A Two-Mode, Adaptive Security Framework for Smart Home Security Applications

DEVKISHEN SISODIA, University of Oregon, USA
JUN LI, University of Oregon, USA
SAMUEL MERGENDAHL, University of Oregon, USA
HASAN CAM, Best Buy, USA

With the growth of the Internet of Things (IoT), the number of cyber attacks on the Internet is on the rise. However, the resource-constrained nature of IoT devices and their networks makes many classical security systems ineffective or inapplicable. We introduce TWINKLE, a two-mode, adaptive security framework that allows an IoT network to be in regular mode for most of the time, which incurs a low resource consumption rate, and to switch to vigilant mode only when suspicious behavior is detected, which potentially incurs a higher overhead. Compared to the early version of this work, this paper presents a more comprehensive design and architecture of TWINKLE, describes challenges and details in implementing TWINKLE, and reports evaluations of TWINKLE based on real-world IoT testbeds with more metrics. We show the efficacy of TWINKLE in two case studies where we examine two existing intrusion detection and prevention systems and transform both into new, improved systems using TWINKLE. Our evaluations show that TWINKLE is not only effective at securing resource-constrained IoT networks, but can also successfully detect and prevent attacks with a significantly lower overhead and detection latency than existing solutions.

Additional Key Words and Phrases: Internet of Things, smart home, security, two-mode security framework

1 INTRODUCTION

The Internet of Things (IoT) continues to pervade our lives. However, as IoT devices are connected by the Internet, they also suffer from the same types of attacks that plague traditional Internet-connected machines. In October 2016, for example, the Mirai IoT botnet, which comprised of up to 100,000 infected IoT devices, launched multiple large-scale distributed denial-of-service (DDoS) attacks [16] all across the Internet. The landscape of IoT security is growing darker and more precarious as new malware strains are being developed and deployed to exploit the many vulnerabilities of IoT devices.

While IoT devices and traditional machines often suffer from the same types of attacks, IoT devices tend to be harder to secure due to some unique properties. They tend to have scarce CPU and memory resources, lower network bandwidth, and limited battery capacity if not plugged into an external power source. These properties, which differentiate IoT devices from traditional machines, severely hinder the deployment of existing security mechanisms in IoT environments.

Cryptographic protocols and intrusion detection/prevention systems (IDSes/IPSes), developed for the traditional Internet, are designed without the assumption of extremely limited resource and computing power. Even systems

Authors' addresses: Devkishen Sisodia, University of Oregon, Eugene, OR, USA, dsisodia@uoregon.edu; Jun Li, University of Oregon, Eugene, OR, USA, lijun@cs.uoregon.edu; Samuel Mergendahl, University of Oregon, Eugene, OR, USA, smergend@cs.uoregon.edu; Hasan Cam, Best Buy, Richfield, MN, USA, hasan.cam3@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2577-6207/2023/11-ART $15.00
https://doi.org/10.1145/3617504

Just Accepted

that are considered extremely lightweight cannot be installed on memory-constrained devices that have less than 1 MB of available memory [38]. For example, Sehgal et. al. [32] show that many IoT devices struggle to run the cryptographic protocol TLS, a traditional Internet security standard. If a security solution needs to probe devices they protect, most devices in an IoT environment may either lack the power or network bandwidth to respond to every probe, or simply wish to stay dormant most of the time. Sometimes a security solution may impose some minor penalties on benign devices while mitigating an attack (e.g., dropping benign traffic from devices to mitigate a DDoS attack). These minor penalties, when moved to an IoT environment, can become a significant hindrance to those benign devices.

In this paper, we focus on the smart home environment where security and privacy are especially important and address the ineffectiveness of classical security mechanisms in the smart home. We introduce a security framework called **TWINKLE**, **TW**o-mode **IN**-home framewor**K** toward **L**ightweight S**E**curity, that supports individual security applications that handle specific attacks in the smart home. By enabling each security application to run in two distinct modes, TWINKLE not only preserves the salient features of classic security solutions, but also addresses the resource limitations that IoT devices face. While plugged into TWINKLE, every security application runs in regular mode for most of the time and incurs a minimal amount of resource consumption, but when it detects any suspicious behavior that an attack must display, it can readily switch to vigilant mode and engage in sophisticated routines for a short time window during which to cope with the suspicious behavior with strong competence. By only running the heavyweight routines when needed, TWINKLE saves precious resources over applications that run these routines either continuously or periodically.

We further implement TWINKLE by addressing key implementation challenges and use it to transform two prior security solutions for the smart home environment. Our evaluation demonstrates that the transformed solutions incur much less overhead than the prior solutions, while achieving equal to better efficacy in detecting and mitigating the attacks.

The work presented in this paper builds upon the TWINKLE framework first introduced by Sisodia et. al. [35], with several new significant contributions. First, the design and architecture of TWINKLE is more comprehensive. We elaborate on the philosophy behind the two-mode design and describe how TWINKLE components interact with each other to form the TWINKLE framework. Furthermore, we discuss the design requirements for TWINKLE security applications, which affect how each TWINKLE component operates. Second, we describe the implementation of TWINKLE in detail. Specifically, we describe how security applications must be represented, explain how each component can be instantiated automatically, and expound on how components interact with each other after instantiation. Finally, we update each case study and enhanced the evaluation of TWINKLE, including using real-world IoT testbeds and introducing more metrics.

2 TWINKLE: DESIGN AND ARCHITECTURE

Many security solutions developed for the traditional Internet, if deployed in an IoT environment such as a smart home, may cause a substantial burden on some IoT devices due to their computing power, resource, and energy requirements. We design the TWINKLE framework in such a way as to not only preserve the salient features of classic security solutions, but also address the resource limitations that IoT devices face. In this section we describe its design, architecture, and how it supports security applications running in a smart home.

2.1 Design with Two Modes for IoT-based Security Applications

A smart home requires many types of security applications. It may face various malicious attacks such as an eavesdropping attack that can spy on the traffic between the smart home devices, a sinkhole attack that can misdirect traffic of devices to a sinkhole, a wormhole attack that can reroute data from the smart home to an attacker outside, or an attack that compromises devices at the smart home and turns them into nodes of a botnet.

Just Accepted

Worse, a smart home may also initiate attacks, such as launching a distributed denial-of-service (DDoS) attack or a phishing campaign through compromised devices at home. The TWINKLE framework thus aims to support various security applications for the smart home, where every security application handles a specific type(s) of attack. We also call these security applications **TWINKLE security applications**. For every TWINKLE security application, the network administrator can plug it into the TWINKLE framework when needed.

The central dilemma facing these security applications is that they must address the inadequacy of computing power and resources available to smart home devices without compromising their own efficacy. A security application may demand resources from a device, such as CPU utilization, memory consumption, power consumption, and bandwidth consumption, such that it is impossible for the device to meet that demand without sacrificing performance of the security application itself, or other applications running on the device.

To address this dilemma, we design the TWINKLE framework that supports security applications to run on top of TWINKLE and operate in two distinct modes: *regular mode* for most of the time which has a low resource consumption rate and *vigilant mode* that potentially incurs a high overhead but is infrequent. In regular mode, a TWINKLE security application invokes functions to detect suspicious behavior that an attack, if occurring, *must* display, whereas those functions must also be lightweight. Note, a legitimate operation may also display a suspicious behavior. Once it detects a suspicious behavior (i.e., an attack *may* be occurring), the security application enters vigilant mode to closely inspect whether an attack is *indeed* occurring and if so, conduct other security operations such as sending an alert of the attack, mitigating the attack, or recovering from the attack. After the attack is handled or the smart home is no longer under this attack, the security application goes back to regular mode.

This two-mode design differs from many classical security applications, which usually run in one mode. Specifically, a classical application usually runs continuously or periodically to monitor security-related events and must minimize both false positives and false negatives. It often employs complicated operations in order to be accurate in detecting attacks, thus consuming resources frequently and heavily. Conversely, a TWINKLE security application, by switching between these two modes but staying in regular mode most of the time, does not consume as much resources as classical applications. In regular mode, it is only concerned about detecting with high sensitivity the suspicious behaviors that attacks in question must demonstrate, even if legitimate operations may also demonstrate such behaviors. In other words, in regular mode it is more concerned with minimizing false negatives but less concerned with minimizing false positives. While every security application defines and handles suspicious behaviors of a different type, nature, or severity, this design choice simplifies the detection of suspicious behaviors for every security application, as they all can rely on vigilant mode to further check if a suspicious behavior is indeed from an attack. By transitioning into vigilant mode for a short period only when needed, the security application can engage in sophisticated operations, including those that may be resource-consuming, to detect or handle an attack in question. Since regular mode is usually less resource-consuming than a classical one-mode system, and by only invoking the resource-consuming vigilant mode infrequently and on demand, a TWINKLE security application overall incurs much less resource consumption than classical security applications, as we quantify in Section 6 for two specific case studies.

This principle difference between the two-mode and one-mode design paradigms can be modeled mathematically. Let us first define the resource consumption of the two-mode paradigm by taking energy consumption as an example. Let $R(t) = \sum_{d \in D}(P_{rd} * t)$ represent the total energy consumption consumed (in Joules) by the network within time period t (in seconds), where d is a device in the set of all devices D and P_{rd} is the average power used per unit (in watts) in regular mode for device d. Similarly, let $V(t) = \sum_{d \in D}(P_{vd} * t)$ represent the total energy consumed by the network at time t (note, P_{vd} is the average power used per unit in *vigilant mode* for device d). Again, P_{rd} and P_{vd} are used here because we are taking energy consumption as an example, but these variables can be substituted for any other resource (e.g., memory consumption). A key point to note is that

Just Accepted

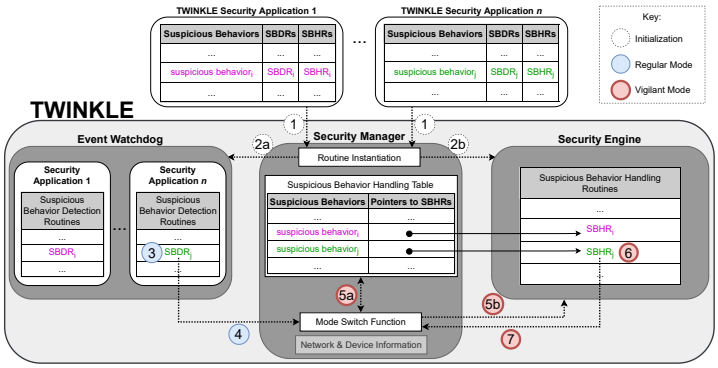


Fig. 1. The basic architecture of TWINKLE.

$P_{rd} < P_{vd}$, or the amount of power a device utilizes in vigilant mode will be greater than the amount of power it utilizes in regular mode due to it running more intensive algorithms in vigilant mode.

In the two-mode paradigm, the network will oscillate between regular and vigilant mode. Therefore, let $R(t_{r1}) + V(t_{v1}) + R(t_{r2}) + V(t_{v2}) + ... + R(t_{rn}) + V(t_{vn})$ represent the total energy consumption of a network in a two-mode design paradigm, or $E_{two-mode}$, where t_{rx} represents the amount of time spent in regular mode the x^{th} time it switched to regular mode (the same applies to t_{vx}, but for vigilant mode), and n is the last oscillation period. This equation can be simplified to $E_{two-mode} = \sum_{i=1}^{n}(R(t_{ri}) + V(t_{vi}))$. Another key point to note is that we assume in most networks $\sum_{i=1}^{n} t_{ri} >> \sum_{i=1}^{n} t_{vi}$, or the amount of total time a network spends in regular mode will greatly outweigh the amount of total time it spends in vigilant mode.

In the one-mode design paradigm (assuming the security applications run continuously), the energy consumption of a network can be represented simply as $E_{one-mode} = \sum_{d \in D}(P_{xd} * t_x)$, where P_{xd} is the average power used per unit and t_x is the total time period (note, for comparison purposes, $t_x = \sum_{i=1}^{n}(t_{ri} + t_{vi})$). The final key point to note is that $P_{rd} << P_{xd} \leq P_{vd}$, or the power a device utilizes when running in one-mode continuously will 1) far outweigh the amount of power it would utilize in regular mode in the two-mode paradigm, and furthermore, 2) either equal or slightly fall short of the amount of power it would utilize in vigilant mode. Therefore, we can confidently conclude that $E_{two-mode} < E_{one-mode}$.

2.2 Architecture of TWINKLE

As shown in Figure 1, TWINKLE is composed of three main components: **security manager**, **event watchdog**, and **security engine**. The security manager is TWINKLE's central component and acts like a security operating system. It is responsible for supporting various security applications, maintaining information about the smart home, instantiating the security functions at the event watchdog and the security engine according to security applications in place, and switching between regular and vigilant mode. The event watchdog is responsible for detecting suspicious behaviors. The security engine is responsible for further handling those suspicious behaviors, such as verifying whether a suspicious behavior is indeed from an attack or not.

In general, the security manager and security engine are deployed at a central node, such as the border router, and an event watchdog can consist of more than one instance by running at multiple devices that can detect suspicious behaviors, such as a smart watch running a monitoring routine, a Raspberry Pi devoted to eavesdropping and monitoring IoT traffic, or any selected devices in the smart home. Certain security applications may instead require a single event watchdog to be installed at the network's border router, as we show in our first case study (Section 4).

Responding to the need of supporting various different security applications for the smart home, the network administrator can plug any security application into TWINKLE as needed (**Step 1** in Figure 1). In doing so, the

Just Accepted

security manager populates data structures and instantiates routines that run on the event watchdog and security engine, all according to the security application in question. On the other hand, in order to be supported by TWINKLE, a security application that handles an attack must define the suspicious behaviors that may be a sign of the attack in question. And for every suspicious behavior, the security application must define the routine that *detects* the suspicious behavior, which we call a **suspicious behavior detection routine (SBDR)**, and the routine that *handles* the suspicious behavior, which we call a **suspicious behavior handling routine (SBHR)**. A general principle here is that an SBDR should be lightweight while an SBHR may be more resource-consuming since, as we explain below, the SBDR runs in regular mode and the SBHR runs in vigilant mode.

After a security application is plugged into TWINKLE, the security manager's **Routine Instantiation** module instantiates the event watchdog by having the event watchdog monitor the set of suspicious behaviors defined by the security application (**Step 2a**). Specifically, the event watchdog begins running the lightweight SBDRs defined by the security application.

TWINKLE provides a dynamic mechanism for a security application to install its event watchdog or security engine at any device needed. At start-up, lightweight processes that can receive, consume, and send messages run at each device that may be a candidate for running an event watchdog or security engine of a security application. When the network administrator deploys a new security application on TWINKLE, and needs to run the event watchdog or security engine code of the application on a device, the security manager can communicate with the device to ship, install, and eventually run the code on the device. Note, depending on the security application, an event watchdog may perform signature-based detection, behavior-based detection, or a combination of both. As described in the previous subsection, the event watchdog ensures that detection is lightweight by not concerning itself with minimizing false positives. When tuned to be lightweight, off-the-shelf intrusion detection systems (IDSes), such as Snort [1], Suricata [2], and Zeek [27], could be used as a basis for the event watchdog.

Next, to instantiate the security engine, the Routine Instantiation module injects the SBHRs defined by the security application into the security engine. Furthermore, the Routine Instantiation module ensures that every defined suspicious behavior is mapped to an SBHR (**Step 2b**). It does so by populating a data structure called the **suspicious behavior handling table (SBHT)**. For each suspicious behavior, the SBHT points to a specific SBHR at the security engine for handling that behavior. While, in general, the security engine runs at a central node, such as a border router, some security applications may require the security engine to run at multiple locations in the network. For example, traffic from a malicious device may need to be dropped before it reaches the border router, and therefore the SBHR that is responsible for dropping malicious traffic should be installed in-network between the malicious device and border router.

Note, security applications may define parameters for various SBDRs and SBHRs. Furthermore, security applications may also require data to be tracked which may be needed for certain SBDRs and SBHRs. Parameters (i.e., threshold values that define various suspicious behaviors) defined by security applications that are required by SBDRs and SBHRs are transferred to the event watchdog and security engine during routine instantiation. Data required by security applications' SBDRs and SBHRs are stored in the security manager's **Network & Device Information** module.

Once the routines are instantiated, the application begins in regular mode, with the event watchdog running. In order to detect suspicious behaviors, in some cases the SBDRs may need to retrieve information stored in the security manager's Network & Device Information module, such as network topology or routing information (**Step 3**), as we will show in our second case study (Section 5). Once the event watchdog detects a suspicious behavior, it notifies the security manager's **Mode Switch Function (MSF)** to handle the suspicious behavior (**Step 4**). Upon receiving the notification, the security manager switches to vigilant mode. The MSF takes the detected suspicious behavior as input, queries the SBHT (**Step 5a**) to determine which SBHR should be invoked (**Step 5b**), and passes the pointer of that SBHR to the security engine (**Step 5c**). The security engine, in turn, invokes the SBHR in question. Generally, the SBHRs are heavyweight and should only be running in vigilant

Just Accepted

Table 1. Taxonomy of common attacks in smart home environments.

Security Requirements	Attacks	Description
Targeting Confidentiality	Side-Channel	Exploits data leakage vulnerabilities to gain sensitive information.
	Brute-Force	Exploits weak credentials/encryption to gain privileged access to devices or sensitive information.
Targeting Integrity	Data Manipulation	Exploits routing protocol vulnerabilities to launch a man-in-the-middle attack to alter packets between a source and destination.
	Voice-Command Injection	Exploits vulnerabilities in speech-recognition systems to inject malicious or inaudible voice commands.
	Event Spoofing	Exploits lack of integrity checks in applications to propagate seemingly legitimate events to devices causing them to react in some way that benefits the attacker.
Targeting Availability	Flooding	Exploits the openness of a victim to launch a DoS attack by overwhelming it with large amounts of traffic.
	Sinkhole/Selective Forwarding	Exploits routing protocol vulnerabilities to launch a DoS attack by dropping all the packets or only forwarding a subset of packets to the destination.
	Jamming	Exploits how transceivers operate to launch a DoS attack that disrupts data transmission and forces devices to repeatedly retransmit packets.
	Battery-Draining	Exploits routing protocol vulnerabilities to launch a DoS attack by depleting devices' battery power.

mode when invoked on demand. SBHRs, like the SBDRs in regular mode, may need to retrieve (and update) network information stored at the security manager (**Step 6**) in order to successfully handle a suspicious behavior. Once the SBHR finishes its execution, the security engine notifies the security manager, and TWINKLE returns to regular mode.

2.3 TWINKLE Security Applications

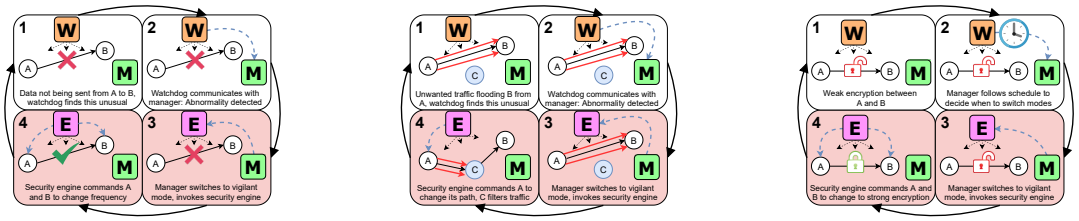
Although TWINKLE security applications are not a part of the TWINKLE framework, they affect how each component in TWINKLE operates. Specifically, the SBDRs and SBHRs, which are critical in detecting and mitigating attacks, are defined by the TWINKLE security applications plugged into TWINKLE. A TWINKLE security application must define (1) a set of suspicious behaviors that may constitute the attack, and (2) a set of routines to detect and handle these suspicious behaviors.

Handling an attack requires defining a set of suspicious behaviors that must be present if that attack is occurring. As mentioned before, suspicious behaviors are defined and detected via SBDRs, which run on event watchdog nodes in regular mode. Depending on the security applications, suspicious behaviors can be of varying granularity. An example of a coarse-grained suspicious behavior is all outgoing traffic exceeding 100 MB at any given time (thus necessitating inspecting the entire network as a whole to determine the suspicious behavior). An example of a fine-grained suspicious behavior is device X sending more than 100 KB traffic every 10 seconds (thus warranting inspecting a specific device to determine the suspicious behavior). Nonetheless, suspicious behaviors must be measurable and detectable by event watchdog nodes. Some suspicious behaviors can be easily measured, such as the amount of outbound traffic, while others may be more difficult, such as the energy consumption of a particular device.

However, suspicious behaviors, if present, are not sufficient in determining if an attack is actually occurring. Further analysis needs to be performed in order to detect if a suspicious behavior or combination of suspicious behaviors should be considered malicious. If an event watchdog detects a suspicious behavior, it triggers the security manager to enter vigilant mode where the security engine performs detailed inspection to determine if the suspicious behaviors are indeed malicious. In particular, a set of routines or functions must be defined to verify and mitigate an attack. These routines are dependent on the attack and vary across different security applications. For example, for certain suspicious behaviors, a security application may want to send probing signals to gain additional information from devices (SEND-SIGNAL), drop outbound traffic (DROP), change certain paths in the network (CHANGE-PATH), change the encryption protocol between certain devices (CHANGE-ENCRYPTION), or change the channel frequency between certain devices (CHANGE-FREQUENCY).

Security applications may handle any number of attacks. Table 1 shows a taxonomy of common attacks in smart home environments. Note, that while IoT devices fall victim to these common attacks, these attacks can also be carried out by compromised IoT devices themselves. TWINKLE provides a framework for *any* security application to utilize the two-mode paradigm for reducing resource consumption in IoT environments. However,

Just Accepted



(a) Jamming example **(b) Flooding example** **(c) Encryption example**

Fig. 2. Three example TWINKLE security applications.

while any security application can be plugged-into TWINKLE, how effectively the application will utilize the two-mode paradigm depends on how the security application itself is written. To exhibit TWINKLE's versatility in protecting an IoT network, we present three example TWINKLE applications, as shown in Figure 2, that use the TWINKLE framework to address jamming attacks, flooding attacks, and weak encryption (which could lead to brute-force attacks). We choose to focus on these three attacks because it is straightforward to show the two-mode nature of TWINKLE with them.

2.3.1 Example Application to Address Jamming Attacks. In Figure 2a, TWINKLE is being used to handle a jamming attack where the link between devices A and B is being jammed by an unknown attacker. In the first two steps (box 1 and box 2), TWINKLE is running in regular mode. In the first step, an event watchdog node in the network, while running an SBDR defined by the security application, detects suspicious behavior. In this case, device B is not receiving traffic from device A, which is abnormal. The event watchdog can detect this abnormality by eavesdropping on the communication between A and B by setting its network interface controller (NIC) to promiscuous mode. Because the watchdog does not receive any traffic from A to B for a prolonged period of time (e.g., the period of time without receiving any traffic surpasses the threshold for what is considered normal), it notifies the security manager at the border router of the suspicious behavior, by invoking the MSF. The security manager switches TWINKLE to vigilant mode (box 3). The security manager's SBHT matches the suspicious behavior detected with the proper SBHR (e.g., CHANGE-FREQUENCY), and the security manager then invokes the security engine to run that SBHR. By running the CHANGE-FREQUENCY routine, the security engine commands A and B to change their communication channel to possibly alleviate the jammed link (box 4). In a worst case scenario, where no SBHR can mitigate the attack (e.g., all channels are being jammed), the security manager can notify the network administrator (we assume that the manager is running on a router with wired ethernet connection, thereby bypassing any jamming that may be present). After the attack is mitigated, either by successfully changing the communication channel or by the network administrator manually identifying and removing the malicious device from the network, the SBHR terminates, and regular mode resumes.

2.3.2 Example Application to Address Flooding Attacks. In Figure 2b, we show how TWINKLE handles a flooding attack where device A is flooding device B with unwanted traffic. Note, the unwanted traffic may have originated from a node other than A, who is simply forwarding the traffic to B, and therefore may not be malicious. Similar to the previous example, the first two boxes show TWINKLE's operation in regular mode. First, the event watchdog, running an SBDR, detects an unusually large amount of traffic being sent to device B by device A, which surpasses the normal threshold. The event watchdog then invokes the security manager, which (by invoking the MSF) switches the security application to vigilant mode (boxes 2 and 3). The SBHT at the security manager points to the security engine's SBHR (e.g., CHANGE-PATH) and, as a result, the security manager invokes the security engine. The security engine then commands A to change its path to route its traffic to a node C which can filter the unwanted traffic and route only the wanted traffic to B (box 4). Here, C can be a machine dedicated to detecting and filtering malicious traffic which may be required for this particular security application. If A does not change its path, and continues to flood B, the security engine may notify the network administrator, or

Just Accepted

```
11 <behavior>
12     <name> ____________________ </name>
13     <SBDR>
14         <name> _________________ </name>
15         <parameter_name> ______ </parameter_name>
16         <parameter_value> _____ </parameter_value>
17         <devices> _____________ </devices>
18         ...
19     </SBDR>
20     <SBHR>
21         <name> _________________ </name>
22         <parameter_name> ______ </parameter_name>
23         <parameter_value> _____ </parameter_value>
24         <devices> _____________ </devices>
25         ...
26     </SBHR>
27 </behavior>
28 ...
```

(a) Behavior element

```
29 <routine>
30     <name> ____________________ </name>
31     <lang> ____________________ </lang>
32     <code> ____________________ </code>
33     <compile> _________________ </compile>
34 </routine>
35 ...
```

(b) Routine element

Fig. 3. A template XML file for representing a TWINKLE security application.

attempt to isolate A from the rest of the network by running other SBHRs. Once the attack had been mitigated, the SBHR terminates and the framework returns to regular mode.

2.3.3 Example Application to Address Weak Encryption. The TWINKLE framework can be used to proactively secure an IoT network that requires expensive but stronger encryption at certain critical periods. Figure 2c shows an example of how TWINKLE can help. For this IoT network, there are certain times during operation that require stronger encryption. During normal operational times, the network is operating under regular mode when weak or no encryption between certain devices, say A and B, is sufficient (box 1). However, once a critical period begins, the watchdog triggers the MSF (box 2), which switches the security application to vigilant mode. Once in vigilant mode, the security manager invokes the security engine to handle the time change and commands the devices to change to strong encryption (e.g., CHANGE-ENCRYPTION; box 4). During the secure period, the event watchdog nodes in the network can ensure that strong encryption is being used between the critical devices. Once the critical period ends, the framework returns to regular mode.

3 TWINKLE IMPLEMENTATION DETAILS

There are three main challenges that must be addressed in order to implement the TWINKLE framework. First, the security application plugged into the framework must be represented in a way as to allow for routine instantiation by the security manager. Second, routine instantiation must be done automatically, without human intervention. Finally, after the routines are initiated, all of the components running the routines must interact with each other in order to detect, verify, and mitigate attacks. What follows is a detailed look at how the TWINKLE framework may be implemented in actuality and how each aforementioned challenge is addressed.

3.1 Representing a Security Application

In order for the TWINKLE framework to support various security applications with minimal effort, each security application must be represented in such a way as to allow for automatic routine instantiation. Generally, either the application's developer or network administrator is responsible for creating such a file. We represent security applications using a markup language such as XML. This allows for the security manager to easily parse the file to find which devices need to run which routines. Figure 3 depicts a template XML file for representing a security application. An XML file represents a single security application, and consists of the following two main type of elements: behavior and routine elements.

The *behavior* element contains information about a suspicious behavior that the security application must detect and handle (lines 11 to 28, as shown in Fig 3a). Note, each suspicious behavior is defined in a separate *behavior* element. The *behavior* element contains a *name* element, which serves simply as a unique identifier to differentiate it from other suspicious behaviors, along with an *SBDR* and an *SBHR* element, which represent the SBDR and SBHR to detect and handle the given suspicious behavior, respectively. The *SBDR* and *SBHR*

Just Accepted

elements further consist of a *name* element that represents the name of the routine, along with *parameter_name*, *parameter_value*, and *devices* elements. The *parameter_name* and *parameter_value* elements define parameters (e.g., thresholds) that the routine takes as input (as command-line arguments). Note again, there must be separate *parameter_name* and *parameter_value* elements for each routine parameter. The *devices* element specifies the devices, as a list of device IDs, on which the given routine will run.

The *routine* element contains the routine code, and information necessary for compiling and running the code (lines 29 to 35, as shown in Fig 3b). It contains a *name* element that represents the name of the routine. The name of the routine must match the name of an SBDR or SBHR specified in a *behavior* element. The *lang* element specifies the programming language in which the routine is written, which the device responsible for running the routine must know because the command to run the routine may depend on the language. For example, if the routine is written in a language that must be interpreted, such as Python (e.g., "*python routine.py arg0, arg1, ...*") and Java (e.g., "*java routine arg0, arg1, ...*"), the device must invoke the appropriate interpreter. The *code* element contains the actual code for the routine. The code of a routine can be as simple as a single function, or as complex as multiple classes that may need to be split across multiple files. The *compile* element specifies the compilation command for the code if it needs to be compiled.

It is important to note that one can introduce additional elements into the XML file when necessary, especially in order to include additional information required for certain security applications. Minimal change to the TWINKLE framework (mainly the security manager) would be required in order to parse any new elements.

3.2 Automated Routine Instantiation

3.2.1 Initialization and component threads. Each TWINKLE component is a lightweight process that contains several threads that are executed at start-up. The network administrator initializes TWINKLE by executing the security manager, event watchdog, and security engine processes. The network administrator provides the security manager process with a configuration file that includes information about devices in the network, such as a device's ID, IP address, and listening port, along with architecture, board, and operating system information, which the security manager process stores in the Network & Device Information module (which can simply be a database). The network administrator provides the event watchdog and security engine processes with a port number on which it should listen on for TWINKLE messages. Each process starts the component threads, which are independent to the security applications plugged into TWINKLE and must be running before the routines are instantiated. We list the threads for each component below and provide a brief description of each. We explain each in detail throughout the rest of this section.

- Security manager threads:
  - parser thread: parses and extracts routines from a security application file
  - assigner thread: assigns routines to the event watchdog and security engine
  - send/recv thread: sends and receives information to and from the event watchdog and security engine
  - mode switch thread: given an alert from the event watchdog, invokes the appropriate SBHR
  - query handler thread: handles queries from the event watchdog and security engine
  - alert thread: sends SMS and email alerts to network administrator
  - cross-compiler thread: cross-compiles a given routine (only when needed)
- Event watchdog and security engine threads:
  - send/recv thread: sends and receives information to and from the security manager
  - compiler thread: compiles a given routine
  - runner thread: runs a given routine

3.2.2 Parsing a security application. Once a security application needs to be plugged into TWINKLE, the network administrator uploads its file onto the security manager. Figure 4 shows a flow diagram of the interactions

Just Accepted

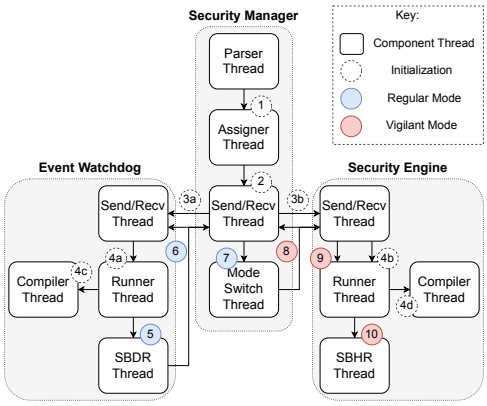


Fig. 4. A flow diagram of the interactions between the main threads in the TWINKLE framework.

between the main threads in the TWINKLE framework. We will refer to it and the circles depicted in it which represent the communications between threads throughout the rest of this section. The **parser thread** reads in the file and parses each element. It parses each *behavior* element, and extracts the necessary information to populate the SBHT and create source code files for the SBDRs and SBHRs. Specifically, it appends a row of two columns to the SBHT, where one element of the row is the name of the suspicious behavior and the other element is a pointer to the SBHR (e.g., the name of the SBHR) that handles that suspicious behavior. Parameter values, as specified in the *parameter_value* elements, are ultimately sent as arguments along with the source code files to event watchdog and security engine devices. In order to correspond parameters to their routines, the parser matches the *name* element in the *SBDR*/*SBHR* element with the *name* element in the *routine* element. In order to correspond routines to the devices on which they will run, the parser parses the *devices* element. Once all parameters are matched with their routines and all routines are matched with devices, the parser parses the *code* element to generate the source code file(s), along with the *lang* and *compile* elements. Finally, it sends all of the parsed information of the security application to the **assigner thread** (circle 1).

3.2.3 Assigning routines to devices. After the assigner thread receives the parsed information from the parser thread, it assigns the SBDR and SBHR source code to the event watchdog and security engine devices, respectively. It does so by mapping each device's ID to its IP address and port, and appends this information to the original parsed information. Then the assigner thread sends all of the information of the security application to the **send/recv thread** (circle 2).

3.2.4 Sending routines to devices. After the security manager's send/recv thread receives a message from the assigner thread for a device, it simply forwards the message as a *routine instantiation information message* to the device (circles 3a and 3b). The send/recv thread creates and manages a TCP/IP socket for each receiving device, where each socket is connected to a receiving device's IP and port, which it fetches from the Network & Device Information module.

3.2.5 Executing routines. After receiving a routine instantiation information message, the device's send/recv thread forwards it to the **runner thread** (circles 4a and 4b). The runner thread first checks if the routine needs to be compiled or can be interpreted. If the former, it invokes the **compiler thread**, which compiles the source code as specified by the given compilation command into an executable (circle 4c and 4d). It then checks the type of the routine. If the routine is an SBDR, the runner runs the routine in a new SBDR thread (circle 5). If the routine is an SBHR, it saves the code to be executed when the SBDR thread detects the suspicious behavior

Just Accepted

in question. Note, by compiling and running routines in such a way, routines can be easily *hot-plugged* into TWINKLE without requiring any of the devices to be restarted or interrupted.

3.2.6 Dealing with errors. If an error is encountered at any point in the compiler and runner threads, the compiler or runner thread will send an *error message* that includes the specific compilation or runtime error, to the security manager. The security manager's **alert thread** can then send an alert to the network administrator that an error during routine instantiation occurred.

3.2.7 Cross-compiling for embedded systems. The security manager's **cross-compiler thread** cross-compiles routines on behalf of extremely lightweight devices, such as an embedded system without an operating system. To do so, the assigner thread invokes the cross-compiler thread before invoking the send/recv thread. The compiler installed on the security manager must be compatible with the target device's processor architecture. Note, because compiling on the target machine is generally more safe and straightforward, cross-compilation is only done when absolutely necessary; capable devices compile routines themselves, as previously described.

3.3 Component Interaction After Routine Instantiation

Next, we explain how the components interact with each other after routine instantiation is complete. For simplicity, we explain the interaction between a single event watchdog device and security engine device.

3.3.1 Generating suspicious behavior alerts. In regular mode, the watchdog device's SBDR thread is running (or multiple SBDR threads are running). Once an SBDR thread detects a suspicious behavior it constructs a *suspicious behavior alert message* that includes the device's ID, name of the suspicious behavior, and a list of parameters to be consumed as arguments by the corresponding SBHR. These parameters are usually identifying features of the entity that caused the suspicious behavior alert (e.g., the source and destination IP:port pairs of a suspicious connection). Note, the SBDR and corresponding SBHR must be coded so that the output of the SBDR can be seamlessly handled as input to the SBHR. The SBDR thread sends the suspicious behavior alert message to the send/recv thread which in turn forwards the message to the security manager's send/recv thread as shown by circle 6 (note, the connection established during routine instantiation is kept alive, thus allowing the two devices to communicate over the same connection).

3.3.2 Mode switching and invoking the SBHR. The security manager's send/recv thread observes that the message is a suspicious behavior alert message, and sends the message to the **mode switch thread** (circle 7). The mode switching thread first extracts the name of the suspicious behavior from the message, looks it up in the SBHT, and finds the name of the corresponding SBHR that should be invoked. It then constructs an *SBHR invocation message*, which includes the name of the SBHR, the event watchdog's device ID, and the list of parameters, and sends it to the send/recv thread, which in turn forwards the message to the security engine's send/recv thread (circle 8).

3.3.3 Running the SBHR. The security engine's send/recv thread receives the message, observes the message is an SBHR invocation message, and sends the message to the local runner thread (circle 9). The runner thread finds the SBHR with the same name as is specified in the message, appends the parameters in the message to the list of arguments, and prepares to run the SBHR. In the same way that the event watchdog's runner thread created a thread for running its SBDR, the security engine creates a thread for running the SBHR (SBHR thread), as shown by circle 10. During the process of running, the SBHR may need to issue an alert to the network administrator, and it does so by sending an *SBHR alert message*, which includes all information regarding the alert, to the security manager. The security manager's alert thread in turn forwards the entire SBHR alert message to the network administrator.

3.3.4 Querying the security manager. Finally, if at any time an SBDR or SBHR thread needs to query the security manager's Network & Device Information module, it sends the query (e.g., an SQL command) encapsulated in a *query message* to the security manager via the local send/recv thread. The security manager's send/recv thread

Just Accepted

forwards the query message to the **query handler thread**, which simply extracts and executes the query. It then encapsulates the result in a *response message*, and sends the message back to the send/recv thread, which forwards it to the receiving device's send/recv thread. Note, the query message may include update commands, which when executed, will update the Network & Device Information module with new information.

4 CASE STUDY I: DDOS ATTACK DETECTION BY TRANSFORMING D-WARD

In this case study, we transform D-WARD, a classic security system for detecting and mitigating DDoS attacks at the source-end of the DDoS traffic, into D-WARD+, a new DDoS defense solution as a security application on TWINKLE.

4.1 DDoS Attacks with IoT Devices

In a DDoS attack, an attacker sends a victim, such as a web server, an overwhelming amount of traffic to make it unavailable. The attacker usually employs a botnet, or a network of compromised devices, to send the traffic. Due to their abundance and the ease to be compromised, IoT devices are easy targets to be recruited by a botnet. As shown in the Mirai attack [16], recent DDoS attacks have been launched from compromised IoT devices and networks [17].

4.2 Prior Art: D-WARD Against DDoS Attacks

A DDoS defense system placed near the victim may struggle with high volume attacks, but because links closer to the attack sources are less likely to be overwhelmed, filtering attack traffic becomes more feasible for source-end defense systems. One source-end solution example is D-WARD [23], which we detail in this subsection.

Deployed at the border router of a policed network, D-WARD consists of an observation module, a rate-limiting module, and a traffic-policing module. The observation module classifies each aggregated flow, or **agflow**, from all devices in the policed network to an entity outside, **receiver**, as good, suspicious, or attack. The classification is based on the ratio of sent packets to received packets of each agflow. Also, each agflow consists of multiple connections where each connection is the traffic from a specific device to the receiver. D-WARD classifies each individual connection as good, transient, or bad, also based on the ratio of sent packets to received packets (smoothed over time) of the connection; a connection is classified as good if its smoothed packet ratio is below a threshold defined by a *legitimate TCP connection model*, transient if there is not enough information about the connection to discern its packet ratio, and bad if it is classified as neither good nor transient. D-WARD's observation component stores information in an *agflow table* and a *connection table*. In the agflow table, D-WARD stores the number of bytes sent, received, and dropped for each agflow, which is used to label the agflow and calculate the potential rate-limit for connections in a suspicious or attack agflow. In the connection table, D-WARD stores the smoothed packet ratio for each connection, which is used to determine if connections follow the legitimate connection model. For agflows that are labeled suspicious or attack, D-WARD first checks the smoothed packet ratio of each connection in the agflows to determine which connections need to be rate-limited; only bad and transient connections are rate-limited. Specifically, the rate limiting module cuts the allowed sending rate of all bad and transient connections in a suspicious or attack agflow to a fraction, f_{dec}, of the afglow's current sending rate. The observation module observes the aglow for a certain period of time, called the *observation interval*. The dynamic rate-limit for the next observation interval is determined by the *agflow compliance factor*, which is the ratio of bytes sent to the sum of bytes sent and bytes dropped; a high compliance factor leads to a relaxed rate-limit in the next observation period. Finally, the traffic-policing module decides whether to forward or drop each outgoing packet. It allows all packets from good connections and transient connections that belong to good agflows to be forwarded, but drops packets based on the current rate-limit from bad and transient connections that belong to attack or suspicious agflows.

Just Accepted

D-WARD is designed for DDoS attacks launched from traditional end-hosts on the Internet, and therefore, several drawbacks may arise when deploying it in a smart home environment that otherwise would not be noticed in a traditional network. First, the memory consumption caused by storing agflow and connection statistics may be too costly in constrained IoT networks. Second, and most importantly, D-WARD could hurt benign devices if their connections are labeled as transient or mislabeled as bad connections, since their traffic, if over the dynamic rate-limit, is dropped. While a traditional benign end-host can recover from the accidental loss of their packets, in an IoT environment such as a smart home, a benign device could instead suffer significantly from such a loss, due to unnecessary retransmissions of lost packets and increased latency. As we show in Section 6.1, unnecessary retransmissions and an increase in connection duration directly leads to an increase in energy consumption.

4.3 D-WARD+: A Two-Mode Approach Against DDoS Attacks

We therefore transform D-WARD into D-WARD+ that runs on TWINKLE. To overcome the aforementioned drawbacks of D-WARD, D-WARD+ significantly reduces memory consumption by not storing any connection-level information in regular mode, and only storing suspicious and bad connection information in vigilant mode. Furthermore, when detecting a DDoS attack from a policed network, D-WARD+ leverages the *fast retransmit* mechanism in TCP congestion control to reduce the sending rate of transient connections, rather than dropping their packets as done in D-WARD. Since these connections could be from benign devices, leveraging fast retransmit does not cause their packets to be dropped, but does lower the amount of DDoS traffic departing from the network. In this subsection, we present the two-mode design of D-WARD+ and explain in detail why D-WARD+ is better suited for an IoT environment as compared to D-WARD.

The XML representation of D-WARD+ consists of a single behavior element for detecting and handling suspicious agflows, and two routine elements. Within the behavior element, the SBDR and SBHR elements each specify the name and input parameters of a routine. The SBDR element defines an *agflow monitoring routine*, which observes and labels each agflow in the network and detects suspicious agflows. The SBHR element defines a *connection monitoring routine*, which inspects every suspicious aglow more closely, including each connection in the agflow. The agflow monitoring routine has a single input parameter value that defines the threshold for determining if an agflow is suspicious in terms of its ratio of sent packets to received packets, while the connection monitoring routine does not have any input parameters. The routine elements provide the programming language, code, and compilation specifics of the SBDR and SBHR.

The security manager, event watchdog, and security engine of D-WARD+, all running at the border router, are designed as follows. The security manager's Network & Device Information module stores the agflow and connection tables, along with keeping track of a fixed rate-limit based on the receiver's TCP receive window (RWIN) for every suspcious agflow, which is detailed in the following paragraphs. The event watchdog consists of the agflow monitoring routine and invokes the security engine when it detects a suspicious agflow. The security engine consists of the connection monitoring routine.

As a security application of TWINKLE, D-WARD+ handles DDoS attacks by switching between the two modes. In regular mode, the event watchdog keeps track of each agflow's sent to received packet ratio and stores this information in the agflow table, located at the security manager. However, unlike D-WARD, the event watchdog does not keep track of any connection-level information in regular mode. If the event watchdog detects a suspicious agflow, it invokes the security manager's MSF to determine what routine to execute at the security engine.

In vigilant mode, the MSF invokes the security engine's connection monitoring routine to handle the agflow in question. The security engine begins keeping track of the smoothed packet ratio for bad and *suspicious connections*, and stores these ratios in the connection table, also located at the security manager. Here, we introduce a new connection class called suspicious connections, which include transient connections and other connections that have only slightly surpassed the smoothed packet ratio threshold (i.e., connections that may be legitimate, but

Just Accepted

were labeled bad by the legitimate connection model). Introducing this new class of connection allows us to reduce collateral damage, especially since false positives caused by strict legitimate connection models leads to unnecessary network overhead (specifically, an increase in retransmissions and connection durations). In addition to the smoothed packet ratio for each bad and suspicious connection, the security engine periodically keeps track of the receiver's RWIN, for each suspicious agflow. Unlike D-WARD, which attempts to guess the maximum sending rate that the receiver can handle by calculating a dynamic rate-limit, D-WARD+ sets a fixed rate-limit at the beginning of each observation period which is set to the current RWIN. Although D-WARD+ is periodically storing an additional value in memory, it ensures that traffic sent from the IoT network never overwhelms the receiver. Furthermore, we will show that D-WARD+ significantly decreases the overall memory consumption as compared to D-WARD in subsection 6.1, and therefore, periodically keeping track of RWIN is feasible. The security engine monitors each suspicious connection of the suspicious agflow; it sends three duplicate TCP acknowledgments to the device of the connection, which, by following the TCP congestion control design, reduces its congestion window by half, thus halving its sending rate. Here, we call the three duplicate TCP acknowledgments a *signal*. In case the device ignores the signal and continues to send its traffic at the original rate, the routine detects it, labels the connection as a bad connection, and drops its packets (note that if a DDoS device follows the signal in the same way as a benign device, it lowers its sending rate and effectively mitigates the DDoS attack). Furthermore, if the traffic volume of the connection is still above the static rate-limit after sending a signal, the routine can send another signal and observe the volume change of the connection, and it can repeat this procedure until the connection is no longer overwhelming its receiver, thus mitigating an ongoing DDoS attack.

Based on the two-mode design above, D-WARD+ is more suitable to a smart home environment than D-WARD. It significantly reduces memory consumption by being selective with what information is stored in the agflow and connection tables. Furthermore, by not dropping packets as in D-WARD, D-WARD+ instead informs devices to transmit more slowly. Doing so avoids retransmissions of packets from benign devices, thus lowering network overhead and power consumption.

5 CASE STUDY II: SINKHOLE ATTACK DETECTION BY TRANSFORMING SVELTE

In this case study, we transform SVELTE, an IDS for detecting sinkhole attacks in 6LoWPAN networks, into a more resource-efficient security application on TWINKLE.

5.1 Sinkhole Attack in 6LoWPAN Networks

6LoWPAN (IPv6 over Low power Wireless Personal Area Networks) is a wireless technology that combines IPv6 and Low-power Wireless Personal Area Networks (LoWPAN) to enable low-powered devices to communicate using an Internet protocol. A 6LoWPAN network uses RPL (Routing Protocol over Low Powered and Lossy Networks) as its routing protocol [40]. For each destination in a 6LoWPAN network to reach, RPL creates a graph called *Destination Oriented Directed Acyclic Graph* (DODAG) where every node is a device in the network and the destination is the root. Each node in a DODAG has a set of parents, including a preferred parent, where every parent is a potential next hop to reach the root. Moreover, every node in a DODAG has a *rank* to represent the distance between the device and the root (the distance can be calculated in a number of ways, the simplest being hop-count).

Each device periodically sends out a *DODAG Information Object* (DIO) message to advertise its rank. An entering device, upon the receipt of DIO messages from its neighboring devices, creates its set of parents, chooses the preferred parent, and calculates its own rank (which is greater than the rank of each of its parents).

The 6LoWPAN network is subject to the sinkhole attack. In such an attack, a compromised device announces a short path toward a destination node to attract traffic from other nodes to the destination, therefore intercepting or dropping the traffic and creating a sinkhole. A sinkhole attack via RPL can happen when a device sends to its

JUST Accepted

neighbors a DIO message to lie that the device has a low rank. It has been shown that RPL's self-healing and repair mechanisms are not resilient against the sinkhole attack [39].

5.2 Prior Art: SVELTE Against the Sinkhole Attack in 6LoWPAN

SVELTE detects sinkhole attacks in 6LoWPAN networks that occur through RPL rank manipulation. It has two main modules running on the border router of a 6LoWPAN network: 6LoWPAN Mapper (6Mapper) that gathers information about the network and determines the DODAG rooted at the border router, and an intrusion detection module that checks the rank inconsistency in data obtained by 6Mapper to detect sinkhole attacks. The 6Mapper sends *probing* messages to nodes in the network at regular intervals (e.g., 2 minutes). Each node runs a 6Mapper client which sends a *response* message to the 6Mapper, which includes its node ID, node rank, parent ID, and all of its neighbors' IDs and ranks.

However, SVELTE's probing mechanism can increase the network overhead, device power consumption, and the latency of detecting sinkhole attacks. Every probe from the border router increases the network overhead. Every response from a device consumes more power. Worst of all, SVELTE has a dilemma in choosing the probing interval: a short interval leads to a low latency in detecting sinkhole attacks, but a large overhead due to frequent probing and responding; a long interval results in a low overhead, but a high latency in detecting sinkhole attacks.

5.3 SVELTE+: A Two-Mode Approach Against Sinkhole Attacks

To be more resource-efficient, we transform SVELTE to SVELTE+ that runs on TWINKLE. The essential difference between SVELTE+ and SVELTE is that the 6Mapper in SVELTE+ does not probe the entire network periodically, and correspondingly, the intrusion detection component does not run periodically, either.

The XML representation of the SVELTE+ security application, like D-WARD+, consists of a single behavior element and two routine elements. The behavior element defines an SBDR element and an SBHR element. The SBDR element defines a *rank advertisement monitoring routine*, which monitors the network for new rank advertisements. The SBHR element defines a *sinkhole handling routine*, which detects sinkhole attacks by inspecting each rank advertisement and attempts to mitigate attacks. Neither routine contains any parameter values. The routine elements implement the SBDR and the SBHR, respectively, by providing the code of the two routines and specifying their programming language and compilation specifics.

When SVELTE+ is plugged into TWINKLE, its security manager, event watchdog, and security engine are as follows. The security manager consists of the 6Mapper (stored in the Network & Device Information module) from SVELTE which runs on the border router. A subset of devices are equipped with an event watchdog which acts as a 6Mapper client and runs the rank advertisement monitoring routine. Specifically, an event watchdog monitors the RPL ranks of its neighbors and sends an alert message to the security manager of a suspicious behavior when it receives a new rank advertisement. The security engine, which also runs on the border router, runs the sinkhole handling routine. Specifically, it inspects the ranks of nodes in the DODAG graph to determine if a sinkhole attack is occurring (i.e., the intrusion detection module from SVELTE) and can mitigate a detected sinkhole attack (functionality that SVELTE+ newly introduces).

SVELTE+ detects sinkhole attacks by switching between the two modes. It begins in regular mode. The event watchdog continuously updates the security manager's 6Mapper whenever it hears RPL control messages. Each time a node advertises a new rank, the event watchdogs that are within range of the advertisement treat the node as a suspect and thereby, detect a suspicious behavior. Each event watchdog then sends an alert message, which includes its own rank and the rank of the suspect, to the security manager.

Note, an event watchdog node may not be directly connected to the border router, and therefore its alert messages must be routed through other nodes before reaching the security manager. To prevent alert messages from being manipulated or dropped by potentially malicious nodes in the network, event watchdog nodes prefer routing alert messages through other neighboring event watchdog nodes. Event watchdog nodes can be deployed

Just Accepted

in the network in such a way that every watchdog can reach the border router via one or multiple paths that include only other event watchdog nodes *en route*. We refer to such paths as event watchdog paths.

Once the security manager receives an alert message, it invokes the MSF to determine the appropriate SBHR. The security manager passes the alert message to the security engine. As soon as the MSF begins its execution, SVELTE+ enters vigilant mode, allowing it to invoke the sinkhole handling routine at the security engine to handle the suspicious behavior. The sinkhole handling routine first queries the 6Mapper in the security manager for an up-to-date DODAG; then, if the event watchdog is a parent (child) of the suspect and its rank is lower (greater) than the rank of the suspect as expected, the routine then has verified the consistency between this event watchdog and the suspect. If it has verified the rank consistency with all of the parents and children (or a threshold number of each) of the suspect, it treats the suspect as a benign node and updates the 6Mapper to add the node to the DODAG, or simply updates the node's rank if it is already in the DODAG. In case the sinkhole handling routine cannot establish the rank consistency between the suspect and its parents and children, it detects a sinkhole attack, labels the suspect as a sinkhole attacker, and further attempts to mitigate the attack as described below. The sinkhole handling routine then finishes its execution, followed by the MSF, and SVELTE+ returns to regular mode.

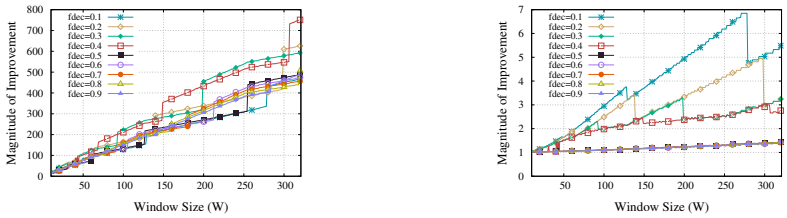
The sinkhole handling routine can remove a sinkhole node from not only the DODAG, but also the records of any device. Specifically, every parent of the attacker removes it as their child. Every child of the attacker removes it as its parent; it may also add a new parent as well as choose a new preferred parent. As a result, the attacker is isolated and can no longer reach any other node.

SVELTE+ outperforms SVELTE in multiple ways. SVELTE+ can reduce the latency in detecting sinkhole attacks to a negligible amount because the event watchdog immediately invokes the MSF whenever a new rank is advertised, without having to wait for the next probing interval, as in SVELTE. SVELTE+ also decreases the network overhead and device power consumption as compared to SVELTE; SVELTE+ may incur more overhead in the beginning as nodes join the network, but as the network stabilizes, the amount of times SVELTE+ switches to vigilant mode is low. An exception here is that a malicious node may frequently advertise a new, legitimate rank, causing SVELTE+ to repeatedly process the suspicious behavior; SVELTE+ sets an upper bound at which a benign node would advertise a new rank and labels a node as malicious if it advertises a new rank too frequently (it can further remove the node using the sinkhole handling routine).

Additionally, when there are multiple collaborating malicious devices which can intercept and manipulate probing and response messages, a smart home running SVELTE is more susceptible to a sinkhole attack as compared to a smart home running SVELTE+. For example, let us assume two devices A and B on the same path towards the border router are compromised, and the device farther away from the border router, say B, advertises an illegitimate rank. In the next probing period, device A can manipulate response messages from devices that received the illegitimate rank advertisement from B so that SVELTE cannot detect the attack. However, with SVELTE+, by leveraging event watchdog paths, each alert message can reach the border router only via event watchdog nodes along one or multiple event watchdog paths, significantly reducing the chances of success for collaborative attacks. In order to launch a successful collaborative attack against SVELTE+, or more specifically, to suppress an alert message, an attacker would have to identify and compromise at least one event watchdog on each event watchdog path used by the alert message.

In summary, by leveraging the two-mode design and event watchdog component of the TWINKLE framework, SVELTE can be improved for the smart home environment. The two-mode, on-demand approach of SVELTE+ can significantly reduce detection latency and network overhead, as compared to SVELTE. Furthermore, leveraging event watchdog nodes allow SVELTE+ to be more resilient against more complex sinkhole attacks, such as attacks in which multiple compromised devices collaborate in hopes to thwart the defense.

Just Accepted



(a) Retransmissions **(b) Connection duration**

Fig. 5. Comparing retransmissions and connection duration under D-WARD and D-WARD+.

6 EVALUATION

We evaluate TWINKLE's two-mode design by showing how D-WARD+ outperforms D-WARD in source-end DDoS defense and how SVELTE+ outperforms SVELTE in sinkhole attack detection. The metrics we focused on are retransmissions, connection duration, energy consumption, and memory consumption for the DDoS case study and network overhead, detection latency, and energy consumption for the sinkhole case study. For the DDoS case study, we additionally compared the effects of D-WARD+ and D-WARD on a naive TCP flooding attack versus a smart TCP flooding attack, and analyze the differences in how the two systems detect and mitigate such attacks. Note that we did not compare SVELTE+ to SVELTE in terms of detection accuracy because SVELTE+ uses the same detection module as SVELTE.

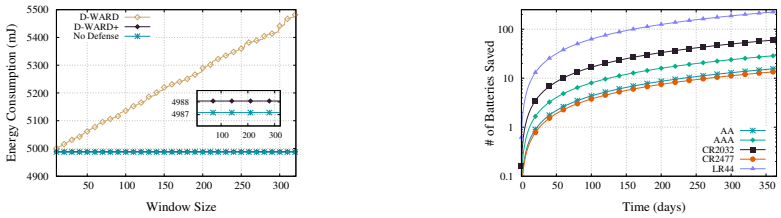
The evaluation results are a mixture of real-world testing, simulation, and formulation, where small-scale testing was performed on a real-world IoT testbed and large-scale results were obtained through simulation and formulation. For small-scale results, we constructed two IoT testbeds, one in which the devices communicated over 802.11b/g/n Wi-Fi, and the other in which the devices communicated over Bluetooth LE. Both testbeds consisted of several low-powered Raspberry Pi Zero W devices (1 GHz single-core CPU, and 512 MB of RAM), and a 2015 Dell XPS (2.2 GHz dual-core CPU, and 8 GB of RAM) as the border router. For large-scale results, we implemented the TWINKLE framework, D-WARD+, D-WARD, SVELTE+, and SVELTE in Java and Python on a 2015 Dell XPS with the same specifications as mentioned previously. When comparing D-WARD and D-WARD+ through formulation, TCP Reno is utilized for congestion control. Lastly, for the large-scale evaluation of SVELTE+ and SVELTE through simulation, we randomly generated mesh IoT network topologies of varying sizes.

6.1 D-WARD+ vs. D-WARD

The main difference between D-WARD+ and D-WARD is that D-WARD+ utilizes the fast retransmit mechanism instead of dropping packets from suspicious connections. The fast retransmit mechanism allows D-WARD+ to throttle DDoS traffic that leaves the source network it polices and avoid resource penalties on benign traffic. In this subsection, we analyzed the attainability of these goals in a smart-home network that utilizes D-WARD+. Specifically, we analyzed the following:

(1) the ratio of retransmissions D-WARD requires of a benign suspicious connection over the amount required by D-WARD+;
(2) the difference in connection duration of a benign suspicious connection under D-WARD compared to that of D-WARD+;
(3) the energy consumed by a benign device under D-WARD and D-WARD+;
(4) the amount of memory consumed by the border router running D-WARD and D-WARD+;
(5) the behavior of a naive attacker under D-WARD and D-WARD+;
(6) the behavior of a smart attacker under D-WARD and D-WARD+.

Just Accepted



(a) Energy consumption **(b) Number of batteries saved**

Fig. 6. Energy consumption under D-WARD and D-WARD+, along with the number of batteries D-WARD+ saves over a one year period.

We hypothesize that due to D-WARD throttling benign suspicious connections, as opposed to D-WARD+ which utilizes the fast retransmit mechanism, D-WARD will incur more retransmissions and longer connections than D-WARD+, which will lead to higher energy consumption. Furthermore, because D-WARD continuously keeps track of connection-level information, as opposed to D-WARD+ which only stores connection-level information in vigilant mode, we hypothesize that D-WARD will incur higher memory consumption than D-WARD+. Lastly, because D-WARD+, unlike D-WARD, sets the fixed rate-limit of each agflow based on RWIN, we hypothesize that D-WARD+ will better handle smart attackers that follow congestion control, but not flow control.

6.1.1 Retransmissions. In order to compare the amount of retransmissions required of a benign suspicious connection by D-WARD and D-WARD+, we examined, through formulation, the number of retransmissions required of a benign suspicious connection that attempts to send 2.5 MB of data outside of the network at a maximum bandwidth of 250 Kb/s under both D-WARD and D-WARD+.

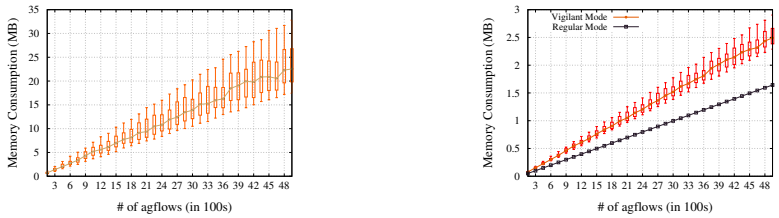
Figure 5a presents the ratio of retransmissions of a benign suspicious connection under D-WARD over D-WARD+. We call this ratio "magnitude of improvement" because it signifies the magnitude at which D-WARD+ prevents unnecessary retransmissions, as compared to its counterpart. We measure the magnitude of improvement with respect to two main parameters: the sender's congestion window size, W, at the time D-WARD or D-WARD+ detects a potentially malicious agflow, and the pre-set fraction of traffic, f_{dec}, that D-WARD or D-WARD+ allows to leave the source network during a suspected DDoS attack. f_{dec} is set to 1/2 by default in [23]. Upon detection of an attack agflow, D-WARD only allows $W * f_{dec}$ segments to the sender each RTT to mitigate any DDoS attacks. Therefore, when the benign suspicious devices follow TCP congestion control, D-WARD drops $W - W * f_{dec}$ segments every 2 RTTs. Thus, as W increases, D-WARD drops more segments which causes more retransmissions. With a large window size and depending on the f_{dec}, D-WARD may require more than 500 times the number of retransmissions than D-WARD+. Even when the window size is less than 10 and given any pre-set fraction of allowed traffic, D-WARD still requires more than 10 times the number of retransmissions than D-WARD+.

6.1.2 Connection Duration. We further compared how long a benign suspicious connection may last under D-WARD and D-WARD+. Clearly, when transmitting the same amount of data, a shorter duration is desired. We examined the duration of a benign suspicious connection that attempts to send 2.5 MB of data outside of the network, again at a maximum bandwidth of 250 Kb/s, under both D-WARD and D-WARD+.

Figure 5b shows the magnitude of improvement in connection duration of D-WARD+ over D-WARD (ratio of average connection duration under D-WARD, over average connection duration under D-WARD+) with respect to the two main parameters W and f_{dec}. When f_{dec} is set low, D-WARD may punish a benign suspicious connection too heavily which leads to long connection durations. However, in cases where f_{dec} is set high (0.5 or above) and W is small, a benign suspicious connection's duration under D-WARD+ is only slightly faster (at most 3 seconds) than if it were under D-WARD (i.e., a small to no magnitude of improvement).

6.1.3 Energy Consumption. Based on the analysis presented by Feeney et al. [12], who estimate the microwatt seconds consumed by a wireless device with respect to the amount of data transmitted, we estimate, through

Just Accepted



(a) D-WARD **(b) D-WARD+**

Fig. 7. Memory consumption for D-WARD and D-WARD+ (in regular mode and vigilant mode). In each graph, 80% of all agflows are good and 80% of all connections are good.

formulation, the energy consumption D-WARD and D-WARD+ requires from a benign IoT device. The results are shown in Figure 6a. Under D-WARD (with an f_{dec} of 1/2), energy consumption increases linearly with respect to the benign device's congestion window size. However, under D-WARD+, energy consumption is static across varying window sizes. Furthermore, D-WARD+ contributes to less than 1 mJ of extra energy consumption for a benign device. This is again due to the fact that D-WARD+ does not throttle traffic and, as a result, does not require a large amount of retransmissions.

Recall in Section 2.1, we modeled and compared the energy consumption of a device under a two-mode paradigm versus a one-mode paradigm. We can observe that under D-WARD's one mode paradigm, devices consume more energy than even D-WARD+'s vigilant mode. Therefore, $P_{rd} \leq P_{vd} < P_{xd}$, and thus $E_{two-mode} < E_{one-mode}$ holds true in this case study.

We further extend the energy consumption results to analyze D-WARD's cost of increased energy consumption by showing how much battery life D-WARD+ can save over D-WARD. We examine the battery life of a benign IoT device under D-WARD and D-WARD+ across five popular IoT batteries (alkaline AA, alkaline AAA, CR2032, CR2477, and LR44) and present how many more batteries an IoT device consumes under D-WARD throughout a year of deployment. These results are shown in Figure 6b. On average, across the batteries tested, a benign IoT device consumes 15.55 less batteries every year under D-WARD+.

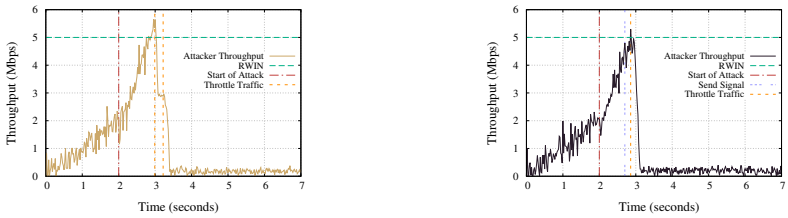
Note that energy consumption should still be a concern for devices that are plugged into an external power source. Energy efficiency is a critical factor for the rise in smart home environments and this is especially true for large-scale environments, such as smart cities [37]. Therefore, a defense system that minimizes energy consumption is preferable in IoT environments, no matter if the devices are plugged-in or battery powered.

6.1.4 Memory Consumption. Next, we analyzed the memory consumption at the border router under both systems. As stated in subsections 4.2 and 4.3, D-WARD maintains information about agflows and connections, while D-WARD+ in regular mode maintains information about agflows, and in vigilant mode, maintains information about suspicious and bad connections. Therefore, the number of agflows and connections per agflow affects the memory consumption of both systems.

Figure 7 shows the large-scale memory consumption results of both systems through simulation (a total of 100 runs). We assume that 80% of all agflows and connections are good. The number of connections per agflow follows a power law distribution, where a few agflows have many connections (over 100), while most agflows have only a few connections each (less than 5).

It is clear that D-WARD (Figure 7a) incurs more memory consumption than D-WARD+ (Figure 7b), especially as the number of agflows increase. This is due to the fact that D-WARD+ does not keep track of connection-level information in regular mode, and only keeps track of suspicious and bad connections in vigilant mode, while D-WARD keeps track of agflow and connection information continuously. Furthermore, the added memory consumption in vigilant mode for keeping track of the RWIN for each suspicious connection is insignificant. The memory consumption D-WARD incurs in a traditional network is more than acceptable (and probably acceptable

Just Accepted



(a) D-WARD **(b) D-WARD+**

Fig. 8. Behavior of a naive attacker under D-WARD and D-WARD+.

in a smart home environment), but not in constrained IoT networks where the border router, like the devices in the network, is memory-constrained.

6.1.5 Naive TCP Flooding Attack. A *naive* TCP flooding attack is one in which the attacker ignores TCP congestion and flow control. In this subsection, we analyze how D-WARD and D-WARD+ handle a naive attacker.

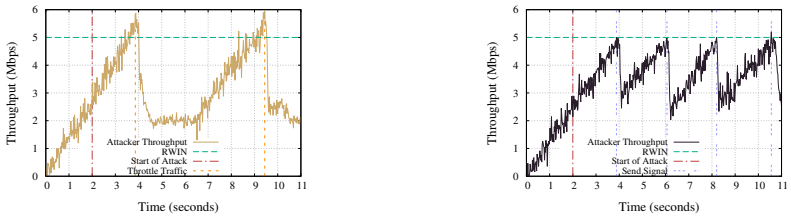
As explained in detail in subsection 4.2, D-WARD classifies connections based on the ratio of the number of packets sent and received, where a ratio that surpasses the maximum threshold indicates an attack. Depending on how the maximum threshold is set, D-WARD may either allow some DDoS traffic to leave the policed network or incorrectly classify benign connections as bad.

In most cases (i.e., not taking into account when significant packet loss is present), when the sender is not overwhelming the receiver, the ratio of the number of packets sent and received stays relatively constant (which we call the normal ratio), regardless of the sending rate. However, as soon as the sending rate begins to overwhelm the receiver (i.e., RWIN is surpassed), the ratio of the number of packets sent and received increases with respect to the sending rate. If the maximum threshold is set below the normal ratio, D-WARD has a devastating impact on benign connections. Namely, whenever a benign connection surpasses the threshold, it essentially throttles continuously causing major collateral damage. Therefore, D-WARD aims to learn the correct normal ratio so it can set the maximum threshold to be higher. However, the larger the difference between the maximum threshold and the normal threshold, the more time it takes D-WARD to detect and throttle DDoS traffic. Mirkovic et al. set the maximum threshold to 3 by default [23], and we therefore use this value in our evaluation.

Figure 8a shows the behavior of a naive attacker under D-WARD in our 802.11b/g/n Wi-Fi testbed. The results of the Bluetooth LE testbed is relatively similar, and therefore, we only present the results of the 802.11b/g/n network. We start the DDoS attack at 2 seconds. At around 3 seconds, the maximum threshold is surpassed, causing D-WARD to throttle the connection and cut the throughput in half (we set f_{dec} to 1/2). Note, in Figures 8 and 9, the green dotted line represents the maximum throughput the receiver can handle before RWIN is surpassed, which we label as "RWIN" for simplicity. While D-WARD throttles the naive attacker, it still allows a small amount of DDoS traffic to leave the network (DDoS traffic here refers to the traffic that surpasses the victim's RWIN). Specifically, since the maximum threshold is met after RWIN is surpassed, the receiver is under DDoS attack for about 300 milliseconds before the connection is throttled. The attacker continues to send at its maximum sending rate even after it's connection is throttled because it ignores TCP congestion control. At around 3.2 seconds, D-WARD again throttles the connection (this time continuously) and classifies the connection as bad.

Unlike D-WARD, D-WARD+ keeps track of RWIN, thereby avoiding the hassle of choosing a correct maximum threshold for the ratio of number of packets sent and received. Figure 8b shows the behavior of a naive attacker under D-WARD+ (again, in our 802.11b/g/n Wi-Fi testbed). At around 2.7 seconds, D-WARD+ sends a signal to the attacker (three duplicate ACKs), which the attacker ignores (because it is ignoring TCP congestion control). After a few milliseconds (the time it takes a packet from the attacker to reach the border router), D-WARD+ notices that the attacker is not complying to the signal, labels the connection as bad, and begins throttling. Note that in these few milliseconds, the attacker's throughput can surpass RWIN, but it is immediately throttled.

Just Accepted



(a) D-WARD **(b) D-WARD+**

Fig. 9. Behavior of a smart attacker under D-WARD and D-WARD+.

In summary, both D-WARD and D-WARD+ handle naive attackers similarly. However, D-WARD+ does a slightly better job of preventing the victim from being overwhelmed with traffic that surpasses its RWIN. The differences between D-WARD and D-WARD+ is more noticeable when handling a smart attacker.

6.1.6 Smart TCP Flooding Attack. A *smart* TCP flooding attack is one in which the attacker follows TCP congestion control, but not flow control. In this subsection, we analyze how D-WARD and D-WARD+ handle a smart attacker.

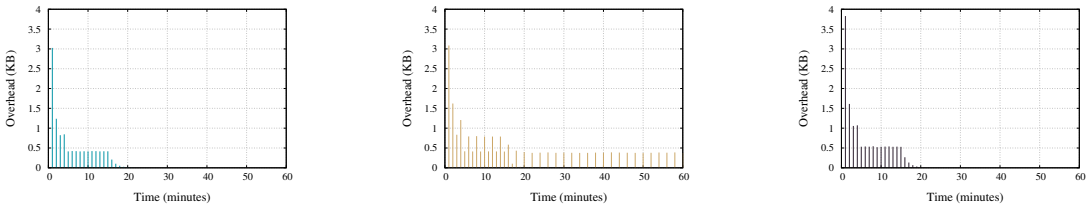
Figure 9a shows the behavior of a smart attacker under D-WARD in our 802.11b/g/n Wi-Fi testbed. The DDoS attack begins at 2 seconds. At around 4 seconds, the maximum threshold for the ratio of the number of packets sent and received is surpassed, causing D-WARD to throttle the connection and cut the throughput in half. Similar to the case of the naive attacker under D-WARD, the receiver is under DDoS attack for about 500 milliseconds before the connection is throttled. However, unlike in the naive attacker scenario, the smart attacker follows TCP congestion control and cuts its window size in half. For the next 2 seconds, D-WARD checks to see if the attacker continues to comply with the rate limit, which, in this case, the attacker does. Once the observation period is over, D-WARD linearly increases the connection's rate limit. At around 9.5 seconds, the connection again surpasses the maximum threshold and is throttled. But again, D-WARD allows DDoS traffic to leave the policed network for about 500 milliseconds. The attacker then cuts its sending rate again in half and complies with the rate limit. This trend continues, allowing the smart attacker to launch a successful periodic DDoS attack on the victim. Also note that if the maximum threshold were higher, the amount and length of the DDoS attack would increase.

Figure 9b shows the behavior of a smart attacker under D-WARD+ (again, in our 802.11b/g/n Wi-Fi testbed). At around 4 seconds, D-WARD+ sends a signal to the attacker, which the smart attacker responds to by cutting its sending rate in half (unlike the naive attacker). However, in this case, unlike D-WARD, no DDoS traffic surpassing RWIN leaves the policed network. The attacker follows TCP congestion control and linearly increases its sending rate (congestion avoidance phase). D-WARD+ again sends the attacker a signal at around 6 seconds. This trend continues. At each peak, D-WARD+ allows the attacker no more than 1/4 RTT (or travel time between attacker and border router) amount of time to send DDoS traffic surpassing RWIN (which, in this case, can be seen on the 4th signal – at around 10.5 seconds). This is significantly less than the amount of DDoS traffic that D-WARD allows to leave the network and makes an attack essentially unfeasible. However, to prevent even a minuscule amount of DDoS traffic from leaving the network, D-WARD+ could drop any traffic surpassing RWIN or preemptively send a signal before traffic surpasses RWIN, but such actions could cause a (relatively small) negative impact on benign devices that behave similarly to smart attackers.

6.2 SVELTE+ vs. SVELTE

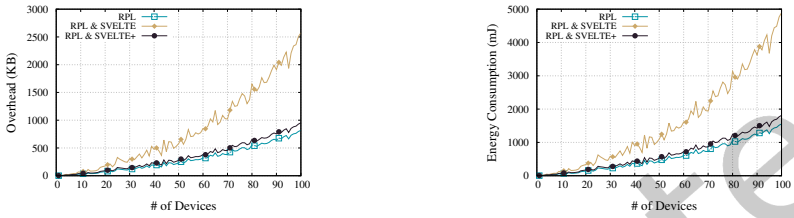
The main difference between SVELTE+ and SVELTE is that SVELTE+ utilizes on-demand probing while SVELTE utilizes periodic probing. In this subsection, we explored how this difference affected network overhead, energy consumption, and detection latency for both security applications. For small-scale results, we use our 802.11b/g/n Wi-Fi testbed which is running RPL, and for large-scale results, we simulate mesh networks of various sizes, also running RPL. We utilize SimpleRPL, which is a Linux-based implementation of RPL as defined in RFC 6550 [40]

Just Accepted



(a) RPL **(b) RPL & SVELTE** **(c) RPL & SVELTE+**

Fig. 10. Network overhead incurred by SVELTE and SVELTE+ in the IoT testbed.



(a) Network overhead **(b) Energy consumption**

Fig. 11. Network overhead incurred and energy consumption by no defense, SVELTE, and SVELTE+ when simulating larger-scale environments.

from the National Institute of Standards and Technology (NIST) to implement RPL in our testbed and simulations [10].

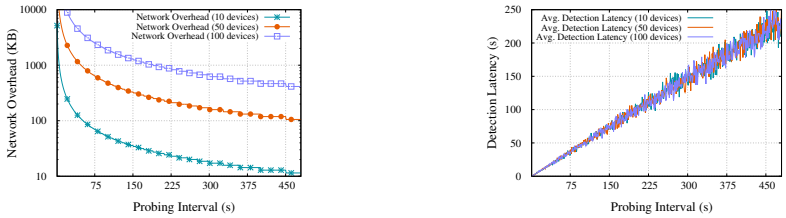
We hypothesize that SVELTE will incur a higher network overhead than SVELTE+ because SVELTE periodically probes the network, as opposed to SVELTE+ which probes the network on-demand. In turn, we expect SVELTE to incur higher energy consumption than SVELTE+. Lastly, we expect SVELTE's detection latency to be completely dependent on its probing interval.

6.2.1 Network Overhead. We analyze network overhead by measuring the number of extra bytes that are sent using SVELTE as compared to SVELTE+. Figure 10 depicts the difference, in terms of network overhead, between an RPL network running 1) no defense, 2) running SVELTE, and 3) running SVELTE+. We obtained these results from our IoT testbed in a 1-hour period.

Figure 10a shows how much overhead RPL alone adds to the network in the 1-hour period. RPL utilizes a trickle timer algorithm to disseminate routing information throughout the network. Each device maintains a trickle timer for periodically broadcasting DIO messages to make sure their neighbors have up-to-date and consistent routing information. As devices begin to join the network, DIO messages are sent more frequently, but as the network stabilizes, each devices' trickle timer increases exponentially, thereby exponentially decreasing the number of DIO messages sent by each device. Figure 10b shows how much overhead SVELTE adds to the RPL network. Raza et. al. set the default probing interval length to 2 minutes [30]. Given this default probing interval length, SVELTE adds around 0.5 KB of overhead every 2 minutes. Figure 10c shows the network overhead incurred by SVELTE+. An event watchdog sends a message to the border router each time it hears a rank advertisement (i.e., when a device sends a DIO message) from a neighboring device, causing the security manager to probe each neighboring device to the device that advertised the rank. The three main events that trigger rank advertisements are when a device first enters a network, when a device's trickle timer expires, and when a device wants to advertise a new rank. Therefore, SVELTE+ incurs more overhead at the beginning when the network is unstable, but incurs much less overhead once the network stabilizes, as compared to SVELTE.

Lastly, we analyze network overhead on a large scale through simulation. The results are shown in Figure 11a. At 100 devices, SVELTE+ incurs about 200 KB more in network overhead than no defense, while SVELTE incurs

Just Accepted



(a) Overhead **(b) Detection latency**

Fig. 12. The effect of probing interval on network overhead and detection latency for SVELTE.

Table 2. Summary of key findings for the DDoS attack case study.

Metrics	Key Findings
Retransmissions	With a large W, D-WARD may require more than 500 times the number of retransmissions than D-WARD+, and with a small W, D-WARD may require more than 10 times the number of retransmissions than D-WARD+.
Connection Duration	When fdec is low and W is high, D-WARD can cause up to 7 times longer connection duration than D-WARD+, and when fdec is high and W is low, connection duration under D-WARD+ is only at most 3 seconds faster than under D-WARD.
Energy Consumption	Under D-WARD, energy consumption increases linearly with respect to window size, while under D-WARD+ energy consumption is static across varying window sizes and contributes to less than 1 mJ of extra energy consumption for a benign device. This leads 15.55 less batteries consumed every year under D-WARD+.
Memory Consumption	On average, D-WARD consumes 10 times more memory consumption than D-WARD+.

about 1500 KB more. Most importantly, an increase in the number of devices has a much larger impact on overhead for SVELTE than SVELTE+. Therefore, we argue that SVELTE+ is far more scalable than SVELTE.

6.2.2 Energy Consumption. Figure 11b shows the estimated energy consumption incurred by SVELTE and SVELTE+. We again use the analysis presented by Feeney et al. [12] to help generate this graph. Clearly, network overhead has a direct impact on energy consumption. This is because it takes energy to send and receive packets. Therefore, similar to network overhead, SVELTE+ may cause devices to consume energy when the network is unstable, but as the network begins to maintain stability, SVELTE+ incurs little to no energy consumption. SVELTE's periodic probing, on the other hand, causes devices to perpetually incur energy costs. Furthermore, the number of devices, again, has a larger effect on energy consumption for SVELTE than SVELTE+.

We can observe that, as noted in Section 2.1, $P_{rd} < P_{vd}$ is true under SVELTE+, as a device is required to send and receive additional packets under vigilant mode. Next, we observe that as the network stabilizes, our assumption that $\sum_{i=1}^{n} t_{ri} >> \sum_{i=1}^{n} t_{vi}$ is also true. Finally, $P_{rd} << P_{xd} \leq P_{vd}$ is true when comparing SVELTE+'s two modes with SVELTE's one mode, and thus $E_{two-mode} < E_{one-mode}$ holds true in this case study.

6.2.3 Detection Latency. We analyze detection latency by measuring the difference in the amount of time it takes SVELTE to detect an attack as compared to SVELTE+. When considering detection latency in our simulations, we did not take into account the negligible round-trip time (RTT) between the border router and each device or processing time at the border router.

Figure 12 illustrates the relationship between network overhead and detection latency for SVELTE. In Figure 12a, network overhead *increases* exponentially as the probing interval *decreases*. In Figure 12b, detection latency *increases* linearly as the probing interval *increases*. Note that unlike in Figure 12a, the number of devices has no impact on detection latency. Unlike SVELTE+, SVELTE must strike a balance between network overhead and detection latency.

The detection latency in SVELTE+ instead is negligible since it immediately responds to a sinkhole attack on-demand. It incurs only some communication delay and processing time for the event watchdog to report a rank advertisement and for the security engine to verify that the advertisement is inconsistent and malicious. Here, the on-demand probing method employed by SVELTE+ is a clear advantage over the periodic probing of SVELTE.

Just Accepted

Table 3. Summary of key findings for the sinkhole attack case study.

Metrics	Key Findings
Network Overhead	For 100 devices, SVELTE+ incurs about 200 KB more in network overhead than no defense, while SVELTE incurs about 1500 KB more, and an increase in the number of devices has a much larger impact on overhead for SVELTE than SVELTE+.
Energy Consumption	For 100 devices, SVELTE+ incurs around 2000 mJ in energy consumption, SVELTE incurs 5000 mJ.
Detection Latency	For SVELTE, due to probing, detection latency increases linearly as the probing interval increases, while the detection latency for SVELTE+ is negligible because it immediately responds to attacks on-demand.

6.3 Evaluation Summary

The various metrics measured and analyzed for the DDoS and sinkhole case studies clearly show that the two-mode design of D-WARD+ and SVELTE+ help make them more suitable in an IoT environment as compared to their counterparts, D-WARD and SVELTE. Table 2 and Table 3 show summaries of the key findings for both case studies. For D-WARD+, not throttling any connections in regular mode, the signal mechanism in vigilant mode, and only throttling when a connection is labeled bad, allows for the reduction in retransmissions and connection duration, which has the positive side effect of reducing overall energy consumption in the network. Furthermore, only storing limited information in regular mode, allows D-WARD+ to significantly reduce its overall memory consumption, as compared to D-WARD. For SVELTE+, allowing devices to behave as if there was no system running in regular mode, and conducting on-demand probing in vigilant mode, allows for the reduction in network overhead and detection latency. Again, the reduction in network overhead in turn reduces overall energy consumption in the network. In conclusion, with D-WARD\D-WARD+ and SVELTE\SVELTE+ as proof, the TWINKLE framework can transform a security application for the smart home environment, into one that achieves equal to better defense efficacy than classical security applications, while consuming significantly less resources.

7 DISCUSSION

One factor in the feasibility of deploying TWINKLE is the potential difficulty of installing components on a smart home's border router. We assume that the border router has enough resources to run TWINKLE's security manager and security engine components. While this may be a safe assumption to make for many commercial home routers, we have yet to evaluate this claim. Also, the feasibility of running TWINKLE on routers is highly dependent on the security application. We show in the DDoS case study that D-WARD+ significantly reduces the memory consumption at the router. SVELTE+, similar to SVELTE, consumes at most 5 KB of RAM (50 KB of ROM) at the router in a smart home containing 16 devices.

Another issue is the feasibility of running event watchdog code on devices in the smart home. We assume that an event watchdog device can run various lightweight processes. However, some devices, such as legacy and extremely resource-constrained devices, may not have the ability to install even lightweight processes. Therefore, in some cases, additional devices need to be added to the network to act as event watchdogs. Furthermore, event watchdogs are required to have enough resources to run the lightweight SBDRs. Again, the feasibility of running SBDRs is dependent on the security application. SVELTE+ consumes at most 0.3 KB of RAM (1.4 KB of ROM) at each device while D-WARD+ does not run any code on the devices.

The placement of event watchdog nodes is another important issue that a network administrator must consider. Event watchdog nodes may be placed in multiple locations in the network in scenarios where suspicious behaviors cannot be detected at a central location, such as the border router. In such scenarios, selecting an effective placement strategy for the event watchdog nodes is paramount to effectively detect and mitigate a potential attack. However, selecting an effective placement strategy is not trivial. This event watchdog placement problem can be represented as a vertex-cover problem, where the constraint is the number of devices that can run the event watchdog code plus the number of event watchdog specific nodes that the network administrator

Just Accepted

Table 4. Comparison of frameworks and systems related to TWINKLE.

Paper	Design Philosophy	Targeted Environment	Real-World Evaluation	Resource Consumption
Bernabe et al. [5]	privacy preservation through contextual management	social IoT	no	not studied
Abie et al. [4]	game theory and risk analysis	eHealth	no	not studied
Celik at al. [8]	policy-based enforcement	trigger-action platforms	yes	not studied
Simpson et al. [34]	entralized and extensible security manager	smart home	yes	not studied
Kang et al. [19]	access control techniques	smart home	no	not studied
Rahmati et al. [28]	risk-based permission	smart home	yes	not studied
Rathore et al. [29]	leveraging deep learning and blockchain technology	5G-enabled IoT	yes	not studied
Blazy et al. [6]	attribute-based security	publish/subscribe systems	no	not studied
Sikder et al. [33]	context-aware security	smart home	yes	not studied
TWINKLE	**two-mode paradigm**	**smart home**	**yes**	**studied**

can add into the network, and the objective is to maximize the number of nodes that are within transmission range of an event watchdog node.

8 BACKGROUND & RELATED WORK

8.1 Smart Home Security Analysis

We begin by studying papers that explore the current state of smart home security and provide suggestions on improvements in this environment. Denning et al. [11] group security and privacy goals into three categories: device goals (device privacy, device availability, command authenticity, and execution integrity), data goals (data privacy, data integrity, and data availability), and environment goals (environment integrity, activity pattern privacy, sensed data privacy, sensor validity, and sensor availability). Notra et al. [26] report vulnerabilities in various household devices, such as the Phillips Hue light-bulb, the Belkin WeMo power switch, and the Nest smoke-alarm. Sivaraman et al. [36] also analyze various smart home devices and rate them in terms of confidentiality, integrity, access control, and their ability to launch reflection attacks. Mare et al. [22] evaluate seven popular smart home platforms, mainly focusing on the extent these platforms support access control, privacy, and automation. Similarly, Celik et al. [7] study the security and privacy of five popular IoT programming platforms through program analysis. The main contribution of [13] is the discovery of security-critical design flaws in the SmartThings capability model and event subsystem. Fernandes et al. [14] introduce a security principle that prevents an attacker from misusing compromised OAuth tokens of trigger-action platforms. These papers give insight into the vulnerabilities and open issues that need to be addressed by smart home security frameworks and systems. Of the seven papers, only [26] and [14] provide security solutions. However, [26] only provides protection via access control rules deployed at the gateway router to prevent unauthorized in-bound and out-bound traffic, and [14] provides a solution to a very specific vulnerability of trigger-action platforms.

Our framework, not only monitors traffic leaving and entering the network, but also monitors device to device communication from within the network. This allows our framework to potentially detect and prevent attacks that cannot be detected or prevented solely at the gateway router. Our framework is also generic in that a network administrator can plug various security applications into it, allowing it to handle various vulnerabilities (including, but not limited to, trigger-action platform vulnerabilities).

8.2 Frameworks and Systems

In this subsection, we survey select papers which introduce security frameworks and systems targeted towards IoT environments. Table 4 summarizes the differences between each of the studied frameworks and systems, and provides a comparison with TWINKLE. In [5], the authors present a security framework based on the Architecture Reference Model (ARM) of the IoT-A EU project. The work in [4], uses game theory and context-aware techniques to create a risk-based adaptive security framework for IoT in an eHealth environment. Both [5] and [4] are proof-of-concept papers that do not provide evidence that the presented frameworks are viable in resource constrained environments.

Just Accepted

The authors of [8] present a policy-based enforcement system that prevents insecure device states that may occur in trigger-action platforms. Again, unlike our framework, this system is targeted towards the specific area of trigger-action platforms and cannot be applied to securing the generic IoT smart home. Furthermore, this system does not concern itself with reducing resource consumption which is a key aspect of our framework.

Similar to our framework, the frameworks presented in [34], [19], and [28] are targeted towards smart home environments. Simpson et al. [34] present a centralized security manager, similar to the security manager component in our framework, whose main purpose is to provide reliable patching and update mechanisms to smart homes. Kang et al. [19] present a security framework that requires kernel-level modifications to provide authentication and access control mechanisms for smart home appliances. Rahmati et al. [28] introduce a secure development methodology which leverages risk-based permissions for IoT networks, instead of permission models used by smart phone operating systems; unlike TWINKLE, this work attempts to improve smart home security by focusing solely on the permission model aspect of IoT devices. However, like [5], [4], and [8] the authors of these three papers targeted to smart home environments do not address the limitations of IoT devices nor provide evaluation results for the resource costs of deploying their solutions.

More recently, the security community has introduced frameworks that leverage concepts such as deep learning, blockchain, attribute-based security, and context-aware security, specifically for securing IoT networks. For example, Rathore et al. [29] propose a deep learning and blockchain-empowered security framework that supports security operations across 5G-enabled IoT networks. In other words, machine learning and blockchain-based security solutions are deployed at the fog and edge computing layers to create a secure environment for 5G-enabled IoT networks. While the authors set up a real-world environment to test their framework, they do not provide any analysis on how their framework affects resource consumption on the 5G-enabled devices. Blazy et al. [6] present a security framework for topic-based publish/subscribe systems which strives to achieve three main goals: subscription confidentiality, publication confidentiality, and payload confidentiality. The framework achieves these goals by leveraging attribute-based cryptography, specifically Attribute-Based Encryption (ABE), Attribute-Based Keyword Search (ABKS), and Attribute-Based Signature (ABS). The authors evaluated their framework through simulations using virtual machines and do not provide any analysis on resource consumption. Sikder et al. [33] introduce AEGIS+, which is a context-aware and platform-independent security framework for the smart home environment. AEGIS+ captures the co-dependence between devices to discern different user patterns to build a contextual model in order to differentiate between malicious and benign behavior. Because Aegis+ is not installed on individual devices, but rather a dedicated central entity, and is only concerned with detecting malicious behavior and not mitigation, it does not consume resources on the devices. However, the centralized design of Aegis+ makes it susceptible to malicious behavior that cannot be detected at a central location (such as sinkhole-type attacks).

In contrast to all of the aforementioned papers, TWINKLE's primary focus is to reduce resource consumption while maintaining a secure environment. Furthermore, we show that our framework can reduce resource consumption through the evaluation of two concrete case studies, which include testing on real-world devices and networks.

8.3 Security Mechanisms for Edge Computing

Because IoT and edge computing are closely related, we analyze existing security mechanisms for edge computing. Specifically, we focus on identification and authentication, which are two well-studied problems in edge computing.

In IP networks, the two main roles an IP address serves are as locator and end point identifier. The Host Identity Protocol (HIP) [24] is a host identification protocol that decouples these two roles by introducing a Host Identity (HI) name space as an end point identifier which is based on public key cryptography. One of the security advantages HIP provides is that it prevents machines on the Internet from directly accessing IoT devices without

Just Accepted

passing the strict security procedure of mutual peer authentication via Sigma-compliant Diffie-Hellman key exchange.

In fact, the preferred way of implementing HIP in edge networks is to use Internet Protocol Security (IPsec) [20] to carry the data traffic. IPsec is a network protocol suite that is used to authenticate and encrypt packets, thereby providing secure communication between two machines in an IP network. Specifically, the only defined method for implementing HIP is to use IPsec's Encapsulated Security Payload (ESP) to carry the data packets. When used in combination, HIP and IPsec not only provides data authentication, integrity, and confidentiality, but allows for secure IP multihoming and mobile computing.

Protocols, such as HIP and IPsec, which enhance the security of communications between IoT devices, are orthogonal to the security benefits that TWINKLE provides. Because HIP and IPsec only deal with the specific problems of host identification, and data authentication, integrity, and confidentiality, TWINKLE, which can handle a plethora of attacks, can be leveraged in combination with HIP and IPsec to provide a more secure environment for IoT networks.

8.4 Motivation for the TWINKLE Design and Possible Extensions

Lastly, we analyze papers that motivate certain design choices and components of TWINKLE. Instead of introducing new security countermeasures, the authors of [18] attempt to strengthen security for smart home networks by making it easier for non-expert home owners to set up secure networks and intuitively manage trust and access to their devices. The research in [25] attempts to provide adequate mechanisms to control the flow of data and enforce policies based on users' preferences. Such work motivates the need for TWINKLE's automated component instantiation which allows users to intuitively and easily plug existing security applications into the framework without having any knowledge about the inner-workings of those applications.

The TWINKLE framework can be extended to include additional features that may work well with the two-mode paradigm, and further increase its defense efficacy and resource efficiency. In [9], the authors utilize special nodes that monitor traffic within the network to detect certain routing attacks. The work in [18], [25], and [9] show the need of user interaction, adjustable policies set by users, and dedicated event watchdog nodes for inspection of in-network communication, respectively. Also, the work in [21] provides motivation for allowing security policies, such as using efficient authentication and key agreement methods. He et al. [15] conclude that per-device granular access control policies are not sufficient, and instead, a combination of per-capability, per-relationship, and per-context granular access control policies are needed. TWINKLE can be extended to allow the user to specify these types of access control policies, and how these policies may change depending on the mode. Furthermore, the substantial research in the area of security in wireless sensor networks (WSNs), such as the work presented by Abduvaliyev et al. [3] and Roman et al. [31], where devices are extremely constrained, can be leveraged to further improve TWINKLE's resource efficiency.

9 CONCLUSION

The staggering growth of the Internet of Things (IoT) brings serious security concerns. However, due to the constrained resources of IoT devices and their networks, many classical security applications become ineffective or inapplicable in an IoT environment. Using the smart home as the battleground, this paper implements a security framework called TWINKLE that endeavors to address a fundamental dilemma facing any security solution for IoT: the solution must consume as little resources as possible while still aspiring to achieve the same level of performance as if the resources needed are abundant. It introduces a two-mode design to enable security applications plugged into the framework to handle their targeted attacks in an on-demand fashion. Every security application can simply run lightweight operations in regular mode most of the time, and only invoke heavyweight security routines when it needs to cope with suspicious behavior.

Just Accepted

This paper elaborates the philosophy behind the two-mode design and detailed how different components in the framework interact with each other. Furthermore, it discusses the critical challenges in implementing TWINKLE in an IoT environment and explained in detail how each challenge is addressed, including how security applications can be represented, how each component can be automatically instantiated, and how the components interact with each other after instantiation. Last but not least, the paper includes detailed studies and evaluations in applying TWINKLE to distributed denial-of-service (DDoS) and sinkhole attacks, showing that previous security solutions can be successfully transformed into effective but more resource-efficient versions using TWINKLE.

ACKNOWLEDGMENTS

This project is in part the result of funding provided by the Science and Technology Directorate of the United States Department of Homeland Security under contract number D15PC00204. The views and conclusions contained herein are those of the authors and should not be interpreted necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security or the US Government.

REFERENCES

[1] 2019. Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/.
[2] 2019. Suricata - Open Source IDS / IPS / NSM engine. https://suricata-ids.org/.
[3] Abror Abduvaliyev, Al-Sakib Khan Pathan, Jianying Zhou, Rodrigo Roman, and Wai-Choong Wong. 2013. On the Vital Areas of Intrusion Detection Systems in Wireless Sensor Networks. *IEEE Communications Surveys & Tutorials* 15, 3, 1223–1237.
[4] Habtamu Abie and Ilangko Balasingham. 2012. Risk-based Adaptive Security for Smart IoT in eHealth. In *Proceedings of the 7th International Conference on Body Area Networks*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 269–275.
[5] Jorge Bernal Bernabe, Jose Luis Hernandez, M Victoria Moreno, and Antonio F Skarmeta Gomez. 2014. Privacy-Preserving Security Framework for a Social-aware Internet of Things. In *International Conference on Ubiquitous Computing and Ambient Intelligence*. Springer, 408–415.
[6] Olivier Blazy, Emmanuel Conchon, Mathieu Klingler, and Damien Sauveron. 2021. An IoT Attribute-Based Security Framework for Topic-Based Publish/Subscribe Systems. *IEEE Access* 9 (2021), 19066–19077.
[7] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2018. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *arXiv preprint arXiv:1809.06962*.
[8] Z Berkay Celik, Gang Tan, and Patrick McDaniel. 2019. IOTGUARD: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Network and Distributed System Security Symposium (NDSS)*.
[9] Christian Cervantes, Diego Poplade, Michele Nogueira, and Aldri Santos. 2015. Detection of Sinkhole Attacks for Supporting Secure Routing on 6LoWPAN for Internet of Things. In *IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, 606–611.
[10] Tony Cheneau. 2013. SimpleRPL. https://github.com/tcheneau/simpleRPL.
[11] Tamara Denning, Tadayoshi Kohno, and Henry M Levy. 2013. Computer Security and the Modern Home. *ACM Communications* 56, 1, 94–103.
[12] Laura Marie Feeney and Martin Nilsson. 2001. Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, Vol. 3. IEEE, 1548–1557.
[13] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security Analysis of Emerging Smart Home Applications. In *IEEE Symposium on Security and Privacy*. IEEE, 636–654.
[14] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized Action Integrity for Trigger-action IoT Platforms. In *Network and Distributed Security Symposium (NDSS)*.
[15] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. 2018. Rethinking Access Control and Authentication for the Home Internet of Things (IoT). In *27th USENIX Security Symposium*. 255–272.
[16] Scott Hilton. 2016. Dyn Analysis Summary of Friday October 21 Attack. https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack.
[17] Ionut Ilascu. 2018. IoT Botnets Responsible for More Powerful DDoS Attacks. https://www.bitdefender.com/box/blog/iot-news/iot-botnets-responsible-powerful-ddos-attacks/.

Just Accepted

[18] Dimitris N Kalofonos and Saad Shakhshir. 2007. IntuiSec: A Framework for Intuitive User Interaction with Smart Home Security using Mobile Devices. In *IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*. IEEE, 1–5.

[19] Won Min Kang, Seo Yeon Moon, and Jong Hyuk Park. 2017. An Enhanced Security Framework for Home Appliances in Smart Home. *Human-centric Computing and Information Sciences*, 1–6.

[20] S. Kent and R. Atkinson. 2015. RFC 2401: Security Architecture for the Internet Protocol. https://www.rfc-editor.org/rfc/rfc2401.html.

[21] Pardeep Kumar, An Braeken, Andrei Gurtov, Jari Iinatti, and Phuong Ha. 2017. Anonymous Secure Framework in Connected Smart Home Environments. *IEEE Transactions on Information Forensics and Security*, 968–979.

[22] Shrirang Mare, Logan Girvin, Franziska Roesner, and Tadayoshi Kohno. 2019. Consumer Smart Homes: Where We Are and Where We Need to Go. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. ACM, 117–122.

[23] Jelena Mirkovic and Peter Reiher. 2005. D-WARD: A Source-end Defense Against Flooding Denial-of-Service Attacks. *IEEE transactions on Dependable and Secure Computing* 2, 3, 216–232.

[24] R. Moskowitz and P. Nikander. 2015. RFC 4423: Host Identity Protocol Architecture. https://datatracker.ietf.org/doc/html/draft-ietf-hip-arch.

[25] Ricardo Neisse, Gary Steri, and Gianmarco Baldini. 2014. Enforcement of Security Policy Rules for the Internet of Things. In *IEEE 10th International Conference on Wireless and Mobile Computing*. IEEE, 165–172.

[26] Sukhvir Notra, Muhammad Siddiqi, Hassan Habibi Gharakheili, Vijay Sivaraman, and Roksana Boreli. 2014. An Experimental Study of Security and Privacy Risks with Emerging Household Appliances. In *IEEE Conference on Communications and Network Security*. IEEE, 79–84.

[27] Vern Paxson. 1999. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks* 31, 23-24 (1999), 2435–2463. http://www.icir.org/vern/papers/bro-CN99.pdf

[28] Amir Rahmati, Earlence Fernandes, Kevin Eykholt, and Atul Prakash. 2018. Tyche: A Risk-based Permission Model for Smart Homes. In *IEEE Cybersecurity Development (SecDev)*. IEEE, 29–36.

[29] Shailendra Rathore, Jong Hyuk Park, and Hangbae Chang. 2021. Deep Learning and Blockchain-Empowered Security Framework for Intelligent 5G-Enabled IoT. *IEEE Access* 9 (2021), 90075–90083.

[30] Shahid Raza, Linus Wallgren, and Thiemo Voigt. 2013. SVELTE: Real-time Intrusion Detection in the Internet of Things. *Ad hoc networks* 11, 8, 2661–2674.

[31] Rodrigo Roman, Jianying Zhou, and Javier Lopez. 2006. Applying Intrusion Detection Systems to Wireless Sensor Networks. In *IEEE Consumer Communications & Networking Conference*. IEEE, 640–644.

[32] Anuj Sehgal, Vladislav Perelman, Siarhei Kuryla, and Jurgen Schonwalder. 2012. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine* 50, 12 (2012).

[33] Amit Kumar Sikder, Leonardo Babun, and A Selcuk Uluagac. 2021. Aegis+ A Context-Aware Platform-Independent Security Framework for Smart Home Systems. *Digital Threats: Research and Practice* 2, 1 (2021), 1–33.

[34] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. 2017. Securing Vulnerable Home IoT Devices with an In-hub Security Manager. In *IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE, 551–556.

[35] Devkishen Sisodia, Samuel Mergendahl, Jun Li, and Hasan Cam. 2018. Securing the Smart Home via a Two-Mode Security Framework. In *Proceedings of the 14th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*. Springer, 22–42.

[36] Vijay Sivaraman, Hassan Habibi Gharakheili, Clinton Fernandes, Narelle Clark, and Tanya Karliychuk. 2018. Smart IoT Devices in the Home: Security and Privacy Implications. *IEEE Technology and Society Magazine* 37, 2, 71–79.

[37] Julie Song. 2019. The Realities Of Smart City Development. https://www.forbes.com/sites/forbestechcouncil/2019/05/14/the-realities-of-smart-city-development.

[38] OSSEC Project Team. 2019. OSSEC: Open source HIDS SECurity. https://ossec.github.io/index.html.

[39] Linus Wallgren, Shahid Raza, and Thiemo Voigt. 2013. Routing Attacks and Countermeasures in the RPL-based Internet of Things. *International Journal of Distributed Sensor Networks* 9, 8, 1–11.

[40] T. Winter, P. Thubert, A. Brandt, J.W. Hui, and R. Kelsey. 2012. RFC 6550: RPL: IPv6 Routing Protocol for Low-power and Lossy Networks. https://tools.ietf.org/html/rfc6550.

Just Accepted