HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps.

Soteris Demetriou
University of Illinois at Urbana-Champaign
sdemetr2@illinois.edu

Nan Zhang
Indiana University, Bloomington
nz3@indiana.edu

Yeonjoon Lee
Indiana University, Bloomington
yl52@indiana.edu

XiaoFeng Wang
Indiana University, Bloomington
xw7@indiana.edu

Carl A. Gunter
University of Illinois at Urbana-Champaign
cgunter@illinois.edu

Xiaoyong Zhou
Samsung Research America
zhou.xiaoyong@gmail.com

Michael Grace
Samsung Research America
m1.grace@samsung.com

ABSTRACT

A new development of smart-home systems is to use mobile apps to control IoT devices across a Home Area Network (HAN). As verified in our study, those systems tend to rely on the Wi-Fi router to authenticate other devices. This treatment exposes them to the attack from malicious apps, particularly those running on authorized phones, which the router does not have information to control. Mitigating this threat cannot solely rely on IoT manufacturers, which may need to change the hardware on the devices to support encryption, increasing the cost of the device, or software developers who we need to trust to implement security correctly. In this work, we present a new technique to control the communication between the IoT devices and their apps in a unified, backward-compatible way. Our approach, called HanGuard, does not require any changes to the IoT devices themselves, the IoT apps or the OS of the participating phones. HanGuard uses an SDN-like approach to offer fine-grained protection: each phone runs a non-system userspace *Monitor* app to identify the party that attempts to access the protected IoT device and inform the router through a *control plane* of its access decision; the router enforces the decision on the *data plane* after verifying whether the phone should be allowed to talk to the device. We implemented our design over both Android and iOS (> 95% of mobile OS market share) and a popular router. Our study shows that HanGuard is both efficient and effective in practice.

CCS CONCEPTS

•Security and privacy → Mobile and wireless security;

KEYWORDS

Android, iOS, wireless networks, IoT, security

ACM Reference format:
Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A. Gunter, Xiaoyong Zhou, and Michael Grace. 2017. HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps.. In *Proceedings of ACM WiSec conference, Boston, USA, July 2017 (WiSec'17),* 12 pages.
DOI: 10.475/123_4

1 INTRODUCTION

The pervasiveness of Internet of Things (IoT) devices has brought in a new wave of technological advances in home automation. According to Gartner [29], 6.4 billion IoT devices will be online in 2016, among which a significant portion are *smart-home* systems like smart thermostats [37, 61], fitness trackers, refrigerators, etc., and the number is expected to go above 20 billion by 2020. Examples of such devices include: the Belkin NetCam [13], a camera for streaming surveillance video to a mobile phone; the iBaby monitor [38], a device for remote babysitting; the Family Hub refrigerator [70], which enables online checking of the fridge's contents. Increasingly, these devices are designed to communicate not only with their servers in the cloud but also with other IoT devices and the user's phone over the Home Area Network (HAN), which is typically built around a Wi-Fi router. For example, Nest Protect Fire sensors [60] are capable of propagating an alarm across multiple sensors installed in different rooms of a house. For the convenience of management, such interconnected IoT equipment often relies on the secure connections of HAN (Wi-Fi authentication) for protection and trusts all the computing systems on the same network. This treatment, however, completely exposes the device to the attacks from compromised local systems, a threat becoming increasingly realistic.

Menace of local threats. Indeed, it has been reported that high-profile WiFi-enabled smart home devices, including the WeMo Switch and motion sensor [15, 45, 62, 79, 80], Belkin NetCam [71], baby monitoring devices [82, 85, 86] and smart light bulbs [32], are all vulnerable to a local attack: an adversary within the same HAN is shown to be able to control those devices or steal sensitive user information, e.g., live video streams [71], from them. Several studies further reveal that this is possible since such devices have poor—or no–authentication mechanisms [3, 35, 36, 46, 52, 63, 64, 73, 95] and therefore easily fall prey to a local attacker.

Defending against such attacks becomes particularly challenging when the IoT devices are controlled by phones: once the same phone also carries malware (even when the app has nothing but the network privilege), protecting the device it controls becomes impossible at the network level, as the phone is completely legitimate to access the device though the malicious app running on it is not. Given the high smartphone penetration rates [67], the millions of available mobile applications on both official and third-party markets [83], and the ease of distribution of such applications [1], devices that can be reached through mobile apps can also become an easy target to adversaries. Unfortunately, such adversaries are not only realistic; they are on the rise [55, 66, 72]. Because of that they become the main subject of study of many other academic works [11, 34, 97, 99] while concerns are also raised on public communication channels [40, 65, 81]. In our research we verified that IoT vendors tend to trust the local network (Section 2). This makes them vulnerable to a mobile adversary as we illustrate with attacks on real-world IoT devices, including the *WeMo Switch*, *WeMo Motion*, *WeMo in.sight.AC1* and *My N3rd*. The demos of these attacks can be found on a private website [7].

Addressing the issue here cannot solely rely on device manufacturers: business factors such as time to market and keeping the cost of the device low but also operational factors such as low power consumption, lead to the production of devices without encryption capabilities [88]. In such cases, response to threats can only be reactive and it would entail manufacturing a new version of the device which would still leave users with the old version susceptible to attacks. To make things worse, device manufacturers can be slow in responding [22, 69] to security and privacy threats. Router vendors have already identified this threat. New hubs and routers pushed onto the market are increasingly armed with various IoT protections (e.g. Microsoft Azure IoT hub [56], Google's OnHub router [33]. Integrating protection and management capabilities in the router has significant benefits as the infrastructure is already in place in most households and it enables unified policy management. However, as mentioned above, security control at the router level cannot succeed without knowledge of the OS-level situation within an authorized mobile phone, particularly whether a request to a target device comes from its official app or an unauthorized party. Fundamentally, a practical solution to the problem needs to bridge the gap between the OS-level observation (apps making network connections on a phone) and the network-layer view (requests from the phone for accessing an IoT device), with minimum modifications on the HAN infrastructure and all the systems involved.

Situation-aware device access protection. A simple solution to the problem is just inferring the identity of the app communicating with an IoT device according to its traffic fingerprint. This approach, however, is unreliable and can be easily defeated by, for example, a repackaged app that closely mimics the authorized program's communication patterns. Also, individual apps' fingerprints need to be reliably generated, deployed and continuously updated, and further to be checked on the router against each communication flow it observes, which adds cost to both the router developer and the user. In this paper, we present a different approach, a new technique that achieves fine-grained, situation-aware access control of IoT devices over a home area network. Our approach, called HanGuard, distributes its protection logic across mobile phones and the Wi-Fi router for jointly constructing the full picture of an IoT access attempt during runtime, which is then utilized to control the access on the network layer. More specifically, on the phone side, the information about the app making network connections is collected and passed to the router; on the router side, security policies are enforced to ensure that only an authorized app can touch a set of functionalities the device provides. In this way, malware on network-authenticated phones can no longer endanger the operations of the IoT devices, even when the IoT devices are not equipped with proper authentication and encryption protection.

HanGuard is designed to directly work on the existing HAN infrastructure, without modifying mobile operating systems or IoT devices. To deploy the system, one only needs to install a *Monitor* app with non-system privileges on mobile phones and update the firmware of the Wi-Fi router with a security patch. A key technical challenge here is how to gather situation information (processes making network connections) on mobile phones, which is not given to a third-party userspace app on both Android and iOS. Although all these systems provide VPN support, the app using the service still cannot observe the process generating traffic and will significantly slow down the network communication of the whole system (Section 3.2). To address the issue, we leverage side channel information for lightweight discovery of runtime situation on Android and utilize the VPN to only mark out authorized apps' traffic on iOS (Section 3.2). Such information is then delivered to the router through a separate control channel, which is synchronized with the traffic generated by the app (over a data channel) and used by the router to determine whether the communication should be allowed to proceed.

We implemented our design over both Android and iOS which cover more than 95% of the mobile OS marketshare [39], and a TP-Link WDR4300v1 Wi-Fi router. Our evaluation shows that HanGuard easily identified and blocked all unauthorized attempts to access IoT devices with negligible overhead in the common case.

Our contributions. The contributions of the paper are summarized as follows:

- *New understanding*. We found that *IoT vendors treat the HAN as a trusted environment* which leaves them vulnerable to malicious apps in the HAN. We demonstrate how a weak mobile adversary can exploit this with real-world attacks on popular devices.
- *New system techniques.* We developed HanGuard, the first practical and backward compatible protection against mobile app attacks on smart-home devices. To the best of our knowledge we are the first to use phones as Monitors for local area SDN. Our design can have applications in enterprise settings, peer-to-peer networks and others.
- *Implementation and evaluation.* We implemented HanGuard on both Android and iOS phones (> 95% of the mobile OS market share), and a commercial router, and evaluated it against attacks on real-world IoT devices and on various performance metrics. Our study demonstrates the practicality and efficacy of the new system.

Roadmap. The rest of the paper is organized as follows: Section 2 presents our study on popular smart-home devices; Section 3 presents HanGuard and Section 4 our evaluation; Section 5 reviews related prior research and we conclude our work's presentation in Section 6.

[1] Android applications can be self-signed.

2 SECURITY OF SMART-HOME DEVICES

We performed an analysis of IoT devices and demonstrate the security implications stemming from a mobile adversary. Our findings informed HanGuard's design decisions.

Methodology. One approach for our study would be to investigate the IoT devices' firmware. That would entail—after identifying such devices–finding images of their firmware or, for each device, buying the device and extracting its firmware. Subsequently, each firmware needs to be analyzed, which is a non trivial task [20]. However, most of these devices are now controlled by mobile apps. Thus their control mechanisms can be examined by analyzing the apps instead of the firmware. Note that, our approach has multiple benefits over analyzing firmware: (1) we can easily acquire Android apps, (2) there is no monetary cost, (3) it is generally easier and faster to analyze mobile apps than an embedded device's firmware.

To discover Android apps for IoT devices, we searched for them at Google Play using keywords such as "home automation" and "internet of things". This, turned out to be not very effective: through manual inspections of search outcomes, we found that many apps identified this way were not related to any IoT systems and in the meantime, popular IoT apps fell through cracks. Our solution is to crawl iotlist.co, a popular site for discovering IoT products. From the list, the crawler we ran collected the meta-data of 353 products, including "Title", "Description", "Product Url", "Purchase Url" and others. Such data was further manually checked to identify a list of package names for the official apps of these devices. Our crawler used this list to download the apps and their meta-data from the Play store. Out of the 353 products, we found that 63% (223) of them have apps on Google Play, 2% (7) are iOS only and the rest are mostly unfinished products (listed on kickstarter.com and indiegogo.com) or are no longer available. This indicates that indeed most IoT devices today are controlled by smartphones. We further used popular reverse engineering tools (e.g. apktool [94], dex2jar [30]) to facilitate manual inspection of their source code.

Focus on home automation IoT. To better understand the operations of smart home devices, we manually went through (1) the meta-data of the collected products, (2) their online documentations and websites, and (3) through their apps' source code when available. Figure 1 illustrates our manual categorization of the IoT products based on their functionality. Note that the *Wearables* category (31%) embodies mostly fitness and location trackers, smartwatches and personal medical devices. We call such devices *personal* devices; these commonly use Bluetooth to connect to a smartphone app. Previous work has already studied the security of personal devices, they found problems with encryption and authentication and proposed solutions [24, 59]. From the figure, we can also see that most of the listed IoT devices (55%) are smart home automation/entertainment/security/hub systems, which are the focus of our study. We call these *shared* devices. Such devices could directly benefit from an access control scheme built within the HAN. Previous work on shared devices, was focused on a single IoT integration platform (hub) [26, 27].

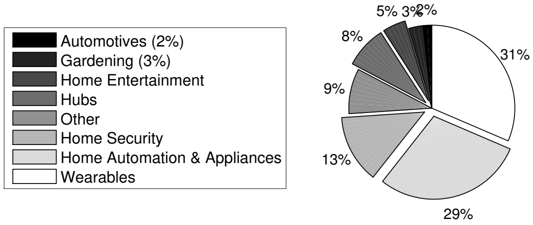


Figure 1: IoT product functionality categorization.

Focus on local connections. Prior research already demonstrated that the interaction between smartphone apps and the cloud is alarmingly unguarded [18, 92]. On the other hand, the local communication between the apps and the devices is not as well understood. In fact *it is unclear whether app developers and IoT device manufacturers treat the local network and everybody connected to it as trusted entities* and whether such treatments leave the devices susceptible to attacks from both local adversaries and remote adversaries that gain access to the HAN. Moreover, even though it has been reported that IoT devices come with serious problems [3, 35, 36, 46, 52, 64, 73, 95], little has been done to understand the security risks stemming from malicious mobile apps. This is particularly important since, IoT devices are controlled by apps which send commands either through the cloud or the local network. Here, we aim to bridge these gaps in knowledge. Our findings build on to the existing evidence which collectively support the need for a unified security and management system built within the HAN to safeguard today's smart-home devices.

In particular, our IoT application study aims to achieve the following goals: (a) Find out whether vendors and developers of WiFi smart home devices/apps erroneously treat the home area network as a trusted environment; (b) Find out whether a mobile adversary can take advantage of such a problematic trust model to attack local smart home devices in practice.

(a) HAN Trust Model. We performed a statistical significance test focused on the following *null* hypothesis (N_0): *HAN apps with only remote connections are equally likely to perform authentication compared to HAN apps with only local connections.* To answer this question we separated our collected IoT apps into two groups. Apps with only remote connections and apps with only local WiFi connections. We used 55 unique Android applications with WiFi/Internet only connections to HAN IoT devices.

To separate the apps into the two groups, we manually went through (1) their online documentations and websites, (2) public forums, and (3) their Java Android code. We found that 22 (40%) do perform some internet socket connection with local discovered devices or fixed local IPs. 25 (45%) were found without local WiFi connections, 5 (9%) we could not determine, for 2(4%) decompilation failed, and 1 (2%) was by that time removed from Google Play. For each of the 2 sets (local; no local) we analyzed them further to discover whether they perform any authentication. For the ones that perform only remote connections, we used a parsing tool that searches for password requests in the layout files of the apps. We then manually verified the existence/absence of a password request. We found that all these apps do perform authentication.

For the 22 apps with local WiFi connections we could not simply use the above tool since it would reveal little to no information on whether a password is used for a connection with the IoT device or the cloud. Thus we manually went through their code looking for network API calls responsible for local connections (e.g. creation of sockets connecting to local IPs, or UPnP discovery). We examined the calls to such APIs and found that 9 of the apps do not authenticate to the IoT device.

To determine whether apps with local connections are less likely to perform authentication one could perform a χ^2-test of independence. In our case this was not suitable due to the small absolute number of relevant available apps derived from iotlist.co. We instead used the Fisher's exact test [28]—a common approach to derive statistically significant results when the sample size is small. We performed the test [17] on our *null* hypothesis (N_0). A 2-sided P value less than 0.05 was considered significant. The test yielded a 2-sided P-value of $0.00036 < 0.05$ and thus we can reject N_0. Therefore, we can confidently say that *HAN apps with local connections are less likely to get authenticated by IoT devices.* This validates an important intuition that IoT vendors consider the HAN to be a trusted environment. However, given the fact that phones are an integral part of such a network and that phones can carry self-signed apps from third-party markets, this treatment becomes detrimental to the security of HAN IoT devices.

(b) The mobile adversary threat. The previous finding is particularly alarming. Next we wanted to validate that a weak mobile adversary can take advantage of this problematic trust model and trivially compromise smart home devices. Towards this end, we cherry-picked four devices with local connections and authentication issues and performed real-world, practical attacks. The devices we picked are listed on Table 1. Our targets include the *WeMo Switch* and *WeMo Motion* [15], the *WeMo in.sight.AC1* [14], and *My N3rd* [57]. The *WeMo* devices are examples of popular plug-and-play devices. Just on Android, the official app of the WeMo devices was downloaded 100,000–500,000 times [2]. Note that all the WeMo devices are manufactured by a single vendor. By focusing on three WeMo devices we want to showcase how an erroneous trust model by a vendor can spread across various of its devices. This suggests that trusting the local network was a design decision and not an implementation issue manifesting in an isolated device. My N3rd, while not yet popular, it is chosen to showcase a new category of do-it-yourself (DIY) devices. It allows one to connect it to any other device enabling turning on/off that device from the My N3rd mobile app. Increasingly more such projects appear on the market with Arduino-based projects taking the lead. While exciting for users, such devices tend to inherit the problematic trust model and allow an adversary to take full control of ones devices. In our experiments we consider a mobile adversary that tries to get unauthorized access to the IoT devices. The mobile adversary can perform an attack from an unauthorized phone, or from an unauthorized app on an authorized phone [3]. To test the above cases, we use 2 Nexus phones. The first one is assumed to be untrusted and the second one is assumed to belong to one of the HAN users. We then tried to access the target IoT devices using both phones. Unfortunately we found that the adversary can trivially connect and control all devices. The video demos of our attacks can be found online [7].

Table 1: Devices used in real-world attack demonstrations.

Target Device	Description	# App Installations
WeMo Switch	Actuator	100K - 500K
WeMo Motion	Sensor	100K - 500K
WeMo Insight Switch	Actuator	100K - 500K
My N3rd	Actuator	100 - 500

3 HANGUARD DESIGN & IMPLEMENTATION

Our previous findings (Section 2) highlight the need for an access control system that can be integrated in home area networks with minimal changes to the existing infrastructure, that is backward compatible, independent of vendor and developer practices and which allows the users the flexibility to manage and control who should communicate to which device. In this section, we elaborate on our design of such a system called HanGuard and its implementation over the HAN and mobile platforms.

3.1 Design Overview

Adversary model. As shown in Section 2, IoT devices are controlled through smartphone apps. These devices are designed to act blindly on the commands from authorized phones (based upon their authentication with the HAN router). This treatment becomes increasingly problematic: while the smartphone may indeed belong to a rightful user, the applications that it runs can come from less known places (e.g., third-party app stores) and less trustworthy developers (e.g., malware authors). Given smartphone penetration [67], prevalence and ease of distribution of mobile applications [83], adversaries can now find their way to the HAN through a legitimate phone with minimal effort. Moreover—as demonstrated in Section 2—given the erroneous threat model of today's IoT devices, which trusts all the requests issued from a trusted source (a router or phone), such malicious applications can easily gain unauthorized control of IoT devices (e.g. turning on/off an actuator, or reading the collected data of a sensor).

Thwarting such attacks is inherently hard. A straightforward solution is to implement a unified security logic in the router, since traffic from applications to IoT devices goes through it. However, the router alone does not have enough information to make any *application level access control* decision. One could resort to traffic fingerprinting techniques to infer the application generating the traffic. The approach can (1) be easily evaded by a malware repackaged from an authorized app, (2) bring in false alarms and (3) impacts the performance of the router.

HanGuard is designed to address the issue through bridging network and application level semantics, associating an app's identity to its traffic to enable a fine-grained access control on IoT devices. In the meantime, it does not modify both software and hardware of these devices, the operating systems of smartphones, and does not make assumptions about the router hardware. For this purpose, our adversary model is focused on the situation where a malicious app is installed on a smartphone device authenticated to the HAN. The adversary is considered to already know the communication protocol used by the victim IoT device. We further assume that the smartphone hosting the app has not been compromised at the OS or hardware level, which limits the adversary to the user land, at the

[2] This is a conservative number as people can download the app from alternative Android app markets or from iTunes for iOS devices.

[3] Note that the case of an unauthorized app on an unauthorized phone trivially reduces to the first case we consider.

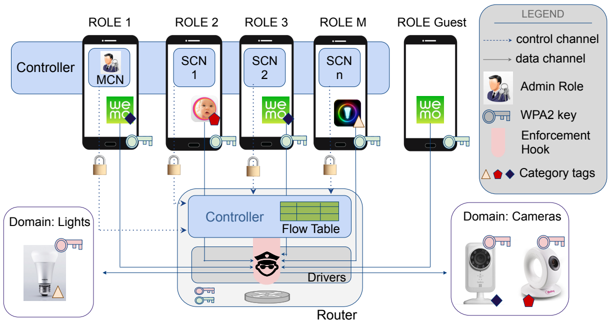


Figure 2: HanGuard high level architecture.

SHA3(username, password)	MAC_P	IP_P	$PORT_P$	IP_I	$PORT_I$	APP	$VERSION_I$	FLAG

- I : IoT device
- P : HAN Phone

Figure 3: HanGuard Control Message delivered over TLS.

app level. Note that though outside our adversary model, HanGuard can also provide coarser-grained protection against guest phones and compromised phones, remote adversaries and more traditional WiFi attacks. Due to space limitations and to avoid confusion, we defer this discussion to the Appendix A.

Idea and architecture. Figure 2 illustrates the architecture of HanGuard. Our design is partially inspired by software defined networking (SDN) (see [49] for a survey), which separates the network traffic (*data*) from its management (*control*). In the meantime, HanGuard is meant to be easily deployed to today's HAN. Serving this purpose is a distributed security control architecture that includes a *Controller* on a HAN router for policy enforcement and a *Monitor* on the user's phone for collecting its runtime situation and making access decisions (which are enforced by the router). To avoid changing the mobile OS, the Monitor is in the form of a user-space app. It detects the app making network communication and its compliance with security policies, and then pushes the access permit to the router's Controller through a secure control channel (Section 3.2). The router utilizes that information to enforce the policy (Section 3.3): only the traffic with a permit from the Monitor is allowed to reach IoT devices.

In essence, this design preserves the data channel within which unmodified information from smartphone apps is propagated to the router, and creates an independent control channel for security decisions. Such a separation, it comes with obvious performance benefits: no extra headers to be processed by the router on a per packet basis in the data channel. It can also guarantee that control information is always transmitted through a secure channel, and allows the router to further enforce policies and ensure, even in periods of heavy congestion, that security decisions are delivered in a reliable manner. In addition, our design allows for a clear separation of tasks: the security policies can be easily managed by the user through a mobile app interface; the router reduces to simply enforcing the flow decisions. This keeps the router as simple as possible and allows for readily updating the security logic with a mere application upgrade.

Policy Model. HanGuard implements an RBAC (*role-based access control*) policy model which leverages *type-enforcement* and *multi-category security* primitives. It uses them in a unique way to create policy rules, to protect smart-home devices. However, HanGuard does not need security experts to create the policies; policies are generated at runtime and transparently to the user. In particular, the user is only expected to perform simple mappings between a finite set of IoT apps, IoT devices and HAN users. Default policies are automatically created during setup to further reduce users' burden. HanGuard's access control model parses such mappings and assigns a *category* tag to each app and its respective IoT device. Further, each IoT device is labeled with a *type*. *Types* can be organized in overlapping groups called *domains*. Each mobile phone is assigned a *role* and each role can be configured to access a number of domains. For example, the iBaby camera can be labeled with the *type* "babyMonitor_t". A *domain* "cameras_d" can be created to encompass the "babyMonitor_t" *type* device among others. Lastly, the *role* of a HAN user's phone (e.g. "Adult") that is supposed to be able to access the cameras, can be configured as eligible to access the "camera_d" *domain* and in extend the "babyMonitor_t" *type* device. The relation between the *role* and the *domain* ensures that an untrusted phone (e.g., a visitor's phone) cannot touch protected devices and even an authorized phone, once compromised, cannot communicate with the IoT devices it is not supposed to talk to. At the same time, and orthogonally to the type-enforcement scheme, the iBaby camera and its official app, can be assigned the *category* "iBaby". The *category* here binds a specific app on a phone to the device the phone is authorized to access. For example, the role "Adult" can be configured to access the domain "cameras_d"; while that stipulates that the adult's phone can control the baby cameras, access is not granted unless the app on her phone and the actual baby camera that it tries to reach are tagged with the same category. Note that more than one category tags can be associated with a domain. This enables the generation of a policy rule which allows an app to access multiple devices of the same type.

By default, a phone registered with the HAN is assigned the role "HAN user", which is allowed to access the "Home" domain. The latter encompasses every newly installed IoT device (which is assigned a unique *type*). However, the access can only succeed when the app on the phone is given the same category tag as the device it attempts to reach. Such an app-device binding is established when the app is used to configure the device, which is established through a special device, a phone or a PC, that takes the role of an *Admin*. This role can configure the router, register other user phones, access all domains and update security policies. During a policy update, new domains, roles and access relations between them can be generated. The policy model also handles unregistered phones (e.g., those belonging to visitors), which connect to the Han as a "Guest", a *role* not allowed to interact with the devices in the "Home" *domain*. A security policy is shared by the phone side and the router side. Although its enforcement happens on the router, its compliance check is performed jointly by the router and the phone. The former ensures that only the authorized phone, as indicated by its *role*, can access the *domain* involving the device. The latter runs the *Monitor* to inspect the app and the target device's *category* tags and asks the router to let their communication flows go through only when the category tags are the same. Next, we describe how individual components of the system work.

3.2 Phone-side Situation Monitoring

In our distributed access-control system, the *Monitors* are deployed as user-space apps. They are aiming at identifying the subject (app) trying to access an IoT device across the HAN, and determine whether it is authorized. Such information is delivered through a control message to the *Controller module* running on the router, informing it the context of the access attempt (since the router cannot see the app initiating the communication), which helps the router enforce appropriate security policies. Note that we designed the system in a way that the workload on the router is minimized, which is important in maintaining the performance level needed for serving the whole local network. More specifically, the *Monitor* launches at boot time to establish an ongoing secure connection with the *Controller module* on the router. Through the channel, the situation on the phone is either *pushed* to, or *pulled* by the router (Section 3.3), enabling it to perform a *per-flow* (instead of *per-packet*) access control. Further, the security policies (Section 3.1) are broken into two parts: the *Monitor* checks whether an app is authorized to access a device and asks the router to enforce its decision, while the router implements a *phone-level* policy check as a second line of defense, which protects the smart-home devices even when a phone is fully compromised.

The communication between the *Monitor* and the router goes through a TLS control channel. The control message delivered through the channel is in the format illustrated in Figure 3. For example, it includes a hash of the user credentials (username, password), the sender phone's MAC address, an identifier for the detected flow (IP/port), an identifier for the app making the request, the policy's version number and a flag indicating whether this flow should be allowed or not. The negative flag is used to mark suspicious behavior (detection). Flow termination is handled by the router (Section 3.3).

Every registered phone on the HAN, can be assigned *roles* instantiating an RBAC (Role-Based Access Control) scheme. Furthermore, the phone used to configure the router is by default designated as the *Master Controller Node* (MCN) and every other phone is designated as the *Slave Controller Node* (SCN). A HAN user can update the policy through the *Policy Update Manager* running in her phone's *Monitor*. A *Monitor* accepts policy updates only when it is running on a master node and after verifying its user's credentials. A distributed *Policy Update Service* intermediates policy synchronization and replication in the system. Every connected (reachable) node gets the latest policy replica as soon as it connects to the network or when there is an update. Unregistered devices are automatically assigned the "Guest" role as soon as they join the network. Each *Monitor* has a local in-memory replica of the policy base, that allows it to make decisions for its own traffic efficiently, alleviating the router from further processing. Having the policy also at the phone side is critical in SDN-like systems since it allows for efficient decision making by the Monitors, reduces the bandwidth on the control channel and keeps the routers simple and fast [49]. This way, Monitors send only their per-flow decision to the router instead of continuously sending all the mobile OS-situation measurements. In the last case, the number of control messages in the HAN would exponentially increase while the router would need to process all the measurements before making a decision, with severe performance degradation.

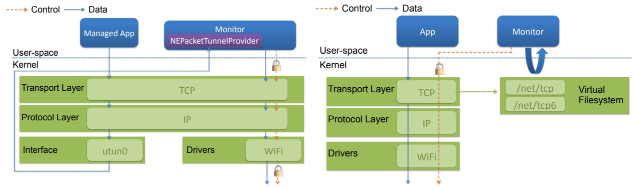
Situation awareness on iOS. As mentioned earlier, the *Monitor* is designed to find out which app is talking to an IoT device under protection. Such information, however, is not directly given to a non-system app on both iOS and Android. To tackle this we utilize a new iOS capability that allows developers to proxy network traffic. Once this functionality is enabled by an app and approved by the user, all network packets from all apps will traverse the network stack and instead of being sent through the physical interface to the remote destination, they end up in a virtual interface (tunnel). The tunnel will redirect those packets to the proxy app running the VPN functionality.

iOS offers developers the capability to proxy network traffic with the `NEVPNManager` APIs). However, blindly tunneling apps' traffic through the VPN is very expensive, often slowing down the mobile system's network performance by an order of magnitude. This workflow is illustrated in Figure 4a: when an app makes a network call this would entail, for every packet, a userspace-kernel context switch, traversing the network stack, trapping the traffic through the tunnel interface and context-switching to userspace again to deliver the network packets to the proxying app. Then the proxying app needs to process the network headers (essentially performing layer 3-4 translations) and then resending the packet.

Our solution is to utilize the VPN in a unique way: instead of running the iOS Monitor to proxy the traffic of *all* apps (through the `NEVPNManager` APIs), which is expensive, requires a remote VPN server and gives little information about the identity of the app generating traffic, our iOS *Monitor* uses the `NEPacketTunnelProvider` APIs with a per-app VPN configuration, to tunnel the traffic *only* from *authorized apps* (the official apps of the IoT devices), while leaving all other traffic outside the tunnel to avoid unnecessary delays. Furthermore, over the tunnel, our iOS *Monitor* does not change the data: it merely acquires packet header information and forwards the packet to its original destination. After authenticating itself to the *Controller module* on the router through TLS and its credentials, the *Monitor* informs the router that the flow in the tunnel is authorized. Other flows towards the IoT devices from the phone are by default considered illegitimate and will all be dropped at the router. In this way, we can strike a balance between the protection of legitimate IoT management traffic and the performance impact of the security control.

Situation awareness on Android. A straightforward way to capture traffic from other apps on Android is to follow a similar process with iOS and utilize the closely equivalent `VPNService` [6] API, introduced in Android 4.0. However, the implementation of VPN on Android is similar to the one in iOS and would entail similar overheads. To collect the situation information in a more lightweight manner, `HanGuard` leverages side channels on Android an approach which results in astounding performance benefits.

The Android *Monitor* we implemented continuously looks at the `procfs` file system (see Figure 4b). `procfs` is a virtual file system which exposes the current status of an Android phone's kernel internal data structures. Particularly the files `proc/net/tcp`, `proc/net/tcp6`, `proc/net/udp` and `proc/net/udp6` disclose the



(a) iOS.

(b) Android.

Figure 4: Traffic Monitoring Workflow.

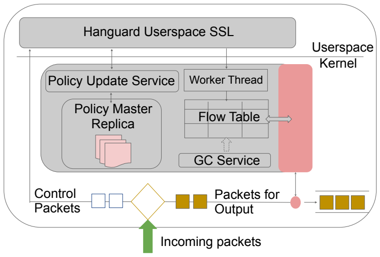


Figure 5: HanGuard Router Controller Module

ongoing TCP and UDP connections between the phone and a remote destination, including the source/destination IP addresses of the ongoing connection and its port numbers, the status of the connection etc [4]. The addresses here can be either IPv4 and IPv6 (with the suffix "6"). These connections are also associated with a specific UID that the *Monitor* can map to an installed app. To minimize operation overheads, the *Monitor* does not open and parse a file for each access. Instead it just checks the file's metadata (i.e. the last modified time or `mtime` in UNIX terms) to determine whether the file has been changed since the last visit. A complication here is that Android often fits an IPv4 address into the IPv6 format before reporting it to the user. Such an address is automatically captured by the *Monitor* and converted back to the IPv4 form. As an example, consider an app on a phone with an IPv4 address *192.168.1.189* that connects to an IoT device with the address *192.168.1.32*. During the app's runtime, the connection may not show up in `proc/net/tcp` but appears inside `proc/net/tcp6` instead with `0000000000000000FFFF 0000BD01A8C0` for the source IP and `0000000000000000 FFFF0000200 1A8C0` for the destination. It is clear that the IPv4 address is enclosed in the 32 least significant bits [5] and the 96 remaining bits are fixed. The *Monitor* detects the address from its fixed part and converts the rest to an IPv4 format before communicating the app's identity to the router through a control message. Note that Android suffers from the repackaged apps problem [97]. To address this the Android Monitor uses a package's signature to verify apps claiming the identity of policy-controlled apps.

3.3 Router-side Policy Enforcement

The design of the controller module mainly focuses on synchronizing security policies across all the systems within the HAN and enforcing these policies on the router, as illustrated in Figure 5. More specifically, the module maintains a *Master Policy Replica*, and runs a *Policy Update Service* responsible for updating the policies and distributing them across registered *Monitors*. Further, the *Controller module* introduces a *Per-Flow Decision Cache* (PFDC) for keeping the access decisions (on the app level) pushed by (or pulled from) the *Monitors*, and a *Garbage Collection Service* (GCS) for maintaining the cache. It also hooks on the router's packet flow for the policy enforcement. Due to space limitations, we omit discussion of the policy synchronization and focus on the policy enforcement.

Receiving decisions. By default the router blocks all flows to IoT devices. As mentioned earlier, app-level access control on the router relies on decisions made by the *Monitor* and delivered to the router through the control channel. To effectively enforce such decisions on a traffic flow, the *Controller module* is designed to efficiently authenticate and process the control messages to avoid holding up the legitimate interactions with the target IoT device. Specifically, the *Controller module* maintains TLS connections with the *Monitors* through a userspace program. When a decision from a *Monitor* arrives, after the successful TLS Monitor certificate validation, the router checks the policy version and the sender user's credentials, and once these are also validated, it passes the decision's *flow ID* (source IP and port, destination IP and port) to the kernel that updates the *PFDC* using the flow ID as the key to record the validation/invalidation decision on the flow, which is then enforced by the router. We highlight that data flows are first checked against a phone-level policy which ensures that the flow comes from a valid HAN phone.

Supporting this decision-making process requires an efficient userspace to kernel communication mechanism (for the router). Although this can be achieved through `system calls`, `ioctl` calls or `procfs` files, these approaches are either complicated to implement or unable to handle asynchronous interactions. Our solution employs the `netlink` socket IPC mechanism for the user-kernel communication, which can be easily built (without changing the kernel) and are asynchronous in nature: it queues incoming messages and notifies the receiver through a handler callback. In our implementation, the callback spawns a *worker thread* that processes the message and updates the *PFDC*, either by inserting a valid flow or removing an invalid flow.

The *PFDC* is loaded at the router's boot time from its persistent storage. It holds the following information per-flow: the *flow ID*, the *flow validation/invalidation flag*, the *requesting app* and the *data last seen time*. This cache is used for enforcing app-level policies (whether a specific app is allowed to access a device), for the purpose of enhancing the existing flow-control capability of the router, which cannot differentiate two flows from the same IP and port but produced by different apps. By searching the cache, the router can apply the app-level access decision upon the whole flow, instead for every individual packet, an advantage over deep packet inspection and traffic fingerprinting techniques. To limit the amount of the resources the cache uses, a *Garbage Collector Service* (GCS) is run to remove the obsolete records with the oldest data last seen time. To prevent DoS attacks where a *Monitor* attempts to flood the cache, a per-phone limit is applied.

[4] Note that iOS does not reveal to an app the information about other processes through its `procfs` file system. Before iOS 9, one could use the system call `sysctl` to access such information. This channel has been closed since then.

[5] in little-endian order, presented using four-byte hexadecimals

Enforcement. The router enforces *phone-level* and *app-level* policies. For the former, it checks every packet to determine whether it originates from a phone which is allowed to access a particular IoT device. Phones and IoT devices are identified based on their MAC addresses. MAC-IP associations are statically defined during a new device enrollment. For *app-level* policies, the router checks the *PFDC* cache to determine whether the flow is generated from a valid app. A technical challenge in implementing the protection is where to place the security control within the existing router infrastructure. On a Linux-enabled system used by the router, once a packet is received, it is put by the link layer into a backlog queue from which the IP layer pulls packets for checksum checking and routing decisions. If the packet is destined for the current machine, it is passed to the transport layer. If not the packet is forwarded. Apparently, the security control should happen on the IP layer (e.g. in the `ip_forward()` function). However, a packet might follow a different path within the kernel depending on whether the current system is configured to run as a bridge or a router. For example, in a bridge mode, no layer 3 operation is involved and as a result the aforementioned function will never operate on the packet. Our solution is to place the Controller hook in `dev_queue_xmit()`, a generic driver function, ensuring that no packet bypasses the check.

To minimize the impact on flows unrelated to the smart-home devices, the `HanGuard`-enhanced router quickly inspects each packet it receives to determine whether further attention is needed. Specifically, a TCP flow is considered interesting if its destination MAC address is associated with an enrolled IoT device. Packets not fitting this description are forwarded without a delay, and others are first handled according to the phone-level policy (whether the phone can access the IoT device) stored at the router, and then the app-level policy (whether the app can do that) which is based upon the validation flag set by the Monitor. For the packet allowed to go through, its flow's last seen time is updated to the packet's arrival time. `HanGuard` helps its users detect and react to spurious access attempts with its *notification mechanism*: `HanGuard` (1) keeps a log, and (2) sends out-of-band notifications to the admin user when a violation or tampering of the policy is attempted.

Flow Termination. A determined adversary could attempt to exploit the fact that a flow from a co-located app is allowed. For example, it could wait for the benign app to release its port and attempt to send a packet before the Monitor informs the router to invalidate the flow. For TCP flows, the router prevents such attacks: it proactively invalidates a validated TCP flow, when it sees its corresponding TCP FIN packet and then handles the session termination. For UDP, the situation is more complex. UDP is an unreliable protocol with no clear indications of a session establishment/termination. `HanGuard` can be configured to handle UDP flows in two ways: (a) in a *STOP-AND-WAIT* mode, for every packet, it *pulls* a decision from its Monitor. If between the time the packet is received by the router and the decision request is received by the Monitor, no other app on the same device attempted to send a UDP packet to the same target IoT device, then and only then the packet is allowed. This is a security stringent policy that prevents the attack. However, it comes with performance penalties since every packet is delayed approximately by one RTT. (b) In the *DETECTION* mode, the Monitor pushes a flow invalidation decision when the benign app releases the port. In this case, a malicious UDP packet from an Android app could make it through before the decision is enforced. However, the Monitor will (a) detect the malicious attempt; (b) can determine the offending app and; (c) can determine the affected device (destination). Once a violation is detected the user is notified to verify the status of the affected device and uninstall the offending app which is also blacklisted in the policy. On iOS such race attacks are always prevented: offending tunelled traffic is blocked on the phone whereas non-tunneled traffic to IoT devices is blocked at the router.

4 EMPIRICAL EVALUATION

We implemented a prototype of `HanGuard`—in *DETECT* mode for UDP (Section 3.3)—on top of a TP-Link WDR4300v1 router with a Gb NIC and a wireless network at the 2.4 GHz band (300Mbps) running OpenWRT Chaos Chalmer with a Linux 4.1.16 kernel, and also Nexus phones running Android 5 (Lollipop) and an iPhone 4S running iOS 9. Our work answers the following research questions:

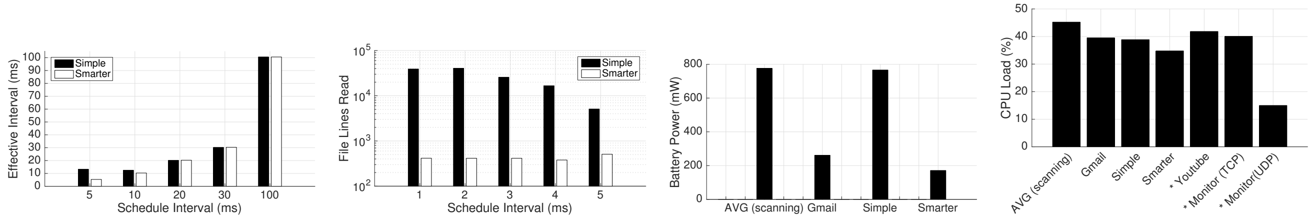
- *RQ1:* Is `HanGuard` effective in thwarting attacks from malicious applications?
- *RQ2:* What is the performance impact and resource consumption of the *Monitors* on the phone side?
- *RQ3:* What is the overall overhead of `HanGuard`?

4.1 Effectiveness

To answer RQ1 and verify `HanGuard`'s backward compatibility and practicality, we repeated our attacks on real world smart-home devices (listed in Table 1). We performed the following two experiments: (A) first we set up the target IoT devices over the "Vanilla" system (without `HanGuard` components), and further installed a repackaged version of their legitimate app on the phone to mimic the adversary; (B) next, we updated the router with `HanGuard`-enhanced firmware, and also put our *Monitor* app on the same phone with a policy that allows the phone and the benign app to access the target IoT device. Under this protected setting, we repeated experiment (A), using the phone with the *Monitor* app to set up the IoT devices. As expected, both the original app and the repackaged one could access the devices in the Vanilla system. With `HanGuard` enabled, only the official apps on the phone running the *Monitor* app could communicate with their respective IoT devices , which confirms the effectiveness of the access control enforced by `HanGuard` and its backward compatibility (see [7] for demos).

4.2 Phone-side Performance

Monitoring cost on Android. On Android, the *Monitor* continuously polls the `procfs` file system to detect ongoing network connections. Here we report our study on two monitoring strategies and their performance impacts. Specifically, we configured the Android *Monitor* on a Nexus phone to inspect the `procfs` file system in different granularity (every 5ms, 10ms, 20ms, 30ms, 100ms). After running for 30 seconds, the *Monitor* went through every single file line to check the presence of interesting network connections, a strategy called the *Naive* mode. The approach was compared with another strategy, called the *Smarter* mode, which first looked at



(a) Polling scheduled frequency vs Actual polling frequency on Android.

(b) File lines parsed for different Android Monitor configurations.

(c) Battery Power on Android.

(d) CPU Load on Android and iOS(*).

Figure 6: Monitoring Costs.

the last modified time of a file before accessing its content. The outcomes of the study are illustrated in Figure 6a. As we can see, the *Smarter* strategy clearly can poll at a finer granularity (5 ms), given that it reads much fewer file lines compared with the *Naive* approach (Figure 6b), which is translated to less work per iteration in the common case.

We further looked into the resource consumption of the *Monitor*. For this purpose, we configured the *Monitor* to poll at 10 ms and recorded its CPU and battery consumption for both the Naive and Smarter mode. On the same Nexus 5 phone, we also ran Trepn [68] by Qualcomm to collect the baseline power profile of the phone for 30 seconds before running our *Monitor* app for 2 minutes. Figure 6c illustrates the average battery consumption that can be attributed to the *Monitor*, and Figure 6d shows the average CPU usage (first 4 bars). To put things into perspective, we compared our *Monitor* with a popular Antivirus app in scanning mode and the de facto mailing app on Android (Gmail). As we can see from the figures, the power consumption of the naive approach is comparable to an antivirus app performing an expensive operation while the smarter mode's is comparable with Gmail which is optimized to always run in the background.

Monitoring cost on iOS. To evaluate the iOS *Monitor*'s resource consumption, we used *Instruments* [9], a performance analysis and testing tool which is part of the official Apple IDE (Xcode [10]). Figure 6d depicts the % CPU utilization that can be attributed to a runtime process, where measurements on iOS are indicated with * (last 3 bars): the *Monitor* when proxying a TCP app that sends 500 messages with payload size equal to one character; the *Monitor* when proxying an equivalent UDP app; and YouTube while streaming a video configured to *auto-select* its quality. The figure reflects the fact that the iOS *Monitor* does a lot of work when proxying TCP traffic: this is expected as TCP is a connection oriented protocol and the *Monitor* needs to guarantee reliable delivery of the packets. For UDP the *Monitor* does very little work. In idle mode (not proxying), the *Monitor* incurred no CPU overhead. *Instruments* can also report the *Energy Use Level* of an app at runtime as a value from 0 to 20. In all experiments the reported value was consistently 0/20.

Detection accuracy. The *Monitor*'s goal is to detect an interesting flow generated on the phone. For iOS the detection accuracy is 100% since all packets from interesting apps are routed through the *Monitor*'s VPN. For the Android *Monitor* though, the situation is more complicated. For example, an interesting app might quickly set up a socket, send a packet and then close the connection. The Android *Monitor*'s detection accuracy depends on whether it can catch such events given its polling interval. To answer this question we created a micro-benchmark that includes a TCP and a UDP app connecting to a TCP and UDP echo server respectively. They both stop the communication once the server response is received. Again, we ran the *Monitor* in the Smarter mode 10 times for each of the following polling configurations: 150ms, 100ms, 30ms and 10ms. We found (see Figure 7) that the 10ms configuration could always detect outgoing TCP and UDP connections.

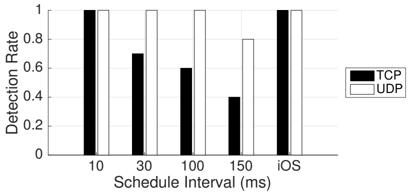


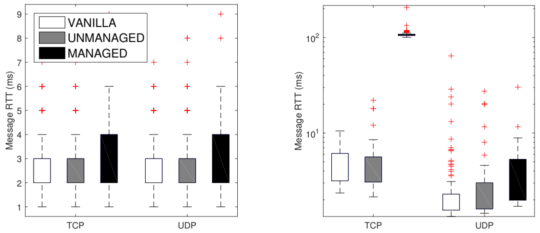
Figure 7: TCP and UDP flow detection accuracy for different Android *Monitor* configurations and for the iOS *Monitor*.

4.3 Communication Overhead

To answer RQ3, we assess the overall performance overhead of HanGuard, as this can be observed from a mobile app. We created a baseline by performing our experiments below on the unmodified system (Vanilla). To evaluate HanGuard communication overheads we repeated the experiments on HanGuard with the respective benchmark app being either policy-protected (Managed) or not protected (Unmanaged).

Application latency. We ran the TCP and UDP apps individually, configured to send 100 messages each. Figure 8a depicts the mean latency in milliseconds (ms) for a TCP message and a UDP message for Android. The latency is measured as round trip time (RTT) on the mobile app. In particular we measured the time interval between the API call to send the message and the time that the message is returned by the server and delivered to the application layer. As we can observe, HanGuard introduces negligible latency for Managed apps on Android.

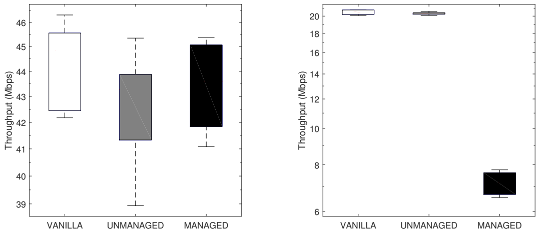
In Figure 8b we can see that there is a big increase on TCP packet latency for the Managed apps on iOS. Nevertheless, in practice this is often tolerable, since most devices are actuators and sensors that



(a) Android.

(b) iOS.

Figure 8: Application-level communication latency.



(a) Android.

(b) iOS.

Figure 9: Application-level throughput (TCP).

create mice flows delivering a small amount of information: for example, it is completely imperceptible when the delay for switching a light grows from a few milliseconds to tens of milliseconds. This Figure also reveals an important benefit of our design: *the security controls have negligible impact on Unmanaged apps, on both Android and iOS devices, for both UDP and TCP.*

Application throughput. To measure HanGuard's throughput overhead, we use our benchmark apps to transmit a file of 20MB to their server counterparts. We repeated the experiment 10 times. Figure 9 plots the throughput CDF for Android and iOS (*). Evidently, HanGuard has negligible impact on throughput for all Android apps and iOS unmanaged apps. Our evaluation also reveals an interesting case: throughput drops significantly—but only—for the iOS Managed apps [6]. This happens because the iOS Monitor implementation uses the built in VPN utility of the OS. Thus, it has to inspect every packet for managed apps (see Figure 4a). This is a security, performance trade-off we had to address. We opted-in for security.

5 RELATED WORK

IoT attacks. Recent works demonstrated attacks on IoT devices [26, 63, 75, 76, 82, 96]. Fernandes et.al. found vulnerabilities on SmartThings' applications [26]. Their work focuses on a specific IoT hub that can integrate third-party IoT devices, whereas HanGuard is applicable to an infrastructure that exists in almost all households with IoT devices. [63, 82], revealed vulnerabilities on smart-home devices. However they consider an adversary on a separate device. [75] considers an intricate mobile adversary which colludes with a cloud. We illustrate that the mobile adversary can succeed with minimal effort. All reported attacks further motivate the need for practical smart-home defenses.

Android side-channels and network monitors. Several works focused on acquiring information for other processes using side-channels on Android [42, 96, 98, 100]. [96] also utilized side channel information for defence purposes. [50] used the VPN service on Android for passive monitoring of mobile apps to collect user traffic information for analysis. However, it redirects all packets to a server that further routes the packets. This raises privacy concerns which we avoid by implementing the routing functionality locally.

Access control. There have been various works on home access control which we classify in three major areas: surveys [25, 31, 90]; access control systems [5, 23, 27, 47, 51, 76]; and user studies for usable policy specifications [48, 54]. More relevant to our work is the second. Nonetheless, most of these systems assume a clean-slate design where the OSes of participating nodes can be modified. Our solution is backward compatible: it requires just a software upgrade on the Home's router and downloading an app on the phone. Other work focused on access control enforced on the mobile phones [16, 24, 77]. Demetriou et. al. [24] enforced local policies to control access to personal devices while our target is to enforce a distributed policy on shared devices.

IDS and Firewalls. Work on intrusion detection systems (IDS), personal and application firewalls [8, 19, 21, 44], focuses either solely at the host or at a network node, or only at the network layer. HanGuard is distributed, consolidating application level semantics from hosts, and network level information from the network node. Furthermore, we do not require experts to set up policies.

6 CONCLUSION

In this work we presented HanGuard, a system that can enforce access control policies in a HAN among user phones and IoT devices. HanGuard uses a new SDN approach applied on HAN: it employs situation awareness on users' phones through a userspace *Monitor* app that detects whether an authorized app is establishing a network flow with a target IoT device; *Monitors* push decisions to the HAN router bridging the gap between network and application-level semantics. This technique allows the router to enforce fine-grained access control based on a global policy protecting access to HAN IoT devices. HanGuard does not require mobile OSes modifications, any IoT device modifications, or new router hardware. It is backward compatible with the existing HAN infrastructure, and was implemented and evaluated in a realistic HAN setting, verifying both its practicality and effectiveness.

7 ACKNOWLEDGMENTS

University of Illinois authors were supported in part by NSF CNS grants 12-23967, 13-30491, 14-08944, and 15-13939. Indiana University authors were supported in part by NSF CNS grants 1223477, 1223495, 1527141 and 1618493, and ARO W911NF1610127. Demetriou and Lee also thank Samsung Research America for supporting this project during their internship. The views expressed are those of the authors only.

[6] In practice this will only affect real-time streaming services offered by such app-device connections. Actuators and sensors will not exhibit a noticeable effect.

REFERENCES

[1] 2008. *ARM Security Technology.* Technical Report. ARM Limited.

[2] 2015. *ARM Strategic Report.* Technical Report. ARM Limited.

[3] 2015. *Internet of things research study.* Technical Report. Hewlett-Packard Enterprise.

[4] 2015. *iOS Security.* Technical Report. Apple Inc.

[5] Gail-Joon Ahn, Hongxin Hu, and Jing Jin. 2008. Towards Role-Based Authorization for OSGi Service Environments *(FTDCS '08)*.

[6] android.com. 2017. VpnService-Android Developers. http://goo.gl/0cKFyO. (2017).

[7] Anonymous. 2017. Demo website. https://goo.gl/dfYeop. (2017).

[8] G. Appenzeller, M. Roussopoulos, and M. Baker. 1999. User-friendly access control for public network ports *(INFOCOM '99)*.

[9] apple.com. 2017. Instruments: iOS performance analysis tool. https://goo.gl/6XnAXF. (2017).

[10] apple.com. 2017. Xcode: Apple's IDE. https://goo.gl/TgMco6. (2017).

[11] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *NDSS*.

[12] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World *(CCS '14)*.

[13] belkin.com. 2017. Belkin Netcam. http://goo.gl/60dfkg. (2017).

[14] belkin.com. 2017. WeMo Insight Switch. http://goo.gl/0WGDFe. (2017).

[15] belkin.com. 2017. WeMo Switch + Motion. https://goo.gl/sjUsi3. (2017).

[16] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies *(USENIX Security 13)*.

[17] Jonathan M Carlson, David Heckerman, and Guy Shani. 2009. Estimating false discovery rates for contingency tables. *Microsoft Res* (2009).

[18] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. Oauth demystified for mobile application developers. In *CCS*.

[19] William R. Cheswick and Steven M. Bellovin. 1994. *Firewalls and Internet Security: Repelling the Wily Hacker.*

[20] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *USENIX Security*.

[21] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. 2007. Traffic Classification Through Simple Statistical Fingerprinting. *SIGCOMM Comput. Commun. Rev.* (2007).

[22] darkreading.com. 2011. Firms Slow To Secure Flaws In Embedded Devices. http://goo.gl/b7Cltt. (2011).

[23] S.R. Das, S. Chita, N. Peterson, B. Shirazi, and M. Bhadkamkar. 2011. Home automation and security for mobile devices *(PERCOM Workshops '11)*.

[24] Soteris Demetriou, Xiaoyong Zhou, Muhammad Naveed, Yeonjoon Lee, Kan Yuan, XiaoFeng Wang, and Carl A. Gunter. 2015. What's in Your Dongle and Bank Account? Mandatory and Discretionary Protection of Android External Resources. *(NDSS '15)*.

[25] Tamara Denning, Tadayoshi Kohno, and Henry M. Levy. 2013. Computer Security and the Modern Home. *Commun. ACM* (2013).

[26] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy*.

[27] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security Symposium*.

[28] Ronald A Fisher. 1922. On the interpretation of χ 2 from contingency tables, and the calculation of P. *Journal of the Royal Statistical Society* (1922).

[29] Gartner. 2015. Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015. http://goo.gl/L9ubfl. (2015).

[30] github.com. 2017. Github: dex2jar. https://goo.gl/Hwx2WX. (2017).

[31] C. Gomez and J. Paradells. 2010. Wireless home automation networks: A survey of architectures and technologies. *Communications Magazine, IEEE* (2010).

[32] Dan Goodin. 2013. Welcome to the"Internet of Things" where even lights aren't hacker safe2 more wireless baby monitors hacked: Hackers remotely spied on babies and parents. http://goo.gl/l0qh05. (2013).

[33] google.com. 2017. OnHub - Google. https://goo.gl/igIM5c. (2017).

[34] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection *(MobiSys '12)*.

[35] tim Greene. 2014. Spike malware toolkit can infect Windows, Linux and ARM-based Linux devices. http://goo.gl/KQgSyT. (2014).

[36] Kashmir Hill. 2015. This guy's light bulb performed a DoS attack on his entire smart house. http://goo.gl/24skXK. (2015).

[37] honeywell.com. 2017. Honeywell. http://goo.gl/9yuiTX. (2017).

[38] ibabylabs.com. 2017. ibabylabs.com. https://goo.gl/y6Gdzd. (2017).

[39] IDC. 2016. IDC: Smartphone OS Market Share. http://goo.gl/y1uN4Q. (2016).

[40] indianexpress.com. 2016. Android malware 'Godless' has affected over 8.5 lakh devices globally. http://goo.gl/RE5ffK. (2016).

[41] infinit.dk. 2012. Nabto. http://goo.gl/ApJo1G. (2012).

[42] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. *(SP '12)*.

[43] J.M Jorup. 2016. "Internet of Things" security is hilariously broken and getting worse. http://goo.gl/PZgKN9. (2016).

[44] P. Judge and M. Ammar. 2002. Gothic: a group access control architecture for secure multicast and anycast *(INFOCOM '02)*.

[45] Isaac Kelly. 2012. Hacking the WeMo WiFi switch Part 1. https://goo.gl/PKeO1A. (2012).

[46] Insoon Kim. 2015. Is CCTV A Spy? Backdoor That Was Secretly Hidden In Chinese Products Were Found. http://goo.gl/3xQ7Dy. (2015).

[47] Ji Eun Kim, G. Boulos, J. Yackovich, T. Barth, C. Beckel, and D. Mosse. 2012. Seamless Integration of Heterogeneous Devices and Access Control in Smart Homes *(IE '12)*.

[48] Tiffany Hyun-Jin Kim, Lujo Bauer, James Newsome, Adrian Perrig, and Jesse Walker. 2010. Challenges in Access Right Assignment for Secure Home Networks *(HotSec'10)*.

[49] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* (2015).

[50] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. 2015. AntMonitor: A System for Monitoring from Mobile Devices *(SIGCOMM '15)*.

[51] A. Lioy, A. Pastor, F. Risso, R. Sassu, and A.L. Shaw. 2014. Offloading security applications into the network *(eChallenges e-2014)*.

[52] Sharon Machlis. 2015. IoT's dark side: Hundreds of unsecured devices open to attack. http://goo.gl/pM9TNk. (2015).

[53] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Capkun. 2014. Smartphones as Practical and Secure Location Verification Tokens for Payments. In *NDSS '14*.

[54] Michelle L. Mazurek, J. P. Arsenault, Joanna Bresee, Nitin Gupta, Iulia Ion, Christina Johns, Daniel Lee, Yuan Liang, Jenny Olsen, Brandon Salmon, Richard Shay, Kami Vaniea, Lujo Bauer, Lorrie Faith Cranor, Gregory R. Ganger, and Michael K. Reiter. 2010. Access Control for Home Data Sharing: Attitudes, Needs and Practices *(CHI '10)*.

[55] mcafee.com. 2013. Mobile Malware -The Rise Continues. http://goo.gl/3f1LP8. (2013).

[56] microsoft.com. 2017. Azure IoT Hub - Microsoft Azure. https://goo.gl/RYZTGC. (2017).

[57] myn3rd.com. 2017. My N3rd: CONNECT AND CONTROL ANYTHING FROM ANYWHERE. http://goo.gl/8gpa0D. (2017).

[58] nabto.com. 2015. Nabto IoT Platform Specifications. https://goo.gl/SekiZV. (2015).

[59] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. 2014. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android.. In *NDSS*.

[60] nest.com. 2017. Nest Protect. https://goo.gl/jM8ALk. (2017).

[61] nest.com. 2017. Nest Thermostat. https://goo.gl/oSIFfQ. (2017).

[62] Matte Noble. 2013. WeMo Hacking. http://goo.gl/C97vKv. (2013).

[63] Sukhvir Notra, Muhammad Siddiqi, Hassan H. Gharakheili, Vijay Sivaraman, and Roksana Boreli. 2014. An Experimental Study of Security and Privacy Risks with Emerging Household Appliances *(M2MSec '14)*.

[64] Pierluigi Paganini. 2013. Internet of Things - Symantec has discovered a new Linux worm. http://goo.gl/DwPGnM. (2013).

[65] Danny Palmer. 2016. This Android malware has infected 85 million devices and makes its creators 300,000 a month. http://goo.gl/4YbaWg. (2016).

[66] Sue Marquette Poremba. 2016. Studies Show Rise of the Mobile Malware Threat. http://goo.gl/VfUKB4. (2016).

[67] Jacob Poushter. 2016. Smartphone Ownership and Internet Usage Continues to Climb in Emerging Economies. *Pew Research Center: Global Attitudes & Trends* (2016).

[68] qualcomm.com. 2017. Trepn Power Profiler. https://goo.gl/KGrswV. (2017).

[69] Juha Saarinen. 2014. Vendors slow to patch OpenSSL vulnerabilities. http://goo.gl/9EFXAT. (2014).

[70] samsung.com. 2017. Samsung Family Hub Refrigerator. http://goo.gl/ddwlxb. (2017).

[71] securityfocus.com. 2013. Belkin WiFi NetCam video stream backdoor with unchangeable admin/admin credentials. http://goo.gl/XnmwAk. (2013).

[72] securityintelligence.com. 2015. 2015 Mobile Threat Report - The Rise of Mobile Malware. https://goo.gl/lhZ1yb. (2015).

[73] Sergey Shekyan and Artem Hartutyunyan. 2013. Watching the watchers:hacking wireless IP security cameras. In *HITB*.

[74] shodan.io. 2017. Shodan. https://goo.gl/vUL10K. (2017).

[75] Vijay Sivaraman, Dominic Chan, Dylan Earl, and Roksana Boreli. 2016. Smart-Phones Attacking Smart-Homes *(WiSec '16)*.

[76] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. 2015. Network-level security and privacy control for smart-home IoT devices *(WiMob '15)*.

[77] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. *(NDSS '13)*.
[78] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android *(NDSS '13)*.
[79] Ms. Smith. 2013. Eavesdropping made easy: Remote spying with WeMo Baby and an iPhone. http://goo.gl/OUxdUy. (2013).
[80] Ms. Smith. 2014. 500,000 Belkin WeMo users could be hacked; CERT issues advisory. http://goo.gl/HBN9HB. (2014).
[81] Howard Solomon. 2016. Mobile malware, unpatched Android devices are increasing problems say studies. http://goo.gl/EUGmDC. (2016).
[82] Mark Stanislav and Tod Beardsley. 2015. HACKING IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities. https://goo.gl/Uh7y4e. (2015).
[83] statista.com. 2016. Number of apps available in leading app stores as of June 2016. http://goo.gl/LO6umz. (2016).
[84] statista.com. 2017. Android version market share distribution among smartphone owners as of September 2016. http://goo.gl/vMm2t2. (2017).
[85] Darlene Storm. 2015. 2 more wireless baby monitors hacked: Hackers remotely spied on babies and parents. http://goo.gl/UIbWvA. (2015).
[86] Darlene Storm. 2015. Eerie music coming from wireless baby cam; is it a haunting? No, it's a hacker. http://goo.gl/49Larp. (2015).
[87] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. 2014. TrustDump: Reliable Memory Acquisition on Smartphones *(ESORICS '14)*.
[88] theverge.com. 2012. Square updates its credit card reader to include hardware encryption. http://goo.gl/G0Vji7. (2012).
[89] throughtek.com. 2015. Kalay Platform. http://goo.gl/t9oGM3. (2015).
[90] Blase Ur, Jaeyeon Jung, and Stuart Schechter. 2013. The Current State of Access Control for Smart Devices in Homes *(HUPS '13)*.
[91] weaved.com. 2015. Weaved Remote Connections. https://goo.gl/elBwS3. (2015).
[92] Fengguo Wei, Sankardas Roy, Xinming Ou, and others. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*.
[93] Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *STC '08*.
[94] Ryszard Winiewski and Tumbleson. 2017. A tool for reverse engineering Android apk files. http://goo.gl/26AzzN. (2017).
[95] yahoo.com. 2014. Proofpoint Uncovers Internet of Things (IoT) Cyberattack. http://goo.gl/GBqies. (2014).
[96] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiaoyong Zhou, and XiaoFeng Wang. 2015. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. *(IEEE Symposium on Security and Privacy)*.
[97] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces *(CODASPY '12)*.
[98] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. 2013. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources *(CCS '13)*.
[99] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution *(SP)*.
[100] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications *(NDSS '13)*.

A BEYOND THE APP-LEVEL ADVERSARY

Access from unauthorized authenticated guest phone. Consider the following cases: (a) a guest phone's role is only allowed to access the *unprotected* domain. If the target IoT device's type is in another domain, the router will reject the offending packets. (b) A guest phone might claim the identity of one of the HAN's Monitors. However, HanGuard uses **static IPs which are associated with phones/devices MAC addresses during setup** by the admin role. Any attempt to claim a reserved IP (*arp-spoofing*) or MAC address (*MAC-spoofing*) is validated through the control channel. The guest phone will fail the validation, the admin user is notified out-of-band and the culprit is removed form the network.

Compromised HAN user phones. Preventing phone compromises is out of the scope of this work since other solutions already exist and even deployed on commodity smartphones [1, 4, 12, 53, 87, 93]. For example, SELinux for Android [78] uses mandatory access control to ensure that even compromised system processes are restricted, and is deployed on all Android phones with version 4.4. and higher (more than 60% in 2015 [84]). Most Android phones are equipped with ARM processors [2] with TrustZone [1] which can be utilized for solutions stemming from the trusted computing domain. TZ-RKP [12] is a real-time kernel protection technique deployed on Samsung Galaxy phones that ensures the kernel integrity using the ARM TrustZone secure world. iOS devices have the Secure Enclave, a secure co-processor that is used to guarantee secure boot [4]. However, even if a phone is compromised, HanGuard can guarantee *phone-level* protection. We illustrate this with the following examples:

(a) An unauthorized phone might attempt to access an IoT device. Assuming it acquires the user credentials and the Monitor certificate, it can try to push a rule to the router to allow its flow. However, the router detects that the rule comes from a device whose *role* is not allowed to access the *type* of the target device and rejects the rule. (b) The phone might attempt to update the policy in its favor. Such attempts by a non-admin device—determined by the device MAC:IP address pair, Monitor certificate, and the user credentials—will be rejected. Even if the *admin* device and its user credentials are compromised and an update is pushed, the admin user always gets notified out-of-band. Thus she can revoke the update and take action. (c) The phone, might try to flood the *flow decision cache*. This would force the *GCS* service to retire older flows, essentially invalidating benign flows and causing DoS. To tackle this, we rate limit the flows a particular device can create. If that limit is surpassed, the device is penalized by having the router dropping all its packets for a few minutes. During that time, no flow entries will be added in the decision cache originating from that device. In all cases, HanGuard triggers its *out-of-band notification mechanism* when it detects a violation.

WPA2-PSK authentication and HanGuard network partition. On a typical WLAN node, once a subnetwork is created it can be configured to use a Service Set Identifier (SSID) and the WPA2-PSK security protocol. WPA2-PSK derives a unique pairwise transient key to encrypt the communication traffic between individual nodes on a HAN and the router. However, all keys are derived from the same SSID and a secret passphrase shared across all the nodes. As a result, a compromised phone could potentially use the key to directly connect to an IoT device, bypassing the router-level protection. To address this threat, HanGuard partitions the HAN into two default subnetworks, each with their own SSID/passphrase pair, one for user phones, PCs and laptops, and the other for IoT devices. This ensures that even a fully compromised phone cannot acquire the secret key used by smart-home devices.

Remote adversary. Commonly, an IoT device behind the NAT initiates a connection to its cloud, through which the cloud learns the device's external IP and port. If this information is exposed to an adversary, she can gain unfettered access to the device [43, 52, 73, 74]. To mitigate this, HanGuard uses a `port-restricted cone NAT` on the router, which ensures that only the flows from the remote IP:port pairs contacted before by a local device can reach that device. Note that this NAT mode is supported by most smart-home devices on the market [41, 58, 89, 91].