2016 IEEE Cybersecurity Development

Applying the Opacified Computation Model to Enforce Information Flow Policies in IoT Applications

Amir Rahmati, Earlence Fernandes, Atul Prakash
University of Michigan
{rahmati,earlence,aprakash}@umich.edu

***Abstract*—Enforcing flow-control is an important, albeit difficult to achieve security goal in practical applications. In this paper, we consider the problem of flow-control in IoT applications and present an approach to enforce information flow policies on them. We describe a model called *Opacified Computation* and its implementation, *FlowFence*, and show how to port an existing IoT application (without flow policy enforcement) to one that uses FlowFence to guarantee protection against data leakage. Finally, we report on performance overhead and discuss directions for future work.**

I. Introduction

Smart phones and IoT devices (*e.g.,* wearables, smart home devices) have unprecedented access to our personal information such as location, contacts, medical records, movement patterns, and video and audio feeds. Apps written for these platforms promise great benefits in functionality, energy efficiency, and health monitoring, but also pose security and privacy risks [1]. Although many of these apps have legitimate reasons to compute on sensitive data produced by cyber-physical platforms, compromised or maliciously designed apps can use their access to steal user information. Traditionally, such platforms have used permissions to regulate apps' access to data and devices [2]. However, permission systems only control *what* resources an app accesses, but cannot control *how* apps use that data. This fundamental discord leads to misbehaving apps (*e.g.,* in smartphones [3], and more recently, in smart homes [4]).

A system that controls how an app uses data needs to know three things: (1) sources of data, (2) sinks of data, and (3) how data *flows* between sources and sinks. Prior work has suggested two strategies to achieve such *flow control*:

- **Taint-tracking.** Such systems use static or dynamic analyses to mark variables with "taints" as they get exposed to sensitive data, then track those taints based on the data flows in the program [5], [6], and finally provide a policy enforcement point when the program attempts to exfiltrate tainted data. Although such approaches might catch some data stealing apps, reasoning about program logic in general can be arbitrarily difficult. Therefore, these systems frequently suffer from incompleteness [7] (missing certain flows), or high overhead [8] (making them unsuitable for constrained platforms). Furthermore, These approaches are language specific and limit developer's choice [9], [10]. Static analysis tools also do not bode well in asynchronous environments such as IoT platforms, where trigger-action programming and device delays are prevalent.
- **Label Based Flow Control.** Such systems define labels for different data sources, and then confine an app's access to sinks based on source-defined declassification policies. An example of this would be a password quality checker that accesses third-party resources for downloading patterns but can only access user passwords when it gains the "password" label which cuts off its outside communication capability [11]. Although such approaches do not face performance overheads that plague taint-tracking systems, they are limited to producer (source) defined policies (*e.g.,* $password \not\rightarrow internet$). While this would not be a shortcoming in platforms such as JavaScript [11] and web [12], it makes these systems less flexible and unsuitable for IoT domain, where diversity of scenarios and requirements arises the need for user-specified discretionary policies such as $location \rightarrow lightSwitch_A$.

In this work, we discuss *FlowFence* [13], and its use in development of IoT apps for which flow policies are enforced. FlowFence draws inspiration from prior systems such as Cowl [11], Hails [12], and Darkly [14], but also extends the concepts so that flow policies can be applied to practical IoT apps.

The key idea behind FlowFence is the notion of *Opacified Computation* in which the system provides sandboxes where an app can access sensitive data. Developers wishing to use sensitive data must split their apps into modules, called Quarantined Modules (*QMs*), that operate on sensitive data within the sandbox. Under this model, taint tracking occurs at the QM level. As a QM accesses sensitive data, its sandbox accumulates taints from data sources. Results returned from QMs take the form of *opaque handles* that carry the taint of the sandbox at the point of return. Outside the sandbox, an opaque handle provides no information about the return results—it cannot be dereferenced or mapped to the sandboxed data. An opaque handle can get dereferenced in two ways: (1) it gets passed into another Quarantined Module or (2) it gets sunk through a trusted API. Sensitive data written to a sink must satisfy flow policies of the form <source taint, sink>. Flow policies are defined in an app's manifest and must be

DOI 10.1109/SecDev.2016.10

approved by the app's user.

In the rest of this paper, we first discuss the two previous approaches to flow-control in more detail (§II). We then examine the architecture of an Opacified Computation system (§III). Next, we walk through an example of how a typical program can be implemented in this model (§IV). Finally, we discuss future work and the associated challenges (§V).

II. Related Work

Taint-Tracking: The most well known technique for monitoring data flows through programs is taint-tracking [15]. Taint-tracking systems typically use two approaches: (1) Dynamic techniques (*e.g.,* TaintDroid [5]) that associate a taint label with variables in applications and instrument the system to propagate taints; and (2) Static taint-tracking techniques (*e.g.,* FlowDroid [6]) in which statically label inputs and analyze data/control flows to determine potential taints at exit points from the program. Dynamic techniques suffer from noticeable run-time overhead (*e.g.,* 14% for TaintDroid [5]), while static techniques depend on access to source code and on language(s) used. Both classes of techniques often suffer from over-tainting or under-tainting [7] and also fall short in handling implicit flows and concurrency. Languages such as JFlow [9] also exist which are designed from ground up to support taint-tracking. However, these require developers to learn a new security-typed language and then can only build their apps in that language.

Label Based Flow Control: In label based systems, information flow control is enforced through a predefined labeling scheme that defines policies regarding accessing and publishing sensitive data. COWL [11], Hails [12], and Flume [16] are examples of these systems which provide flow control for webapps, Javascript and OS processes respectively. The main shortcoming of these systems is their reliance on producer (source) defined labels and policies. This limits their applicability to dynamic environments like the IoT where multiple devices and platforms may interact with each other in unpredictable patterns. An example of this would be a camera which could be used to provide a web feed, or perform face recognition to open a door, or any other number of functions depending on its deployment.

III. Opacified Computation Model

The Opacified Computation model, first introduced in our recent work [13], requires each app to consist of: (1) Sensitive sources and sinks that are only accessible inside sandboxes; (2) a set of Quarantined Modules (see Introduction) that implement functions to process sensitive data and that execute in sandboxes, and (3) an unguarded non-QM code that orchestrates execution of the program. A trusted system tracks taints for QMs as they execute, treating the function as a blackbox, and not at the instruction level. This allows for both efficient taint-tracking as well as use of any language inside a QM function. Programmer chooses the decomposition into QMs to control over-tainting.

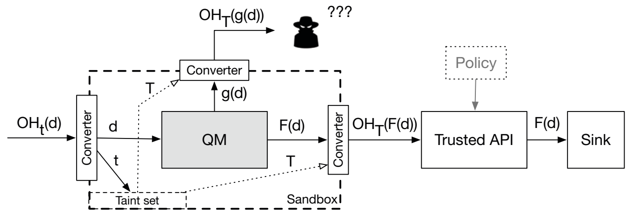

Fig. 1. Data flow in and out of a sandbox. Opaque handles ($OH_t(d)$) are transparently dereferenced and their taints (t) are added to sandbox taint set (T). Any attempt to use sinks ($F(d) \rightarrow Sink$) has to go through trusted API which is subject to <source,sink> flow policies. Non-QM code cannot dereference an opaque handle ($OH_T(g(d))$) to access its data.

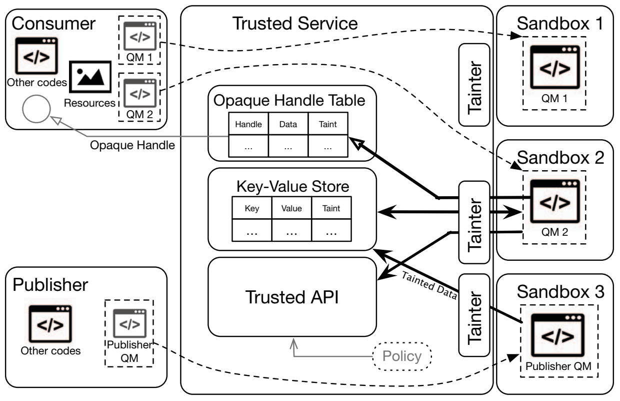

Fig. 2. Architecture of FlowFence [13], an Opacified Computation system for IoT frameworks. Developers split apps into Quarantined Modules, that run in sandbox processes. Data leaving a sandbox is converted to an opaque handle tainted with the sandbox taint set.

As discussed, results returned by a sandbox are converted to opaque handles. Opaque handles are hidden references to sensitive data. Each opaque handle is associated with a set of taints that represent the type of sensitive value they represent. An opaque handle can only be dereferenced inside a sandbox when passed to another QM. Figure 1 shows how such a transaction would work. When program passes an opaque handle to the QM, the Trusted Service transparently dereferences it and passes the data to the QM. Trusted Service also adds the taint associated with the handle to the taint set associated with QM's sandbox. When a QM returns data, the Trusted Service again creates an opaque handle of the data, tainted with the sandbox taint set. To write data to a sink, a QM must use a FlowFence-provided trusted API, which verifies whether the flow from QM's taint set to the sink is allowed. If a violation is detected, an exception occurs inside the QM which is not exposed to non-QM code. FlowFence has two main mechanisms for data sharing: Key-value store, and event channels. Each QM has a key-value store which is read-only for other QMs and contains <key,sensitive value,taint label> entries. This allows publishers and consumers in a device-agnostic manner. Each producer can also have multiple event channels which allow data consumers to register for callback from data publisher. We direct readers to the FlowFence [13] paper for in-depth discussion on architecture and other details.

Figure 2 presents an overview of FlowFence [13] architecture, a system implementing the Opacified Computation model for IoT frameworks. FlowFence does not suffer from under-tainting, but could incur over-tainting if an app is not correctly modulated into QMs. However, over-tainting is detectable when comparing requested flows with authorized flow policies. In the next section, we will discuss the steps for developing an app in FlowFence platform.

IV. Developing Apps under OC model

To illustrate the model's applicability, we will go through the steps for porting an existing SmartThings app to a FlowFence-based application. The app accesses the user's location and automatically turns on (and off) a light if the user is within (or outside) a predefined geo-fenced area.[1] The app can provide a guarantee to the user, via its manifest, that location data is not transferred to Internet and only used to control a light switch.

In its manifest (Listing 1), application requests $location \rightarrow switch$ data flow. This flow is either previously approved by data producer (*i.e.,* location service) or must be approved by user at install time. FlowFence ensures that no other flow of location data will occur, say to the Internet, even if application attempted it or was exploited to attempt it.

Figure 3.a provides an overview of the main functions of the original (permission-based) SmartThings app. A presence detector function (Listing 3) monitors user's location. Upon detecting a change, it activates the *toggleSwitch* function (Listing 6) which turns the light on or off. In the original app, the user has no flow guarantees without inspecting and analyzing code in depth – location can potentially be leaked to the Internet.

Figure 3.b presents how the SmartLight application has to be modified to use FlowFence. Two main changes must be made: (1) The location service developer needs to define a QM for presence detector (Listing 5) that monitors location (sensitive source and tainted with `locationTaint`), updates a Key-Value store to record presence status, and fires channel event when detecting a change; and (2) SmartLight developer needs to define a QM for light switch (Listing 7) to subscribe to the channel, read presence status via events (thus also getting tainted with `locationTaint`, and configures the switch, a sink, to which flow is permitted by policy. Event channels are declared in the publisher's manifest (see Listing 2) and allow both intra-app and app-to-app communication of tainted data between QMs. A QM is structurally similar to a typical Java class, but needs to use FlowFence primitives to access and share sensitive data. This familiarity makes it easy for developers to adopt their programs to function within FlowFence framework. In our evaluation, it took a programmer unfamiliar with framework's API 2 days to port SmartLight application.

[1]Note that we simplify the code and focus on relevant parts of it because of space constraints.

```
<?xml version="1.0" encoding="utf-8"?>
<FlowFenceManifest ...>
  <policy>
    <allow flowfence:src="locationTaint"
    ↪ flowfence:sink="SmartThings.SmartSwitch" />
  </policy>
</flowfenceManifest>
```

Listing 1: SmartLight FlowFence policy. $location \rightarrow switch$ data flow is requested by the application. To establish, this flow either has to be pre-approved by data producer (location service) in producer's manifest (Listing 2) or by the user during install time.

```
<?xml version="1.0" encoding="utf-8"?>
<FlowFenceManifest ...>
  <event-channel flowfence:name =
  ↪ "presenceUpdateChannel" flowfence:exported =
  ↪ "both" />
</flowfenceManifest>
```

Listing 2: Location service FlowFence policy. $location \rightarrow switch$ flow has not been pre-approved, thus SmartLight will need to get user's approval to establish it. Location service however has defined an event channel which can be used by SmartLight to subscribe for updates.

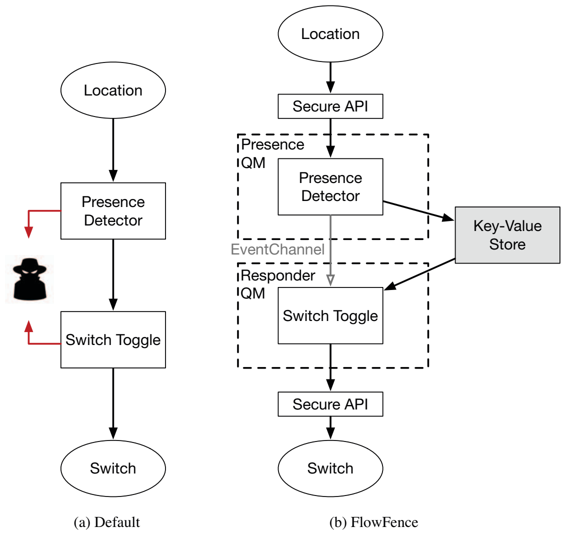

Fig. 3. Normal structure of the SmartLight app compared to the FlowFence model.

Listing 5 presents the definition of presence QM. The QM writes presence value to the KV store. It also sends an update to the channel if there is change in presence value so subscribing QMs can get notified.

Notice that the presence change detector function is moved into the QM and modified (Listing 4) to store presence in KV store, so that it can be accessible by other QMs. Toggle switch function is also moved inside a QM. (Listing 7). This QM reads the presence status from KV store and adjust the switch value accordingly. Neither QM can leak location data to the Internet, since access to Internet in a QM is only available

```java
firebaseRef.child(LOC_KEY).addValueEventListener
  ↪  (new ValueEventListener()) {
  public void onDataChange (DataSnapshot
  ↪  dataSnapshot) {
    String presence = (String)
  ↪  dataSnapshot.getValue();
    toggleSwitch(presence);
  }
  ...
}
```

Listing 3: The original (permission-based) presence function. It calls `toggleSwitch` when detecting a presence change.

```java
QM.S1<String, Void> presenceKV =
  ↪  oconn.resolveStatic(void.class,
  ↪  PresenceQM.class, "putLoc", String.class);
...
firebaseRef.child(LOC_KEY).addValueEventListener
  ↪  (new ValueEventListener()) {
  public void onDataChange(DataSnapshot
  ↪  dataSnapshot) {
    String presence = (String)
  ↪  dataSnapshot.getValue();
    putLoc.arg(presence).call();
  } ...
}
```

Listing 4: FlowFence presence function. Function updates a KV store value when detecting a presence change.

```java
public class PresenceQM implements Parcelable
{
  public static void putLoc(String presenceVal)
  {
    //Write presence value to KV store
    SharedPreferences myprefs =
FlowFenceContext.
    getInstance().getSharedPreferences
    ("presenceKVS", Context.MODE_WORLD_READABLE);
    SharedPreferences.Editor edit =
    myprefs.edit();
    edit.putString("location", presenceVal);
    ...
    //fire an event to any listening QM
    IEventChannelAPI eventApi = (IEventChannelAPI
    )FlowFenceContext.getInstance().getTrustedAPI
    ("event");
    eventApi.fireEvent(builtTS, ComponentName.
    unflattenFromString("presenceChannel"));
    Log.i("PresenceQM", "updated KV with value: "
  ↪  + presenceVal + ", and fired channel
  ↪  event");
  }
...
}
```

Listing 5: presenceQM. The QM defines a KV store to record presence state and fires an event to subscribed QMs whenever a change in key value occurs.

via a Trusted API that enforces flow policies.

Performance: Using the modified SmartLight application caused modest latency increase in app response time. In our setup, we used an LG Nexus 4 running a modified version of Android 5.0 augmented with FlowFence as our IoT hub. In our experiments, the time it took for the switch to turn on after user entered the geo-fenced area increased by $90ms$. We considered this extra latency to be acceptable for this class of IoT applications given the higher assurance user is provided about the confidentiality of their private data.

```java
private void toggleSwitch(String presence){
  if(!history.equals(presence)) {
    if (presence.equals("home")) {
      Log.i(TAG, "let there be light!");
      List<SmartSwitch> switches =
  ↪  SmartThingsService.getInstance().⌋
  ↪  getSwitches();
      if(switches != null) {
        for (SmartSwitch ssw : switches) {
          SmartThingsService.getInstance().⌋
  ↪  switchOnOff("on",
  ↪  ssw.getSwitchId());
        }
      }
    } else if (presence.equals("away")) {
      Log.i(TAG, "lights off!");
      List<SmartSwitch> switches =
  ↪  SmartThingsService.getInstance().⌋
  ↪  getSwitches();
      if(switches != null) {
        for (SmartSwitch ssw : switches) {
          SmartThingsService.getInstance().⌋
  ↪  switchOnOff("off",
  ↪  ssw.getSwitchId());
        }
      }
    }
    history = presence;
  }
}
```

Listing 6: The original (permission-based) toggle switch function. Program sends a command to turn lights on/off depending on the presence value.

V. Future Research Directions

- **Information flow tracking across multiple environments:** One major shortcoming shared across all approaches discussed in this paper is their limitation to one environment (*e.g.,* a single home). The growth of IoT systems and cloud computing however, creates the need for information flow control across multiple environments and platforms [17]. In an IoT ecosystem, multiple environments with different communication channels, computational capabilities, sensors and actuators need to interact with each other and the surrounding environment (*e.g.,* a smart city consisting of smart buildings and roads). Future flow control systems need to propagate information flow across these different platforms to provide comprehensive control. Hardware security extensions such as Intel SGX [18] or ARM TrustZone [19] can provide the required integrity to create a trusted computing base in such scenarios.
- **Mitigating side-channels:** At their best, information flow control systems can control the course of data from sources to sinks but they are still susceptible to side-channel attacks that leak data about system events. These side-channels can be intentional in form of covert channels (*e.g.,* using switch toggles in SmartLight app to transmit data) or unintentional (*e.g.,* power variations caused by different computational loads, program control flow, timing). Limiting such attacks

```
public class ResponderQM implements Parcelable
{
  public static void pollPresenceAndCompute()
  {
    // Read updated presence value from KV store
    SharedPreferences presencePrefs = FlowFenceContext.getInstance().createPackageContext("presenceQM", 0).getSharedPreferences("PresenceKVS", Context.MODE_WORLD_READABLE);
    String presence = presencePrefs.getString("location", "null");

    // Read previous presence value from KV store
    SharedPreferences myprefs = FlowFenceContext.getInstance().getSharedPreferences("hist_store", Context.MODE_WORLD_READABLE);
    String history = myprefs.getString("history", "");

    // Toggle switch function
    if(!history.equals(presence)) {
      String op = null;
      if (presence.equals("home")) {
        Log.i(TAG, "let there be light!");
        op = "on";
      } else if (presence.equals("away")) {
        Log.i(TAG, "lights off!");
        op = "off";
      }

      if (op != null) {
        ISmartSwitchAPI switchAPI = (ISmartSwitchAPI) FlowFenceContext.getInstance().getTrustedAPI("smartswitch");
        List<SmartDevice> switches = switchAPI.getSwitches();

        if(switches != null) {
          for (SmartDevice ssw : switches) {
            switchAPI.switchOp(op, ssw.getId());
          }
        }
      }

      history = presence;
      // Store new presence value in KV store
      SharedPreferences.Editor edit = myprefs.edit();
                         edit.putString("history", hist);
                         edit.commit();
    }
  }
}
```

Listing 7: Responder QM reads an updated presence value in KV store, and toggle switches based on it.

fall outside the scope of access-control systems but remain a major concern for these systems and remains an active research area [20]. FlowFence does not eliminate these side-channels, but mitigates them by coaxing developers toward a more structured use of privacy sensitive resources through usage of QMs and preventing unnecessary access of program functions to these resources by taint-tracking flow of data between sources and sinks.

- **Policy Management:** While there has been major progress in enforcing flow-control across systems, meaningful representation of these decisions to both admins and users still remains a challenge. Managing, comprehending, and delegating policies in a multi-stakeholder environment (*e.g.,* smart city) creates new challenges that lie outside of traditional flow control systems. In addition, meaningful visualization of these policies remain a major interface challenge, especially given existing problems in this space [2], [21].

VI. Conclusion

Current access control system suffer from well-known but difficult-to-solve problems. In this work, we discussed the Opacified Computation model as an approach to achieve fine-grained access control—we showed how it draws inspiration from previous systems and we highlighted its principles that improve upon the current state of the art in flow control. We walked through an example of how developers could modify their applications to fit this model while highlighting some of its shortcomings and discussing future research directions.

Acknowledgment

We thank Justin Paupore, Daniel Simionato, and Mauro Conti for collaborating on the original FlowFence project. This research is supported in part by NSF grant CNS-1318722 and by a generous gift from General Motors. Any opinions, findings, conclusions, and recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] T. Denning, T. Kohno, and H. M. Levy, “Computer security and the modern home,” *Communications of ACM*, 2013.

[2] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, “Android permissions: User attention, comprehension, and behavior,” in *SOUPS*, 2012.

[3] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, “Taming information-stealing smartphone applications (on Android),” in *Trust and Trustworthy Computing*, 2011.

[4] E. Fernandes, J. Jung, and A. Prakash, “Security Analysis of Emerging Smart Home Applications,” in *IEEE S&P*, 2016.

[5] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Transactions on Computer Systems*, 2014.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in *PLDI*, 2014.

[7] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar, “On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices.” in *SECRYPT*, 2013.

[8] J. Paupore, E. Fernandes, A. Prakash, S. Roy, and X. Ou, “Practical always-on taint tracking on mobile devices,” in *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[9] A. C. Myers, “Jflow: Practical mostly-static information flow control,” in *POPL*, 1999.

[10] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, “Flexible dynamic information flow control in Haskell,” in *Haskell Symposium*. ACM SIGPLAN, 2011.

[11] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, “Protecting users by confining javascript with cowl,” in *OSDI*, 2014.

[12] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, “Hails: Protecting data privacy in untrusted web applications,” in *OSDI*, 2012.

[13] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks," in *USENIX Security Symposium*, 2016.

[14] S. Jana, A. Narayanan, and V. Shmatikov, "A scanner darkly: Protecting user privacy from perceptual applications," in *IEEE S&P*, 2013.

[15] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE S&P*, 2010.

[16] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *SOSP*, 2007.

[17] M. Clark and P. Dutta, "The haunted house: Networking smart homes to enable casual long-distance social interactions," in *IoT-App*, 2015.

[18] V. Costan and S. Devadas, "Intel sgx explained," Tech. Rep.

[19] ARM, "Arm security technology - building a secure system using trustzone technology," Tech. Rep., 2012.

[20] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres, "Addressing covert termination and timing channels in concurrent information flow systems," in *SIGPLAN Notices*, 2012.

[21] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, "A conundrum of permissions: installing applications on an android smartphone," in *FC*, 2012.