Automated Black-Box Security Testing of ~~Smart~~ Embedded Devices

Andrea Continella

UNIVERSITY OF TWENTE.

$whoami

Assistant Professor @ University of Twente - SCS group

Cybersecurity @ SCS

- Data Security
- Systems Security

Research Interests: Systems Security

- Malware Analysis & Defenses
- Threat Detection & Response
- Automated Security Testing & Patching

CTF Competitions

- Member of Shellphish & (previously) ToH & mhackeroni
- Mentor Twente Hacking Squad (THS)

Cooking today

Automated vulnerability research for smart embedded devices

- Challenges in firmware testing
- Black-box fuzzing
- Device firmware update
- Conclusions & future directions

Today’s IoT Landscape

SUPPLY CHAIN SECURITY

Nuki Smart Lock Vulnerabilities Allow Hackers to Open Door

Security re
attackers t
Nuki offers
walking in

By Ionut Arghire
July 27, 2022

IoT Botnets Fuel DDoS Attacks – Are You Prepared?

July 26, 2022 / 8:38 am

The Botnet That Broke the Internet Isn't Going Away

TechNewsWorld > Security > Privacy | **Next Article in Privacy**

Webcam Maker Takes FTC's for Internet-of-Things Secu Failure

20y old vulnerabilities are back!

Featuring
Stack Overflows
Gets()
Scanf()
ASLR who?

Firmware Testing: Challenges

Hardware-dependent

Unique, minimal **environments** with **non-standard** configurations

Several different architectures
- ARM, MIPS, x86, PowerPC, etc.
- Sometimes proprietary

Manage **external peripherals**, often using custom code

Firmware Testing

- Dynamic Analysis
  - Emulation, coverage-guided fuzzing, etc...
  - Currently **not generic**, too **unreliable**

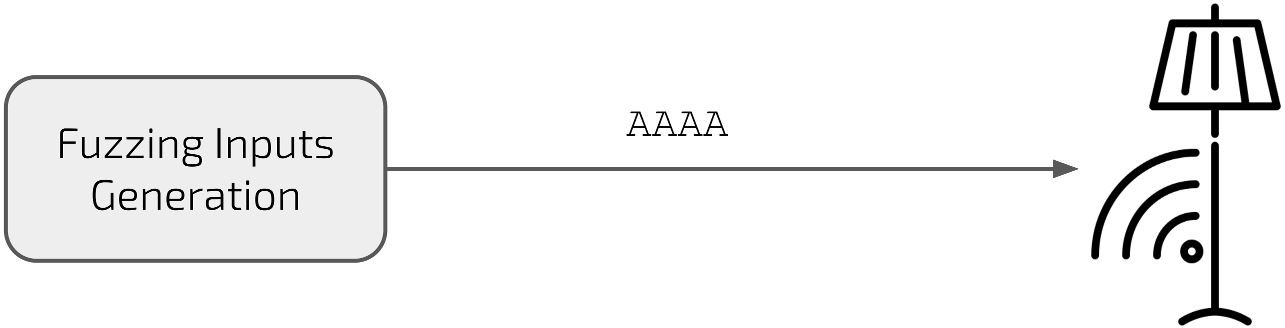
- Static Analysis
  - Too many **false positives**
  - Need to take into account the **multi-binary** aspect

KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware. *IEEE Symposium on Security & Privacy (S&P)*, 2020

What if we do not have access to the firmware image?

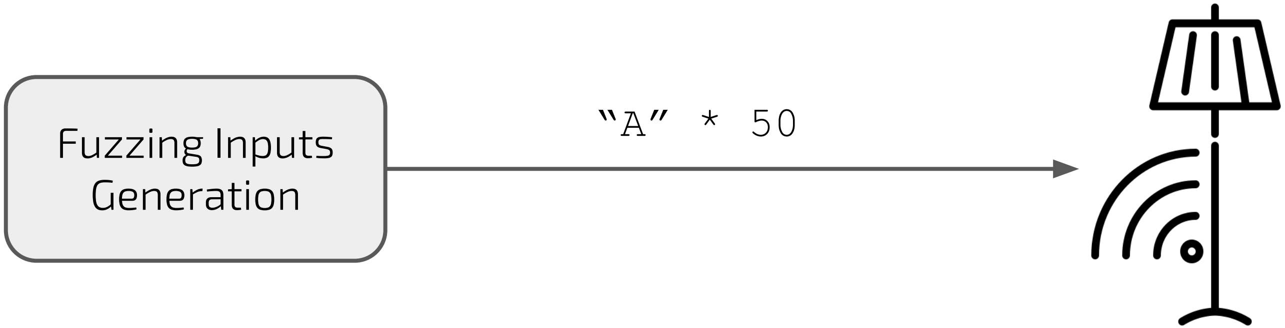
Black-box Fuzzing

Black-box Fuzzing




```
char data[100];
...
read(socket, buff, 500);
...
strcpy(data, buff);
```

Black-box Fuzzing

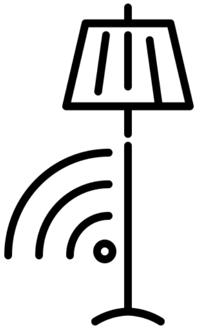



```
char data[100];
...
read(socket, buff, 500);
...
strcpy(data, buff);
```

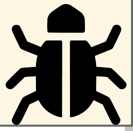
Black-box Fuzzing

Fuzzing Inputs Generation

"A" * 300



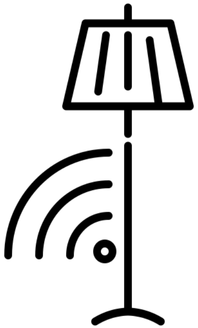
```
char data[100];
...
read(socket, buff, 500);
...
strcpy(data, buff);
```



Black-box Fuzzing

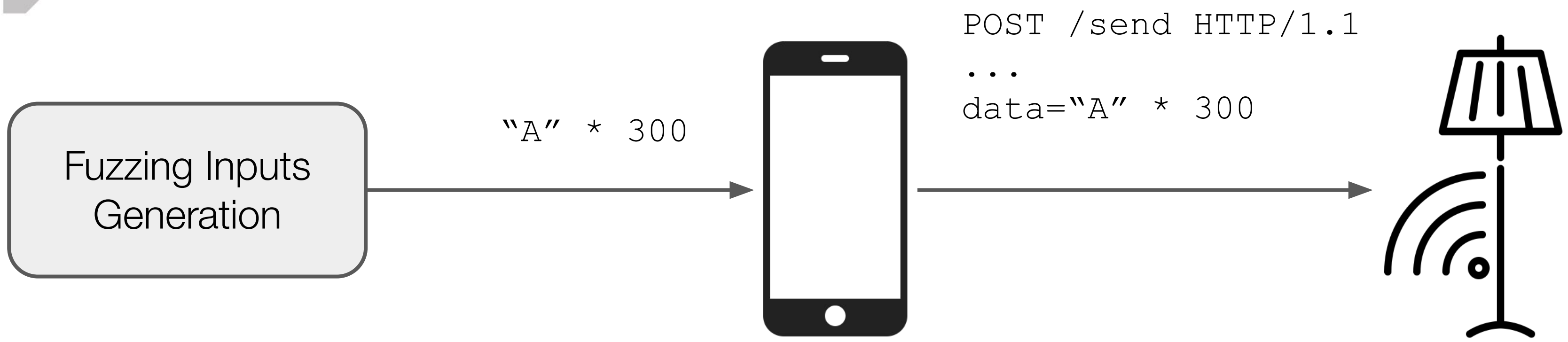
Fuzzing Inputs Generation

"A" * 300



```
char data[100];
...
read(socket, buff, 500);
if (!valid_http_req(buff))
    return;
...
strcpy(data, buff);
```

Smarter Black-box Fuzzing




```
char data[100];
...
read(socket, buff, 500);
if (!valid_http_req(buff))
    return;
...
strcpy(data, buff);
```

Smarter Black-box Fuzzing

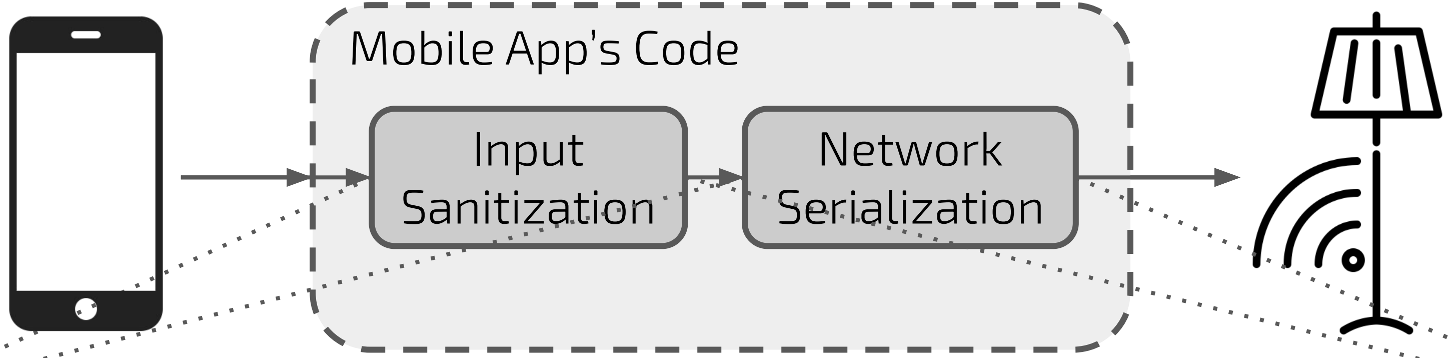
Black-box techniques require knowledge of the valid data format

IoTFuzzer uses companion apps to create fuzzing inputs

- Finds UI elements that generate network traffic
- Fuzzes functions' arguments containing UI data

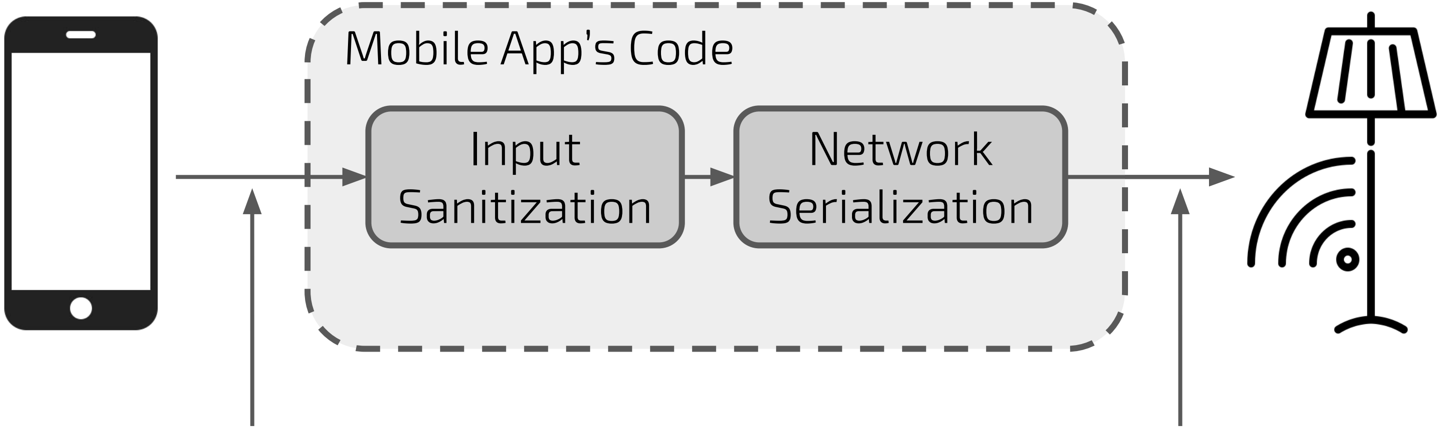
```
public void getBrFromUI(String val) {
      // ...
      process_brightness(val);
}

public void process_brightness(String msg) {
     byte[] cnt = encode(msg);
     send_to_device(cnt);
}
```

Fuzzing IoT Devices ©




```
...
String json = "{\"op\": \"auth\", \"pass\":" + adminPwd "}";
String encoded = Base64.encode(json);

httpSend(DEVICE_IP; encoded);
```

Fuzzing IoT Devices ©



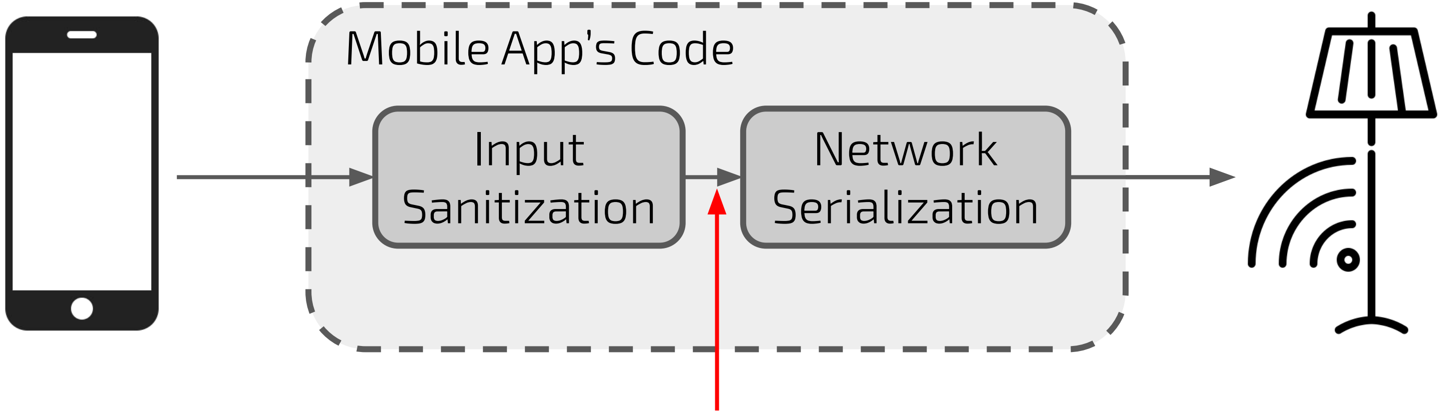
UI-level

Limited by app sanitization ❌

Network-level

Invalid inputs ❌

Fuzzing IoT Devices ©



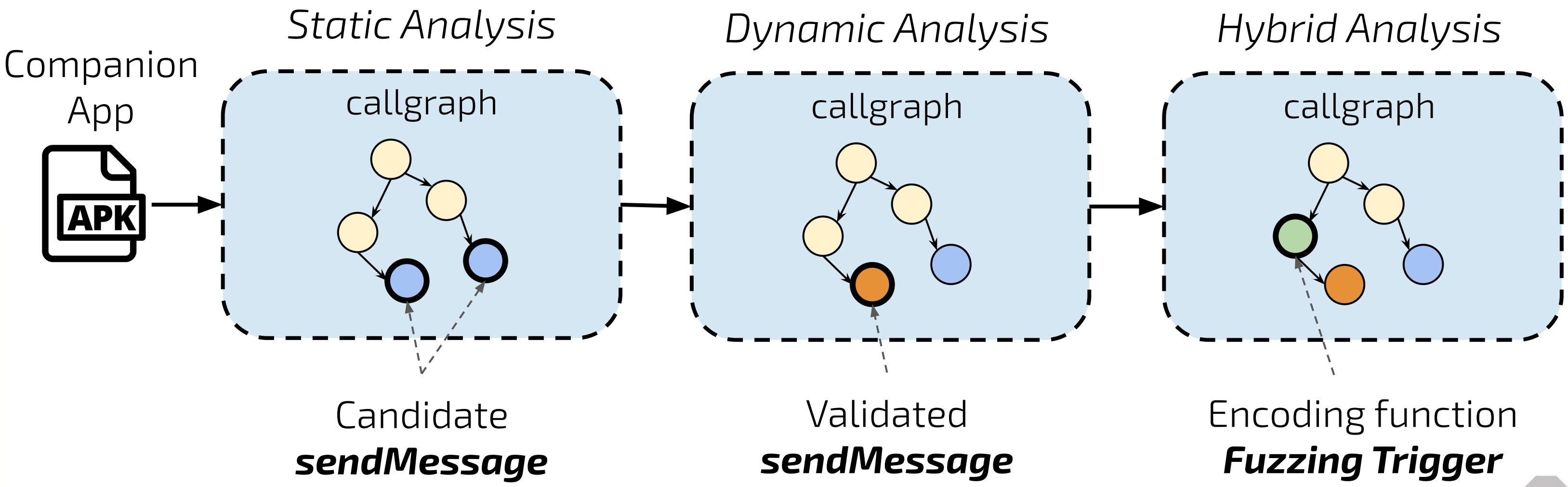
Our Approach

Valid inputs ✓

Not limited by app-side input sanitization ✓

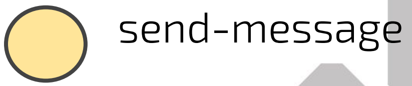
Diane: Overview

Fuzzing triggers: functions between app-side validation & data-encoding



Fuzzing Triggers

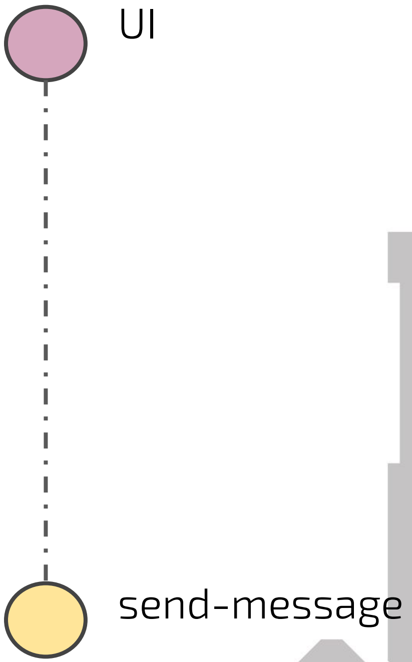
Bottom-up approach to identify fuzzing triggers



Fuzzing Triggers

Bottom-up approach to identify fuzzing triggers

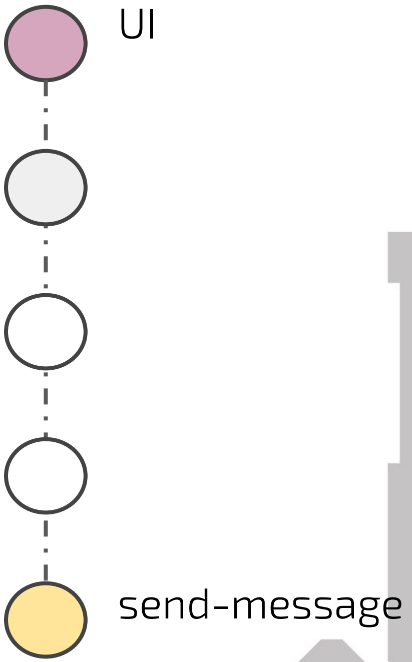
- Perform a backward slice up to the UI/input



Fuzzing Triggers

Bottom-up approach to identify fuzzing triggers

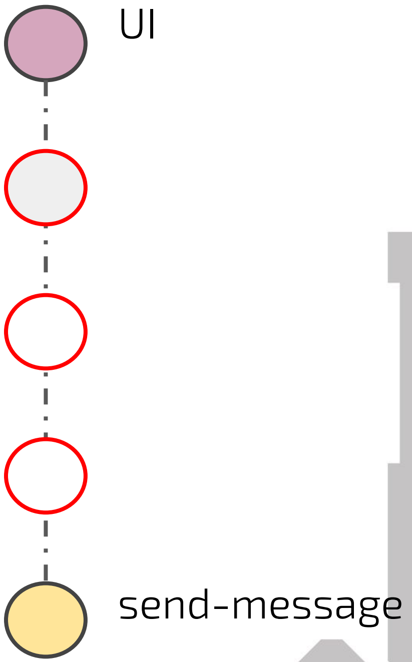
- Perform a backward slice up to the UI/input
- Identify traversed functions



Fuzzing Triggers

Bottom-up approach to identify fuzzing triggers

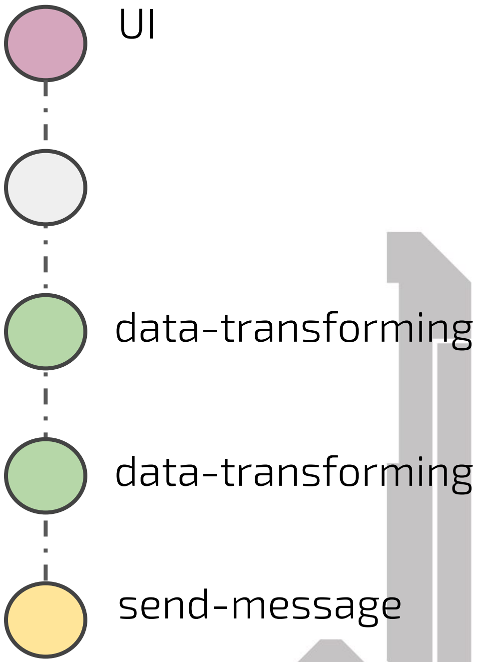
- Perform a backward slice up to the UI/input
- Identify traversed functions
- Dynamically hook funcs and calculate entropy



Fuzzing Triggers

Bottom-up approach to identify fuzzing triggers

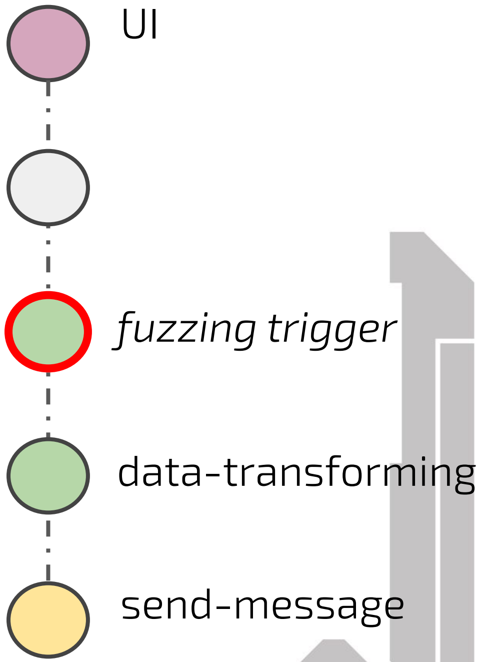
- Perform a backward slice up to the UI/input
- Identify traversed functions
- Dynamically hook funcs and calculate entropy
- Data-transforming funcs if increase entropy >= T



Fuzzing Triggers

Bottom-up approach to identify fuzzing triggers

- Perform a backward slice up to the UI/input
- Identify traversed functions
- Dynamically hook funcs and calculate entropy
- Data-transforming funcs if increase entropy >= T
- Identify data-transforming funcs not dominated by other data-transforming funcs (**fuzzing triggers**)



Example

```
public void setDeviceName(String oname) { // UI
      String name = substring(oname, 15);
      setDeviceInternal(name);
}

public byte[] encode(String s) {
      byte[] enc;
      // encode cmd
      return enc;
}

public byte[] setDeviceInternal(String name) {
      byte[] e = encode(name);
      return sendToDevice(e);
}

public byte[] sendToDevice(byte[] c) { /* … */ }
```

Example

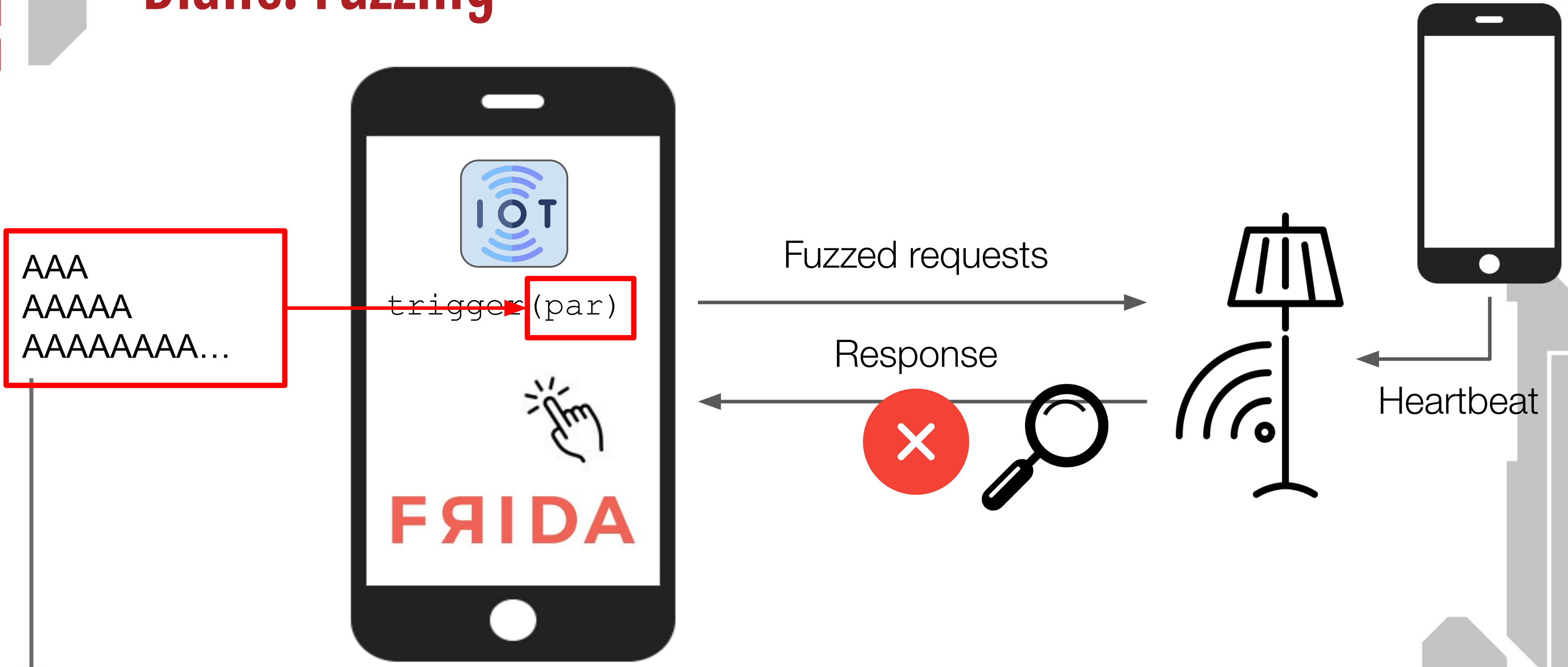
```
public void setDeviceName(String oname) { // UI
      String name = substring(oname, 15);
      setDeviceInternal(name);
}

public byte[] encode(String s) {
      byte[] enc;
      // encode cmd
      return enc;
}

public byte[] setDeviceInternal(String name) {
      byte[] e = encode(name);
      return sendToDevice(e);
}

public byte[] sendToDevice(byte[] c) { /* … */ }
```

Entropy < T

Entropy > T

Entropy < T

Diane: Fuzzing



Experimental Results

Tested on 11 IoT devices; different brands and categories

7/11 companion apps contain input sanitization

- On a larger scale, **663/1304 (~51%)** companions apps have sanitization

On the 11 companion app/devices

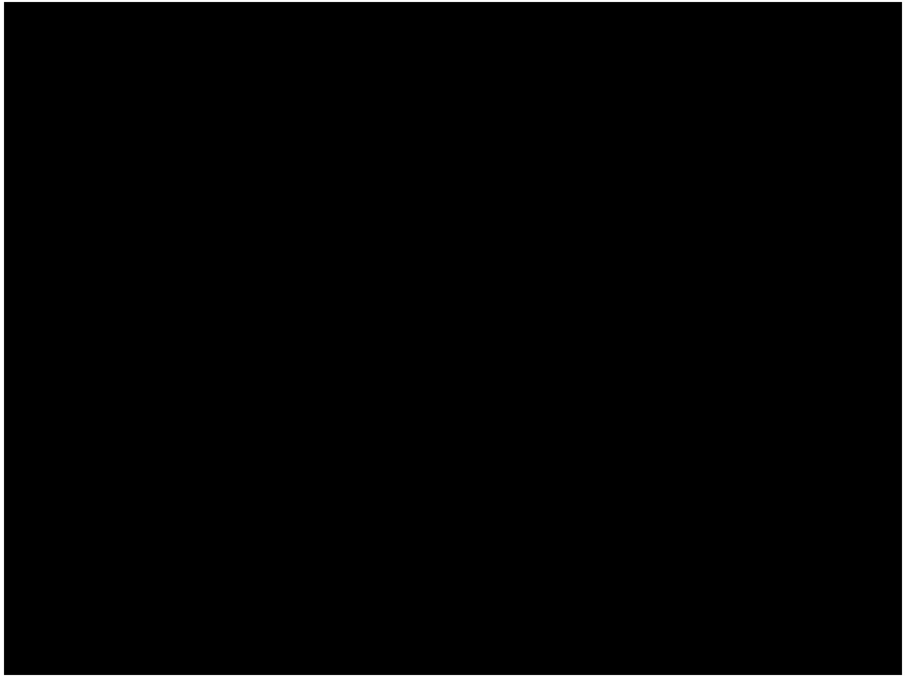
- Diane identified **54** fuzzing triggers
  - **5** false positives
  - **5** fuzzing triggers == send_message functions

Experimental Results

	DIANE					IoTFuzzer		
Device ID	No. Generated Alerts	No. Bugs	Zero-day	Vuln. Type	Time [hours] (No. Generated Inputs)	No. Fuzzed Functions	No. Bugs	Time [hours]
1	1	1	✓	Unknown	$\leq$ 0.5 (60,750)	• 1	0	N/A
2	3	7	✓	Buff overflow	$\leq$ 0.5 (322)	5	2	0.98
3	1	1		Unknown	$\leq$ 1.2 (7,344)	1	1	4
4	1	0		N/A	N/A	• 1	0	N/A
5	1	0		N/A	N/A	• 1	0	N/A
6	4	1		Unknown	$\leq$ 10 (34,680)	1	1	$\leq$ 10
7	3	0		N/A	N/A	N/A	N/A	N/A
8	3	0		N/A	N/A	N/A	N/A	N/A
9	0	0		N/A	N/A	3	0	N/A
10	1	0		N/A	N/A	N/A	N/A	N/A
11	0	† 1	✓	Unknown	2.2 (3,960)	N/A	N/A	N/A

* All bugs were responsibly disclosed following the community guidelines

Use case: Popular Smart Lock



Research Outcomes

DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices
N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, G. Vigna
In Procs. of the IEEE Symposium on Security & Privacy (S&P), 2021

DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices

Nilo Redini*, Andrea Continella†, Dipanjan Das*, Giulio De Pasquale*, Noah Spahn*, Aravind Machiry‡, Antonio Bianchi‡, Christopher Kruegel*, and Giovanni Vigna*
*UC Santa Barbara †University of Twente ‡Purdue University
{nredini, dipanjan, peperunas, ncs, chris, vigna}@cs.ucsb.edu
a.continella@utwente.nl, {amachiry, antoniob}@purdue.edu

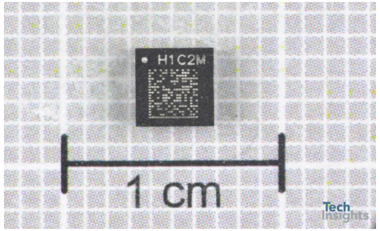
***Abstract*—Internet of Things (IoT) devices have rooted themselves in the everyday life of billions of people. Thus, researchers have applied automated bug finding techniques to improve their overall security. However, due to the difficulties in extracting and emulating custom firmware, black-box fuzzing is often the only viable analysis option. Unfortunately, this solution mostly produces invalid inputs, which are quickly discarded by the targeted IoT device and do not penetrate its code. Another proposed approach is to leverage the companion app (i.e., the mobile app typically used to control an IoT**

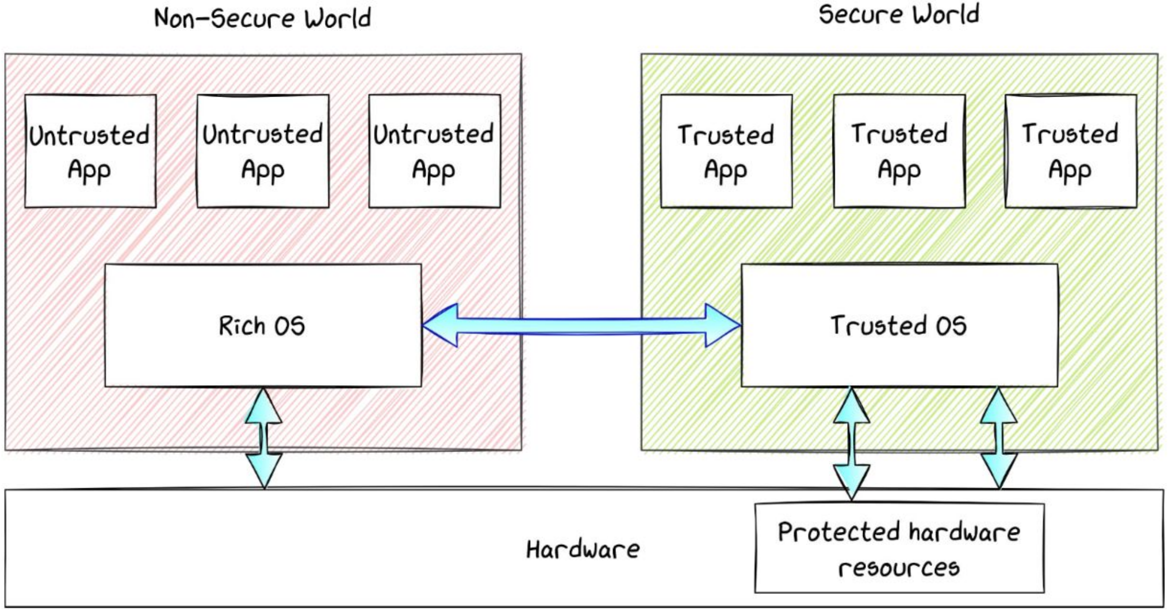
however, present several limitations. First, obtaining the firmware running on an IoT device is difficult: Extracting the firmware from a device typically requires *ad hoc* solutions, and vendors hardly make their software publicly available [70]. Second, unpacking and analyzing a firmware sample is a challenging task: Firmware samples may be available in a variety of formats, and may run on several different architectures, often undocumented. Furthermore,

Other targets?

Google Titan M Chip

External Coprocessor: Trusted Execution Environment (TEE)





Fuzzing Titan M

Table 1: Results of fuzzing the Titan M firmware, version *0.0.3/brick_v0.0.8232-b1e3ea340*

Task	Command	Bug	Detection	Return code	Avg. # of messages
Identity	`ICPushReaderCert`	Buffer overflow	Chip reboots	2	74
Identity	`ICsetAuthToken`	Buffer overflow	Stack canary	2	475
Identity	`WICaddAccessControlProfile`	Null-pointer dereference	Chip halts	4	57
Identity	`WICbeginAddEntry`	Null-pointer dereference	Chip halts	4	99
Identity	`WICfinishAddingEntries`	Null-pointer dereference	Chip halts	4	82
Identity	`ICstartRetrieveEntryValue`	Null-pointer dereference	Chip halts	4	105
Keymaster	`FinishAttestKey`	N/A	Chip reboots	2	257
Keymaster	`IdentityFinishAttestKey`	N/A	Chip reboots	2	192

Table 2: Results of fuzzing the Titan M firmware, version *0.0.3/brick_v0.0.8292-b3875afe2*

Task	Command	Bug	Detection	Return code	Avg. # of messages
Identity	`WICfinishAddingEntries`	Null-pointer dereference	Chip halts	4	72
Identity	`ICstartRetrieveEntryValue`	Null-pointer dereference	Chip halts	4	126

* All bugs were responsibly disclosed following the community guidelines

Research Outcomes

Reversing and Fuzzing the Google Titan M Chip
Damiano Melotti, Maxime Rossi-Bellom, Andrea Continella
In Procs. of the Reversing and Offensive-oriented Trends Symposium (ROOTS), 2021

Reversing and Fuzzing the Google Titan M Chip

Damiano Melotti
University of Twente & Quarkslab
dmelotti@quarkslab.com

Maxime Rossi-Bellom
Quarkslab
mrossibellom@quarkslab.com

Andrea Continella
University of Twente
a.continella@utwente.nl

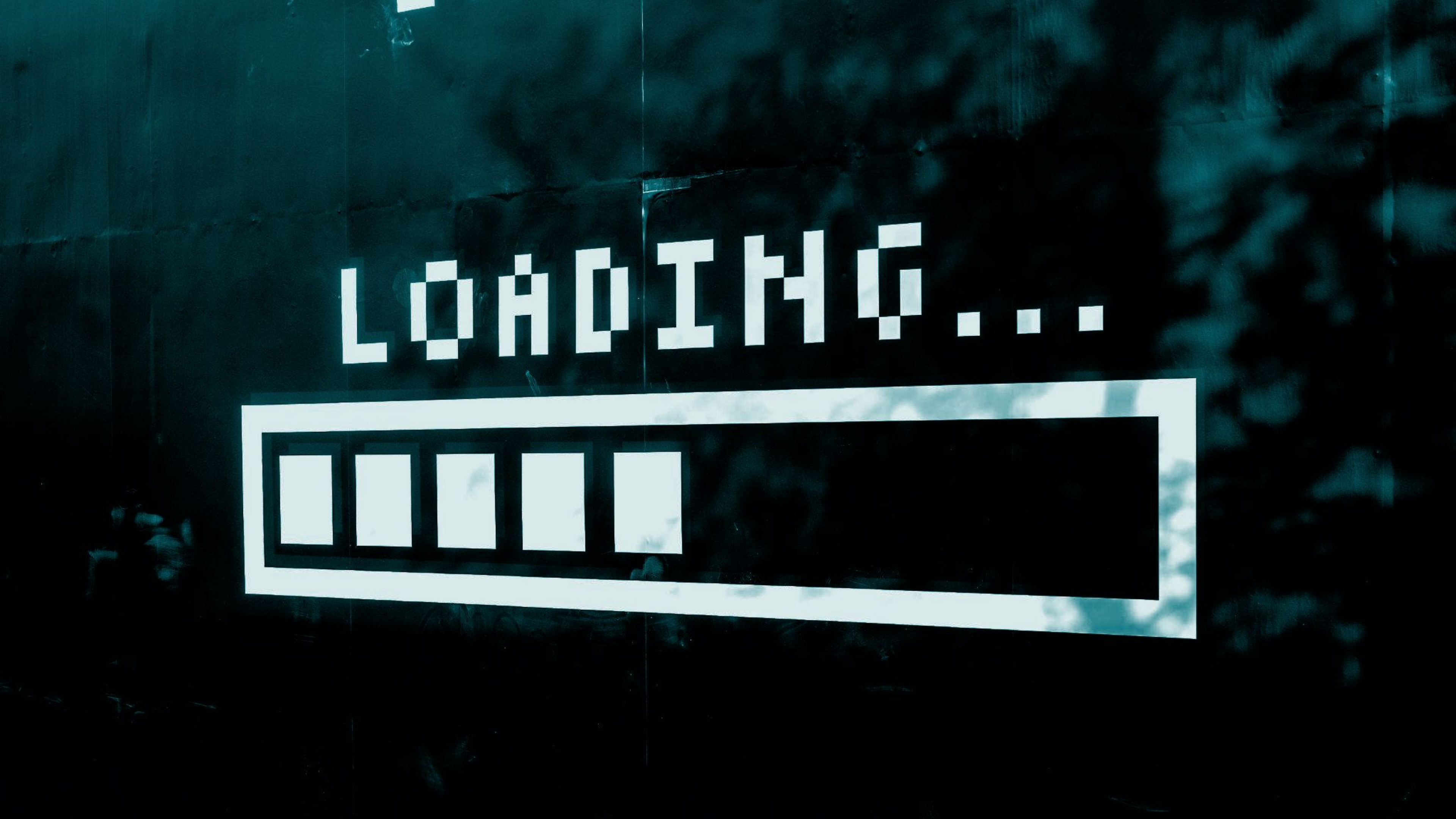
ABSTRACT

Google recently introduced a secure chip called Titan M in its Pixel smartphones, enabling the implementation of a Trusted Execution Environment (TEE) in Tamper Resistant Hardware. TEEs have been proven effective in reducing the attack surface exposed by smartphones, by protecting specific security-sensitive operations. However, studies have shown that TEE code and execution can also be targeted and exploited by attackers, therefore, studying their security lays the basis of the trust we have in their features.

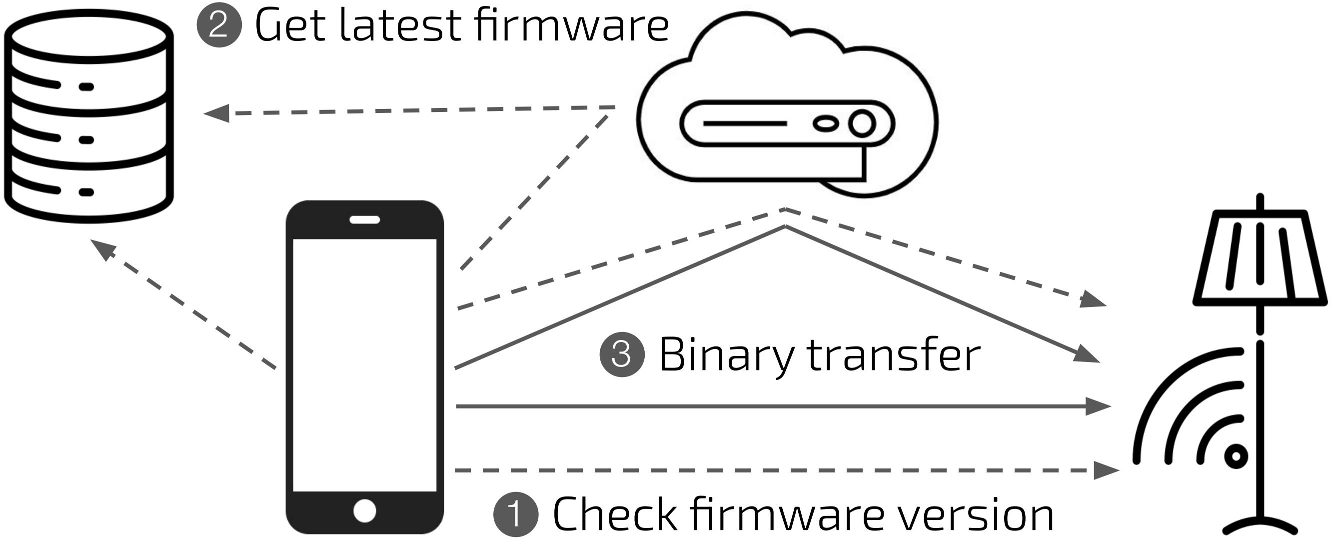
In this paper, we provide the first security analysis of Titan M. First, we reverse engineer the firmware and we review the open source code in the Android OS that is responsible for the communication with the chip. By exploiting a known vulnerability, we then dynamically examine the memory layout and the internals of the chip. Finally, leveraging the acquired knowledge, we design and

Deploying security measures at the hardware level is not new, as described in Section 2. However, it is not so common for mobile devices to have a dedicated chip, physically separated from the main CPU, implementing a Trusted Execution Environment (TEE) and ensuring tamper-resistant properties.

When the chip was announced, Google reported that its firmware would be open source [33]. To date, no source code has been published and not much information is available about the internals of this chip. Despite that, to motivate researchers into investigating this module, Google introduced a special reward of one million dollars for whoever can find a full-chain remote code execution exploit with persistence [27]. Indeed, Titan M represents the so-called *Root of Trust* of a device, the baseline all security features rely upon: in case of compromise, the target falls completely under the attacker's control.


LOADING...

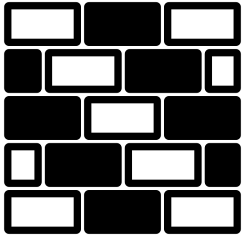
IoT Device Firmware Update (DFU)



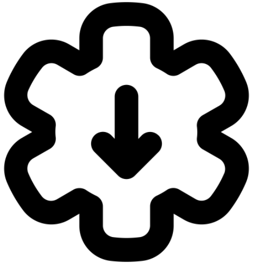

What could
possibly go wrong?

Threats

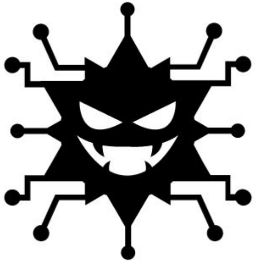
Apps, networks, & cloud servers might be compromised



Device Bricking

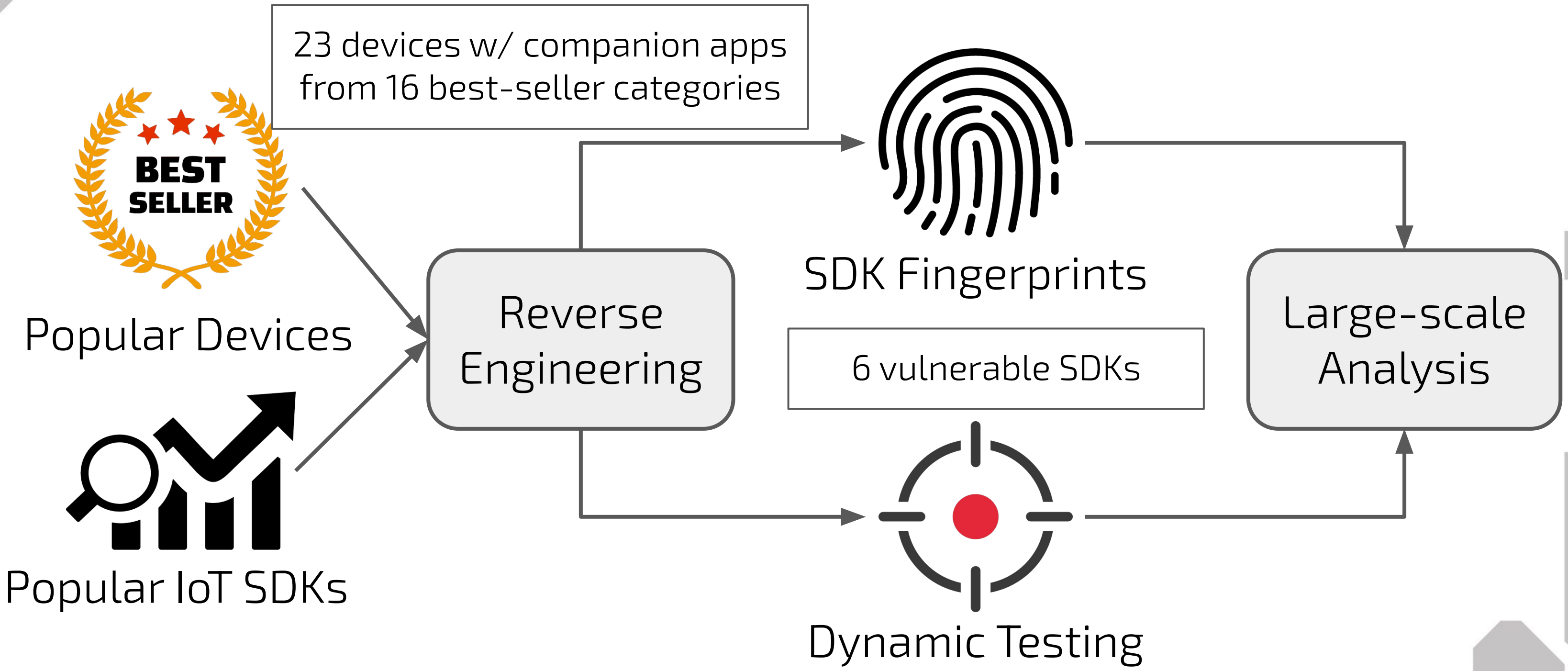


Firmware Downgrade



Firmware Modification

Methodology



Large-scale Analysis

Dataset: 37,783 IoT companion apps (Android)

1,356 apps on the Google PlayStore use at least one of the 6 vulnerable SDKs

- **1,347** apps vulnerable to **ModAttack** → also **Brick/DownAttack**
- **1** app only vulnerable to **BrickAttack**
- **8** apps only vulnerable to **DownAttack**

24 apps control **61** potentially vulnerable devices among the top 50 best-sellers

Research Outcomes

AoT - Attack on Things: A security analysis of IoT firmware updates
Muhammad Ibrahim, Andrea Continella, Antonio Bianchi
In Procs. of the IEEE European Symposium on Security and Privacy (EuroS&P), 2023

AoT - Attack on Things: A security analysis of IoT firmware updates

Muhammad Ibrahim, *Purdue University, West Lafayette, USA*, *ibrahi23@purdue.edu*

Andrea Continella, *University of Twente, Enschede, Netherlands*, *acontinella@iseclab.org*

Antonio Bianchi, *Purdue University, West Lafayette, USA*, *antoniob@purdue.edu*

***Abstract*—IoT devices implement firmware update mechanisms to fix security issues and deploy new features. These mechanisms are often triggered and mediated by mobile companion apps running on the users' smartphones. While it is crucial to update devices, these mechanisms may cause critical security flaws if they are not implemented correctly. Given their relevance, in this paper, we perform a systematic security analysis of the firmware update mechanisms adopted by IoT devices via their companion apps. First, we define a threat model for IoT firmware updates, and we categorize the different potential security issues affecting them. Then, we analyze 23 popular IoT devices (and corresponding**

IoT devices can miss critical security patches or can be compromised by executing malicious code.

Previous works [10], [22], [33], [46], [68] identified specific vulnerabilities in the firmware update mechanisms of some IoT devices. However, the state-of-the-art lacks a comprehensive and systematic picture of DFU issues in the IoT ecosystem. In fact, existing works only focus on a few selected products from specific vendors and do not provide a scalable categorization approach. Besides, the previously investigated attacks require access to the hardware of the IoT devices, significantly limiting the

Conclusions

Embedded devices require **re-thinking** automated security analyses

Understanding and modeling the **interactions** of their firmware is crucial

More effective approaches and tools to identify vulnerabilities

Now, how do we automatically prevent and patch vulnerabilities?

Ongoing/Future Research

Injecting patches into monolithic firmware by static re-writing

Identifying and isolating components in monolithic images

Building a "living" IoT lab for data collection & experimentation

Lightweight runtime detection of anomalies

Thank you!
Questions?

Andrea Continella
<acontinella@iseclab.org>
https://conand.me
@_conand