ARM TrustZone for Secure Image Processing on the Cloud

Tiago Brito, Nuno O. Duarte, Nuno Santos
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
{*tiago.de.oliveira.brito,nuno.duarte,nuno.m.santos*}*@tecnico.ulisboa.pt*

***Abstract*—Nowadays, offloading storage and processing capacity to cloud servers is a growing trend. This happens because high storage capacity and powerful processors are expensive, whilst cloud services provide a cheaper, ongoing, and reliable solution. The problem with cloud-based solutions is that servers are highly accessible through the Internet and therefore considerably exposed to hackers and malware. In this paper, we design and implement Darkroom, a secure image processing service for the cloud leveraging ARM TrustZone technology. Our system enables users to securely process image data in a secure environment that prevents exposure of sensitive data to the operating system. We evaluate our system and observe that our solution adds a small overhead to image processing when compared to computer platforms that require the entire operating system to be trusted.**

I. INTRODUCTION

Over the latest years, the cloud has become immensely popular due to the proliferation of numerous online services for storage, streaming, and processing of content. However, these services frequently handle user sensitive data which have security and privacy requirements that are not always properly considered by the providers of these services. In some cases, this negligent behavior has even lead to serious scandals such as celebrity photo [1], and user document [2] leaks. To make matters worse, instead of trying to enforce security protocols capable of preventing this type of accidents, these providers end up stitching user contract terms so they can be absolved of these incidents, giving a bad reputation to cloud providers in general [3].

To secure user generated content in the cloud, such as personal documents, images, or videos, a commonly used approach has been to encrypt the content at the client side before it is sent to the cloud. While this approach is effective whenever the cloud is used for persistent storage, it can no longer be applied in scenarios where content needs to be processed by the cloud service. Notable examples include cloud services such as Facebook or Instagram which apply transformation functions to the images uploaded by the users, for example to rescale, rotate, or reencode images. To perform such operations, the image data must be in its unencrypted format, point at which it may become vulnerable, e.g., to an attacker that managed to exploit some critical bug in the application code or in the operating system.

A promising alternative to encryption is to leverage Trusted Execution Environments (TEE) in order to safely perform image transformations at the cloud server without the need to rely on the rich operating system running on the server. Thus, if the OS is compromised, the TEE ensures that an attacker cannot access the memory regions allocated to the TEE where security-sensitive images are located. This approach has been recently adopted in many mobile device studies, whether to provide safe storage [4], enforce authentication mechanisms [5], [6], or provide OS introspection and monitoring capabilities [7], [8], [9]. One of the reasons this approach has been so popular in the mobile landscape, has to do with ARM TrustZone [10], a technology that allows the implementation of TEE systems and is present in the majority of mobile device processors.

In this paper, we explore the adoption of ARM TrustZone technology in order to provide an isolated environment for processing images securely on the cloud. Specifically, we present the design and implementation of Darkroom, a system that leverages ARM TrustZone to offer a secure image processing environment for cloud-hosted services. Next, we start by describing the design of our system.

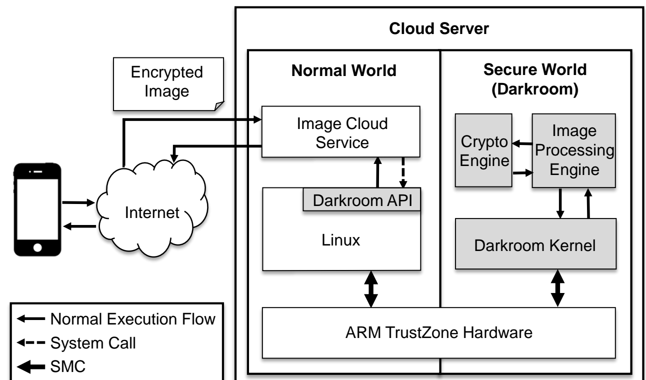

Figure 1. Architecture

II. DESIGN

This section describes Darkroom, a TrustZone-based system which allows for secure image processing on the cloud. This system is capable of processing images without exposing them to the operating system. This is done by storing image data in encrypted form and using TrustZone-enabled ARM processors to securely process such images in isolation from the operating system.

Figure 1 shows the components of our solution and represents a possible execution flow for the Darkroom system. The flow starts in a client application, which is

represented by a mobile device. The client application sends an encrypted image to the Image Cloud Service component, which can store the image data locally. Both the Image Cloud Service and the operating system run in the normal world context of the TrustZone-enabled processor. After uploading the image data, the client application issues a transformation request for the image. Upon receiving a transformation request, the Image Cloud Service sends the encrypted image data to the secure world. It also sends the transformation request and triggers a world switch via a Secure Monitor Call (`SMC`). This `SMC` gives control to the secure world which decrypts the image data and executes the requested transformation on it. After processing, the image is encrypted once again and sent to the normal world.

Note that we are primarily concerned about potential security breaches resulting from software exploits to normal world programs. In particular, we want to prevent external attackers that manage to take over control of the application or the operating system residing in the normal world from violating the confidentiality of users' images (e.g., by leaking them out). We assume that both cloud provider and cloud administrators are trusted. The trusted computing base (TCB) of our system comprises the hardware platform and the Darkroom components hosted in the secure world.

A. Rich OS Isolation Using ARM TrustZone

To isolate the rich operating system from Darkroom, we leverage ARM TrustZone. ARM TrustZone is a security extension present in ARM processors that provides a hardware-level isolation between two execution domains: *normal world* and *secure world*. Compared to the normal world, the secure world has higher privileges, as it can access the memory, CPU registers and peripherals of the normal world, but not the other way around. Isolation is enforced by checking and controlling the state of the CPU through a NS bit, and partitioning the memory address space into secure and non-secure regions. To perform a context switch between the different worlds, TrustZone offers the *SMC* instruction, which generates a software interrupt that is then handled by the secure monitor. TrustZone follows a similar approach with interrupts. Basically, Interrupt Requests (IRQs) and Fast Interrupt Requests (FIQs) can be configured to be either secure or non-secure interrupts. This means that secure interrupts can only be intercepted by the secure world and not the normal world. Both interrupt management and memory isolation mechanisms are fundamental in guaranteeing peripheral access isolation between worlds. Additionally, TrustZone features a secure boot mechanism that ensures the integrity and authenticity of the system running in the secure world.

B. System Deployment on ARM-based Cloud Servers

We envision Darkroom to be deployed on cloud servers maintained by the service provider. As opposed to Intel-based servers commonly adopted in today's cloud infrastructures, Darkroom's target servers must be based on ARM TrustZone technology. To deploy the system, the cloud provider must flash the firmware of its ARM servers so that the servers bootstrap into the secure world and run Darkroom's setup code before switching to normal world and loading up a rich operating system or hypervisor.

In this process, the cloud provider must generate a *root key* for each server. A root key is a public key pair KR whose private part (KR^{-}) is bundled into the Darkroom image right before the image is flashed onto a server's firmware. The private part of the root key will be accessible only by Darkroom within the secure world and there will be a unique root key for each cloud server. The public part KR^{+} is certified by the cloud provider in order to assert the authenticity of this key. This key is fundamental to ascertain the authenticity of the Darkroom platform to a remote client and to securely share secrets with the Darkroom runtime without the need to trust the rich operating system. Note also that, in addition to the root key, the Darkroom image contains public keys of cloud administrators authorized to perform some security-critical operations, e.g., setting up image transformation functions, as explained next.

C. Setting Up Image Transformation Functions

Once a cloud server is up and running, the rich operating system takes up control of system resources and Darkroom is suspended until requests arise from the normal world to perform image transformation operations (e.g., image rescaling). To allow for additional flexibility, rather than hardcoding transformation functions (TF) into the firmware, these functions can be loaded dynamically by cloud administrators into the Darkroom runtime. This makes it easy to upgrade the system with new or more efficient functions without the need to reflash the servers' firmware.

To support dynamic loading of transformation functions, we must ensure that the code to be loaded is trustworthy. This is because such code will have access to the raw image data submitted by client applications and can potentially compromise that data, e.g., by leaking it from the server. In addition, we must provide mechanisms that allow transformation functions to be uniquely identified in order to ensure that the correct transformation is applied to a given image. Such mechanisms must be able to tolerate modifications in the set of available transformation functions, as these can be installed from the system.

To ensure that the transformation functions installed into the Darkroom runtime are trustworthy, we require that such functions are installed or uninstalled only by trusted cloud administrators. In particular, to install a TF, a cloud administrator must issue a *load* request which must be signed by the administrator key (KA). Darkroom validates the request against the public part of the authorized admin key $KA+$ and allows the operation to proceed if the test

passes; otherwise the operation is aborted. To uninstall a TF from the system, the corresponding *unload* request must also be signed by a trusted cloud administrator. To allow for unique identification of a given TF, Darkroom leverages the hash of the TF binary. This binary is included into the (signed) load request provided to Darkroom. This request contains additional meta-data that indicates the TF name and the type of input parameters, e.g., *rotation (int degree)*.

D. Performing Image Transformation Operations

Normally, performing image transformations involves a three-stage life cycle. First stage is to securely *upload* a security-sensitive image from the client to the cloud server. In this process, we must ensure that the image is encrypted in such a way that it can only be decrypted within the Darkroom runtime. Second stage corresponds to *transforming* the image by invoking a transformation function of Darkroom. Note that one or more transformations can be chained together (e.g., rotation followed by scaling, etc.). Third stage is to *download* the result of the transformed image while allowing that only the client is able to decrypt the resulting image. In addition, the client must be able to trace the authenticity of the transformation sequence in order to ensure that the resulting image derives from a valid sequence of transformations properly applied to the original image submitted by the client to the cloud provider.

But before explaining these stages, we must describe a necessary pre-configuration step. In particular, Darkroom must be set up with a public key pair called *service key* (KS). The role of this key is to associate the image transformation operations to the context of a specific cloud service (whose logic runs in the rich OS and client). Furthermore, it also aims to allow image transformations to be performed on any of the cloud servers properly configured with a root key. To this end, cloud administrators must generate a public key pair KS and securely load it to the Darkroom runtime of each cloud server. Security is achieved by mutually authenticating the cloud administrator's key with the server's root key. Once the service is loaded into each server, it is now possible to perform the following three stages:

1. Image upload: To securely upload the image to the cloud server, the client simply generates a symmetric key KI and encrypts the image with that key. Then, KI is encrypted with the public key KS^+ to ensure that the image can be decrypted only by Darkroom-enhanced servers allocated to the service to which KS^+ is bound. For integrity verification, the client also computes the HMAC of the message using key KI. The resulting blob $\{I\}_{KI}hmac\{KI\}_{KS^+}$ is then uploaded to the cloud service. We name this blob: *envelope*.

2. Image transformation: Whenever the cloud service needs to perform some specific transformation to the encrypted image, it makes a request to Darkroom by invoking a specific system call (through the Image Cloud Service). Through this system call, the service provides as input the image envelope, the ID of the transformation function to be invoked, and any additional parameters required by the transformation function. The system call issues a world switch to Darkroom, which performs three steps: 1) obtains the encryption key KI by decrypting it with the private part of the service key (KS^-), 2) based on KI, recomputes the HMAC of the message and validates the message integrity, and 3) decrypts the image using KI. If all goes well, Darkroom executes the requested transformation function and produces another envelope containing the resulting image encrypted with the symmetric key KI (which means that only the original message owner and Darkroom-enhanced servers will be able to decrypt the resulting image). To be able to trace the transformations that were applied to a given image, Darkroom builds a cryptographic-protected log which is included in the resulting envelope. The log contains a hash chain of the history of transformations applied to the image.

3. Image download: Finally, the last stage in the image transformation life cycle is to download the resulting image to the client and recover it. Since the client has access to the key KI, it can perform the same sequence of steps as Darkroom in order to validate the integrity of the image and decrypt it. In addition, since the envelope contains a history of transformations performed to the image, it is possible to verify that the received image has resulted from a sequence of transformations applied to the original image. (For space constrains, we omit the details of this verification.)

III. Implementation

We implemented a prototype of Darkroom for the Freescale NXP i.MX53 Quick Start Board. A fundamental concern when building our system was to keep a small Trusted Computing Base (TCB), which essentially consists of the components that live in the secure world: Darkroom Kernel, Cryptography Engine, and Image Processing Engine. Next, we describe the most relevant implementation details of the components of our prototype.

The Darkroom kernel is a fundamental component of our system. It lives in the secure world and is responsible for memory management, thread execution, and context switch operations between worlds. To reduce the chance of code vulnerabilities and keep the kernel size small, we adopted the Genode [11] framework to build our secure world kernel. Genode contains a custom kernel called *base-hw* which runs in the secure world and has a codebase of 20 KLOC (thousand lines of code), which is much smaller than the Linux kernel. In addition, Genode implements a Virtual Machine Monitor (VMM) which can manage a paravirtualized full-featured operating system running in the normal world. In the normal world, we run a paravirtualized Linux kernel, custom-made for Genode. This is because resources such as the framebuffer, and signals such as the data abort interrupt must be trapped and managed by the secure world.

Name	Description
T1	Grey-scale transformation
T2	Color invert transformation
T3	Color swap transformation
T4	90 degree rotation transformation
T5	180 degree rotation transformation
T6	Mirror transformation

Table I
DESCRIPTION OF THE TRANSFORMATION FUNCTIONS IMPLEMENTED.

```
for(i = 0; i < length(oldp); i++) {
    color = oldp[i];
    alpha = (color >> 24) & 0xff;
    red   = (color >> 16) & 0xff;
    green = (color >>  8) & 0xff;
    blue  =  color & 0xff;
    lum = (red * 0.299 + green * 0.587 + blue * 0.114);
    newp[i] = (alpha<<24) | (lum<<16) | (lum<<8) | lum;
}
```

Listing 1. Sample code of gray-scale transformation function.

Genode also provides basic mechanisms for context switching. The VMM saves the processor state (registers and stack) when a `SMC` instruction is executed and restores this state whenever the control is given back to the normal world. Although Genode implements a VMM for the normal world OS, the secure world needs to identify the source of `SMC`. To solve this problem we used CPU registers to send arguments for the secure world to interpret. Since the VMM saves the state of the processor before switching security contexts, the secure world can read the saved register values and interpret them as arguments.

To communicate between worlds, we adopt a shared memory strategy. ARM TrustZone provides low-level mechanisms that allows for memory regions to be securely shared. In Darkroom, we use the so called *watermarking* feature implemented by the i.MX53 QSB board to protect secure world's memory regions from normal world accesses. However, using this feature it is possible to allocate a region of memory in the normal world and access it in the secure context once the secure world controls the execution. This effectively creates a shared memory region between both worlds which can be used for the exchange of messages. We leverage this mechanism in Darkroom as follows. Whenever the system call is invoked by an image server, we allocate a memory region with the same size as the image to be processed using `kmalloc`. After allocating this memory region the `virt_to_phys` function is used to translate the array's virtual address to a physical address so it can be used by the secure world to retrieve the image.

Before triggering the `SMC` instruction, the array's physical address is written to register one (`r1`) so it can be used by the secure world. Darkroom must then copy the contents of the shared memory region to a secure memory region so it can be processed and avoid exposing the decrypted data to normal world memory regions.

The Cryptographic Engine runs on top of the secure world kernel and must support the management of both symmetric and asymmetric cryptographic keys and implement core cryptographic algorithms, including a random number generator. In order to maintain a reduced code base, we used the AES-128 implementation from the *mbed TLS* library [12] as the symmetric key encryption and decryption algorithms. We also adapted RSA from *mbed TLS*.

The Image Processing Engine runs on top of the Darkroom kernel and manages the transformation functions loaded into the system. These functions are responsible to effectively process the image data sent from the client in an isolated environment managed by the kernel. In our current prototype, we implemented a small set of simple transformation functions just to demonstrate the feasibility of our approach. In a real world setting, more sophisticated functions could be developed in order to serve the needs of external services such as image managing websites, social networks and personal record management services. The transformation functions currently implemented in our system are listed in Table I. All transformations offered by this component were implemented from scratch without having to rely on any image library. Listing 1 provides the sample code of a transformation function to change the color palette of the image to gray scale.

IV. EVALUATION

To study the performance of our prototype, we measure the execution time of the image transformations it supports through micro-benchmarking. To be able to do a more in-depth analysis of these numbers, we performed measurements both in the normal and secure worlds. Our evaluation testbed consisted of an i.MX53 Quick Start Board, featuring a 1 GHz ARM Cortex-A8 Processor, and 1 GB of DDR3 RAM memory. The board executed our system from a mini SD card, which was flashed with the modified Genode and Linux versions. For each experiment, we report a mean of 50 runs, and provide no standard deviation values because these were negligible (less than 3 milliseconds). We also stress that the results recorded in these experiments were collected after both the normal and secure world components were compiled with the O3 optimization flag.

Table I presents the six different image transformations supported by our system, as well as the names we picked to identify them during our result analysis. To get a clear perspective of the performance costs of these transformations, we chose to measure their execution time over a single image. The image we picked is 1024x1024 pixels, not only because it is a resolution supported by many of today's mobile devices, but also because it is a common resolution for network computing devices.

Figure 2 shows the execution times of all these transformations. As we can see, the grey-scale transformation (T1) is by far the most expensive. This happens because even though every transformation consists on a simple for-loop with

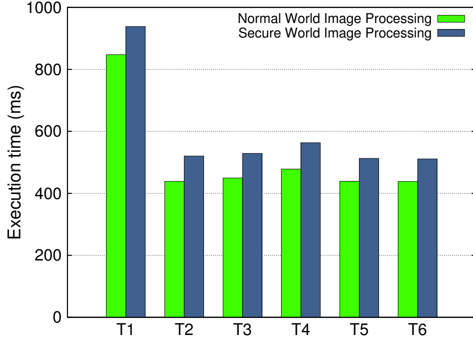

Figure 2. Execution time of all transformations.

complexity *O(N × M)*, where *N* is the width and *M* is the height of the image, T1 features additional math and bit shift operations. In addition, since this transformation requires mathematical and bit shift operations, the execution time difference becomes more prominent because the compiler cannot optimize it as much as the other transformations. Regarding the execution time differences between both worlds, we observe an almost constant penalty in the secure world, associated to the world switch operation, but mostly to buffer allocations necessary to share data between worlds.

To study the performance variations of these transformations, we measured the execution time of transformation T1 for different image resolutions. We focused on T1 since it is the most computationally demanding of all transformations supported by the system. For consistency reasons, we downgraded the original 1024x1024 picture resolution and tested T1 with three additional lower resolutions. Figure 3 shows the impact of T1 on a 128x128, 256x256, 512x512, as well as on the original 1024x1024 picture. The execution time is broken up into three parts: the transformation itself, context-switch, and cryptographic operations. As we can see, the cryptographic operations, more specifically decryption of input image and encryption of the resulting image, are slower in the secure world than in the normal world. This happens because the measurements in the secure world involve copying contents from a world-shared buffer to a buffer in secure memory, and then from this buffer back to the shared buffer after the transformation. On the other hand, the context-switch times are almost negligible, corresponding to the times taken by the system to execute our custom syscall, allocate memory for the shared buffer, translate a virtual to a physical address, and call `SMC`, which triggers the context-switch. In this case, the larger the image, the longer it takes to allocate the shared memory region, which explains why we see an increase in the time the context-switch takes from smaller images to bigger images. Regarding transformation times, we can see the execution times between the normal and secure worlds are very similar.

V. Related Work

Over the past years, extensive work has been focused on data security for untrusted cloud-hosted storage services.

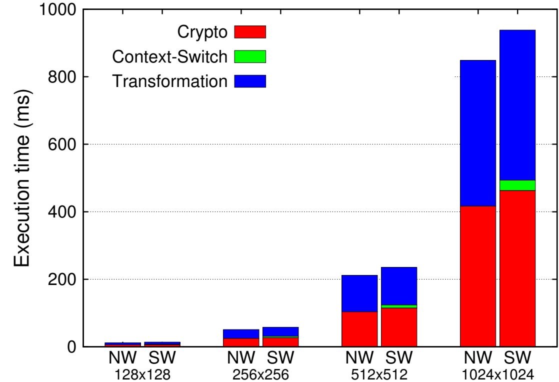

Figure 3. Execution time of the grey-scale transformation.

Systems such as Venus [13] or DEPSKY [14] protect data confidentiality by relying exclusively on cryptographic techniques performed at the client side. This approach tends to be attractive for users because it precludes the need to trust in specific components controlled by the cloud provider. The downside of this approach, however, is that encrypted data cannot be processed by applications, e.g., for performing image transformations, which reduces the applicability of this technique in the cloud. To overcome this limitation, researchers have studied ways to allow for encrypted data processing by leveraging advanced cryptographic techniques. CryptDB [15] is a representative example of such a system which employs homomorphic encryption to enable SQL query processing over encrypted relational databases. Nevertheless, systems such as CryptDB tend to limit the functions that can be executed, introduce considerable performance overheads, and depend on weaker cryptographic schemes when compared with traditional ones.

An alternative approach to securing cloud processing is to leverage Intel's upcoming technology Security Guard Extensions (SGX). SGX is a set of hardware extensions to the Intel architecture which enables processes to maintain secure address space regions. These regions are called *enclaves* and provide a Trusted Execution Environment (TEE) for running security-sensitive application code in isolation from the OS. Although this approach requires users to trust the SGX-enabled hardware deployed by the cloud provider, enclaves can run arbitrary functions at native processor speed, hence faster than homomorphic encryption schemes, and provide strong security properties. In particular, enclaves' internal state cannot be accessed neither by privileged system code nor through memory probe attacks. Projects such as Haven [16] and V3 [17] have started to explore ways to run unmodified legacy code and verified code, respectively, inside enclaves. However, some fundamental limitations need to be overcome in order to make this technology fully practical and robust [16]. Thus, similarly to SGX, we take a TEE-based approach in the design of Darkroom, but explore the use of ARM TrustZone technology, which is more mature than SGX and available in commodity hardware.

Given that the majority of mobile devices are equipped with ARM processors, ARM TrustZone has been studied mostly to overcome security issues on mobile platforms. Many authors propose solutions based on TrustZone-enabled TEE for hosting mobile security services, which allow for: detecting and preventing mobile app ad frauds [18], implementing OS introspection mechanisms [7], enabling secure storage of sensitive data [4], providing secure authentication mechanisms [5], implementing one-time-password solutions [6], or providing forensic tools for trusted memory acquisition [8]. Other systems provide general-purpose frameworks for splitting mobile app code and run it in the TEE [19] or enabling trusted I/O between the user and TrustZone-based services [20]. Existing systems, however, are not directly applicable to the cloud setting due to the cloud's specificity in terms of application requirements and system administration model. Brenner et al. [21] took some first steps to using ARM TrustZone on the cloud by building a TEE-protected privacy proxy for Zookeeper. Their system provides a confidential coordination service for distributed applications, which constitutes a different application scenario than the focus of our work.

VI. Conclusion

We presented Darkroom, a system that leverages ARM TrustZone to bootstrap trust in a cloud based image processing service. Darkroom provides clear isolation between a potentially compromised cloud server OS and a smaller trusted execution environment and guarantees that all users' data stored or processed in a cloud server is handled by a smaller code base. For Darkroom, we designed a set of cryptographic protocols that builds trust when cloud administrators add dynamic code in the server, and provides integrity and authenticity properties to users' image processing requests to cloud servers. The image processing performance in Darkroom incurs a reduced time penalty.

Acknowledgments: This work was partially supported by the EC through project H2020-645342 (reTHINK), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

References

[1] BBC, "FBI investigates 'Cloud' celebrity picture leaks," http://www.bbc.com/news/technology-29011850.

[2] N. Security, "Google Drive security hole leaks users' files," https://nakedsecurity.sophos.com/2014/07/10/google-drive-security-hole-leaks-users-files.

[3] C. Weekly, "Why unfair contract terms put end-user trust in cloud at risk," http://www.computerweekly.com/blog/Ahead-in-the-Clouds/Why-unfair-contract-terms-put-end-user-trust-in-cloud-at-risk.

[4] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena, "DroidVault: A Trusted Data Vault for Android Devices," in *Proc. of ICECCS*, 2014.

[5] D. Liu and L. P. Cox, "Veriui: Attested Login for Mobile Devices," in *Proc. of HotMobile*, 2014.

[6] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens," in *Proc. of CCS*, 2015.

[7] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture," in *Proc. of MoST*, 2014.

[8] H. Sun, K. Sun, Y. Wang, and J. Jing, "Reliable and Trustworthy Memory Acquisition on Smartphones," *Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2547–2561, 2015.

[9] B. Yang, K. Yang, Y. Qin, Z. Zhang, and D. Feng, "DAA-TZ: An efficient DAA scheme for mobile devices using ARM TrustZone," in *Trust and Trustworthy Computing*, 2015, pp. 209–227.

[10] ARM, "ARM Security Technology – Building a Secure System using TrustZone Technology," ARM Technical White Paper, 2009.

[11] "The Genode OS Framework," http://genode.org/.

[12] "mbed TLS," https://tls.mbed.org/.

[13] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for untrusted cloud storage," in *Proc. of CCSW*, 2010.

[14] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds," in *Proc. of EuroSys*, 2011.

[15] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *Proc. of SOSP*, 2011.

[16] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proc. of OSDI*, 2014.

[17] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in *Proc. of IEEE S&P*, 2015.

[18] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure Online Mobile Advertisement Attestation Using Trustzone," in *Proc. of MobiSys*, 2015.

[19] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications," in *Proc. of ASPLOS*, 2014.

[20] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building Trusted Path on Untrusted Device Drivers for Mobile Devices," in *Proc. of APSys*, 2014.

[21] S. Brenner, C. Wulf, and R. Kapitza, "Running ZooKeeper Coordination Services in Untrusted Clouds," in *Proc. of HotDep*, 2014.