2014 IEEE 6th International Conference on Cloud Computing Technology and Science

STOVE: Strict, Observable, Verifiable Data and Execution Models for Untrusted Applications

Jiaqi Tan, Rajeev Gandhi, Priya Narasimhan
Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA
Email: tanjiaqi@cmu.edu, rgandhi@ece.cmu.edu, priya@cs.cmu.edu

***Abstract*—The massive growth in mobile devices is likely to give rise to the leasing out of compute and data resources on mobile devices to third-parties to enable applications to be run across multiple mobile devices. However, users who lease their mobile devices out need to run applications from unknown third-parties, and these untrusted applications may harm their devices or access unauthorized personal data. We propose STOVE, a data and execution model for structuring untrusted applications to be *secure by construction*, to achieve strict and verifiable execution isolation, and observable access control for data. STOVE uses formal logic to verify that untrusted code meets isolation properties which imply that hosts running the code cannot be harmed, and that untrusted code cannot directly access host data. STOVE performs all data accesses on behalf of untrusted code, allowing all access control decisions to be reliably performed in one place. Thus, users can run untrusted applications structured using the STOVE model on their systems, with strong guarantees, based on formal proofs, that these applications will not harm their system nor access unauthorized data.**

I. Introduction

There has been massive growth in the number and capabilities of personal mobile devices in use in recent years. More than 2 billion mobile devices (tablets and smartphones) were sold in 2014 alone [1]. Just as the massive increase in the availability of enterprise-level compute resources, such as servers, gave rise to cloud computing and the leasing out of enterprise compute resources to third-parties, this massive availability of personal mobile devices will inevitably lead to the leasing out of the compute and data resources on personal mobile devices to third-parties. In fact, systems for mobile leasing to allow third-parties to harness both the compute and data resources of mobile devices have been proposed ([2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]). However, the leasing out of mobile devices is very different from the leasing out of enterprise compute resources in cloud computing, as mobile devices are inherently personal. Hence, third-party code and applications from unknown entities running on personal mobile devices can alter or modify these devices, and they may make unauthorized accesses to privacy-sensitive personal data belonging to users on these devices. Current security models for cloud computing focus on enterprise concerns, such as securely running trusted applications on untrusted hosts [14], [15], [16]. Hence, the existing security models for cloud computing are inadequate for addressing the threats posed to personal data on mobile devices with mobile resource leasing. Security models for mobile resource leasing must both provide protection for executing untrusted application code from unknown developers, and provide privacy controls for allowing untrusted applications to access only authorized data on the mobile device.

Traditionally, software applications are assumed to be written by reputable software developers. These applications are able to use the full functionality of the interface provided by the operating system to user applications, hence, these applications are implicitly trusted. Although modern OSes provide isolation between the OS and applications, and between different applications, user applications are still allowed to access most OS facilities via system calls, and applications are free to access user data. However, this “reputable developer” trust model does not hold for recent emerging or even existing classes of applications. In particular, users may not want to trust applications which have been developed by unknown entities. Users need to run untrusted applications which have been developed by unknown entities when they lease their compute resources out to third-parties, such as in cloud computing. This problem is made worse in the emerging trend of mobile resource leasing, because the systems leased out to third-parties are mobile devices belonging to individuals which contain personal data, as compared to enterprise-level servers in cloud computing, and these third-party applications may need to make use of the user’s personal data. Thus, any malicious actions by untrusted applications can lead to significant personal data loss and damage to personal devices. Hence, strong guarantees are needed for applications developed by unknown entities which are untrusted, so that users can be confident that they can run these applications on their personal devices without other applications, or the OS, being harmed, and that these untrusted applications cannot access any unauthorized data.

Systems with Untrusted Applications: Apart from resource leasing in cloud computing and clouds of mobile devices, there are a number of existing and proposed systems where users need to run potentially untrusted applications developed by unknown entities. **Volunteer computing** systems enable users to volunteer compute time on their systems to run small parts of large computations, such as scientific applications (e.g. SETI@Home [17] and BOINC [18]). Users have to trust the authors of these large computations, although these projects are typically sponsored by reputable scientific institutes which users can plausibly trust. **Web browsers** allow users to install browser extensions, which contain executable code, and they can display rich content, such as executable scripts and multimedia content (e.g. Abode Flash). Web browsers present a very complex environment, and the rich content and extensions [19] supported can be exploited to compromise users’ web browsers. Users must trust the authors of websites they visit and extensions they install, as web browsers effectively execute remote content. **Cloud computing** allows owners of idle compute resources to lease out idle systems to third-parties to

978-1-4799-4093-6/14 $31.00 © 2014 IEEE
DOI 10.1109/CloudCom.2014.116

run applications on them. Cloud computing typically involves the leasing out of compute resources at an enterprise-scale, such as in the Amazon EC2 compute cloud. Nonetheless, resource owners (e.g. Amazon in the case of EC2) need to trust the tenants of these systems and the applications they run, as tenants are free to run any application on these leased resources. **Crowd-sourcing applications** typically provide a client which allows users to contribute answers to questions posed by application owners, either through user responses, or using the system's data. Such crowd-sourcing applications typically have clients on mobile devices (e.g. smartphones), and they collect or elicit responses such as the users' locations, to enable applications such as large-scale urban transit planning [2], mobile question-answering [12] and video collection [13]. Users need to trust these clients as they receive instructions from remote servers, and they access data from the users' systems. **Cooperative mobile computing** systems extend crowd-sourcing systems by enabling clients to communicate not just with a central server, but also amongst themselves to organize computation and exchange data. Clients may cooperate to jointly complete a task, such as sensor data collection [9]. In such systems, clients need to trust other clients in the system as well, whose identities are most likely unknown. Hence, there are many examples of systems where users run applications from unknown entities, necessitating a secure by construction approach for untrusted applications.

Proposed System: In this paper, we propose STOVE, a data and execution model (henceforth the "STOVE model") for untrusted applications developed by unknown entities, which provides strong guarantees that (i) when the code of an untrusted application runs on a system, it will not be able to harm or interfere with the OS or other applications on the system, and that (ii) the untrusted application can only access data authorized by the user. *STOVE represents a secure by construction approach to security for untrusted apps, requiring that untrusted apps be constructed so that they can be run securely without harm to their host systems.* Thus, users can run untrusted applications which are built using the STOVE model on their system, with the confidence that the untrusted application will not harm the user's system, nor make unauthorized access to the user's data. The STOVE model consists of two parts. First, STOVE strictly limits untrusted applications, and requires its code to be fully isolated from the system it runs on, preventing the code from directly accessing any data. This strict isolation of application code is verified by STOVE using formal logic. The execution of untrusted application code is strictly separated from any access to user data, and we verify this isolation of the untrusted code formally. Second, STOVE performs all data accesses on behalf of the untrusted application, so that all data accessed is observable. The STOVE model is realized by a verifier which statically proves the isolation of untrusted code, and a runtime system which accesses data on behalf of untrusted applications, providing access control. We note that while mobile resource leasing is an important use-case for the STOVE model, the STOVE model is general and applies to any untrusted application. *To the best of our knowledge, STOVE is the first model which allows the automatic proving of safety properties in unmodified machine-code binaries without a prior safe rewriting step, and is the first model which uses execution isolation to provide strict and observable access control for data accesses.*

II. Related Work

Security for Untrusted Code Execution: There are two main classes of techniques for ensuring that untrusted code is securely executed without the executing host coming to harm. First, the untrusted code can be isolated to limit or prevent damage due to its execution. Virtualization [20] can be used to run untrusted applications in an environment which is isolated from trusted applications, but this will prevent the untrusted application from accessing any data from the host, whereas untrusted applications may need access to host data. Software Fault Isolation (SFI) [21] and XFI [22] provide protection from untrusted code modules by rewriting them to render potentially dangerous operations (e.g. memory writes and jumps) safe. XFI improves on SFI by providing a verifier which verifies that the rewritten module meets the desired safety properties. The isolation properties which the STOVE model requires of untrusted application code are similar to the safety properties which SFI and XFI aim to provide, and like XFI, STOVE verifies that security properties are met using a formal logic. However, STOVE does not rewrite untrusted code, unlike XFI. Second, guarantees can be established to ensure that the untrusted code conforms to particular desired security policies. Proof-carrying code [23] (PCC) allows a code consumer to formally validate that a piece of untrusted code meets a desired security policy; PCC requires the code producer to prove that the code meets the security policy. Like PCC, STOVE also requires untrusted code to meet a security policy, but unlike PCC, consumers of untrusted code in STOVE perform the proof for themselves. PCC [23] does not specify how the proof for untrusted code is obtained, and allows for manual proofs. In STOVE, we aim to reduce the burden on code developers, and we aim to automatically prove that the untrusted code meets the desired security policy. Like the initial version of PCC [23], STOVE proves that untrusted code meets desired security policies at the machine code level.

Security Measures for Cloud Computing: SecureMR [14], Sedic [15] and Airavat [24] provide security for MapReduce [25], a popular distributed computing framework for cloud computing. SecureMR [14] tackles the threat of untrusted nodes, Sedic [15] provides privacy-aware scheduling for ensuring private data is not processed on untrusted nodes, while Airavat [24] provides security measures to protect a trusted MapReduce cluster which is executing untrusted jobs. STOVE tackles the problem of running untrusted third-party applications on a trusted system, whereas Sedic and SecureMR address the running of trusted code on untrusted systems. Similarly to Sedic and SecureMR, Juels [26] and Gennaro [16] propose cryptographic solutions to tackle the problems of untrustworthy cloud storage services and untrustworthy cloud compute services by providing provable data retrieval and computation. Only Airavat addresses the same threat model as STOVE of untrusted code running on trusted systems, but requires operating system-level modifications by using SELinux to impose Mandatory Access Control.

Security Measures for Mobile Resource Leasing: Most security mechanisms proposed for mobile resource leasing focus on preserving the integrity and privacy of data exchanged between users and application owners (e.g. in a crowd-sourcing system). Proposed privacy and reliability measures include privacy-preserving tasking [27] and data collection [4], [3],

[28], and the collection of accurate data [27]. Such techniques are orthogonal to STOVE: they provide system-level privacy and integrity for applications which lease mobile resources, whereas STOVE protects the local security and privacy of mobile devices when they lease their resources to other, unknown entities. PRISM [10] and RACE [8] address the local security of mobile devices running code from unknown entities, but they use a sandbox to restrict the behaviors of untrusted code. Software sandboxes need to restrict all behaviors which may be dangerous, and they must be correctly designed and implemented to fully prevent all dangerous behaviors. It is challenging to prevent all dangerous behaviors, whereas STOVE provides strict isolation of untrusted code which is verified using formal logic, providing a stronger guarantee.

III. Overall Approach

A. Definitions, Assumptions and Threat Model

In this paper, we consider the scenario where a user executes an application from an entity unknown to the user, on a system (which can be a full computer or a mobile device) owned by the user. We call this system the "host" of the application, and because the user owns the host, the host contains personal data belonging to the user, which the user may consider privacy-sensitive. As the user does not know the entity which developed the application, we assume that the user does not trust the application. We call the application the "untrusted application", or "untrusted app". Nonetheless, we assume that the user wishes to run the untrusted app on the host he owns because the user receives utility from running the untrusted app (e.g. contributing compute cycles to a volunteer computing project, or contributing data to a crowdsourcing system to receive results from other participants; we do not consider how to incentivize users to participate in such systems). We also assume that the untrusted app will require access to some of the user's potentially private data stored on the host, and that the user is willing to share some, but not all, of his data with the untrusted app. Finally, we assume that the untrusted app does not require a user interface, and performs only computation on data. This is due to the restrictions imposed by the STOVE model (limits on system call invocations) on untrusted apps.

We consider arbitrarily malicious and powerful attackers, i.e. untrusted apps may contain arbitrarily malicious code which may crash or alter the hosts they run on. We assume that hosts faithfully execute untrusted apps as provided by developers and return correct results (i.e. we do not concern ourselves with the correctness of results provided by potentially dishonest hosts). Instead, we focus on protecting hosts from the arbitrary code they receive in untrusted apps.

B. Goals and Non-goals

The main objective of STOVE is to protect hosts when they execute potentially malicious code in untrusted apps. We aim to meet the following goals in our design and implementation: (1) be transparent to underlying operating systems and platforms (e.g. smartphone platforms such as Android), and not require any changes to them; (2) not require access to the source-code of the untrusted app; (3) provide strong, provable guarantees of execution isolation, to provably ensure that untrusted apps cannot circumvent or affect the execution of any other applications or the operating system of the host; (4) not need to trust any runtime mechanisms (e.g. sandboxes) for code isolation; (5) ensure that all data accessed by untrusted apps is observable; (6) provide hosts with fine-grained and uncircumventable access control over data provided to untrusted apps; (7) minimize the Trusted Computing Base (TCB) of our system, by minimizing the system components which we must trust for the isolated execution of untrusted apps.

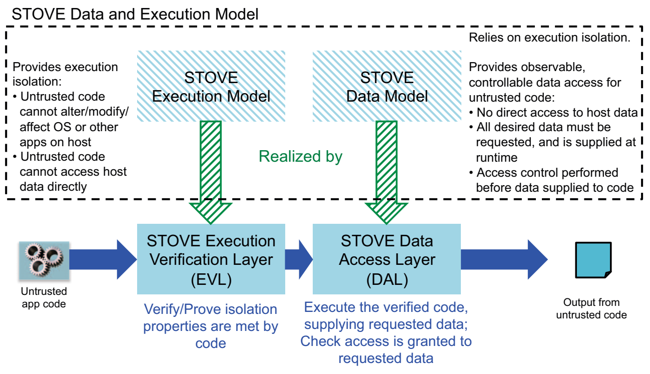
STOVE Data and Execution Model
Relies on execution isolation.
Provides execution isolation:
• Untrusted code cannot alter/modify/ affect OS or other apps on host
• Untrusted code cannot access host data directly
STOVE Execution Model
STOVE Data Model
Provides observable, controllable data access for untrusted code:
• No direct access to host data
• All desired data must be requested, and is supplied at runtime
• Access control performed before data supplied to code
Realized by
Untrusted app code
STOVE Execution Verification Layer (EVL)
STOVE Data Access Layer (DAL)
Output from untrusted code
Verify/Prove isolation properties are met by code
Execute the verified code, supplying requested data; Check access is granted to requested data

Fig. 1. Overview of the STOVE Data and Execution Model, and our approach.

We focus on the security and privacy issues facing hosts when they run code from, and provide data to untrusted apps. We do not address the security and privacy challenges that face application authors whose applications rely on outputs from individual device owners. For instance, we do not address the challenge of ensuring that participating hosts in volunteer computing or crowd-sourced systems return correct results and supply timely and correct data [27]. These challenges are orthogonal to the challenges facing hosts. Network attackers who may spoof, alter, or destroy network traffic are not relevant to our threat model as we consider only local threats facing the hosts when they run untrusted apps.

C. Overview of the STOVE model

The STOVE Data and Execution Model is a two-layered architecture to allow code from untrusted apps to be securely executed on a trusted host, while still allowing untrusted apps to access potentially private data on the host, without unauthorized data accesses. The key idea of STOVE is to isolate untrusted code as it runs on the host, to both (i) prevent the untrusted code from affecting (modifying or tampering with) the OS and other running applications, and (ii) prevent the untrusted code from accessing any data on the host directly. Thus, all data accesses are made on behalf of the untrusted app, which allows for all accessed data to be directly observable. This provides a single point to easily enforce access control policies on data accesses by the untrusted app. Figure 1 illustrates the properties provided by the STOVE Data and Execution Models, how they are realized by the Execution Verification Layer (EVL) and Data Access Layer (DAL), and the relationship between the Data and Execution Models. First, the Execution Model specifies the isolation requirements for code in untrusted apps. STOVE requires developers of untrusted apps to ensure that their code meets the isolation requirements specified by STOVE before their app is allowed to be executed on a host. Code which meets these isolation requirements specified by STOVE will (i) not be able to modify or affect the execution of the OS or any other application, and (ii) not be able to directly access any data on the host. Second, the Data Model requires that untrusted apps cannot directly access any host data (this is ensured by our execution model), and that untrusted apps must request desired host data. Then, this data will be retrieved from the host on behalf of the untrusted app and supplied to it. The STOVE model is realized via: (i) the STOVE Execution Verification Layer (EVL), which statically proves that untrusted code meets the required isolation requirements, and (ii) the STOVE Data Access Layer (DAL), which is a runtime layer that accesses data on behalf of the untrusted code, supplying that data to untrusted apps, while providing access control for host data. The STOVE EVL provides the basis for the strong security and privacy guarantees of the STOVE Execution Model, while the STOVE Data Model relies on the STOVE Execution Model to guarantee that the only host data which untrusted apps can access is the data provided to the app by the STOVE DAL, while providing access control for host data.

IV. STOVE Execution Verification Layer (EVL)

The STOVE Execution Verification Layer (EVL) realizes the isolation requirements of the STOVE Execution Model for code from untrusted apps. The EVL requires untrusted apps to provide code which meets a set of predetermined isolation properties (specifically, memory and control-flow safety, §IV-A). When these safety properties are met by the untrusted code, the code will not be able to alter, modify or affect the execution of the OS nor other applications on the host it runs on, ensuring that the host is secure even when it runs the untrusted code. The EVL statically verifies that the code from the untrusted app meets these safety properties by automatically proving safety theorems on the code. These safety theorems are a formal statement that the untrusted code meets the required safety properties. The EVL proves that the safety theorems hold on the code before executing it, and if the proof process fails, the code is rejected and is not executed.

A. Isolation: Memory and Control-flow Safety

Concretely, to ensure that hosts are safe and will not be affected by running untrusted code, the EVL requires that all untrusted code must meet memory and control-flow safety properties. Specifically, memory safety requires that all memory read and write instructions in the untrusted code must have target addresses restricted to user-addressable parts of virtual memory, while control-flow safety requires that all jump targets are restricted to addresses in the text section of the binary. The EVL also requires that all untrusted app code is statically compiled, and that no external library code is called. Thus, the memory and control-flow safety properties ensure that the host is isolated from the execution of the untrusted code, and the untrusted code will not be able to affect the execution of any other process on the host. Additionally, we restrict the system calls that can be made by the code, and we allow the code only the `write` system call, and with only file descriptor `1` (i.e. standard output). Later, the Data Access Layer (DAL) runtime supplies the code with any required input via standard input. The restriction on system calls strengthens the isolation of the untrusted code, and ensures that the code is not able to use system calls to directly access any host data, providing the isolation guarantee required by the Data Access Layer to ensure that the DAL can effectively provide access control for host data at a single point.

B. Proof Process and Automation

The STOVE EVL requires untrusted app code to be supplied as machine code programs, and the proof of the code possessing the required isolation properties is carried out on machine code programs. By proving our desired isolation properties on the machine code, we can exclude any compilers used to compile the code (as compared to proofs on high-level languages), and we can exclude any intermediate virtual machines (as compared to proofs on intermediate byte-code, e.g. Dalvik or Java byte-code), from our TCB. Hence, the TCB which we must trust for isolated execution of untrusted code consists only of the formalization of the machine code in logic [29], the HOL4 proof assistant [30], and the correct implementation of the processor in hardware.

The EVL formally proves the isolation of the untrusted code using a Hoare logic over machine code [29] and program analysis. We first decompile the machine code programs to a formalization of machine code instructions for the particular architecture (e.g. Intel x86, x86-64, ARM) of the machine code instructions in the logic of the HOL4 proof assistant [30]. While the machine code Hoare logic which we use [29] supports the 3 above architectures, we are currently developing automated proof techniques for the ARM architecture because of the widespread use of ARM processors in mobile devices, and because the ARM machine model formalization in [29] has been thoroughly validated [31]. Then, we automatically instantiate the desired memory and control-flow safety properties in the logic to obtain a theorem specific to the code being verified which asserts that the code meets our desired safety properties. Finally, the EVL automatically proves that the safety theorem holds, by mechanizing the proof steps, and using program analysis. We have described details of our initial work on the proof automation and mechanization for safety properties in [32]. The proof process of the EVL is static, and does not require executing the code being verified. As our proof process works on machine code, we do not require the source code of the untrusted app.

C. Developer Assistance

As the STOVE EVL requires untrusted apps to have code which meets our specified isolation properties, we intend that the STOVE EVL can also be used as a development tool to help developers write code which meets our required properties. The static verification process in the STOVE EVL can be used on the machine code of programs under development. Then, information from proof failures for non-conforming code can be used to suggest to developers changes at the source-code level, such as adding safety checks, to help their code meet our isolation properties. Note that the EVL does not need the source-code for proving safety properties on the machine-code, but the source-code can be used with information from proof failures to help developers write safe code. [32] describes some of our planned work in developer assistance.

V. STOVE Data Access Layer (DAL)

The STOVE Data Access Layer (DAL) serves as the single point through which all host data that untrusted apps wish to access must pass. The STOVE Execution Model ensures that untrusted apps are not able to directly access any host data, thus ensuring that the DAL is the only way that the untrusted app can access any host data. Untrusted apps specify the host data which they wish to access in a high-level description separate from their executable code. Then, the DAL runtime provides access control for host data, and the DAL runtime retrieves the host data requested by the untrusted app (if it is allowed) on behalf of the code, and the runtime supplies this data as input to the untrusted app. Thus, all accesses to host data by the untrusted app are made explicit, and the DAL does not have to control how the untrusted code may access host data (as compared to a sandbox approach).

A. Data Specification Language

We envision that untrusted apps specify the data they wish to access from the host using a high-level description. This is a logical description of the data source, such as sensor data from mobile devices (e.g. GPS or location data, accelerometer data), or photos and videos stored locally, or the address book or calendar of the host owner. The untrusted app can also specify the rate at which the data should be sampled, and the number of samples, for certain data sources, such as sensors. The DAL will also allow for untrusted code to receive remote inputs, for instance, in the case of volunteer computing projects, where untrusted apps may need to process data received from the volunteer project (e.g. receiving radio signal data to process for SETI@Home).

B. Fine-grained Access Control

The DAL also provides the host with an interface for specifying access control policies for host data, and for enforcing these access control policies when supplying data to untrusted apps. Since the DAL is able to observe all host data accessed by the untrusted app (because the DAL retrieves all data on behalf of the untrusted app), the DAL can enforce sophisticated access control policies which can be based on the actual data content. For instance, the DAL can enforce access control policies which allow host owners to limit the sampling rate and number of samples accessed by untrusted apps, which the DAL can enforce without having to monitor the execution of the untrusted app. Also, as the DAL retrieves all data for the untrusted app, the DAL can also transform the data before supplying it to the app. Thus, the DAL can perform data sanitization or anonymization, such as adding noise to GPS data, or pixelating photos. Other content-based fine-grained access control mechanisms, such as using face recognition and other image processing algorithms to identify privacy-sensitive photos [33], can also be used.

C. Logic-based Authorization

To further improve user confidence in the access control for user data by our DAL runtime, we intend to use access control logics to express access control policies, so that our runtime can justify based on user-specified preferences why each access control decision is allowed.

VI. Example Applications

Volunteer Computing: To recap, we illustrate how the STOVE model can be realized in a volunteer computing system such as BOINC [18]. Currently, some volunteer computing systems provide users with clients which download or receive executable programs from the volunteer computing project's servers. These programs are then run on the user's host to perform the volunteered task. Users who wish to participate in such volunteer computing system must trust that the authors mean no harm, and that their executable programs will not harm their host systems.

A volunteer computing platform which is developed using the STOVE model would comprise of code which meets the isolation requirements of the STOVE Execution Model. The authors of the volunteer task would first use the STOVE EVL to verify that their code meets the isolation requirements of the STOVE model, and they have to ensure that their code does not make use of any system calls (except for the `write` system call to return output to `stdout`), and that their code essentially performs only computation with no external communication. They can make use of our planned developer tools (§IV-C) to help them write code which meets our isolation requirements. Then, the authors of the volunteer task will use the high-level data specification language of the STOVE DAL to specify what inputs the untrusted code requires. Thus, the code of the app, together with the data request of the app, make up the STOVE-compliant app. Next, the volunteer computing project will make available their STOVE-compliant app to users. Then, on each user's host system, the STOVE EVL will first prove that the code of the untrusted app possesses the required isolation properties. Once the proof succeeds, the code can be executed. The STOVE DAL runtime executes the verified code, supplies the code with its requested inputs (as allowed by the user's access control policy), and collects the code's output at the end of the execution to return to the project's servers.

Mobile Resource Leasing: Mobile Edge Clouds: Next, we illustrate how the STOVE model would be realized in a mobile resource leasing scenario, in a mobile edge-cloud system. Consider a compute cloud made up entirely of mobile devices, or edge-devices, which allows an application owner to submit his application to run across multiple mobile devices belonging to individuals to make use of, or lease, the compute and data resources on individual users' mobile devices [34], [35]. Then, consider an mobile edge-cloud application which surveys its users to find out how many hours in a day each user spends walking or jogging. Owners of mobile devices would consider such an mobile edge-cloud application to be untrusted, as the users do not know the identity of the application developer. Such an untrusted app can be developed using the STOVE model to gain user confidence that the app is safe to run, and will not make unauthorized data accesses.

Our example application might request 30-second windows of accelerometer readings at a 5 Hz frequency at 5-minute intervals (i.e. every 5 minutes, the app is provided with a 30-second window of data) over a 24-hour period. Next, the untrusted app code will receive raw accelerometer readings from the STOVE DAL, and process these readings to determine if the user was walking, running, or not physically active. Then, the code will write its output as a time-series of whether the user was walking, jogging, or not active to its standard output. Finally, our runtime returns the code's results to the app owner which initiated the app. When our runtime receives the untrusted app, it will first extract the binary code, and the STOVE EVL will decompile this machine code to logic, and instantiate the safety theorems specifically for the code. The STOVE EVL automatically proves that the safety theorem is met before executing the code; if the proof fails, the untrusted app is rejected and the code is not executed. Next, the STOVE DAL runtime extracts the high-level description of the data requested by the edge-cloud app. If the app is allowed to access the requested data, the STOVE DAL collects and prepares the data requested by the app to pass to the untrusted code as input. Then, the runtime runs the untrusted code as a separate user process, and it passes the assembled input data to the standard input of the new process. Finally, after the edge-cloud code has completed its execution, the runtime collects the data written to the standard output of the process and returns it to the app owner.

VII. Conclusion and Future Work

We have presented the design of our STOVE Data and Execution Model for untrusted applications. STOVE provides a ground-up, *secure by construction* approach to securely execute and share data with code in untrusted apps by (i) restricting untrusted code to isolate it, verifying this isolation using formal logic, and (ii) performing all accesses to system data on behalf of the untrusted code. We have begun implementing the STOVE EVL, and we can currently automatically instantiate safety properties in Hoare logic theorems for ARM machine code programs. We plan to implement and evaluate the performance and security of both our Execution Verification (EVL) and Data Access (DAL) Layers, and demonstrate their usage in scenarios such as mobile edge-cloud applications. We also intend to explore the use of the STOVE model in other environments, such as crowd-sourcing, cloud computing, and cooperative mobile computing applications.

References

[1] "Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments to Grow 4.2 Percent in 2014," Jul 2014, http://www.gartner.com/newsroom/id/2791017.

[2] G. Chatzimilioudis, A. Konstantinidis, C. Laoudias, and D. Zeinalipour-Yatzi, "Crowdsourcing with Smartphones," *IEEE Internet Computing*, Sep/Oct 2012.

[3] E. Cristofaro and C. Soriente, "Short Paper: PEPSI: Privacy-Enhanced Participatory Sensing Infrastructure," in *ACM WiSec*, 2011.

[4] M. Asghar, A. Gehani, B. Crispo, and G. Russello, "PIDGIN: Privacy-Preserving Interest and Content Sharing in Opportunistic Networks," in *ASIACCS*, 2014.

[5] N. Lane, E. Miluzzo, H. Lu, D. Peebles, and T. Choudhury, "A Survey of Mobile Phone Sensing," *IEEE Communications*, Sep 2010.

[6] W. Zhang, Y. Wen, J. Wu, and H. Li, "Toward a Unified Elastic Computing Platform for Smartphones with Cloud Support," *IEEE Network*, Sep/Oct 2013.

[7] N. Aharony, W. Pan, C. Ip, I. Khayal, and A. Pentland, "Social fMRI: Investigating and shaping social mechanisms in the real world," in *IEEE PerCom*, 2011.

[8] B. Chandramouli, J. Claessens, S. Nath, I. Santos, and W. Zhou, "RACE: Real-time Applications over Cloud-Edge," in *SIGMOD*, 2012.

[9] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song, "CoMon: Cooperative Ambience Monitoring Platform with Continuity and Benefit Awareness," in *ACM MobiSys*, 2012.

[10] T. Das, P. Mohan, V. Padmanabhan, R. Ramjee, and A. Sharma, "PRISM: Platform for Remote Sensing using Smartphones," in *ACM MobiSys*, 2010.

[11] C. Shi, V. Lakafosis, M. Ammar, and E. Zegura, "Serendipity: Enabling Remote Computing among Intermittently Connected Mobile Devices," in *MobiHoc*, 2012.

[12] T. Yan, V. Kumar, and D. Ganesan, "CrowdSearch: Exploiting Crowds for Accurate Real-time Image Search on Mobile Phones," in *ACM MobiSys*, 2010.

[13] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan, "Scalable Crowd-Sourcing of Video from Mobile Devices," in *ACM MobiSys*, 2013.

[14] W. Wei, J. Du, T. Yu, and X. Gu, "SecureMR: A Service Integrity Assurance Framework for MapReduce," in *ACSAC*, 2009.

[15] K. Zhang, X. Zhou, Y. Chen, X. Wang, and Y. Ruan, "Sedic: Privacy-Aware Data-Intensive Computing on Hybrid Clouds," in *ACM CCS*, 2011.

[16] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *CRYPTO*, 2010.

[17] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An Experiment in Public-Resource Computing," *Communications of the ACM (CACM)*, Nov 2002.

[18] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *IEEE/ACM Grid*, 2004.

[19] A. Barth, A. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *NDSS*, 2009.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[21] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," in *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[22] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula, "XFI: Software Guards for System Address Spaces," in *OSDI*, 2006.

[23] G. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *OSDI*, Oct 1996.

[24] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and Privacy for MapReduce," in *NSDI*, Apr 2010.

[25] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, Oct 2004.

[26] A. Juels and B. Kaliski, "PORs: Proofs of Retrievability for Large Files," in *ACM CCS*, 2007.

[27] A. Kapadia, D. Kotz, and N. Triandopoulos, "Opportunistic sensing: Security challenges for the new paradigm," in *COMSNETS*, 2009.

[28] Q. Li and G. Cao, "Providing Privacy-Aware Incentives for Mobile Sensing," in *IEEE PerCom*, 2013.

[29] M. Myreen, A. Fox, and M. Gordon, "Hoare Logic for ARM Machine Code," in *Fundamentals of Software Engineering (FSEN)*, 2007.

[30] K. Slind and M. Norrish, "A Brief Overview of HOL4," in *TPHOLs*, 2008.

[31] A. Fox, "Formal specification and verification of ARM6," in *TPHOLs*, 2003.

[32] J. Tan, U. Drolia, R. Gandhi, and P. Narasimhan, "Poster: Towards Secure Execution of Untrusted Code for Mobile Edge-Clouds," in *ACM WiSec*, 2014.

[33] J. Tan, U. Drolia, R. Martins, R. Gandhi, and P. Narasimhan, "Short Paper: CHIPS: Content-based Heuristics for Improving Photo Privacy for Smartphones," in *ACM WiSec*, 2014.

[34] U. Drolia, R. Gandhi, and P. Narasimhan, "Enabling Edge-clouds," Tech. Rep. CMU-PDL-13-114, Oct 2013.

[35] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. Narasimhan, "The Case for Mobile Edge Clouds," in *IEEE Ubiquitous Intelligence and Computing (UIC)*, Dec 2013.