Vigilia: Securing Smart Home Edge Computing

Rahmadi Trimananda
University of California, Irvine
rtrimana@uci.edu

Ali Younis
University of California, Irvine
ayounis@uci.edu

Bojun Wang
University of California, Irvine
bojunw2@uci.edu

Bin Xu
University of California, Irvine
xub3@uci.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

Guoqing Xu
University of California, Los Angeles
harryxu@cs.ucla.edu

***Abstract*—Smart home IoT devices are becoming increasingly popular. Modern programmable smart home hubs such as SmartThings enable homeowners to manage devices in sophisticated ways to save energy, improve security, and provide conveniences. Unfortunately, many smart home systems contain vulnerabilities, potentially impacting home security and privacy. This paper presents Vigilia, a system that shrinks the attack surface of smart home IoT systems by restricting the network access of devices. As existing smart home systems are closed, we have created an open implementation of a similar programming and configuration model in Vigilia and extended the execution environment to maximally restrict communications by instantiating device-based network permissions. We have implemented and compared Vigilia with forefront IoT-defense systems; our results demonstrate that Vigilia outperforms these systems and incurs negligible overhead.**

***Keywords*-smart home; security; privacy; programming model;**

I. Introduction

Smart homes enable appliances to be controlled locally via the network and typically enable more sophisticated control systems [67]. Companies have launched a wide range of smart-home devices, many of which have serious security issues. A study reported vulnerabilities in 70% of the devices investigated [17]. Bugs have been found in a wide range of devices including routers [69], [71], smartcams [58], [14], [73], baby monitors [56], [60], [31], smart hubs [72], sprinklers [6], smart plugs [35], and smart fridges [1]. The problems in these systems are more basic than missing buffer checks—some of these devices have unsecured embedded web servers that allow anyone to update the firmware, have default passwords, use insecure authentication, or use clear text communications. To demonstrate the severity of the problem, we assigned public IPs to our webcams (§VII-C). All of them were hacked *within 15 minutes*!

Part of the promise of smart home systems is the ability of collections of devices to work together to be smarter and more capable than individual devices. Achieving this requires integration between different devices, which may come from different manufacturers with entirely different software stacks, *e.g.*, Nest Thermostat, Wemo Switch, etc. Smart home hubs support integration between these disparate devices, but existing hubs including SmartThings have serious security weaknesses.

SmartThings: SmartThings is a smart home environment created by Samsung [61]. This environment allows smart home devices (*e.g.*, SmartThings and third-party devices) to be connected to a home network, monitored, and controlled through the SmartThings phone app. Among these devices, Zigbee or Z-Wave devices are connected to the LAN via the SmartThings smart hub, while WiFi devices are directly connected to the LAN.

The SmartThings environment also allows the user to create smart home apps (SmartApps) to manage connected devices to perform specific functionality. For example, a smart switch app that manages motion sensors and switches could use the sensors to detect motion as a trigger to turn on a switch. SmartApps communicate with devices through device handlers. A device handler exposes device capabilities that allow the SmartApp to access device features, *e.g.*, `switch.on()` and `switch.off()` for a switch. Most SmartApps and device handlers run on the SmartThings cloud and have the local smart hubs relay commands to the physical devices. SmartApps are written in Groovy, a managed programming language running on top of the JVM [27].

The SmartThings environment has the following weaknesses:

(1) Device Vulnerabilities: Many IoT devices connect directly to the home Internet connection and communicate with the hub via the LAN or the cloud. Many of these devices either intentionally trust communication from the local area network (*e.g.*, Wemo, LiFX), use inadequate authentication mechanisms (*e.g.*, a short PIN in the case of D-Link), or have backdoors (*e.g.*, Blossom sprinkler) that make them vulnerable to attack.

*(2) Trusted Codebases with Bad Security Records (*e.g.*, JVMs):* The SmartThings system executes device drivers and applications on a JVM and relies on the JVM to provide safety. Bugs in the JVM could potentially allow applications to subvert the capability system and access arbitrary devices.

(3) Excessive Access Granted to Cloud Servers: The SmartThings system executes most applications and device handlers on their cloud servers and uses the hub to relay

commands to the local devices. The hub punches through the home firewall to give the SmartThings cloud servers arbitrary access to communicate with any local device. Note that while compromised firmware updates could conceptually be used to obtain similar access, the scenarios are fundamentally different because firmware updates are often signed. Thus, with appropriate key protection mechanisms, they can be made difficult for attackers to compromise.

(4) Excessive Access Granted to Device Handlers or SmartApps: SmartThings device handlers have the ability to capture all SSDP network traffic to the hub [37], communicate with arbitrary IP addresses and ports by reconfiguring the device's network address, and send arbitrary commands to arbitrary Zigbee devices [20].

When a homeowner purchases a new IoT device, they first make it available to their SmartThings hub. SmartThings provides drivers for a wide range of third party devices; users can also write their own drivers or import third-party driver code. Some popular devices such as the Nest thermostat can only be integrated into SmartThings via third-party drivers that are not subject to any code review process.

When a SmartApp is first installed, the user configures it by selecting the devices to be monitored and controlled. This process grants the SmartApp the capabilities to access those devices. While the SmartThings capability system appears at first glance to provide strong security assurances, it can be easily subverted. For example, a SmartApp can conspire with a device handler to subscribe to all SSDP traffic to the hub, open arbitrary connections to cloud servers, or obtain arbitrary access to LAN and Zigbee devices.

Our initial goal was to secure a popular real-world system such as SmartThings. However, SmartThings is closed source—we could not directly enhance it as we do not have access to its source code. As a result, we had to develop a new distributed IoT infrastructure that closely follows the programming and computation model of SmartThings. We demonstrate the viability of our approach by implementing Vigilia on top of this new system. Our idea is generally applicable to SmartThings and any other smart home IoT infrastructure that uses similar models.

Vigilia Approach: We developed Vigilia, a new cross-layer technique to harden smart home systems. First, Vigilia restricts network access—Vigilia uses a similar programming model as SmartThings but leverages the configuration information that is already available to also restrict network access. Vigilia makes the network primarily responsible for the security of IoT devices—Vigilia implements a default deny policy for all IoT devices and smart home applications. Access is only granted when user has explicitly configured a smart home application to use a specific device. A key advantage of this approach is that it becomes less critical that end users keep every IoT device fully patched. At the same time, by leveraging the configuration information that is already present, Vigilia's security mechanisms never get in the way of legitimate computations.

Second, Vigilia provides more fine-grained access control to specific devices. In Vigilia, a smart home application controls a specific device via a device driver. The interaction between the smart home application and the device driver occurs through *remote method invocation* (RMI). Device features are exposed as API methods in the device's driver class. This is implemented as *capability-based RMI* that only allows a limited set of API methods to be called depending on the configuration (§III). Thus, this mechanism provides more fine-grained access control to specific devices on top of the network policy restrictions.

Vigilia implements a lightweight approach to securing smart home systems at the network and operating system layers. This work leverages the observation that *most IoT devices are not general-purpose; they do not need to communicate with arbitrary machines and thus do not require full network access.* By enforcing access at the network level, Vigilia shifts the primary burden for security from individual devices to the network. The net effect is that system security no longer relies on every device manufacturer securing their devices and end users keeping devices patched—helping users secure IoT devices when manufacturers do not.

This paper makes the following contributions:

- **Automatic Extraction of Enforcement of Security Policies:** It presents techniques that automatically extract and enforce fine-grained security policies on applications written using a programming model that is similar to SmartThings.
- **Secure Enforcement Mechanism:** It uses a set of router-based techniques including modifications to the WiFi stack that ensure that compromised devices cannot subvert the enforcement mechanisms by masquerading as the router or another device.
- **No Spurious Failures:** It statically checks that programs will respect the policies at runtime and thus will never spuriously fail due to the security enforcement mechanisms.
- **Implementation:** It provides an open implementation of a smart home programming model that is similar to mainstream (close) platforms. We have made this implementation available at `http://plrg.eecs.uci.edu/vigilia/`.
- **Evaluation:** We have evaluated Vigilia on four smart home applications that control commercially available IoT devices. Our results demonstrate that Vigilia, among existing commercial and research systems, is the best at protecting these applications from various attacks with only minimal overhead.

II. Threat Model and Guarantees

Vigilia protects IoT devices from attacks resulting from overprivileged network access. We use the following threat

model: 1) the IoT devices have *vulnerabilities*, 2) attackers have *full knowledge* of Vigilia, and 3) attackers have *access* to the home network via a *compromised* laptop or device, *not* physical access.

Our threat model is *stronger* than those assumed in the existing IoT-defense systems that we are aware of. Commercial systems [16], [15] typically assume that threats come from the outside network and the home network is well-guarded. In the research community, systems such as HomeOS and HanGuard [13], [10] assume that attacks can come from the home network, but they focus on PC and smartphone apps vulnerabilities. IoTSec [62] mainly safeguards against arbitrary port accesses. Our comparison (§VII-B) between Vigilia, and commercial and research systems demonstrates that the threat model we use enables stronger protection of IoT devices than these systems.

We do *not* trust application developers—Vigilia ensures that applications can only perform network, Zigbee, and file accesses allowed by the user configuration. We do *not* assume that application processes are trusted. The attacker may tamper with the source/binary code of the IoT program or the language runtime such as the JVM, *e.g.*, exposing device driver objects to applications that are not supposed to access those devices. In such cases, the unspecified communications will be blocked by the WiFi router or the Zigbee gateway—we trust the integrity of the Vigilia WiFi router and the Zigbee gateway. We do *not* trust the wireless stack of any smart home device. This includes not trusting devices to use the assigned MAC or IP addresses.

We assume a partial trust of the OSs (*i.e.*, TOMOYO Linux [65]) running on Raspberry Pi nodes. Defending against attacks on the OS is out of scope. Note that even if the OS is compromised, the attacker can only obtain the permissions of other Vigilia components on the same device as the router enforces inter-device permissions.

Vigilia provides the following *guarantees*: (1) all communications that are *not* explicitly configured by the user with a Vigilia component or IoT device are blocked; (2) a Vigilia component is only allowed to perform actions permitted by the capabilities it is granted; (3) smart home applications developed by honest developers will never be blocked by Vigilia's checks.

III. Example

Figure 1 presents a block diagram of an example smart irrigation system that connects a set of IoT devices, Raspberry Pis, Zigbee gateways, and Zigbee devices with a router. An *IoT device* is a smart home device connected to the WiFi network such as a sprinkler. Similar to the SmartThings system, each device has a *device driver* that interfaces between the device and smart home applications. The device driver runs on a Raspberry Pi running Raspbian. Similar to SmartThings, we expect that device drivers will be written either by the device manufacturer, Vigilia developers,

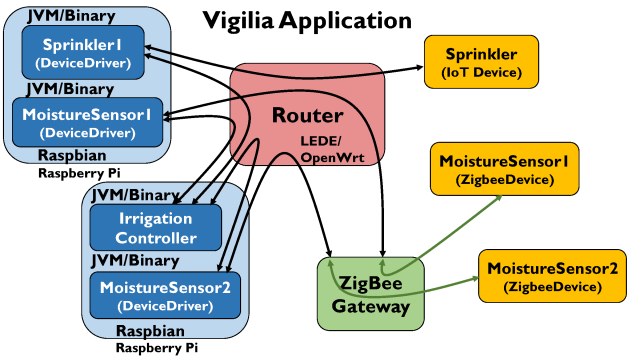

Figure 1. A closer examination of an irrigation system.

or third-party hobbyists. For standard classes of devices, we expect that the Vigilia developers would define standardized APIs to support compatibility much like SmartThings ecosystem. For each smart home application, there is an *application* that interacts with the drivers of the involved devices to achieve certain smart home functionalities. In our example, the application talks to a set of moisture sensors (discussed shortly) to measure soil moisture, which will be used to adjust the irrigation schedule for the sprinkler. The application thus needs to communicate with the drivers of the sprinkler and the moisture sensors.

One significant difference between Vigilia and SmartThings is that Vigilia runs applications on local compute nodes. This has significant advantages in that Vigilia applications can operate even if Internet connectivity is lost. The application also runs on a Raspberry Pi, which may or may not be the same one that hosts the drivers. A smart home system often has multiple applications and thus multiple applications may exist simultaneously. The drivers and the application may be developed by different developers and/or in different languages. For example, if they are written in Java, they are executed by JVMs; if they are C++ programs, their binary code is directly executed. In this paper, we refer to device drivers or applications as *components*.

Zigbee is a standard communication protocol that connects devices with small, low-power radios. A smart home system may also contain Zigbee devices that connect to the home WiFi through a Zigbee gateway. In this case, the Zigbee gateway has an IP address from the LAN while the Zigbee devices do not support TCP/IP and only have Zigbee addresses. Hence, device drivers must communicate with Zigbee devices via requests made to the Zigbee gateway.

IV. Architecture and Programming Model

Figure 2 depicts Vigilia's architecture. Vigilia implements key components of the SmartThings programming model and system architecture to ensure that our techniques are applicable to real smart home systems. We cannot directly

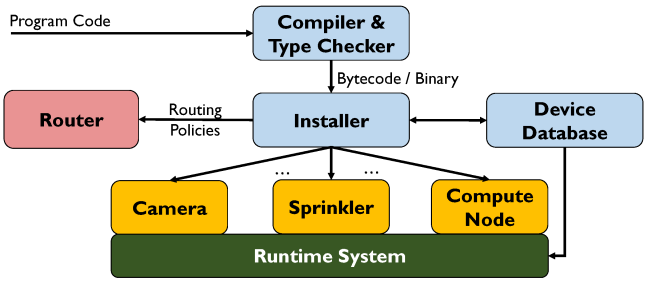

Figure 2. Vigilia system architecture.

```
1   public class IrrigationController extends
2     Application implements Irrigation {
3   @config Set<Sprinkler> sprinklers;
4   @config Set<MoistureSensor>
5     moisturesensors;
6   @config Relation<MoistureSensor,
7     Sprinkler> sensortosprinklers;
8   @config Set<Gateway> phone;
9   @config Set<Address> weatherforecast;
10  //Interface method containing initialization ↩
        logic
11  public void init() {
12    ...
13  }
14  //Other computation methods
15  private void turnOn() {
16    ...
17  }
18 }
```

Figure 3. Example application code in Java.

use SmartThings as key components are cloud-based and unavailable and thus not amenable to modification.

Applications are compiled using the Vigilia tool chain. The tool chain checks that applications will never violate the declared permissions at runtime. Applications are then deployed using the Vigilia installer. The deployment process involves the end user specifying how the application should be configured for the given house. For example, this process might specify which switches and light bulbs an application has access to, and which switches should control which light bulbs. The Vigilia installer then computes a set of permissions that is required for the given installation. Finally, the Vigilia runtime enforces these permissions.

To enable applications to fully realizing the potential benefits of smart home systems, systems like SmartThings implement and expose rich APIs—potentially increasing their attack surface. Complex interactions among different devices requires a programming framework that makes it easy for components to interact when desired while at the same time blocking undesired interactions. Like SmartThings, Vigilia users implicitly grant permissions to a smart home application when they configure the application to implement the desired functionality. The permissions required are partly determined by the application's intended function—thus, some information about the nature of the permissions required by smart home applications must be specified by the developer. However, the developer does not know the specifics of a given deployment. For example, the developer would typically not know how many light bulbs or cameras an end user has installed (or what rooms these devices are installed in). Instead, developers only have a high-level view of which type of device the application needs and the required relationships between devices (*e.g.*, that they are in the same room).

Like SmartThings, Vigilia employs an object-oriented component model. Each device driver or smart home application has a corresponding *class*. Vigilia classes can declare *sets* and *relations*. These sets and relations are declared as data fields in these classes. Sets represent abstract communication permissions. In the SmartThings programming model, the same information is specified using the `preferences` keyword. Vigilia extends the SmartThings programming model by using RMI to both isolate components and to support distributed applications. Communication with devices or other smart home applications are implemented using *remote method invocation* or RMI. As IoT systems may contain components written in different languages, Vigilia provides cross-language support for RMI. Vigilia contains a RMI compiler that parses policy files defining the capabilities of a component to generate code that implements the RMI stubs and skeletons.

Irrigation Application Code: To better explain the programming model, we show a code example for a smart irrigation application. Figure 3 presents Java code for the example. In this example, the application communicates with a set of sprinklers to water the lawn and a set of moisture sensors to monitor soil moisture. The `IrrigationController` class implements the smart irrigation application. The irrigation application uses information from several moisture sensors to adjust watering schedules and thus must communicate with the moisture sensors. Each application class extends the Vigilia `Application` class and implements the `init` method. This method will be invoked by the Vigilia runtime during application startup.

Abstract Permissions: In general, developers only know which types of devices an application needs to communicate with; the exact device instances in each class are specified during the site-specific installation process. Vigilia provides the developer with an *abstract permission* model to specify the permissions required by a given application. These abstract permissions are specified in terms of members of sets (similar to SmartThings preferences). The Vigilia installer (like the SmartThings installer) then *instantiates* these permissions by specifying the exact members of sets.

For example in Line 4 of Figure 3, a developer specifies an abstract permission that allows communication between the application and the generic type of moisture sensor by declaring

`@config Set<MoistureSensor> moisturesensors` in the `IrrigationController` application class. This declares that the application has the abstract permission that allows it to talk to moisture sensors at runtime. The Vigilia programming model uses annotations either in the code (Java) or in a separate file (C++) to allow the developer to express this information.

In the above example, the developer does not need to worry about how to create the set object and the contained `MoistureSensor` objects in the program as these objects are created by the runtime system. For example, if the end user configures two moisture sensors for the application, then the Vigilia runtime would create two `MoistureSensor` objects and insert both objects into the `moisturesensors` set. When the program is executed, the Vigilia runtime system initializes this set with references to the appropriate sensor objects. Vigilia components such as applications and device drivers run in separate processes (*i.e.*, JVM/binary). Since communication between components is implemented via RMI, a reference from the `moisturesensors` set can be used to directly communicate with the sensor. Components in Vigilia can only communicate with other components that are specified by this set-based model.

Application Installation: During the installation process, the end user configures the application for their home. This configuration process is not unique to Vigilia, most smart home systems include a similar process in which the end user must specify which devices should be controlled and how they should be. Moreover, the Vigilia installation process for an application is similar to SmartThings. The Vigilia installer asks the end user to configure the concrete device instances to be used by an application. For example, a sprinkler controller may ask which *moisture sensors* should be used to monitor soil moisture. The end user specifies which specific moisture sensors the application should use by defining the devices that comprise the set of moisture sensors. Finally, the Vigilia installer uses abstract permissions and user configuration to generate concrete permissions. Abstract permissions are generic for the application, while concrete permissions are specific to installations and grant access to physical devices.

Vigilia extends the set-based model with *relations*, specifying relations between devices and communicating configuration information. For our irrigation example, the application must know which sprinklers are located near which moisture sensors. During installation, the user provides this information in relations as it is specific to their installation. Line 6 of Figure 3 declares the `sensortosprinklers` relation that maps moisture sensors to the nearby sprinklers. Similar to sets, relation objects are also constructed by the runtime system.

Communication: Line 8 of Figure 3 declares a set of gateways for smartphones. Devices like tablets/smartphones/laptops can be used to provide a user interface,

```
1  class SpruceSensor : public Device,
2                       public MoistureSensor {
3     private:
4        Set<ZigbeeAddress*> sprucesensor;
5        Set<DeviceAddress*> zigbeegateway;
6        double moisture;
7        double temp;
8     public:
9        void init();
10       double getMoisture();
11       double getTemperature();
12 }
```

Figure 4. Example device driver header in C++.

through which users can input application parameters. Finally, Line 9 declares a set of addresses of cloud-based servers that provide weather forecast information. Vigilia uses an oblivious cloud-based key-value store to provide secure storage and communication even in the presence of malicious cloud servers.

Device Drivers: Figure 4 presents a device driver class declaration in C++ for the moisture sensor used by our irrigation example. Our irrigation example uses a Spruce moisture sensor [63], which is a Zigbee-based wireless sensor. To communicate with the sensor, the device driver must send packets to the sensor via a Zigbee gateway. Thus the driver needs two addresses: (1) the IP address for the Zigbee gateway and (2) the Zigbee network address for the Spruce sensor.

Device drivers use the same set-based mechanism to obtain direct access to network-based devices. The installation process stores the system configuration parameterized by the devices' MAC addresses, and the Vigilia runtime maps the MAC addresses to the corresponding IP addresses. Network access is only permitted via runtime provided IP address/port pairs, and thus the Vigilia runtime knows which devices a driver may communicate with. The Vigilia runtime uses this information to configure the routing policies. Device drivers may declare a set of public methods such as `getMoisture` for the application to get/set information from/to the device.

V. Vigilia Security Mechanisms

We next discuss the security mechanisms Vigilia implements for the SmartThings' programming model.

Checking: One challenge is *how to statically eliminate permission bugs*, in which an application accidentally exceeds its declared permissions and thus fails at runtime when the Vigilia runtime enforcement framework blocks the illegal access. The Vigilia static checking framework is designed to help honest developers ensure that their applications never fail at runtime because of Vigilia's runtime enforcement framework. *It is important to note that Vigilia does not rely on the static checks for security—applications that attempt to violate their permissions will be blocked by runtime checks.* The static checker needs to notify the developer of any network accesses that are doomed to be blocked by runtime

checks. For example, an application could potentially violate its permissions if it were to obtain a reference to a device object from some other component and then attempt to use that reference to access the underlying device. Such an access would fail at runtime and potentially cause the application to crash.

Vigilia supports both Java and C++. One goal of Vigilia is to make it easy to support new languages and thus we minimize the dependence on specialized compiler passes for static checking. To the degree possible Vigilia uses the existing language type system to check for permission violations. Vigilia implements these checks via the Vigilia RMI compiler. The Vigilia RMI compiler uses the declared types to ensure that the existing language type system will catch any accidental sharing of references to device objects by an application.

SmartThings applications have full Internet access. A malicious app can easily leak private information. Internet access may also provide a conduit to attack benign applications. On the other hand, some functionality requires Internet access to implement. Thus, Vigilia supports managed access to TCP/IP sockets. This ensures that Vigilia is aware of any potential TCP/IP accesses. If a program were to attempt other accesses, they would be blocked by the Vigilia enforcement framework. The Java implementation of Vigilia's checker uses a type checker to ensure that Java Vigilia applications do not attempt to directly use raw TCP/IP sockets for communication. The C++ implementation does not implement this particular check—note that this does not impact security, but developers could potentially attempt direct network accesses that would be blocked at runtime.

Vigilia Installer: The Vigilia installer manages the installation of new devices and smart home applications. A major issue with the SmartThings system is that it trusts that devices on the home network are not malicious. Under SmartThings, a single malicious device on the home network has full network access to all other devices. Vigilia fully isolates each IoT device from every other device on the network, permitting communication only when applications are explicitly configured to use a device during the installation process. When a new device is installed, Vigilia must update its database to include a record of the device's MAC address and type. To prevent MAC address spoofing or sniffing attacks from circumventing Vigilia's access control, Vigilia assigns a unique *pre-shared key* (PSK) to each device. The Vigilia router ties each unique PSK to a specific device MAC address. Note that while some Android and iOS devices implement MAC randomization, it is used only when probing for wireless networks. Thus, our approach is compatible with modern smart phones. Finally, the installer maps the device to a specific driver.

The Vigilia installer also manages the addition of new smart home applications. Installing a new smart home application requires specifying the device instances that the smart home application can control. For each type of abstract permission the smart home application has requested, the Vigilia installer presents the list of devices that could provide those capabilities. The user then selects the subset of devices she wishes the application to use. For relations, the user specifies the pairs that comprise the relation (*e.g.*, that a moisture sensor is close to a given sprinkler head).

Enforcement: Vigilia implements its security model by combining a range of known techniques. It begins with a modified wireless router based on LEDE—now merged with OpenWrt [53]. Many commercially available routers are built using a similar core code base, so it should be relatively straightforward to modify existing routers to implement the necessary functionality. The Vigilia router allows wireless devices on the same wireless network to have different PSKs. This allows the router to prevent both MAC spoofing and sniffing attacks. The effect is that Vigilia can trust the MAC address of a device and that the wireless communications between the router and other devices are secure. Vigilia then uses firewall rules to prevent IP spoofing so that it can trust IP addresses.

Compute nodes can run more than one computation and these computations may have different permissions. Vigilia assigns different ports to different computations on the same node so that other devices can identify a communication's source. Vigilia sandboxes client code using TOMOYO Linux to ensure that client processes cannot fake port numbers. TOMOYO Linux also ensures that processes do not access the files of other processes.

Vigilia implements concrete permission checks by translating each access permission into a corresponding firewall rule. Vigilia's *default* policy is to *block communications*—e.g., *unused smart home devices are not allowed to communicate with anything*.

So far we have only discussed restricting network accesses. However, devices may have many features (*e.g.*, read temperature and set temperature), and it is important to restrict accesses to only the necessary features. To support restrictive feature access, Vigilia employs a *capability-based RMI*—device features are often exposed as API methods in the device's driver class and thus accessing device features is often done through remote invocations on the corresponding methods. A *capability* in Vigilia is a device feature that consists of a set of methods from its corresponding class. A component can declare multiple capabilities and the capabilities can contain overlapping methods.

Components declare the capabilities they require from other components. The RMI compiler uses these policy files to generate stubs and skeletons that only provide access to the declared capabilities. Although Vigilia's security guarantees for capabilities are enforced dynamically, this code generation strategy enables the existing C++ or Java compiler to statically check that a component does not

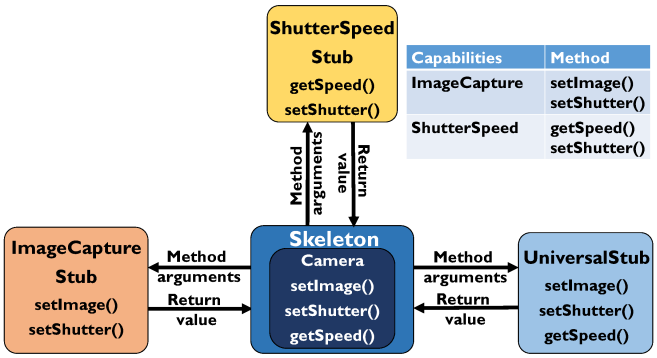

Figure 5. Capability-based RMI example.

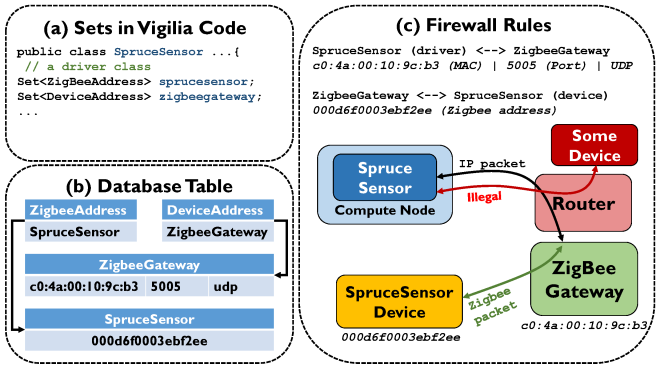

Figure 6. Vigilia program (*i.e.*, irrigation system) (a), device database (b), and instantiated firewall rules (c).

exceed its declared capabilities. This ensures that a well-behaved component will never fail a runtime security check.

Figure 5 shows an interaction between a `Camera` object and the stubs generated from the original `Camera` interface. The `Camera` interface has two capabilities, namely `ImageCapture` and `ShutterSpeed`. Each of these capabilities has two methods and three stubs (`ShutterSpeedStub`, `ImageCaptureStub`, and `UniversalStub`) are generated based on each combination of the capabilities. The skeleton supports all the methods. Vigilia's capability-based model is complementary to firewall rules.

This means the problem of restricting feature accesses can be reduced to restricting remote method invocations. Vigilia enforces capabilities by using request filters in its RMI request server—these filters are automatically configured by Vigilia, and use the source port and IP address to determine whether a given request is allowed.

Figure 6 shows the relationship between the programming model, Vigilia's configuration database, and the firewall rules. The developer specifies that the Spruce sensor driver communicates with the Spruce sensor. Since the driver runs on a Raspberry Pi while the Spruce sensor is a Zigbee device that needs to communicate via a Zigbee gateway, the developer adds a second set that enables the driver to obtain a reference to appropriate the Zigbee gateway. These two abstract permissions have two separate effects. They mean that the code can only communicate with the Zigbee gateway specified by the `DeviceAddress` object and can only communicate with the `ZigbeeAddress` for the Spruce sensor. These abstract permissions will be concretized into concrete permissions at installation, which, together with the network configuration in the device database (Figure 6(b)), will be used by Vigilia to generate the firewall rules for the router (Figure 6(c)). As a result, the router will block any communication inconsistent with these permissions.

VI. Vigilia Runtime System

The Vigilia runtime system is a distributed system with a master and several slaves.

Startup: The master manages the application startup process. The master generates a deployment plan for an application, configures the appropriate firewall rules for both the router and every compute node, and then sends requests to slave processes to startup the components. Each component is started inside of a sandbox that constrains the component to the specified ports.

Wireless Network Filtering: In the default configuration, a standard firewall will not filter traffic between devices on the same wireless network as the traffic never passes through the firewall. Access points typically offer two modes of operation: the standard mode, which forwards all traffic between clients, and the client isolation mode, which blocks all traffic between clients. However, the Linux kernel firewall can be configured to filter these packets. This is implemented by: (1) enabling access point isolation, (2) turning on bridge hairpin mode (also called 'reflective relay') for the wireless LAN interface to force the traffic through the kernel firewall, and (3) then using iptables to filter the traffic.

Vigilia modifies the WiFi stack to secure it against network-level attacks such as snooping, ARP-spoofing, and MAC-spoofing that would otherwise subvert Vigilia. Most IoT devices only support the pre-shared key (PSK) mode of WPA/WPA2 and do not support WPA/WPA2 Enterprise mode. This introduces a potential attack—even though each device eventually negotiates its own key, in the pre-shared key mode all devices on the same network know the same initial shared key. Any device that knows the pre-shared key and monitors the key negotiation can extract the private key.

Surprisingly, it turns out that it is possible to assign a unique PSK to each MAC address without breaking the WPA/WPA2 protocol. This prevents devices from computing the private keys of other devices, ensuring that malicious devices cannot masquerade as the router. This approach also effectively locks a physical device to a specific MAC address—malicious devices cannot spoof the MAC addresses of other devices as they do not know the MAC-specific PSK. The Vigilia router also enforces that MAC

addresses are locked to the specific assigned IP address—any spoofed traffic is dropped.

Vigilia uses an Android app to configure new devices on the network. The app generates a new PSK and sends the PSK to the router using ssh. The router then changes the default password for the network to this PSK to allow the new device to join the network. It then detects the MAC address of the new device, adds the MAC address-PSK pair to its database, and reverts to the default PSK.

The shared group key, which is used for broadcasting messages, can also be misused by attackers. Vigilia addresses this issue by assigning a unique randomized group key to each device (the router then unicasts group packets) and combining this with proxy ARP [55]. Note that while these options are present in the *hostapd* code, they do not work and required us to fix them.

Application Sandboxing: Vigilia can run multiple applications on the same host. This brings the possibility that a malicious application can masquerade as another application on the same host by stealing the other application's port. Alternatively, a malicious application might try to access or modify files that are owned by another application. To prevent these attacks, Vigilia sandboxes applications using TOMOYO Linux [65]—components are restricted to their own ports and files.

Zigbee Support: An issue with SmartThings is that any driver that obtains the Zigbee address of any Zigbee device can send commands to it [20]. The problem is that device drivers explicitly build low-level Zigbee packets. These packets include the destination address for the commands and the address where responses should be sent. Thus, SmartThings trusts that device drivers are not malicious. Malicious device drivers can easily communicate with any Zigbee device whose address they have.

Vigilia guarantees that device drivers cannot interact with the wrong Zigbee devices. Vigilia's Zigbee support consists of four components: (1) language support for communicating Zigbee addresses to device drivers, (2) language support to ensure that honest device drivers do not manually produce Zigbee address objects, (3) a Zigbee abstraction that separates the specification of addresses from device commands, and (4) a Zigbee firewall that verifies that the given device driver has permission to communicate with the specific Zigbee device.

At the language level, Vigilia uses the same basic set-based abstraction that it uses for both RMI and IP addresses to check for permission bugs in Zigbee accesses. It then enforces these properties using runtime permission checks in the Zigbee gateway. The Zigbee gateway checks are configured automatically by the Vigilia master to implement the permissions granted by the end user. These checks use the source port and IP address to verify that a given Zigbee device driver has been granted permission to communicate with the specific Zigbee device address.

Table I
LINES OF CODE IN VIGILIA APPLICATIONS.

Application	Application LOC	Driver LOC	Library LOC	Android LOC
Irrigation	4,075	2,975	401,843	208
Lights	1,683	3,456	401,843	N/A
Music	1,237	2,434	25,254	641
Home Security	2,299	4,177	401,843	187

Some Zigbee requests can leak information about other devices or configure a Zigbee device to interact with other devices. Thus the Zigbee gateway limits the types of messages a device driver can send to prevent the device driver from directly performing commands such as device discovery. The Zigbee gateway also filters incoming messages to ensure that device drivers only receive messages about the relevant device.

Incoming messages are often reports that are generated by a network node. For a node to receive information from another network node it must tell that node to send reports using a ZDO bind command. The Zigbee gateway remembers which driver performed a ZDO bind command, and to which node and cluster. When a report arrives from a Zigbee node, the gateway consults a table to determine which driver should receive it.

VII. EVALUATION

We deployed Vigilia on a test bed that consists of the following devices: 2 Raspberry Pi 2 compute nodes, a Google Nexus 5X smartphone, a Netgear Nighthawk R7800 wireless router, 2 LIFX Color 1000 bulbs, 4 Amcrest IP2M-841 ProHD 1080P cameras, a XBee S2C Zigbee module attached to a Raspberry Pi 1 (Zigbee gateway), a Spruce soil moisture Zigbee sensor, a Blossom sprinkler controller, 2 iHome iWS2 AirPlay speakers, a D-Link DCH-S220 siren, 3 Samsung SmartThings Zigbee sensors (motion, water-leak, and multi-purpose), and a Kwikset SmartCode 910 Zigbee lock. We have made the implementation of Vigilia publicly available at `http://plrg.eecs.uci.edu/vigilia/`.

Table I presents the lines of code for our applications.

Our test bed is built in a smart home lab environment. Figure 7 shows the hardware setup in the lab.

A. Applications

We implemented four applications on our test bed. Table II presents the summary of these applications.

Irrigation: The irrigation application optimizes watering to conserve water. It uses the Spruce moisture sensor to measure soil moisture. The system makes use of weather forecasts to determine the expected precipitation. When people walk on a lawn, they stress the lawn and thus it requires more water [2], [29]. It uses cameras to monitor lawn usage and thus whether it requires extra water. The Spruce moisture sensor uses Zigbee to communicate; we

Table II
SUMMARY OF VIGILIA APPLICATIONS.

Application	Smart Home Devices	Security Properties
Irrigation	1 Spruce soil moisture sensor 1 Blossom sprinkler controller 1 Amcrest camera 1 Google smartphone	This benchmark uses the device drivers for camera, Spruce moisture sensor, and sprinkler controller. It also includes a Zigbee gateway that relays messages to the Spruce sensor. Vigilia generates firewall rules that only allow the following communication: (1) the application can communicate with the drivers, phone, and the weather forecast website and (2) each device driver can communicate with its respective device. Each communication channel is isolated from the others and from all outside devices by (1) the compute node firewall and (2) the router firewall. The runtime system sends filtering rules also to the Zigbee gateway, ensuring that the Spruce driver can only communicate with the Spruce sensor.
Lights	2 LIFX light bulbs 2 Amcrest cameras	This benchmark uses the device drivers for camera and light bulb. Vigilia generates firewall rules that only allow the following communication: (1) the application can communicate with the device drivers and (2) each device driver can communicate with its respective device (*i.e.*, light bulb or camera). Each communication channel is isolated in a way similar to Irrigation.
Music	2 iHome speakers 1 Google smartphone	This benchmark uses a phone app and two speaker drivers. Vigilia generates firewall rules that only allow the following communication: (1) the main music application can communicate with the speaker drivers and the phone app, and (2) each of the device drivers can communicate with its respective speaker. Each communication channel is isolated in a similar manner.
Home Security	3 Samsung SmartThings sensors 1 Kwikset door lock 1 Amcrest camera 1 D-Link siren 1 Google smartphone	This benchmark uses the device drivers for camera, siren, door lock, and SmartThings sensors. Vigilia generates firewall rules that only allow the following communication: (1) the main home security application can communicate with it device drivers and the cloud, and (2) each device driver can communicate with its respective device. Each communication channel is isolated in a similar manner.

Table III
ATTACKS PERFORMED ON DEVICES.

No.	Attack	Application	Detail
1.	Sprinkler attack	Irrigation	A rogue program that controls the sprinkler (*i.e.*, turn on valves, reconfigure wireless connectivity, and update the firmware based on a non-documented, non-secured RESTful API to port 80 [6]).
2.	Light bulb attack	Lights	A rogue program that issues commands to turn the light on and off (port 56700).
3.	Speaker attack	Music	A rogue program that sends and plays music file on the speaker (port 80).
4.	Camera attack	Home Security	A HTTP URL is used to view the main/sub stream via a web browser (port 80).
5.	Siren attack	Home Security	A rogue program that launches a brute-force attack to guess the PIN code of the siren; an attacker can use this PIN code to perform a valid authentication (port 80).
6.	Deauth. attacks	All	A jammer is used to deauthenticate a specific device (*i.e.*, sprinkler, light bulb, speaker, camera, or siren) from its original *access point* (AP) to let it join a malicious AP with the same SSID and PSK as the ones used for the actual AP. Thereafter, the device is attacked using the attack for the specific device (*i.e.*, attack 1, 2, 3, 4, or 5).

Figure 7. Vigilia hardware setup.

have implemented a driver for this sensor that uses the sensor to monitor soil moisture. An Amcrest camera monitors the usage of the lawn to adjust the soil moisture target. An Android app provides the user interface. Finally, a Blossom sprinkler controller actuates the sprinklers.

Lights: The light application attempts to save energy by turning lights off in unoccupied spaces, and to improve sleep by adjusting brightness and color temperature to match the sun's color [32], [28], [9]. The application uses cameras combined with image processing to detect people. We use two Amcrest cameras to monitor rooms and control the two LIFX light bulbs.

Music: The music application tracks people using WiFi-based indoor localization of their cell phone and plays music from the closest speakers. An Android phone is used to implement localization and play music through two iHome speakers.

Home Security: The home security application is modeled after commercial home security products. Such applications usually consist of multiple sensors that can detect in-

Table IV
VIGILIA COMPARISON WITH OTHER SYSTEMS.

Attack	Normal	IoTSec	Vigilia
Sprinkler cont. attack	✓	✓	×
Light bulb attack	✓	✓	×
Speaker attack	✓	×	×
Camera attack	✓	✓	×
Siren attack	✓	×	×
Deauthentication + sprinkler cont. attack	N/A	N/A	×
Deauthentication + light bulb attack	N/A	N/A	×
Deauthentication + speaker attack	N/A	✓	×
Deauthentication + camera attack	N/A	N/A	×
Deauthentication + siren attack	N/A	✓	×
✓= successful attack	× = thwarted attack		

trusions/anomalies and sound an alarm. Our test bed uses an Amcrest camera, three Samsung SmartThings sensors (*i.e.*, motion, water-leak, and multi-purpose sensors), a Kwikset door lock, and a D-Link siren as the alarm. Sensor and door lock drivers communicate with the three sensors and the door lock through the Zigbee gateway. Finally, an Android app implements a UI through the secure cloud (§IV).

B. Comparisons

We next compare Vigilia with existing commercial (Norton Core [16] and Bitdefender BOX 2 [15]) and research systems (HanGuard [10] and IoTSec [62]).

Attacks: We designed a set of direct attacks, under our threat model (Section II), against our smart home devices. The sprinkler, speaker, camera, and siren communicate through port 80 using the HTTP protocol. The speaker also uses other ports as it communicates using the AirPlay protocol [34]. The sprinkler particularly has a known vulnerability that can be exploited through a non-documented and non-secured RESTful API [6].

The light bulb communicates through port 56700, through which all LIFX bulbs listen [39]. The deauthentication attack is a more sophisticated attack that we use in combination with the first five attacks that directly target the devices. This attack deauthenticates a device, and makes it leave its router to join a malicious router that has the same SSID and PSK. When the device joins the other router, we can forcefully launch a direct attack to the device. Table III summarizes all of them.

For every system that we evaluated, we connected the smart home devices to the system and we performed the direct attacks. When a direct attack failed, we performed a combination attack. We first deauthenticated the device, let it join the malicious router that we have prepared, and performed the direct attack. Table IV summarizes the results. We also performed the attacks on a normal router to establish a baseline. The normal router does not have any of the security properties that the Vigilia router has.

SmartThings: We implemented several previously known attacks against the SmartThings hub. In our first attack, we modified a device handler to subscribe to all LAN traffic. When we installed this device handler, there was no notification that it might access all SSDP network communications. We then ran the handler and could observe all SSDP traffic to the hub

We next modified the service manager component of the Wemo Switch driver to change the IP address and port of a device after installation. This allowed us to control arbitrary devices on the LAN. Since third party drivers are commonly used to control smart home devices under SmartThings (*e.g.*, the only driver for Google Nest is a third party driver written by a hobbyist), this is a significant threat. This hack can be used to communicate with any device on the LAN.

We then implemented the same type of attack on Zigbee drivers and have discovered that Zigbee drivers can contact arbitrary Zigbee devices and send arbitrary Zigbee commands [20].

None of these attacks are possible under Vigilia. Vigilia blocks all network traffic by default and thus components can only access network traffic that they have been explicitly configured to access and that was explicitly intended for the component. Drivers under Vigilia are subject to the fine-grained access controls for both the TCP/IP and Zigbee networks and thus can only access the devices they were explicitly configured for. Moreover, our Zigbee framework prevents issuing commands that would cause a Zigbee device to interfere with other Zigbee devices.

Finally, as part of our general attacks reported later in this section, we sent commands directly to smart home devices. SmartThings does not block any such attacks. Vigilia blocks all such attacks.

Commercial Systems: We selected Norton Core and Bitdefender BOX 2, which are two leading secure routers that protect smart home IoT devices [16], [50], [15], [51]. They both use machine learning to learn the normal behavior of smart home devices. Their system compares device behavior against their database that contains information about vulnerabilities, attacks, viruses, malicious activities, etc., and warns users when it detects anomalies.

We first connected our devices to Norton Core and Bitdefender BOX 2. Next, we performed direct attacks against the smart home devices. The attacks were successful and thus we categorize these systems under the normal router category in our results.

Further inspection revealed that these systems operate under a different threat model—they only defend against attacks that come from outside. A device inside the local network is considered safe and trusted—it is allowed to generate any traffic to any of the other local devices. Hence, they do not defend against our attacks that come from compromised local devices.

Research Systems: For research systems, we evaluated HanGuard [10] and IoTSec [62]. To the best of our knowledge, these systems are the closest to Vigilia in terms of the threat model.

HanGuard uses SDN-like techniques to learn the normal traffic between smartphone apps and their respective smart home devices. A *Monitor* app runs on the phone to identify any attacks and inform the router through the system's *control plane*. The router then enforces policies in the *data plane* after verifying the party that attempts to access the device. Unfortunately, we could not obtain the implementation of HanGuard. Thus, we could not compare HanGuard with Vigilia. However, the paper [10] implies that HanGuard would leave IoT devices vulnerable to the combination attacks that can be thwarted by Vigilia.

IoTSec has two phases: *profiling* and *deployment*. During profiling, it attempts to learn the normal traffic of devices, *e.g.*, legitimate source and destination IP addresses, port numbers, protocols, etc. Then, a set of firewall rules will be generated and can be deployed on the router. Similarly to Vigilia, IoTSec reduces the attack surface with firewall while trying to maintain full functionality of devices.

To evaluate IoTSec, we connected our devices to a router running the IoTSec profiler. We then executed the four Vigilia applications, but turned off Vigilia's firewall protection. The IoTSec profiler learned the normal traffic of the four applications and generated a set of firewall rules for all devices. We deployed the firewall rules on the router and restarted the applications.

A key weakness of IoTSec is that it relies entirely on profiling. For most of our devices, this approach worked because they always use the same IP address, port numbers, and protocols. However, the iHome speaker randomly selects a port number and the generated firewall rules disrupted the speaker's operation—these rules assume devices always use the same port numbers. In addition, profiling may not exhibit all behaviors of a system. For example, during profiling, we did not trigger the siren to let it go off—deliberately triggering the home alarm to enable the home security system is not a normal behavior. The profiler did not learn the siren's traffic and thus the generated firewall rules disabled the siren.

We performed direct attacks on the devices. The attacks against the sprinkler, light bulb, and camera were successful because the generated firewall rules allowed them to communicate through their respective port numbers. During profiling, IoTSec does not learn the source IP addresses—it assumes that devices are allowed to communicate through their respective ports regardless of the source IP addresses. Hence, the firewall rules are not fine-grained enough to block communications from illegal sources.

The attacks against the speaker and siren failed because the incomplete firewall rules meant that they did not function at all. We then performed the deauthentication attack to both devices. After they joined our malicious router, we successfully attacked them.

Vigilia: We performed the same attacks against the devices under Vigilia. We connected every device using a unique PSK to Vigilia's router. We ran the four applications simultaneously and attacked them.

Under the protection of Vigilia's firewall and sandboxing mechanisms, all of the applications and devices were fully functional, and all of the attacks were successfully thwarted. The direct device attacks were blocked by the deployed firewall rules on the router and the compute nodes. The deauthentication attack also failed as none of the devices could join the malicious router. Even though the malicious router was configured with the same SSID and PSK as the Vigilia router, the devices did not use the router's default PSK—every device was connected to the Vigilia router using a unique PSK.

Table V
STATISTICS OF ACCESS ATTEMPTS FOR THE PUBLIC IP EXPERIMENT; 'A' IS A PLACEHOLDER FOR `128.200.150` AND 'B' IS FOR `calplug.uci.edu`; COLUMN **DS** REPORTS THE NUMBER OF DISTINCT SOURCES; **TCP** AND **UDP** REPORTS NUMBERS IN THE FORM OF X/Y WHERE X AND Y REPRESENT THE NUMBERS OF TOTAL AND DISTINCT ADDRESSES, RESPECTIVELY.

IP	Domain	Total	DS	TCP	UDP	ICMP
A.130	iot1.B	2,944	1,411	1,992 / 340	334 / 60	218
A.131	iot2.B	2,791	1,451	2,039 / 343	256 / 84	69
A.132	iot3.B	3,255	1,405	1,947 / 350	203 / 62	693
A.133	iot4.B	2,841	1,364	1,934 / 344	219 / 73	271
A.134	iot5.B	2,769	1,422	2,043 / 349	233 / 62	82
A.135	iot6.B	2,792	1,416	2,024 / 353	281 / 65	69
A.136	iot7.B	3,284	1,443	2,106 / 342	276 / 64	496
A.137	iot8.B	3,006	1,507	2,084 / 316	272 / 88	246
A.138	iot9.B	3,000	1,433	2,028 / 316	353 / 72	231
A.139	iot10.B	2,620	1,370	1,862 / 283	244 / 62	169
A.140	iot11.B	2,692	1,419	1,983 / 316	258 / 69	66
A.141	iot12.B	2,709	1,429	2,018 / 267	262 / 69	93
A.142	iot13.B	3,582	1,397	2,042 / 352	287 / 63	838
A.143	iot14.B	2	2	0 / 0	2 / 2	0
A.144	iot15.B	3	2	0 / 0	3 / 2	0
A.145	iot16.B	6	2	0 / 0	6 / 1	0
	Total	**38,296**				

Table VI
STATISTICS OF PUBLIC IP EXPERIMENT ON CAMERAS; 'A' IS FOR `128.200.150`; **ATT, SRC, PKT** REPRESENT THE NUMBER OF ACCESS ATTEMPTS, SOURCES, AND NETWORK PACKETS, RESPECTIVELY; **U/T** STANDS FOR UDP/TCP.

IP	With Vigilia (Att / Src / Pkt)	Ports (U/T)	With pwd only (Att / Src / Pkt)	Ports (U/T)
A.134	106 / 96 / 114	6 / 23	5,337 / 117 / 9,658	39 / 48
A.135	111 / 100 / 115	7 / 23	20,172 / 124 / 40,998	47 / 46
A.136	206 / 97 / 208	6 / 22	1,201 / 98 / 2,039	19 / 43
A.137	128 / 109 / 135	7 / 21	4,520 / 119 / 8,889	17 / 51

C. Public IP

To further evaluate Vigilia, we conducted another experiment, in which we assigned public IP addresses to our devices. While other secure routers generally claim to protect smart home IoT devices when they are connected to a local network behind *Network Address Translation* (NAT), we let our devices be *exposed to the open Internet*. For this experiment, we assigned a public IP address for

every device, ran the four applications, and let Vigilia set up firewall rules on the router. We ran this experiment for approximately 10 days.

Table V summarizes the results of the experiment. The table reports, for a device, the IP address, its domain name, the total number of access attempts for this device, the number of distinct sources these attempts came from, the number of total and distinct TCP attempts, the number of total and distinct UDP attempts, as well as the number of ICMP packets. The 16 public IP addresses generated 38,296 access attempts—approximately 3,629 access attempts per day and 240 access attempts per day per device.

All the attempts were thwarted by the Vigilia firewall rules set up on the router. No device responded to any of the sources, except for the ICMP packets. The network trace suggests that most of the access attempts were either *ICMP ping* or *TCP SYN/ACK port scanning* [70], which are the two approaches attackers commonly use to "test the waters". Since our devices only replied to ICMP pings, there were no further packets from more sophisticated attacks.

Real Attacks on Cameras: We conducted an additional experiment with our Amcrest cameras and exposed them to real attacks. This experiment was done under three scenarios: 1) cameras were protected under Vigilia, 2) cameras were protected with passwords, and 3) cameras were unprotected. Each scenario lasted for 14 hours.

Table VI summarizes the results of the experiment for the first two scenarios. In the first scenario, the first camera with address `128.200.150.134` received 106 access attempts from 96 distinct sources with 114 packets of total traffic under Vigilia's protection—the attempts targeted 6 distinct UDP ports and 23 distinct TCP ports, and were all thwarted. In the second scenario, the same camera received many more access attempts. Although the camera had not been compromised, it could have been had we extended the duration. This is especially the case when people use generic/default passwords for their cameras, as shown in a study on the Mirai botnet attack [3]—*there was even a ... Mirai infection on Amcrest cameras despite strong passwords* [24].

In the third scenario, it took *just 15 minutes*, for *all of the four cameras* to be hacked and crippled—the user interface was completely broken although it was still able to stream out video. Each attack session for each camera just took around 172 - 362 packet exchanges between each camera and the attacker. The network trace in the log file suggests that the attackers used a technique called *XML-RPC attack* [59], which typically brings down web services by executing *remote procedure call* (RPC) commands via the HTTP protocol.

D. Performance Microbenchmarks

Vigilia's primary enforcement is implemented by firewall rules. The other components are not on the hot paths and should add minimal overhead. This subsection evaluated the overhead of Vigilia's routing policies on network throughput. We measured the network bandwidth under three different router configurations: normal mode, hairpin mode, and hairpin mode with policies. We performed each of these measurements under two different setups: (1) a node-to-node bandwidth measurement using the Apache HTTP server on a Raspberry Pi 2 and (2) a node-to-LAN bandwidth measurement using the Apache HTTP server on an Intel Core i7-3770 CPU 3.40GHz machine running Ubuntu. We ran `wget` on another Raspberry Pi 2 to retrieve a 30 MB file from both the Raspberry Pi 2 and the Ubuntu machine. All equipment was placed in a Faraday cage to limit interference. We report average bandwidth over 20 runs.

Table VII
VIGILIA MICROBENCHMARK RESULTS.

	Node-to-Node	Overhead	Node-to-LAN	Overhead
Normal	2.91 MB/s	N/A	5.64 MB/s	N/A
Hairpin	2.78 MB/s	4.5%	5.62 MB/s	0.3%
Hairpin + Policies	2.75 MB/s	5.5%	5.62 MB/s	0.3%

Table VII reports the average bandwidths. Under the node-to-node scenario, hairpin mode introduces a 4.5% overhead since it forces traffic to exit the driver and go through the kernel firewall. Under the node-to-LAN scenario, the lower overhead is not surprising as node-to-LAN traffic already exits the driver before going through the firewall. The firewall policies introduce almost negligible overhead for both setups. Node-to-node results show lower bandwidths as communication must take two hops on the same WiFi channel. Overall, the overheads are relatively small.

VIII. DISCUSSION

Vigilia Techniques: The techniques used in Vigilia, *i.e.*, static checking, router policy enforcement, process sandboxing, and capability-based RMI (§V) along with WiFi network filtering and Zigbee firewall (§VI) could also be deployed in existing systems, *e.g.*, SmartThings. The major issue with directly implementing our approach on SmartThings is that the SmartApps run on their cloud servers and none of the source code for their software infrastructure for running SmartApps is available. Nevertheless, assuming that we had access to their software infrastructure and extended it to execute applications and device drivers on the local network, it would be straightforward to deploy the static checking, router policy enforcement, process sandboxing, and capability-based RMI. The techniques to secure the WiFi network against snooping, ARP-spoofing, and MAC-spoofing could be applied directly to the router, while the Zigbee firewall could also be integrated fairly easily into the smart hub.

IX. Related Work

Network-based policy checking is not a new idea and it has been studied in the *software-defined networking* (SDN) community [26], [8], [7]. For example, Ethane [7] requires each application to specify a manifest of its required communication and then checks packets against security rules and installs forwarding rules as required. While Ethane is applicable in our setting, it is designed primarily for enterprise networks that have a large number of switches; it requires a sophisticated controller that performs authentication, registration, and checking. It also requires expert administrators to develop routing policies—a task that is beyond the abilities of most end users. Moreover, IoT devices typically communicate via WiFi and often do not support the enterprise security modes. Thus, malicious devices can masquerade as other devices to bypass the SDN protections.

IoT Security: Denning et al. [11] identified emergent threats to smart homes due to the use of IoT devices. Recent Work [20] discusses scenarios in which hackers can weaken home security through compromising these devices. A study by Ur et al. [66] on the access control of the Philips Hue lighting and the Kwikset door lock found that each system provides a siloed access control system that fails to enable essential use cases.

Many other projects have made a similar observation that IoT devices have highly structured communication patterns. The Bark policy language uses manually created policies [33]. This policy language provides five types, *i.e.*, *who*, *what*, *where*, *when*, and *how* to capture the high level information (*e.g.*, devices, apps, types of service, etc.) needed to construct network level policies. A similar approach uses manually created policies [4], while other approaches propose learning policies [74], [46]. These approaches suffer from similar challenges to IoTSec, in that they can generate overly relaxed policies that allow attacks or overly restrictive policies that break applications. Moreover, compromised devices can easily bypass the policies by masquerading as other devices. For simple IoT control rules of the form used by IFTTT, automated analysis can generate rich policies that only grant permissions under specific criteria (e.g., one can only turn on the heat if it is cold) [22].

Bluetooth devices face similar issues to Zigbee devices regarding access control. Previous work [38] has explored access control for Bluetooth devices. Low-level protocol differences mean that the solution for Bluetooth devices does not solve the problem for Zigbee.

There are two main categories of work in current smart home security research, focused primarily on *devices* and *protocols*, respectively [23], [30], [52]. On protocols, studies found various flaws in the Zigbee and Z-Wave devices [25], [41], [20], [68].

Much work has been done to limit the privilege of networked systems, but this is difficult to achieve due to the lack of programming language and system support. Felt et al. [19] found that more than 300 Android apps were overprivileged. Fernandes et al. [20] found that IoT devices are also overprivileged due to the framework design itself. HomeOS supports smart home devices using a PC-like abstraction [13]. HomeOS only provides support for placing restrictions on modules running on the PC—other devices on the network are free to attack smart home devices. Vigilia provides much stronger security guarantees—it can defend against attacks from other smart home applications, device drivers, and arbitrary devices on the home network. FlowFence [21] provides security by requiring consumers of sensitive to declare their data flow patterns. ContexIoT [36] is a context-based permission system provides contextual integrity by supporting fine-grained context identification for sensitive actions.

Capability-based Object Model: Capability-based object models are used to control object accesses according to a certain set of capabilities. This term was coined in 1959 by Dennis et al. [12]. Miller et al. [47] compared ACL with the capability-based model. In [44], [43], [57], [45], capability-based models have been used in different contexts. Unix-like operating systems, *e.g.*, SELinux [42], have implemented ACL and MAC, which are orthogonal to capability-based object model.

Routing Policy Derivation: Researchers have developed tool kits for managing firewalls. The Firmato [5] toolkit allows administrators to specify rules in terms of a higher-level model. This model is specified by the administrator and thus has no direct relationship to code—errors in specifying the model can either open the system to attacks or block desired communications. It is likely to be unreasonable to expect end users to develop such models for their home networks. There also exists work on information flow systems [48], [49], [64], [18], [54], [75], [76], [40], most of which is orthogonal.

X. Conclusion

We present an approach for building secure systems out of insecure components in Vigilia. Our approach moves the burden of securing the system from the device manufacturers to the platform, reducing concerns about the long-term availability of security patches.

We have implemented 4 applications in Vigilia using commercially available IoT devices. The intended deployment of the IoT devices used by each application had least one vulnerability. Vigilia successfully defended all our applications against all attacks.

XI. Acknowledgment

We would like to thank our anonymous reviewers for their thorough comments and especially our shepherd Klara Nahrstedt, who has helped us improve the paper's readability. We would like to also thank Changwoo Lee, Jiawei Gu, Yuting Tan, and Jiman Jeong, who helped develop the Zigbee device drivers; Hyeongtag Chi, Bowon Ko, Kevin Cong

Truong, and Brian Truong, who helped develop the phone application; Dohyun Kim and Janghoi Koo, who helped with a benchmark application.

This project was partly supported by the National Science Foundation under grants CCF-1319786, CNS-1613023, CNS-1703598, CNS-1763172, OAC-1740210 and by the Office of Naval Research under grants N00014-16-1-2913 and N00014-18-1-2037.

References

[1] Samsung smart fridge leaves Gmail logins open to attack. http://www.theregister.co.uk/2015/08/24/smart_fridge_security_fubar/, August 2015.

[2] Lawn watering tips - best times & schedules. http://www.scotts.com/smg/goART3/Howto/lawn-watering-tips/33800022/12400007/32000006/18800019, April 2016.

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[4] D. Barrera, I. Molloy, and H. Huang. Idiot: Securing the internet of things like it's 1994. *CoRR*, abs/1712.03623, 2017.

[5] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Transactions on Computer Systems*, 22(4):381–420, November 2004.

[6] M. Bergin. Unplugging an IoT device from the cloud. https://blog.korelogic.com/blog/2015/12/11/unplugging_iot_from_the_cloud, December 2015.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, Aug. 2007.

[8] M. Casado, T. Garfinkel, M. Freedman, A. Akella, D. Boneh, N. McKeowon, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proc. Usenix Security Symposium*, August 2006.

[9] A.-M. Chang, F. A. J. L. Scheer, and C. A. Czeisler. The human circadian system adapts to prior photic history. *The Journal of Physiology*, 589(5):1095–1102, March 2011.

[10] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. Gunter, X. Zhou, and M. Grace. Guardian of the HAN: Thwarting mobile attacks on smart-home devices using OS-level situation awareness. https://arxiv.org/abs/1703.01537, 2017.

[11] T. Denning, T. Kohno, and H. M. Levy. Computer security and the modern home. *Commun. ACM*, 56(1):94–103, Jan. 2013.

[12] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, Mar. 1966.

[13] 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*, 2012.

[14] L. DROLEZ. Wanscam JW0004 IP Webcam hacking. http://www.drolez.com/blog/?category=Hardware&post=jw0004-webcam, July 2015.

[15] M. Eddy, V. Song, and J. R. Delaney. Bitdefender box 2. https://www.pcmag.com/review/357433/bitdefender-box-2, November 2017.

[16] M. Eddy, V. Song, and J. R. Delaney. Symantec norton core router. https://www.pcmag.com/review/355417/symantec-norton-core-router, September 2017.

[17] H. P. Enterprise. Internet of things research study: 2015 report. http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA5-4759ENW&cc=us&lc=en, 2015.

[18] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, 2014.

[19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, Chicago, IL, USA, October 2011 (CCS '11)*.

[20] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP), Oakland, CA, USA, May 2016 (Oakload '16)*, pages 636–654, May 2016.

[21] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *USENIX Security*, pages 531–548, 2016.

[22] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action IoT Platforms. In *22nd Network and Distributed Security Symposium (NDSS 2018)*, Feb. 2018.

[23] D. Fisher. Pair of bugs open honeywell home controllers up to easy hacks. https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/, 2015.

[24] A. Forum. Mirai infection. https://amcrest.com/forum/technical-discussion-f3/mirai-infection-t3686.html, October 2017.

[25] B. Fouladi and S. Ghanoun. Honey, i'm home!!, hacking zwave home automation system. In *Black Hat USA*, 2013.

[26] O. N. Foundation. Software-defined networking (sdn) definition. https://www.opennetworking.org/sdn-resources/sdn-definition, 2017.

[27] T. A. S. Foundation. The apache groovy programming language. http://groovy-lang.org/, 2003-2018.

[28] J. J. Gooley, K. Chamberlain, K. A. Smith, S. B. S. Khalsa, S. M. W. Rajaratnam, E. V. Reen, J. M. Zeitzer, C. A. Czeisler, and S. W. Lockley. Exposure to room light before bedtime suppresses melatonin onset and shortens melatonin duration in humans. *Journal of Clinical Endocrinology & Metabolism*, 96(3):E463–E472, March 2011.

[29] J. Hartin, P. M. Geisel, and C. L. Unruh. Lawn watering guide for california. Technical Report ANR 8044, University of California – Agriculture and Natural Resources, http://anrcatalog.ucanr.edu/pdf/8044.pdf, 2001.

[30] A. Hesseldahl. A hackers-eye view of the internet of things. http://recode.net/2015/04/07/a-hackers-eye-view-of-the-internet-of-things/, 2015.

[31] K. Hill. The half-baked security of our 'Internet Of Things'. http://www.forbes.com/sites/kashmirhill/2014/05/27/article-may-scare-you-away-from-internet-of-things/.

[32] D. C. Holzman. What's in a color? the unique human health effects of blue light. *Environmental Health Perspectives*, 118(1):A22–A27, January 2010.

[33] J. Hong, A. Levy, L. Riliskis, and P. Levis. Don't talk unless i say so! securing the internet of things with default-off networking. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 117–128, April 2018.

[34] A. Inc. Airplay. https://developer.apple.com/airplay/, 2018.

[35] IOActive. Belkin WeMo home automation vulnerabilities. http://www.ioactive.com/pdfs/IOActive_Belkin-advisory-lite.pdf, 2014.

[36] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *NDSS*, 2017.

[37] Security of the Local LAN? https://community.smartthings.com/t/security-on-the-local-lan/41585, May 2018.

[38] A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein. Beetle: Flexible Communication for Bluetooth Low Energy. In *Proceedings of the 14th International Conference on Mobile Systems, Applications and Services (MobiSys)*, June 2016.

[39] LIFX. Device messages. https://lan.developer.lifx.com/docs/device-messages, 2018.

[40] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM 2009 Symposium on Operating Systems Principles and Implementation*, 2009.

[41] N. Lomas. Critical flaw identified in zigbee smart home devices. http://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/, 2015.

[42] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[43] T. Luo and W. Du. *Contego: Capability-Based Access Control for Web Browsers*, pages 231–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[44] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 125–140, 2010.

[45] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium*. Internet Society, 2010.

[46] M. Miettinen, S. Marchal, I. Hafeez, T. Frassetto, N. Asokan, A. R. Sadeghi, and S. Tarkoma. IoT Sentinel: Automated device-type identification for security enforcement in IoT. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017.

[47] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished, 2003.

[48] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[49] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedfings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[50] B. Nadel. Norton core router review. https://www.tomsguide.com/us/norton-core-router,review-4827.html, November 2017.

[51] B. Nadel. Bitdefender box (2018) review: Flexible protection. https://www.tomsguide.com/us/bitdefender-box,review-3766.html, January 2018.

[52] T. Oluwafemi, T. Kohno, S. Gupta, and S. Patel. Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security. In *Proceedings of the LASER 2013, Arlington, VA, USA (LASER 2013)*, pages 13–24. USENIX, 2013.

[53] OpenWrt. https://openwrt.org.

[54] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[55] Proxy arp. http://www.cisco.com/c/en/us/support/docs/ip/dynamic-address-allocation-resolution/13718-5.html, January 2008.

[56] Rapid. HACKING IoT: A case study on baby monitor exposures and vulnerabilities. https://www.rapid7.com/docs/Hacking-IoT-A-Case-Study-on-Baby-Monitor-Exposures-and-Vulnerabilities.pdf, September 2015.

[57] S. Saghafi, K. Fisler, and S. Krishnamurthi. Features and object capabilities: Reconciling two visions of modularity. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 25–34, New York, NY, USA, 2012. ACM.

[58] Samsung SmartCam. https://www.exploitee.rs/index.php/Samsung_SmartCam%E2%80%8B#Fixing_Password_Reset_.22Pre-Auth.22, August 2014.

[59] J. Schwenn. How to protect wordpress from xml-rpc attacks on ubuntu 14.04. https://www.digitalocean.com/community/tutorials/how-to-protect-wordpress-from-xml-rpc-attacks-on-ubuntu-14-04, February 2016.

[60] S. Shekyan and A. Harutyunyan. To watch or to be watched: Turning your surveillance camera against you. https://conference.hitb.org/hitbsecconf2013ams/materials/D2T1%20-%20Sergey%20Shekyan%20and%20Artem%20Harutyunyan%20-%20Turning%20Your%20Surveillance%20Camera%20Against%20You.pdf.

[61] S. SmartThings. Samsung smartthings website. http://www.smartthings.com, 2018.

[62] D. A. Sorensen, N. Vanggaard, and J. M. Pedersen. IoTsec: Automatic profile-based firewall for IoT devices. http://projekter.aau.dk/projekter/files/260081086/report_print_friendly.pdf, June 2017.

[63] Spruce - the smart irrigation controller. http://www.spruceirrigation.com, April 2016.

[64] A. J. Summers and P. Muller. Freedom before commitment-a lightweight type system for object initialisation. In *Proceedings of the 2011 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2011.

[65] Tomoyo linux. https://tomoyo.osdn.jp/index.html.en, April 2017.

[66] B. Ur, J. Jung, and S. Schechter. The current state of access control for smart devices in homes. In *Proceedings of Workshop on Home Usable Privacy and Security, Newcastle, UK, July 2013 (HUPS)*, 2013.

[67] S. H. USA. What is a smart home? https://www.smarthomeusa.com/smarthome/, 2018.

[68] Veracode. The internet of things: Security research study. https://www.veracode.com/sites/default/files/Resources/Whitepapers/internet-of-things-whitepaper.pdf, 2015.

[69] G. Wassermann. ZyXEL NBG-418N, PMG5318-B20A and P-660HW-T1 routers contain multiple vulnerabilities. http://www.kb.cert.org/vuls/id/870744, October 2015.

[70] A. Whitaker and D. Newman. Penetration testing and network defense: Performing host reconnaissance. http://www.ciscopress.com/articles/article.asp?p=469623&seqNum=3, June 2018.

[71] Z. Whittaker. Hackers exploiting 'serious' flaw in Netgear routers. http://www.zdnet.com/article/hackers-exploiting-serious-flaw-in-netgear-routers/, October 2015.

[72] Wink hub. https://www.exploitee.rs/index.php/Wink_Hub%E2%80%8B%E2%80%8B#Wink_Hub_.22.2Fvar.2Fwww.2Fdev_detail.php.22_SQLi_for_root_command_execution.

[73] K. York. Dyn statement on 10/21/2016 ddos attack. http://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/, October 2016.

[74] 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 (HotNets)*, 2015.

[75] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[76] 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*, 2008.