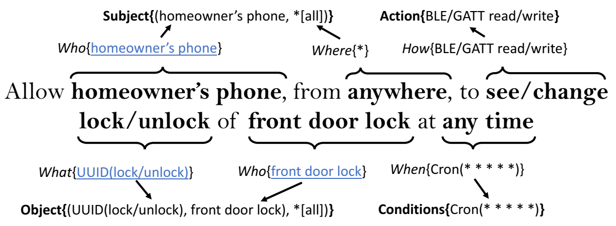
Cloud Monitoring: Applications in the cloud provide remote monitoring and control of IoT devices. We list rules for a cloud-based HVAC application with access to a smart thermostat and sensors in the user's home. The thermostat and sensors expose `temperature` services. The cloud application allows the user to view heating history and make adjustments remotely in a web interface. The cloud application can read the temperature service from the sensors and the thermostat at all times but can only set the temperature when a SMS message confirms that user has instructed it to do so. This requires two rules (Figs. 13 and 14).

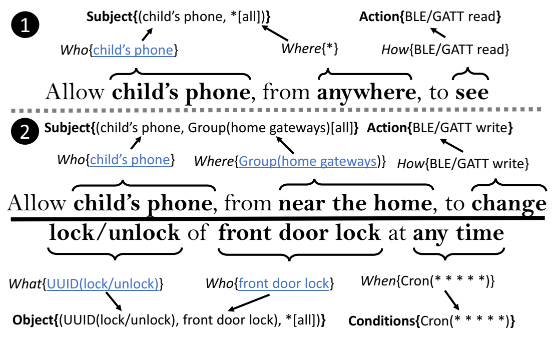

Subject{(CloudMonitor, *[all])}
Action{BLE/GATT read}
Who{CloudMonitor}
How{BLE/GATT read}
Allow the cloud monitor to see the temperature of sensors and thermostat at any time
What{UUID(temperature)}
Who{Group(sensorsU{thermostat})}
When{Cron(* * * * *)}
Object{(UUID(temperature), Group(sensorsU{thermostat})[all]), *[all])}
Conditions{Cron(* * * * *)}
Fig. 13: Rule to allow passive monitoring of the HVAC system.

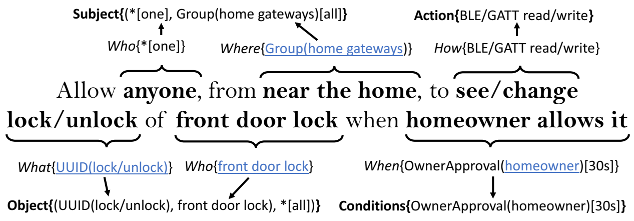

Subject{(CloudMonitor, *[all])}
Action{BLE/GATT write}
Who{CloudMonitor}
How{BLE/GATT write}
What{UUID(temperature)}
Allow the cloud monitor to change the temperature of thermostat at any time when Alice allows it to
Who{thermostat}
When{Cron(* * * * *)}
When{UserAttestation(Alice)[30s]}
Object{(UUID(temperature), thermostat, *[all])}
Conditions{Cron(* * * * *)∧UserAttestation(Alice)[30s]}
Fig. 14: Rule to allow the changing of the temperature setting.

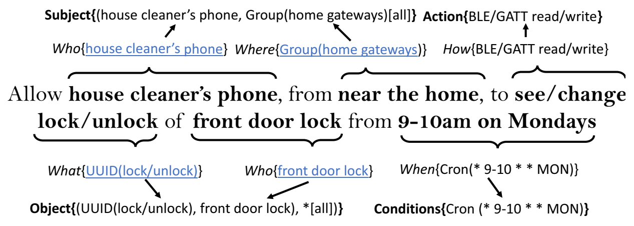
Smart Lock: Locks require policies that grant access to very specific groups of users under various contexts. We describe a generic lock connected to a gateway. A user's phone connects to the lock through the gateway.

Users for this system include (but are not limited to) a homeowner, child, guest, and house cleaner. These users' phones are added as *who*s in Bark and each is given a different set of permissions appropriate to their person. The access rules that we proposed in §VI-C are expressed in figs. 15 to 18.


Subject{(homeowner's phone, *[all])}
Action{BLE/GATT read/write}
Who{homeowner's phone}
Where{*}
How{BLE/GATT read/write}
Allow homeowner's phone, from anywhere, to see/change lock/unlock of front door lock at any time
What{UUID(lock/unlock)}
Who{front door lock}
When{Cron(* * * * *)}
Object{(UUID(lock/unlock), front door lock), *[all])}
Conditions{Cron(* * * * *)}
Fig. 15: Rule for a homeowner, who can observe and change the lock from anywhere at all times.


1
Subject{(child's phone, *[all])}
Action{BLE/GATT read}
Who{child's phone}
Where{*}
How{BLE/GATT read}
Allow child's phone, from anywhere, to see
2
Subject{(child's phone, Group(home gateways)[all]}
Action{BLE/GATT write}
Who{child's phone}
Where{Group(home gateways)}
How{BLE/GATT write}
Allow child's phone, from near the home, to change
lock/unlock of front door lock at any time
What{UUID(lock/unlock)}
Who{front door lock}
When{Cron(* * * * *)}
Object{(UUID(lock/unlock), front door lock), *[all])}
Conditions{Cron(* * * * *)}
Fig. 16: Two rules for the child, who can observe the lock from anywhere but only unlock when nearby.


Subject{(*[one], Group(home gateways)[all]}
Action{BLE/GATT read/write}
Who{*[one]}
Where{Group(home gateways)}
How{BLE/GATT read/write}
Allow anyone, from near the home, to see/change lock/unlock of front door lock when homeowner allows it
What{UUID(lock/unlock)}
Who{front door lock}
When{OwnerApproval(homeowner)[30s]}
Object{(UUID(lock/unlock), front door lock), *[all])}
Conditions{OwnerApproval(homeowner)[30s]}
Fig. 17: Rule for the guest, who needs approval from the homeowner to unlock the door. After the homeowner authorizes access, a new rule is generated using $*[one]$.


Subject{(house cleaner's phone, Group(home gateways)[all]}
Action{BLE/GATT read/write}
Who{house cleaner's phone}
Where{Group(home gateways)}
How{BLE/GATT read/write}
Allow house cleaner's phone, from near the home, to see/change lock/unlock of front door lock from 9-10am on Mondays
What{UUID(lock/unlock)}
Who{front door lock}
When{Cron(* 9-10 * * MON)}
Object{(UUID(lock/unlock), front door lock), *[all])}
Conditions{Cron (* 9-10 * * MON)}
Fig. 18: Rule for the house cleaner, who has periodic access.

Unlike HomeOS [26], which uses smart locks with two-factor authentication as an example application, Bark introduces multiple factors in the network without needing a custom application. In this way, other applications can directly benefit without needing to convolute their logic.

D. Performance Considerations

Although optimizing for performance is not the primary objective of the Bark language, we demonstrate that enforcing Bark rules does not significantly degrade performance even on modest hardware (Raspberry Pi 3 in our experiments). Enforcing Bark can introduce latency in three ways. (1) Rules that require a user-in-the-loop rules can delay or timeout traffic. (2) Enforcing Bark adds logic into the network to inspect packets prior to forwarding. (3) Gateways must query for and evaluate rules. Factors such as user response time and unreliability in receiving responses (e.g., SMS) are out of scope for this paper. We will discuss (2) and (3).

Fig. 19 measures our Wi-Fi prototype with 1,000 rules installed on the gateway. Since rules whitelist access, evaluation stops as soon as one is satisfied. Static elements of rules such as *who* and *where* can be matched when retrieving rules from a local database using SQL, however, conditions and wildcards require additional evaluation. In the worst case, several overlapping rules are applicable, but conditions evaluate to `false`. Our experiments show that, for a reasonable number of applicable rules (<10), the additional latency is unnoticeable to users. Large numbers of nearly redundant rules (≥ 30) introduce latency and variance for flows with more data, but we believe that there is ample room for improvement with a more efficient firewall implementation (and better written rules if this becomes the case).

Adding Bark to a multi-hop Beetle network does not introduce significant performance overhead. Latency and throughput values (and their variance) for BLE are dominated by the connection interval at the BLE link layer (Fig. 20).

VII. Discussion

Making communication for the IoT *default-off* yields substantial security benefits. In this section, we discuss using and extending Bark to enable access when communication is off by default.

A. Interacting With and Writing Bark Rules

Explainability of rules to end-users is required to manage devices, incorporate new devices, and delegate permissions to apps and other people. End-users interact with IoT devices physically and digitally. Bark hides firewall details from the end-user, presenting rules as high-level subjects, objects, actions, and conditions that are explainable by device function.

While our measurements of commercial IoT devices demonstrate that it is possible to map devices' raw communication patterns into Bark types, doing so can require knowledge and skills beyond those of the average user (for example, understanding DNS, UPnP, and NAT traversal protocols or vendor-defined APIs such as `/upnp/control/basicevent1`). However, once types such as *who*, *what*, *how*, and *when* have been defined once by an expert, they can be copied and used by end-users in their rules. For instance, when a user adds a new WeMo smart plug as a *who*, they name the device and indicate to the policy server the device model. Device model then determines the *what*s and *how*s that the device supports.

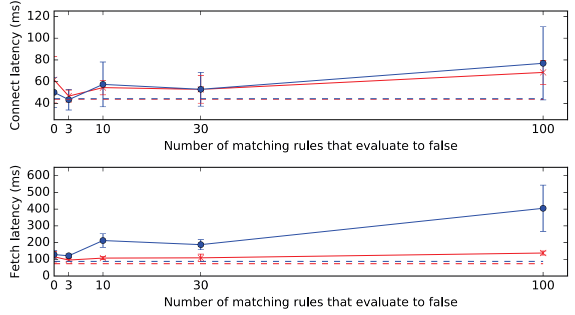


Fig. 19: The solid lines show connection and request latency when fetching 1KB (×) and 10KB (●) of data with the Bark daemon running and a variable number of matching rules where conditions are `false`. Dashed lines are baseline without the daemon. Latency increases as the number of rules that are statically matched and fruitlessly evaluated increases, but remains small for reasonable numbers of matched rules.

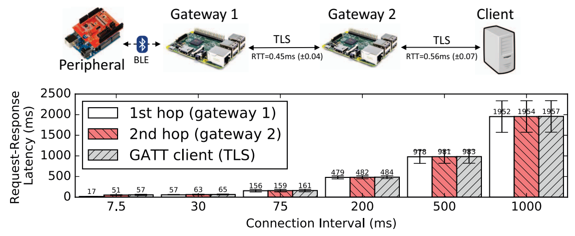


Fig. 20: End-to-end request-response latency for a read transaction across two Beetle gateways.

Improving the user experience for composing rules is an interesting area of future research. A more intelligent interface might suggest policies based on rule templates, written and verified by IoT vendors or community experts. Likewise, smart gateways could learn default rules through traffic analysis or trust on first use (the device accesses all of the resources it requires after registration).

B. External Oracles for When

Binding a large number of devices to two-factor authentication with SMS or passwords will create usability problems for users. External oracles can maintain security, while replacing user prompts in most use cases by evaluating conditions on information from outside of the network. For example, an oracle can require "an app to be in the foreground on the homeowner's phone" before it can access peripherals, bypass owner approval when "the homeowner is at home," or allow the cloud monitoring system to adjust temperature if the "ambient temperature falls below $26°C$." Expressing these conditions with an external oracle means that Bark can act on complex events, without the need to understand applications and their data.

C. Extensions to Bark

The goal of extending Bark is to enable new network types to be supported and new policies to be written. New networks and gateways will require mappings for *who*, *what*, *where*, and *how* that are appropriate to translate information about network traffic to entities and concepts that users can understand. Bark's rule structure (consisting of subject, object, action, and conditions) remains the same. As mentioned previously, oracles for *when* provide a way to improve the expressiveness of policies while keeping Bark generic.

VIII. Related Work

Smart home applications have given rise to commercial frameworks such as HomeKit [30], Weave [31] and SmartThings [32]. Such frameworks include built-in systems to delegate application permissions to users and accounts. However, these settings alone are insufficient under Bark's threat model. Applications in these systems can have vulnerabilities [33] and they cannot stop attackers from changing device behavior once a device is compromised.

Existing frameworks ([26], [34], [35]) and libraries [36] require modification to applications or modules to adapt existing APIs. HomeOS [26] achieves centralized management through device-specific drivers. SIFT [35] focuses on detecting safety violations and policy conflicts for applications expressed with high-level intents. In contrast, Bark focuses on IoT communication and interposes in the network. Bark is complementary to systems like IFTTT [19] and Amazon Echo [25], targeting the security of rather than the automation of device APIs.

Access control is a well-studied topic of operating systems, file systems, and traditional networks. Existing works include both policy specification and enforcement mechanisms.

ABAC [37] and RBAC [38] are common models for access control, using attributes and roles, respectively. Bark can be seen as a subset of ABAC (and awkwardly as RBAC with roles defined for each device). To use these models in practice, a policy specification language is needed. Languages such as XACML [39], while expressive, are not tailored to an IoT application and network setting.

Taos [40] introduced rule logic and authentication in an OS setting, and subsequent policy-driven access control systems adopt similar forms of formal rules and logic. For mobile OSes, CRePE [41] and Saint [42] propose context-aware policies and permission assignment, which use conditions like Bark. HomeOS [26] expresses access control with datalog rules, and BOSS [43] introduces an authorization service within their building architecture. Labels in information flow control ([44], [45]) do not map cleanly to the IoT device semantics that Bark targets.

Network firewalls and content filters are often configured in a whitelist or blacklist setting. For IoT devices, blacklisting is either too permissive or leads to an explosion of rules. *Default-off* and whitelisting are not new ideas [46], but traditional firewall implementations are too coarse and do not exploit the narrow functionality of IoT devices for configuration and maintenance. Moreover, complexity is introduced by priorities, ordering, negation, and chaining of rules [47], degrading the understandability of the system.

Smart, commercially-obtainable IoT APs [48] automatically inspect traffic and mitigate attacks. Bark applies similar techniques of deep packet inspection to IoT traffic but also gives users visibility into the permissions and rules.

Usability requires that humans are able to comprehend and conceive policies in the system. End-users are already familiar with managing apps on phones and online accounts; Bark provides the technical means to analogously manage IoT permissions. At a developer level, network behaviors in Bark are similar to *intents* in Android ([41], [42]). These entities must be specified by some human (not necessarily the end-user) for access control to be semantically meaningful. Statistical inference [29] has a potential role in making security for the IoT seamless, but cannot replace all human configuration and decision making: for instance, when new permissions need to be granted to a guest.

IX. Conclusion

Recent attacks warrant heightened concern over the security of IoT devices and their networks. Fortunately, the security of these devices and networks can be greatly improved by a *default-off* communication model. This paper proposed Bark, a new policy language for whitelisting access for IoT applications when communication is *default-off*. Bark enables semantically meaningful access control policies that are both transparently enforceable in existing networks and presentable to users (in terms of natural language and physical applications).

Billions of IoT devices already exist in the wild. Many will remain deployed and vulnerable for years to come. Bark demonstrates that better access control policies in the network can mitigate the threats to and posed by the Internet of Things.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and feedback. This work is supported by Intel/NSF CPS Security grant #1505684, the Secure Internet of Things Project, the Stanford Data Science Initiative, and gifts from Google, VMware, Analog Devices, and Qualcomm.

References

[1] E. Ronen, A. Shamir, A. O. Weingarten, and C. OFlynn, "IoT Goes Nuclear: Creating a ZigBee Chain Reaction," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 195–212.

[2] P. Oltermann, "German parents told to destroy doll that can spy on children," Feb. 2016. [Online]. Available: https://www.theguardian.com/world/2017/feb/17/german-parents-told-to-destroy-my-friend-cayla-doll-spy-on-children

[3] Proofpoint, "Your Fridge is Full of SPAM: Proof of An IoT-driven Attack," 2014. [Online]. Available: https://www.proofpoint.com/us/threat-insight/post/Your-Fridge-is-Full-of-SPAM

[4] S. Edwards and I. Profetis, "Hajime: Analysis of a decentralized internet worm for IoT devices," Oct 2016. [Online]. Available: https://security.rapiditynetworks.com/publications/2016-10-16/hajime.pdf

[5] O. Bilodeau and T. Dupuy, "Dissecting Linux/Moose - The Analysis of a Linux Router-based Worm Hungry for Social Networks," May 2015. [Online]. Available: https://www.welivesecurity.com/wp-content/uploads/2015/05/Dissecting-LinuxMoose.pdf

[6] US-CERT, "Alert (TA16-288A): Heightened DDoS Threat Posed by Mirai and Other Botnets," 2016. [Online]. Available: https://www.us-cert.gov/ncas/alerts/TA16-288A

[7] S. Hilton, "Dyn Analysis Summary Of Friday October 21 Attack," 2016. [Online]. Available: https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/

[8] NIST, “National Vulnerability Database - CVE-2017-7240,” March 2017. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-7240

[9] D. Desai, “IoT devices in the enterprise - A look at the enterprise IoT device footprint and IoT traffic analysis,” Nov 2016. [Online]. Available: https://www.zscaler.com/blogs/research/iot-devices-enterprise

[10] M. Weiser and J. S. Brown, “Beyond calculation,” P. J. Denning and R. M. Metcalfe, Eds. New York, NY, USA: Copernicus, 1997, ch. The Coming Age of Calm Technolgy, pp. 75–85. [Online]. Available: http://dl.acm.org/citation.cfm?id=504928.504934

[11] A. A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein, “Beetle: Flexible Communication for Bluetooth Low Energy,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’16. New York, NY, USA: ACM, 2016, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906414

[12] Philips, “Wireless and smart lighting by Philips - Meet Hue,” 2018. [Online]. Available: https://www2.meethue.com/

[13] MySmartBlinds, “Smartphone Controlled Blinds,” 2016. [Online]. Available: https://www.mysmartblinds.com/

[14] Fitbit, “FitBit: Official Site for Activity Trackers and More,” 2018. [Online]. Available: https://www.fitbit.com/

[15] August, “August Smart Lock - Your Smart Home Starts at the Door,” 2018. [Online]. Available: https://august.com/

[16] Anova, “Introducing the Anova Precision Oven,” 2016. [Online]. Available: https://anovaculinary.com/introducing-the-anova-precision-oven/

[17] Cellnovo, “The Cellnovo Diabetes Management System,” 2016. [Online]. Available: https://www.cellnovo.com/

[18] Nest, “A Nest Thermostat for every home,” 2018. [Online]. Available: https://nest.com/thermostats/

[19] IFTTT, “IFTTT helps your apps and devices work together,” 2018. [Online]. Available: https://ifttt.com/

[20] *Backdooring the Frontdoor: Hacking a “perfectly secure” smart lock.*, 2016. [Online]. Available: https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Jmaxxz-Backdooring-the-Frontdoor.pdf

[21] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta, “The Internet of Things Has a Gateway Problem,” in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile ’15. New York, NY, USA: ACM, 2015, pp. 27–32. [Online]. Available: http://doi.acm.org/10.1145/2699343.2699344

[22] B. Krebs, “Self-Service Food Kiosk Vendor Avanti Hacked - Krebs on Security,” 2017. [Online]. Available: https://krebsonsecurity.com/2017/07/self-service-food-kiosk-vendor-avanti-hacked/

[23] Belkin, “WeMo Mini Smart Plug,” 2018. [Online]. Available: http://www.belkin.com/us/p/P-F7C063/

[24] TP-Link, “Smart Wi-Fi LED Bulb with Dimmable Light - LB100,” 2018. [Online]. Available: https://www.tp-link.com/us/products/details/cat-5609_LB100.html

[25] Amazon, “Echo & Alexa Devices,” 2018. [Online]. Available: https://www.amazon.com/

[26] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, “An Operating System for the Home,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 25–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228332

[27] A. Networks, “Private Pre-Shared Key (PPSK),” 2018. [Online]. Available: https://www3.aerohive.com/solutions/technology/ppsk.html

[28] Bluetooth SIG, “Bluetooth Core Specification 4.2,” 2014.

[29] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, “Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV. New York, NY, USA: ACM, 2015, pp. 5:1–5:7. [Online]. Available: http://doi.acm.org/10.1145/2834050.2834095

[30] Apple, “iOS - Home,” 2018. [Online]. Available: https://www.apple.com/ios/home/

[31] Nest, “Weave,” 2018. [Online]. Available: https://nest.com/weave/

[32] Samsung, “SmartThings - Add a little smartness to your things,” 2018. [Online]. Available: https://www.smartthings.com/products

[33] E. Fernandes, J. Jung, and A. Prakash, “Security Analysis of Emerging Smart Home Applications,” in *Proceedings of the 37th IEEE Symposium on Security and Privacy*, May 2016.

[34] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, and J. Kim, “FACT: Functionality-centric Access Control System for IoT Programming Frameworks,” in *Proceedings of the 22Nd ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT ’17 Abstracts. New York, NY, USA: ACM, 2017, pp. 43–54. [Online]. Available: http://doi.acm.org/10.1145/3078861.3078864

[35] M. C.-J. Liang, B. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu, “SIFT: Building an Internet of Safe Things,” in *IPSN (International Conference on Information Processing in Sensor Networks)*. ACM, April 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/sift-building-an-internet-of-safe-things/

[36] A. L. M. Neto, A. L. F. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. F. Loureiro, D. F. Aranha, H. K. Patil, and L. B. Oliveira, “AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle,” in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, ser. SenSys ’16. New York, NY, USA: ACM, 2016, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/2994551.2994555

[37] E. Yuan and J. Tong, “Attributed Based Access Control (ABAC) for Web Services,” in *Proceedings of the IEEE International Conference on Web Services*, ser. ICWS ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 561–569. [Online]. Available: http://dx.doi.org/10.1109/ICWS.2005.25

[38] D. Ferraiolo, J. Cugini, and D. R. Kuhn, “Role-based access control (RBAC): Features and motivations,” in *Proceedings of 11th annual computer security application conference*, 1995, pp. 241–48.

[39] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, “First Experiences Using XACML for Access Control in Distributed Systems,” in *Proceedings of the 2003 ACM Workshop on XML Security*, ser. XMLSEC ’03. New York, NY, USA: ACM, 2003, pp. 25–37. [Online]. Available: http://doi.acm.org/10.1145/968559.968563

[40] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, “Authentication in the Taos Operating System,” in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’93. New York, NY, USA: ACM, 1993, pp. 256–269. [Online]. Available: http://doi.acm.org/10.1145/168619.168640

[41] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, “CRePE: A System for Enforcing Fine-Grained Context-Related Policies on Android,” *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 5, pp. 1426–1438, Oct 2012.

[42] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, “Semantically Rich Application-Centric Security in Android,” in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 340–349. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2009.39

[43] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler, “BOSS: Building Operating System Services,” in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 443–457. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/dawson-haggerty

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, “Making Information Flow Explicit in HiStar,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’06. Berkeley, CA, USA: USENIX Association, 2006, pp. 263–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298481

[45] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, “Securing Distributed Systems with Information Flow Control,” in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI ’08. Berkeley, CA, USA: USENIX Association, 2008.

[46] *Off by Default!* ACM, November 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/off-by-default/

[47] H. Welte and P. N. Ayuso, “The netfilter.org project.” [Online]. Available: http://www.netfilter.org/

[48] Symantec, “Norton Core Router - Introducing the Future of WiFi,” 2018. [Online]. Available: https://us.norton.com/core

[49] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, “Bolt: Data Management for Connected Homes,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr 2014, pp. 243–256. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/gupta