TinMan: Eliminating Confidential Mobile Data Exposure with Security Oriented Offloading

Yubin Xia†, Yutao Liu†, Cheng Tan†, Mingyang Ma†, Haibing Guan§, Binyu Zang†, Haibo Chen†

Shanghai Key Laboratory of Scalable Computing and Systems
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
§Department of Computer Science, Shanghai Jiao Tong University

Abstract

The wide adoption of smart devices has stimulated a fast shift of security-critical data from desktop to mobile devices. However, recurrent device theft and loss expose mobile devices to various security threats and even physical attacks. This paper presents *TinMan*, a system that protects confidential data such as web site password and credit card number (we use the term *cor* to represent these data, which is short for Confidential Record) from being leaked or abused even under device theft. *TinMan* separates accesses of *cor* from the rest of the functionalities of an app, by introducing a trusted node to store *cor* and offloading any code from a mobile device to the trusted node to access *cor*. This completely eliminates the exposure of *cor* on the mobile devices. The key challenges to *TinMan* include deciding when and how to efficiently and transparently offload execution; *TinMan* addresses these challenges with security-oriented offloading with a low-overhead tainting scheme called asymmetric tainting to track accesses to *cor* to trigger offloading, as well as transparent *SSL session injection* and *TCP payload replacement* to offload accesses to *cor*. We have implemented a prototype of *TinMan* based on Android and demonstrated how *TinMan* protects the information of user's bank account and credit card number without modifying the apps. Evaluation results also show that *TinMan* incurs only a small amount of performance and power overhead.

1. Introduction

Mobile devices are superseding desktop computers as a primary computing platform, thanks to the mobility, constant connectivity and application diversity. Nowadays, such devices are routinely used in security-sensitive contexts, e.g., personal finance, enterprise business [31], or even military [41]. As a result, mobile devices have been aggregating an increasing amount of users' sensitive data.

Problem. Our motivation comes from two facts: first, the plaintext of confidential data may exist in memory on mobile devices for a long time after being used. Yang et al. [61] analyzed 14 popular Android apps and found sensitive data (like login password) from the memory dump and disk dump in 13 apps after 10 minutes. Second, an attacker can steal the confidential data by getting the content of device memory and disk either by rootkit or spyware or gaining physical control. This is because current software-based access control mechanisms require a large TCB (including both Linux and Android framework) and all protections can be bypassed by physical attacks like mobile device cold-boot attack [44]. Even if a user has wiped all of the data, an attacker can still recover critical data from the phone [6].

Previous solutions. To protect confidential data in the storage of mobile devices, conventional wisdom suggests using encryption and deletion. For example, a user can adopt full-disk encryption [19, 30, 40] or BitLocker [4] to ensure that data on disk is encrypted. However, the decryption key is usually stored in memory in plaintext. There has also been extensive research on in-memory data protection by leveraging a trusted server [23, 61]. For example, CleanOS [61] encrypts sensitive data that is not recently accessed and saves the encryption key on a trusted cloud. However, the confidential data has to be ultimately decrypted in memory when being used, and thus may be vulnerable to both malware and physical attacks. Hence, it mainly limits but not mitigates data exposure. Further, the TCB (Trusted Computing Base) for commodity mobile devices is huge that includes nearly the whole software stack. Thus, an attacker can trivially exploit the OS's security vulnerabilities or leverage phishing attack to get all the keys and steal the user's confidential data.

Challenges. There are several challenges on protecting in-memory confidential data. First, once some confidential

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741977

data has been accessed on a device, it is likely to have residue in either memory [29] or storage [27, 52, 63], or both, as data must be in the form of plaintext when being processed. For example, many bank web sites require the client to hash the plaintext of the user's account and password, and use the hash value for login [1, 21]. However, after the login, the plaintext of password or hash value might remain on the device for a long time [61]. Even if an app explicitly deletes the confidential data after use, the underlying operating system may still have data residue in many places, such as the file system log, page cache, socket buffer, swapped files, or any other freed but not cleared memory pages and disk blocks [47]. An attacker may retrieve the confidential data from these locations. Second, it is hard to identify all the residue of confidential data on a device. Simply searching the plaintext of confidential data in memory and storage is not enough, since the data might be encrypted while the key is vulnerable. Third, the design of new solutions should be practical and deployable. It should be general and require no modification to existing apps, while the overhead introduced on both performance and power consumption should be modest.

Our solution. Since it is almost impossible to ensure the security of confidential data if it has ever existed on devices, we use a different method: storing no confidential data on the device, and thus the device will *have nothing to lose* when it is stolen. We classify sensitive data on a mobile device, and focus on the most critical one, named *cor* (short for COnfidential Record), as shown in Figure 1. Examples of *cor* include password, bank account, social security number, credit card number, whose access patterns differ significantly from regular private data such as emails, messages, photos, which are directly manipulated by the user. Our observation is that since the privacy of the *cor* is of vital importance, the plaintext of *cor* is usually not displayed on mobile devices, even to the users themselves. Thus, from user's perspective, there is no difference between accessing *cor* locally and remotely, which makes our *on demand offloading* possible.

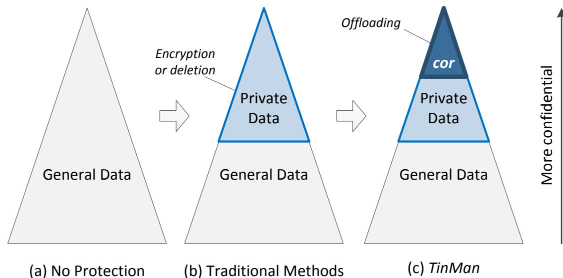


Figure 1: *TinMan* separates *cor* from regular private data and enforces its protection

In this paper, we present *TinMan*, a system that aims at protecting *cor*s by completely eliminating them on a device, so that there is no way for an attacker to steal *cor* from the device even if she can physically control the device. *TinMan* achieves this by ensuring that all *cor*s are stored and accessed exclusively on a *trusted node*, which can be a server inside a company or a virtual machine on a trusted cloud, other than the device itself. Instead of requiring applications to split the logic of accessing *cor* and use some new APIs provided by the trusted node, *TinMan* introduces a security-oriented offloading mechanism to seamlessly support existing applications. *TinMan* stores a placeholder for each *cor* on the device, while all of the *cor*s are stored on a trusted node. Each placeholder has the same size as the corresponding *cor* and is tracked using tainting mechanism. When an application accesses some tainted placeholder, *TinMan* migrates the application execution to the trusted node to access the corresponding *cor*, and migrates it back after that.

Different from traditional offloading mechanisms like COMET [25], which are typically implemented at the application level, security-oriented offloading usually involves multiple levels including the native library and the OS, as Figure 2 shows. In *TinMan*, we develop two techniques to enable multi-level offloading, named *SSL session injection* and *TCP payload replacement*, which are used to make states synchronized at SSL layer and TCP layer, respectively. The application level states are synchronized based on COMET [25]. We also develop an optimization named *asymmetric taint tracking* to minimize the overhead introduced by taint tracking on the mobile device.

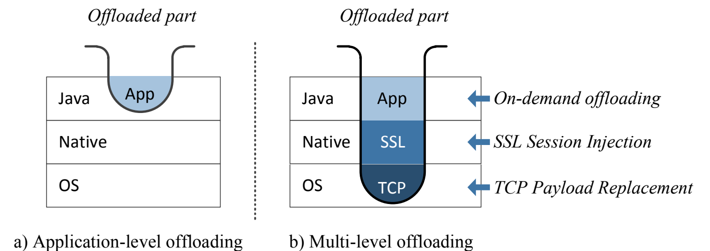


Figure 2: System involved offloading

Evaluation on real-world applications. We have implemented a prototype of *TinMan* with on-demand offloading, and asymmetric taint tracking on Android[1] to demonstrate its effectiveness and efficiency. We use the BankDroid app and the default Android web browser as examples to demonstrate how *TinMan* protects a user's account information and credit card numbers. Our evaluation shows that *TinMan* only introduces small performance and power overhead, since the portion of offloaded code (e.g., web login and payment) in an app is small.

Contributions. In summary, *TinMan* makes the following contributions:

1. A novel scheme for protecting *cor*s on the mobile device by completely eliminating their exposure on the device.

[1] The mechanism of *TinMan* is general and not necessarily coupled with Android.

2. A design of security-oriented offloading that contains multi-level of state synchronization, which requires no application modification.
3. A working prototype based on Android, and case studies using BankDroid and the default Android web browser. The evaluation results illustrate the effectiveness and small overhead incurred in *TinMan*.

2. Problem and Overview

2.1 Motivation

In this section we further describe our motivation in two aspects: critical data exposure and data stealing through either software or hardware attack.

For the first aspect, Yang et al. [61] analyzed 14 popular Android apps by searching critical data like login password in memory dump, and found that 13 apps left such data in plaintext in memory 10 minutes after using them. Such data could exist in all kinds of system cache, swapped files, any freed but not cleared memory pages or disk blocks, and so on [47]. Even worse, since it is usually not convenient to type on small screens, most users choose to save some confidential data (e.g., passwords) on the device to avoid typing them everytime, which could further increase data residues in the system.

For the second aspect, the openness of the mobile software stack and its wide adoption have stimulated not only a burst of mobile applications, but also an explosive growth of malware [68] that divulges users' security-sensitive data. As long as the user's device is infected by rootkit malware, any data on the device is in danger since the operating system which can read any data in memory is now controlled by the malware. Even if a mobile device is not rootkited, there are also many ways for an attacker to gain full control over it if given physical access, even with its screen locked. Many bugs have been reported that allow attackers to bypass the lockscreen [3, 7–10]. For example, on a 2013 Sony Xperia phone, an attacker could easily bypass the lockscreen by entering "*#*#7378423#*#*" as an emergency call and then clicking on "NFC Diag Test" [10].

Worse still, the convenience for carrying also makes mobile devices highly vulnerable to theft and loss [5, 16]. This exposes users' confidential data to not only software-based threats [3, 7–10] , but also physical attacks [11, 44]. In more sophisticated attacks, Thomas Cannon [11] has demonstrated cracking a locked HTC mobile phone by a special SIM card and a specially formatted MicroSD card and gained full access. Muller et al. [44] issued cold boot attacks on a Samsung Galaxy Nexus phone, which had an encrypted disk partition to store user's data. They retrieved the encryption key stored in memory and successfully decrypted the disk partition. Thus, how to protect confidential record in memory against software or even physical attack, is an important and urgent problem.

As mentioned, *TinMan* distinguishes *cor* (e.g., password and credit card number) from other private data (e.g., photos and emails), and offers stronger protection for *cor*. We argue that the security of *cor* is more significant than the rest. Most apps are backed by powerful cloud servers and act as an entry point to the cloud, thus usually the phone itself only stores a small piece of data. For example, the default Mail app on iPhone just shows the latest 50 mails. However, if a *cor* (e.g., the password of user's account) is leaked, an attacker can access *any* data on the cloud. Further, many *cor* are related to finance, e.g., credit card number. If such data are retrieved by an attacker, it could directly result in financial loss to users or companies. Finally, the US military announced that it would equip soldiers with Android equipment for accessing classified documents [41], which makes the *cor* stealing problem even more serious.

2.2 Goals

Our primary goal is to offer strong protection of confidential data (*cor*s) even under physical threats. More specifically, the goals are as following:

1. *No plaintext of* cor *on mobile devices:* In order to ensure zero residue of *cor* on the mobile device at any time, our design requires that no plaintext of *cor* would ever exist on the device.
2. *Minimal TCB on mobile device:* Our system should not depend on the security of software stack on the mobile device.
3. *Transparency to applications:* The system should mostly retain backward compatibility to support existing apps so that it is not very hard to adopt in the existing software stack.
4. *Design for mobiles:* The design should consider the resource scarcity of mobile devices, especially the battery consumption.

2.3 Threat Model

Our threat model considers any form of data stealing, including application-level, OS-level and even hardware-level in the mobile devices. *TinMan* makes no assumption on any software running on the mobile device. This assumption significantly reduces the TCB on the mobile device comparing with other systems such as KeyPad [23] or CleanOS [61] that need to trust all software running on the device. An attacker can install any malware or even reinstall a malicious OS on the device, or tamper with the device physically at any time.

The trusted node used for storing and accessing *cor*s is trusted at all times. It can be considered as a service provided by the device manufacturer as a value-added service.As long as a user trusts the device vendor, it is straightforward for them to trust the trusted node. Another way is for the users to deploy such services themselves. For example, a company can use its private cloud to do so. The trusted node is con-

sidered to be more secure than mobile devices thanks to its professional administration and physical protection.

We assume that the initialization of *cor* on the trusted node is done in a safe environment in advance, which is a one-time effort. When users want to access *cor* on the device, they can select from a list of *cor* description instead of typing (e.g., the password) into the device. An example is shown in Section 4.1.

2.4 Background on COMET

TinMan leverages the offloading engine of COMET [25]. COMET is an open-source project based on Android's Dalvik virtual machine that offloads computating-intensive workloads from mobile phone to a more powerful server. It uses DSM (Distributed Shared Memory) to synchronize memory states between endpoints, which also enables full multi-threading support and migration of a thread at any point during execution. By taking full advantages of Java's memory consistency model, COMET establishes a "happen-before" relationship between operations by synchronization primitives within an app, and minimizes the need of state synchronization. Such design greatly simplifies the system and also reduces the performance overhead.

2.5 Design Overview of *TinMan*

In order to eliminate *cor* exposure, *TinMan* introduces a trusted node to store and access *cor*. It introduces a new mechanism, named *on-demand offloading*, that migrates only the logic that processes *cor* to the trusted node for execution, and leaves most logic of an app executing on the mobile device, thus can keep good user experience and cause little performance overhead. In order to avoid app modification, *TinMan* saves placeholders on the client. It taints these placeholders and tracks their data flow using a modified TaintDroid. As long as a placeholder is accessed, the app is migrated to the trusted node. After the access, the app is migrated back to the mobile device, as Figure 3 shows. More details are in Section 3.1. We also optimize the tainting mechanism to reduce the overhead, as presented in Section 3.5.

On-demand offloading brings a new problem on SSL based network I/O: how can the trusted node transparently join an SSL session established between the client and a web site? For example, when a client needs to send a *cor* (e.g., credit card number) to Amazon, it does not have the real data of *cor*, which is stored on the trusted node. Thus, the process of *cor* sending must be done by the trusted node. However, since the connection session is established between the app on the client and Amazon, the trusted node must join the session without awareness of either the app or Amazon. *TinMan* solves this problem by supporting multi-level offloading, which involves not only at the application level, but also the native library and the operating system. Thus, the entire process of *cor* sending is migrated to the trusted node, and the states of SSL and TCP session are kept

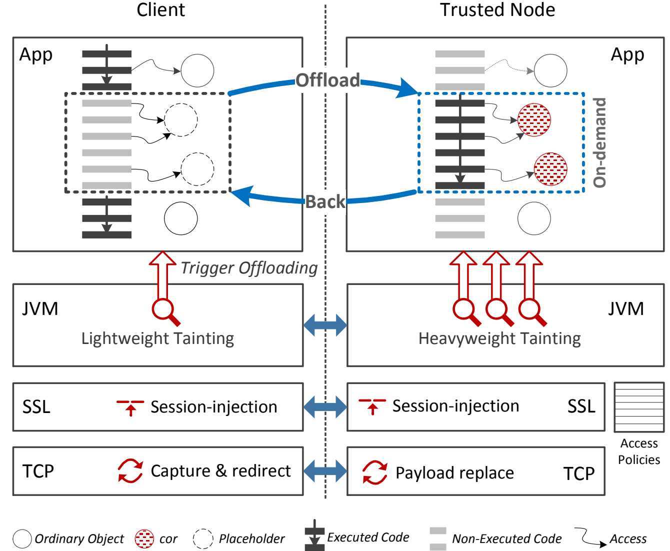


Figure 3: Design Overview of *TinMan*

synchronized between the client and the trusted node to keep apps unmodified, as shown in Figure 4.

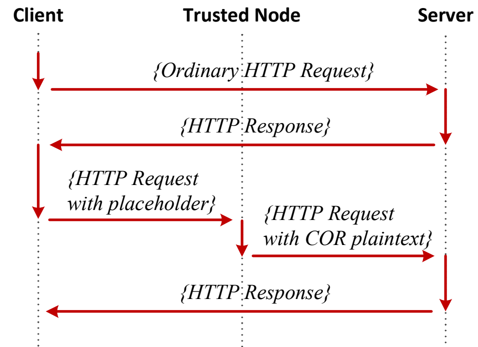


Figure 4: Communication between client, trusted node and server. The trusted node sends HTTP requests containing *cor* and syncs states with the client.

3. Security-oriented Offloading

This section describes the design and implementation details. There are two questions need to be considered:

1. Which parts of an app should be offloaded and how to identify these parts automatically?
2. How to enable multi-level offloading without modification to apps?

For the first question, *TinMan* uses taint tracking to monitor *cor* accessing as trigger points of offloading, and ensures that all the access to *cor* are executed on the trusted node. For the second question, since *TinMan* leverages COMET for application-level offloading, this section will focus on two other layers: SSL and TCP offloading.

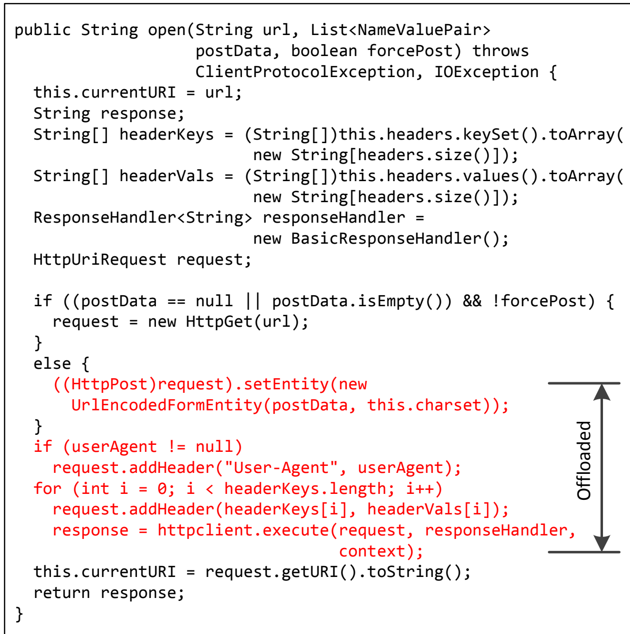
3.1 On-demand Offloading

A *cor* is a piece of data that is extremely critical and that it is even not displayed to users. *TinMan* uses five metadata to represent each *cor*, as shown in Table 1. Each *cor*'s plaintext is tainted with a unique ID on the trusted node, and the corresponding placeholder is tainted with the same ID on the client. The initialization of *cor* tainting is done by the user without entering the plaintext of *cor* into the device. Examples of initialization can be found in Section 4.

Field	**Description** *cor*
ID	Unique for each *cor*
Plaintext	Stored exclusively on the trusted node
Placeholder	Dummy data on client with the same size of *cor*
Description	Shown to the user for selection (optional)
Whitelist	A list of legal domains that the *cor* could be sent to

Table 1: *cor* abstraction

We have two observations on the access pattern of *cor*, which show that it is practical to offload *cor* accessing. First, the amount of code that accesses *cor* in an application is relatively small. Take BankDroid as an example, only the code shown in figure 5 is offloaded, which indicates that most of the app logic is executed on the client.



```
public String open(String url, List<NameValuePair>
                  postData, boolean forcePost) throws
                  ClientProtocolException, IOException {
  this.currentURI = url;
  String response;
  String[] headerKeys = (String[])this.headers.keySet().toArray(
                        new String[headers.size()]);
  String[] headerVals = (String[])this.headers.values().toArray(
                        new String[headers.size()]);
  ResponseHandler<String> responseHandler =
                        new BasicResponseHandler();
  HttpUriRequest request;

  if ((postData == null || postData.isEmpty()) && !forcePost) {
    request = new HttpGet(url);
  }
  else {
    ((HttpPost)request).setEntity(new
      UrlEncodedFormEntity(postData, this.charset));
  }
  if (userAgent != null)
    request.addHeader("User-Agent", userAgent);
  for (int i = 0; i < headerKeys.length; i++)
    request.addHeader(headerKeys[i], headerVals[i]);
    response = httpclient.execute(request, responseHandler,
                                  context);
  this.currentURI = request.getURI().toString();
  return response;
}
```


Figure 5: The offloaded part of BankDroid code. The *postData* and *request* are *cor*. No change to app is needed.

Our second observation is that *cor* access has both spatial and temporal locality. We tracked the propagation of *cor* and data generated from *cor*, and found that most of the tainted data during *cor* accessing on the trusted node are temporary and do not need to be synchronized to the client, which significantly reduces the data transferred through network.

TinMan builds the application-level offloading engine on COMET [25], which is a DSM based offloading mechanism. In order to support security-oriented offloading, it is not allowed to synchronize the plaintext of *cor* to the mobile device. In another word, for an app, the memory on both the client and the trusted node are identical, except for the *cor* parts. For *cor*, the offloading engine will only transfer its ID between the mobile device and the trusted node, then the two sides will fill the memory with placeholder and *cor* plaintext, respectively.

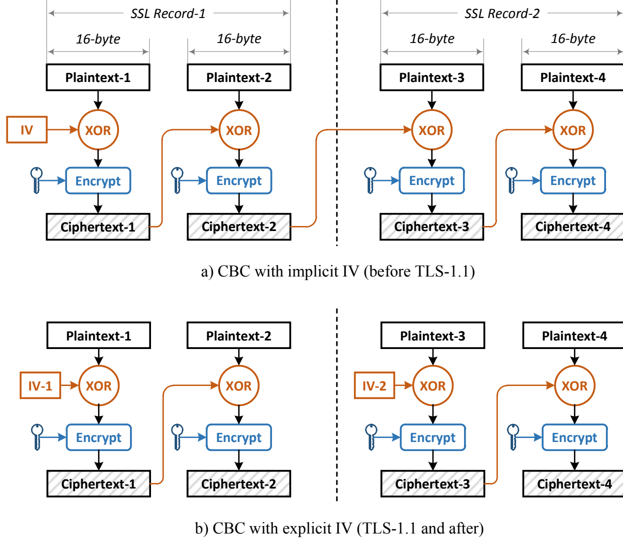
To enable on-demand offloading, *TinMan* tracks the dataflow of placeholders on the client. An app will be migrated to the server only when it accesses a placeholder. The app will be migrated back to the client in two cases: (1) *cor* has not been accessed on stack for a predefined threshold of duration; (2) the app invokes a non-offloadable native function, such as I/O operations or functions provided by third party libraries. One exception is that, if a *cor* is being sent through network, the operation must be done by the cooperation of the client and the trusted node, as described below.

3.2 SSL Offloading: Session Injection

TinMan implements security-oriented offloading by migrating a part of an SSL session from the mobile device to the trusted node. The basic process works like this: an app establishes an SSL session with some server and communicates with the server using it. When the app needs to send a *cor* to the server, it is migrated to the trusted node to send one or more SSL records, and migrated back after that. Thus, the *cor* is actually sent by the trusted node. Two questions arise: which states of an SSL session should be synchronized? Does the state synchronization violate our security requirements?

The first question depends on the encryption methods used by different versions of SSL. If an app uses stream encryption, such as RC4 or CBC, such offloading needs to synchronize SSL states. For different cipher algorithms, the states of an SSL session that need to be synchronized between the mobile device and the trusted node are different. For example, if RC4 is used, each block is independent of the others. Thus, only the metadata (e.g., session key, encrypted method) of SSL session will be sent from mobile device to the trusted node. If an app uses CBC, then the state synchronization depends on the version of the algorithm. Before TLS-1.1 (Transport Layer Security), the process of CBC in SSL was shown in the top part of figure 6. Each SSL record uses the last ciphertext block of the previous record as its IV (Initialization Vector). The first SSL record uses a random number as its IV, and attaches the IV with the record. Such mechanism is known as *implicit IV*. However, the practice of re-using the last ciphertext block of a message as the IV for the next message is widely known to be insecure. [2] Thus, from TLS-1.1, each SSL record uses a separated IV, aka., *explicit IV*, as shown in the bottom part of figure 6.

[2] BEAST attack [2] (CVE-2011-3389)



a) CBC with implicit IV (before TLS-1.1)

b) CBC with explicit IV (TLS-1.1 and after)

Figure 6: CBC using implicit and explicit IV.

For apps using *explicit IV*, since each SSL record is independent of the others, *TinMan* can be used without synchronizing IVs back to the mobile device. In order to make *TinMan* work even if an app uses CBC with *implicit IV*, both the mobile device and the trusted node need to send their last ciphertext block to each other as IV, which might leak the plaintext of *cor* to the mobile device.

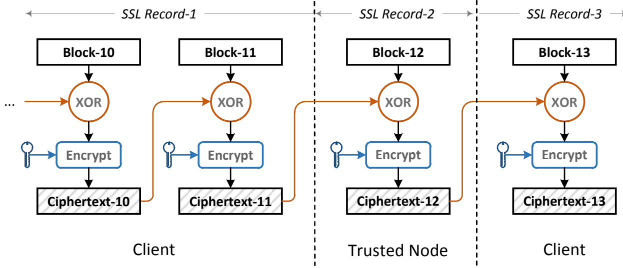


Figure 7: An example of secret leakage through synchronizing IV in CBC using implicit IV.

Take figure 7 as an example, the mobile device needs to send ciphertext-11 to the trusted node as IV, and the trusted node also needs to send back ciphertext-12 to the mobile device to continue the SSL session. From the mobile device's perspective, the IV it gets from the trusted node is actually the ciphertext of block-12. Thus, it is easy for the mobile device to derive plaintext of block-12 by decrypting the ciphertext of block-12 and then XORing with ciphertext-11, as following:

$$P_{12} = decrypt(C_{12})_{key} \oplus C_{11}$$

where P stands for *plaintext*, C stands for *ciphertext*, and $\oplus$ stands for XOR operation. Now the mobile device is able to get the plaintext of block-12, which contains *cor*.

In order to protect against such an attack, *TinMan* modifies the SSL library on the client to make it use an SSL version newer than TLS-1.0. In the SSL protocol, the client and server negotiate the version of SSL to use during the handshake phase: In a *ClientHello* message, the client first sends the highest version number that it supports. The server is then supposed to pick the most recent protocol version that both support. In our evaluation, the Android SSL library and all the web sites we test support TLS-1.2. Thus, we modify the Android SSL library to check and ensure that the SSL version used by apps must be newer than TLS-1.0.

3.3 TCP Offloading: Payload Replacement

In order to keep *cor* from the mobile device, the SSL record containing *cor* must be sent by the trusted node. One naive way is that, before the client sending the SSL record with placeholder, it synchronizes the TCP states, including the sequence number etc., to the trusted node. The trusted node uses the states to generate a TCP header and the *cor* to generate the packet payload, and sends the packet to the web server. After that, the trusted node synchronizes the TCP states back to the client, which continues to run as if it has sent the packet.

However, such an implementation is intrusive since it needs to change the TCP stack of both the client and the trusted node. In order to avoid such a modification, *TinMan* adopts a mechanism named *payload replacement*. Basically, the client first generates a packet with the placeholder. The packet is then redirected to the trusted node instead of being sent to the server. The trusted node replaces the payload of the packet with *cor*, and then sends it out. The packet is sent by the trusted node on behalf of the client, thus the server is not aware of the process of payload replacement.

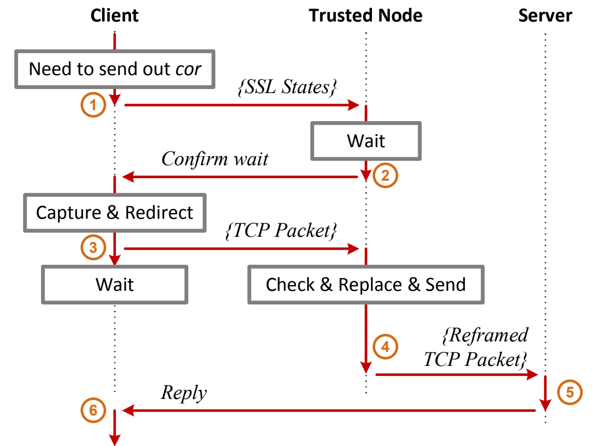


Figure 8: Process of payload replacement

The detailed process is shown in figure 8. ① As long as the client needs to send *cor* through network, the SSL library will first send its SSL states to the trusted node, including the SSL session key. ② The trusted node then waits for a redi-

rected packet. ③ The client sets a packet filter to capture the packet. It then continues to send the data, which goes through the TCP stack to generate a packet. The packet is captured by the filter, and is redirected to the trusted node. Now, the client is waiting for a reply from the server. ④ The trusted node first checks the access policies (see Section 3.4), and then generates a new payload with *cor* encrypted with the SSL session key, and replaces the original payload of the packet. The TCP header is not changed, whose source address is the client's IP and destination address is the server's IP. After that, the trusted node sends the reframed packet out. ⑤ Once the server receives the packet, it just processes as if the packet is sent by the client, and sends a reply to the client. ⑥ The client continues to run after it receives the reply.

3.4 Security Enforcement on Trusted Node

An attacker who has gained control over the client may cause the trusted node send *cor*s to some malicious server to steal *cor*s by running malicious code on the client which will then be offloaded to run on the trusted node. In order to defend against such attacks, *TinMan* introduces two kinds of binding on the trusted node to restrict the way of accessing *cor* by the offloaded code.

The first binding is between the application and the *cor*. For example, *TinMan* can set a rule to ensure that the password of Facebook could only be accessed by Facebook's official app. The trusted node identifies an application by the hash of its dex [3] file. Every time a *cor* is accessed, the trusted node will calculate and check the hash to ensure that the running application has the access permission. This binding can also prevent phishing attacks, where a malicious app pretends to be Facebook's official app and accesses user's login password. Once the app is offloaded, the trusted node will identify that it has no permission and refuse its access to *cor* due to the mismatch of hashes. Meanwhile, a user can request the trusted node to revoke access permission of *cor* to prevent abusing *cor*. For example, if a user realizes that her phone is stolen, she can revoke all the *cor* access permissions from the lost device.

The second binding is between *cor* and a target domain. For example, *TinMan* can set a rule to state that "the password of Facebook could only be sent to the domains belong to Facebook website", or "the private key of bitcoin cannot be sent out", etc. Thus, when a *cor* is sent out, the target domain will be checked to ensure the user-defined rules are not violated. One potential problem is that the granularity of domain sometimes can be coarse-grained. For example, an attacker who has controlled a mobile device may make the device post the user's Facebook password to the attacker's own Facebook page as a comment. In this attack, the password is still sent to some IP within Facebook's domain, thus the trusted node will send the password to that IP. We observe that most well-known web sites use dedicated machines for authentication, like Facebook, Google, Linkedin, etc. The IP addresses of the authentication machines are different from other machines. Thus, the trusted node will reduce the range of the whitelist to only the IPs that are responsible for authentication for password to solve the problem above. The address list is kept by the trusted node, which will be updated periodically.

The trusted node also deploys app analysis tools to improve the security. When an application is offloaded, the trusted node first checks whether the application is one of the known malware by calculating the hash of the code and search in a malware hash database. Currently we only apply a relative small database with around 1,000 malware. It is straightforward to leverage existing static analysis based on our framework. It is our future work to deploy more dynamic analysis methods on *TinMan*. Meanwhile, all of the *cor* access activities on the trusted node are logged for auditing. Each record includes timestamp, application hash, *cor* ID and network domain. Any abnormal activity will be reported to the user.

3.5 Optimization: Asymmetric Tainting

In *TinMan*, tainting is used on both the mobile device and the trusted node. On the mobile device, tainting is used only to trigger offloading, which is a much simpler scenario than the one on the trusted node. We classify the propagation of taints into four types, and find that only two of them are needed on the client. Thus, we adopt a lightweight tainting mechanism on the mobile device for optimization, while leave the full-fledged one on the trusted node.

There are two types of data in Java: primitive-type data (e.g., int, double, long, object reference), and objects. The stack of a Java application only contains primitive-type data, while the heap only contains objects. Correspondingly, there are four types of data transmission, as shown in Table 2.

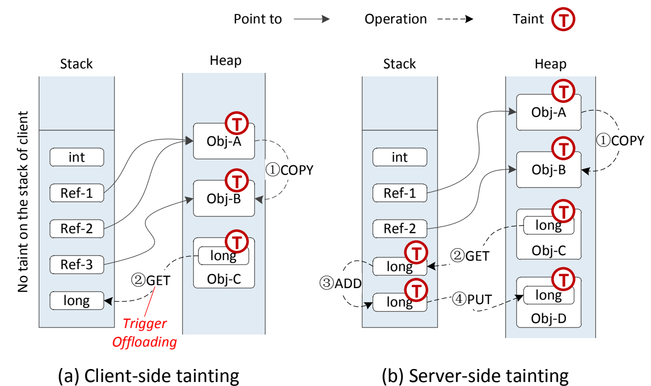


Figure 9: Asymmetric tainting on the mobile device and the trusted node. Only heap-to-heap and heap-to-stack are needed on the mobile device.

Since the JVM ensures that each data must be moved from heap to stack before being accessed, the mobile device only needs to handle heap-to-heap propagation of taints, and

[3] A dex file is a compressed file that contains Android executable files

Type	Example	On client	On trusted node
heap to heap	① in fig 9-a, ① in fig 9-b	Yes	Yes
heap to stack	② in fig 9-a, ② in fig 9-b	Yes	Yes
stack to stack	③ in fig 9-b	No	Yes
stack to heap	④ in fig 9-b	No	Yes

Table 2: Four types of taint propagation

handle heap-to-stack by triggering offloading and leaving everything else to the trusted node. The other two types of operations will not occur on the mobile devices.

More specifically, the heap-to-heap operations include object *clone*, *arraycopy* and *memcopy*. Take *clone* as an example, when an object is being cloned, the JVM creates a new object on the heap and fills it with data from the original object. In this process, both objects are on the heap and no data are transferred to the stack. Only the generated object needs to be tainted. The operations of object *clone* are rare in a typical Java program. A more common case is copying a reference instead of cloning an object. It is noted that a reference of a tainted object is not tainted itself. Copying such a reference will not trigger any taint propagation, since the new reference also points to the same tainted object.

The taint propagations between stack and heap are a little more complicated. We use following piece of code to explain.

```
1.  // passwd is tainted
2.  concat_1(String s, String passwd) {
3.     char c = passwd.charAt(0);
4.     char d = c;
5.     s.append(d);
6.  }
```

Figure 10: Example of taint propagation between stack and heap

Initially, the string *passwd* is tainted and *s* is not tainted; both objects are on the heap. *c* is a variable of primitive-typed *Char* on the stack. At line 3, when *c* is assigned the value returned by method *charAt*(), there is a *GET* operation that transfers data from the heap to the stack, and *c* is then tainted. Line 4 is a stack-to-stack operation, where *d* will be tainted on the stack. At line 5, *d* will be PUT from the stack to *s* on the heap, and the object *s* is then tainted.

```
1.  // passwd is tainted
2.  send(Socket s, String user, String passwd) {
3.    String httpRequest;
4.    httpRequest = "username=" + user;
5.    httpRequest += "&passwd=";
6.    httpRequest += passwd; // migrate to server
7.    s.send(http_request);  // migrate back
8.  }
```

Figure 11: Example of offloading triggering

During the four types of taint propagation, stack-to-stack operations are the most common, and thus bring most performance overhead. In *TinMan*, tainting on the client is used only for triggering offloading. There are two types of operations on tainted objects that will cause offloading: first is a heap-to-stack operation, such as the instruction at line 3 in the above *concat_1* code snippet. The second is heap-to-heap operation that will generate a new *cor* ID, such as the code at line 6 of the *send* code snippet, where the data of *passwd* is concatenated to *httpRequest*. The other two types of operations of *cor* do not exist on the client, since if so, there must be some heap-to-stack operations before, which will trigger the offloading. Thus, the two types of taint propagation do not need to be implemented on the client.

The trusted node adopts a full-fledged taint tracking mechanism, which tracks the tainted data in both stack and heap to keep the accuracy of tainting. It not only handles data propagation for primitive types on the stack, such as *add*, *move* or *multiple*, but also handles data transferring between heap and stack. It intercepts Java data movement opcodes to implement taint propagation, which is similar to prior work [20]. Such tainting in the trusted node is precise and heavyweight, and is used as to support extensions of security monitoring (e.g., *cor* access control) on the trusted node by combining the two types of tainting mechanisms together. *TinMan* incurs low performance overhead on the mobile device, while still keeping the accuracy of tainting.

Another way to reduce the performance overhead on the mobile device is to adopt selectively tainting, which enables tainting only for certain security critical apps instead of all. A user needs to specific those critical apps.

3.6 Other Implementation Details

Instead of building all functionalities from scratch, *TinMan* extends COMET [25] to support security oriented on-demand offloading, and ports TaintDroid [20] with asymmetric tainting. The tainting module on both client and server is about 3,870 LOC (Lines Of Code) by using *#ifdef* to share the code. To implement on-demand offload, we modify around 1,730 LOC, including functions for offloading triggering, *cor* synchronization, SSL session injection and TCP payload replacement. We also optimize to synchronize less states on the initialization stage. On the client, we slightly modify the Android application framework to change the edit widget to display *cor* list for the user to select, as well as I/O operations, including accessing disk for storing and loading taint label. This adds around 550 LOC. Also on the client, we use iptable to capture the packet containing *cor* placeholder. Our modified SSL library will mark such packets by writing a specific value in *type* field of SSL record. (There only 4 types of SSL record while the *type* field has 8 bits.)

4. Use Cases

TinMan offers a platform to offload security-critical operations from mobile devices to a trusted node. Such platform is general and can be leveraged to benefit different apps. In this

section we present two typical examples: protecting password of various banks used in BankDroid, and credit card number used in a web browser.

4.1 Protecting Password in Applications

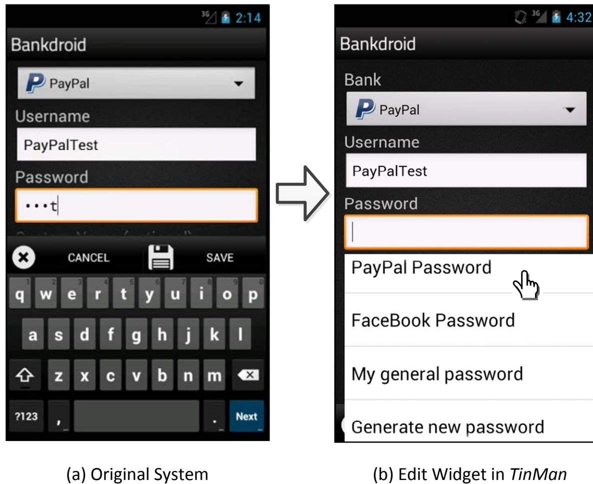


Figure 12: BankDroid. User selects a password instead of typing it.

We use BankDroid, a popular bank account management application on Android, as an example. BankDroid is an application that can retrieve user's bank transactions and show them in a nice form. We assume that the user has already initialized all the passwords as *cor* on the trusted node, with proper descriptions.

When BankDroid is opened for the first time, it will ask the user to enter account and password for different banks. A modified password widget will display a list of description of all the *cor*, as shown in Figure 12. The widget is a part of Android framework, thus no modification of the app is needed. Instead of typing the password, a user selects the corresponding password from the list. Then the app will use the corresponding *cor* placeholder as password, which is tainted and tracked. The entire process is transparent to BankDroid.

Some bank site requires hash of account/password for login. When BankDroid calculates the hash of password, it actually accesses the tainted placeholder and triggers offloading. Then, the client migrates BankDroid to the trusted to calculate the hash of account/password. The tainting mechanism on the trusted node ensures that the hash value is a new *cor*.

When the calculation is done, BankDroid is migrated back to the phone. From the app's perspective, it now gets a hash value, which actually is another placeholder. It then tries to send the placeholder of hash value to the bank site. The SSL layer then synchronizes session states to the trusted node, and marks the SSL record. Once the packet with the marked data is sent out, it is intercepted and redirected to the trusted node. The trusted node is waiting for the packet since SSL state synching. It receives the packet, checks the access control policies, and reframes the packet with *cor* as its payload. It then sends the packet out to the web server on behavior of the client. The web server, which is not aware of the whole payload replacement process, receives the request and replies.

Since *TinMan* allows a user to select password instead of typing them, then some others (like his kids) might also be able to access those apps if the user does not use a PIN code to lock the phone. In order to ensure only the user can access *cor*, one solution is to add a local authentication every time a *cor* is selected. Such check has nothing to do with either the application or the server. It can be done either by software like a PIN code or some existing hardware devices such as a fingerprint checker.

4.2 Protecting Credit Card Number in Browser

It is common to use credit cards for online payment. For example, when registering to attend a conference, the user is required to fill a web form and use credit card to pay the registeration fee. All of the credit card information is needed, including the card number, the expiration date, the security code, and other information. If the user enables the auto-fill option of browser, these data may be on the device permanently and could be retrieved by an attacker.

By using *TinMan*, we modify the rendering engine of default browser to add a dropdown list widget adjacent to each input widget. A user will choose the *cor* of credit card number, security code, etc. from the menu instead of typing them. Similar with the protection of password in last section, only placeholders of credit card information are stored on the mobile phone.

On the trusted node, we could make following access control policies. First, the target must be in a pre-defined domain whitelist. Second, the access must be done during a time window, e.g., 10:00 am to 10:00 pm. Third, the access frequency could not exceed a preset limitation, e.g., four times per day. Fourth, all of the access operation will be logged for future auditing.

5. Security Analysis and Discussion

Given physical access to a mobile device, an attacker has three ways of attacking to retrieve or access the *cor*: 1) scanning the entire memory and storage of the phone and searching residues of *cor*, 2) install malware/rootkit on the device to steal *cor* from memory, or 3) attacking the trusted node to retrieve *cor* directly. This section will discuss each in detail, as well as the limitations of *TinMan*.

5.1 Eliminating *cor* Exposure on Device

The first attack is eliminated in *TinMan*, since the system ensures that no plaintext of *cor* will ever exist on a mobile device. However, since we require that the *cor*-stub must have equal size with *cor*, the length of *cor* is not protected.

5.2 Defending Malware and Phishing Attack

TinMan can protect *cor* from phishing attacks using access control on the trusted node. For example, a user can bind the password of Citibank with only the official app and the domain "citibank.com". Suppose that the mobile phone has been compromised and shows a fake web site or malicious app to the user. The user selects the password instead of typing it, as mentioned in Section 4.1, and then clicks the "login" button. Then the application or browser is migrated to the trusted node to execute. The password will not be sent to the fake site or be used by the malicious app.

5.3 Security of the Trusted Node

TinMan relies on the security of the trusted node, in both the storage of *cor* and the execution of offloaded code. However, the trusted node faces threats from both outside hackers and inside malicious administrators (if the trusted node is deployed in a company or on a public cloud). An attack can also pretend to be a legal user and migrate some malware on the trusted server to execute, which may further get control of the server and steal other users' *cor*. In order to enforce the security of the trusted node, a lot of traditional methods could be leveraged, including cloud security, web security, intrusion detection, etc., which are beyond the scope of this paper.

Users may be unwilling to store passwords for personal accounts (e.g., Facebook) on a server owned by his employer for privacy reasons. In such case, a user can deploy different trusted nodes for different passwords to avoid putting all eggs in one basket. Further, deploying passwords on multiple sites can also tolerate various kinds of service failure.

By decoupling the *cor* storage and access from mobile device to a trusted node, it also brings several new benefits for security. For example, more powerful static and dynamic analysis can be deployed on the trusted node, as well as secure auditing, etc. The trusted node can check a migrated app before running it to filter any known malware, and can also monitor its behavior during runtime. Any abnormal activity will be reported to the user. All of the *cor* access activities on the trusted node could be logged for auditing.

5.4 Discussion

Non-*cor* private data. *TinMan* does not protect the data that must be visible to users. For example, although the password of an email account is protected, the data of email itself is in plaintext on the device for being displayed. The protection of general privacy data could be done through existing systems such as KeyPad and CleanOS, which is orthogonal to our work. *TinMan* focuses on protecting *cor* and could be seen as a complementary to previous work.

Protect *cor* from abusing. *TinMan* can protect the confidentiality of the *cor*, but cannot eliminate *cor* abuse. For example, if an attacker steals a mobile phone that is not locked, and uses it to login Facebook and check the phone owner's chatting history, then *TinMan* cannot defend against such attack, but can rely on users' explicit revocation of accesses of the lost device. Such an attack can be defeated by using more secure authentication technologies, such as biological authentication. Another way is to adopt more effective dynamic analysis on the trusted node, which can detect user's abnormal behavior and give some warnings. Even without these technologies, *TinMan* can still ensure that user's password of Facebook is not exposed to the attacker, and any login activity is logged and cannot be denied. Once the user notices that the device is lost, he/she can revoke the permission of *cor* access to prevent future attacks.

Attack time window. An app might not require password to login everytime it is activated. One scenario is that after the first time of login using password, the app saves a temporary token generated by the server for authentication in later communication. Since the token is not visible to the trusted node, it is not tainted or tracked. Although such token can reduce the overhead of performance and network traffic of *TinMan*, it also causes a time window for attacker to bypass *TinMan*. Shortening the time-to-live of the token can shrink the attack time window. Such attack has many in common with the *cor* abusing, but *TinMan* ensures that in either case the *cor* itself is still protected which can prevent password reuse attack. For some *cor*s like credit card number, each access requires a new authentication. Thus there is no such attack time window in these cases.

Connectivity requirement. *TinMan* requires that the mobile device be online while accessing *cor*. If the device is offline, e.g., during a flight, the *cor* is not available. We think this requirement will affect little users' experience. We argue that most scenarios of accessing *cor* would already require network connectivity, such as web login or online banking, so *TinMan* does not add more requirements. Another issue is that an attack may use DoS attack on the trusted node which leads to DoS on the user.

Network policy on the trusted node. We require to deploy the trusted node on a machine without IP egress filtering, or make the machine set policies to allow legal IP address switching. Otherwise, if the trusted node employs egress filtering, it might treat the TCP header switching as IP spoofing attack and refuse to send the reframed packet.

Usability implications. *TinMan* changes a user's behavior when using password or credit card number. First, the user needs to setup her/his own *cor* on the trusted node. This task is relatively trivial since the number of *cor* is usually small. For example, typically a user has less than five passwords [18] for all of her/his online accounts. This process can also be done in a batch if the user uses some password management tools such as LastPass. Second, when a user needs to use *cor*, e.g., during web login, she/he will select one from the list instead of typing them on the device. If a user reuses a password for a non-critical account, which is a common case, she/he can initialize a password with the

name "My General Password" and select it for multiple sites. If a user needs to create a new account with a new password, she/he can choose "Generate New Password" in the menu to let the trusted node create a password. By avoiding typing on the small screen, *TinMan* can actually improve the user experience.

Compatibility. *TinMan* modifies the Android framework as well as the JVM. It also requires apps to use system's SSL library, which is modified to enable session injection. If an app uses its own SSL library, then *TinMan* currently cannot be used to protect its *cor*. In our evaluation of top 100 apps on Google Play, all of them use system's SSL library.

6. Performance Evaluation

In this section, we evaluate the performance overhead of the *TinMan* system. The evaluation is conducted on a Samsung Galaxy Nexus smartphone, with Android 4.1 installed as the default system. The phone has a 1.2 GHz TI OMAP4460 CPU, a 1 GB memory, 16 GB internal storage, a 1750 mAh Battery and 1280x720 display. We deploy the trusted node in a PC with 2.8 GHz Intel i5-2300 quad-core CPU, 8 GB memory, 500 GB disk, and 100/1000 Mbps NIC. We conduct the end-to-end evaluation in both Wi-Fi and 3G network environment.

6.1 Micro-benchmark Performance

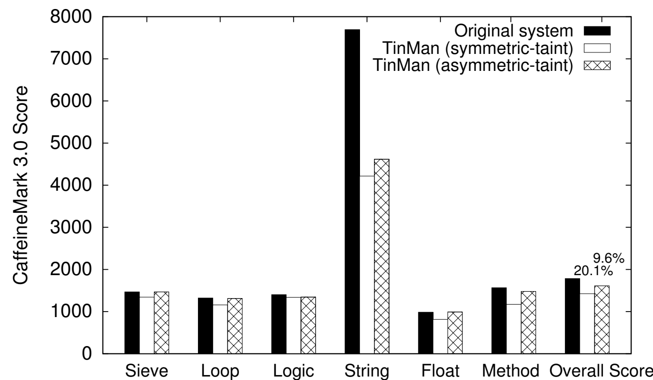


Figure 13: Caffeinemark results

We first use Caffeinemark, a computation-intensive workload, to evaluate the performance overhead of *TinMan*. We use three configurations: the original one, the one with full-fledged tainting, and the one with asymmetric tainting. The results show that the average performance overhead of *TinMan* using asymmetric tainting is only 9.6%. If the client uses a full-fledged tainting mechanism (similar with TaintDroid), then the average overhead is 20.1%. *TinMan* achieves better performance in asymmetric tainting since it only tracks heap-to-stack and heap-to-heap operations, but not stack-to-heap and stack-to-stack operations.

The original Android platform has significantly better performance than *TinMan* in the *String* benchmark. This is because some optimizations of string operations enabled in the original Android system are disabled in *TinMan* for tainting. Meanwhile, the ratio of heap-to-stack operation is very high in the *String* benchmark, which is also one of the reasons of the performance degradation. Since there is no *cor* data in these benchmark, the overhead is mainly caused by the tainting mechanism. For normal apps without accessing *cor*, the overhead should be lower since a typical app usually spends most of the time waiting for users' input and is not computation intensive.

6.2 End-to-end Performance

We measure the latency of the different applicaitions login process under both Wi-Fi and 3G network using the original Android and our *TinMan* system. The offloading engine requires to transfer the entire app binary (i.e., the dex file) from the mobile phone to the trusted node the first time, which is a one-time effort. The measurements in the *TinMan* system are done after warm-up (i.e., the trusted node already has the dex files of the apps). The results are shown in Figure 14 and Figure 15. In the Wi-Fi network environment, the average latency of application login using *TinMan* increases from 4.0s to 5.95s, where the DSM-based offloading takes 0.8s in average and SSL/TCP offloading related overhead is 1.2s in average. When using the 3G network, in average, the latency increases from 5.4s to 8.2s, where the offloading takes 1.2s and the other overhead is 1.6s. As login is a kind of low frequency operation, such overhead is acceptable.

The time for warm-up depends on the size of apps' dex files. For example, the dex file of paypal app is around 2MB and it takes about 8 seconds to transfer from the client to the trusted node. The warm-up is needed only once for each app, thus it does not hurt the performance of normal runs.

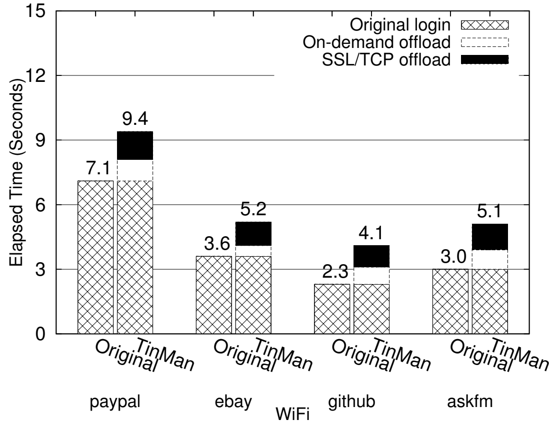
6.3 Offloading Overhead

We also count the code that will be offloaded to the trusted node and measure the network consumption when running login operations in different applications.

Apps	Off. Code	Sync. Times	Off. Init. (KB)	Off. Dirty (KB)
paypal	10274 (4.7%)	2	768.5	24.3
ebay	2835 (2.4%)	4	759.8	16.6
github	1672 (2.0%)	3	603.0	4.9
askfm	1791 (1.7%)	4	716.6	18.7

Table 3: Offload code, times of synchronization and the network consumption of login for different applications

To count the percentage of the code offloaded when a *cor* is accessed, we log every function invocation in the trusted node, and count the overall function invocations during the login phase. As shown in table 3, the number in the first column (e.g., 10274) represents the total number of method invocation in the trusted node, and the percentage (e.g., 4.7%) is the proportion of the offloaded code in the total



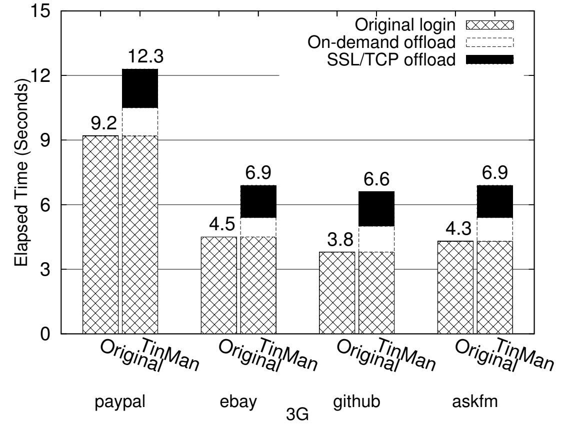
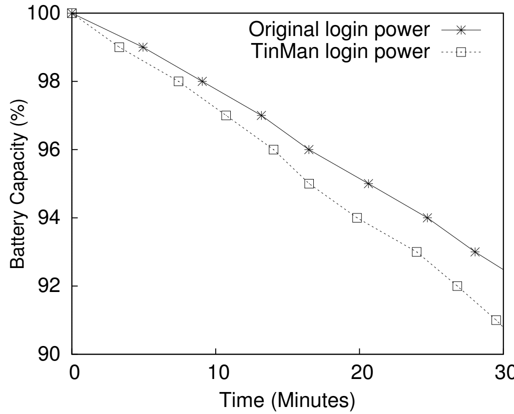


Figure 14: *Break down of applications login time in Wi-Fi network environment, after warming up*

Figure 15: *Break down of applications login time in 3G network environment, after warming up*



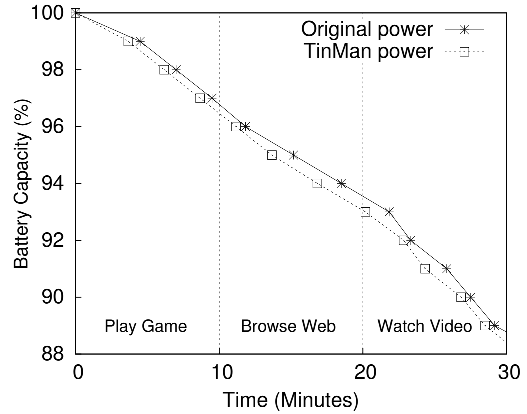


Figure 16: *Battery level changing, runs stress test to repeat login for 30 minutes*

Figure 17: *Battery level changing, each 10 minutes runs different workloads, without login operations*

number of method invocation. The data show that less than 5% of the code is offloaded, thus most of the code runs on the mobile device.

Besides, we count the number of times of DSM-based synchronization, either from the mobile phone to the trusted node, or vice versa. We find that in these test cases, the synchronization happens for three reasons: first, when the code in the mobile phone accesses a tainted object; second, when the code in the trusted node invokes a native method that cannot be offloaded; third, when a "happen-before" relationship needs to be established while the lock is held by the other end (as happens in the *github* test). In all the cases, the times of synchronization are less than 4, since the accesses to *cor* typically involve only a small portion of data.

Table 3 also shows the amount of state transferred during login operations. At the very beginning of the offloading, one initial heap synchronization will happen. This first sync phase will transfer a large number of states, most of which will never change later. Thus we show the amount of data transferred during the subsequent dirty field synchronization, which is a few to tens of KB in most cases.

6.4 Power Consumption

In order to illustrate the power consumption impact of *TinMan*, we conduct a stress test on login operations. We consecutively run PayPal login for 30 minutes in both Android and *TinMan*, and get the remaining battery every 10 seconds. As shown in Figure 16, after 30 minutes, Android system has 93% battery left, while *TinMan* has 91%. Since the offloading just happen in a very small period of time, it has little affect on the power consumption.

We also evaluate the power consumption caused by client side tainting. The tainting mechanism on the mobile device must be enabled at all times to monitor if any *cor* placeholder

is getting accessed, thus we focus on how tainting on the phone will affect the power consumption. Our evaluation has three phases to run three typical applications, each with 10 minutes. In the first phase, a user plays AngryBird for 10 minutes. In the second phase, the phone is used to skim various Wikipedia web pages, with figures and texts. In the third phase, the user plays a local 720p video for 10 minutes. The battery level is recorded every 10 seconds. We perform the test on Android as well as using *TinMan*, and the results are shown in Figure 17. The curves indicate that *TinMan* only occurs small amount of power overhead.

7. Related Work

TinMan retrofits some prior techniques such as taint tracking and computation offloading, but is the first to combine and extend them to protect (non-interactive) confidential data without trusting the entire mobile stack. This section briefly relates *TinMan* to state of the art.

Taint Tracking: Taint tracking, also known as information flow tracking, has been intensively used for defending attacks and privacy protection [13, 66]. For example, Haldar et al. [28] use taint tracking to defend against SQL injection attacks by instrumenting the Java String class. Chandra et al. [12] instrument Java byte-code to support fine-grained information flow tracking. Similarly, Nair et al. [45] instrument the Kaffe JVM. TaintDroid [20] implements system-wide information flow tracking on the Android system. Compared to prior work, *TinMan* leverages a new asymmetric tainting mechanism across the mobile device and trusted node, which is lightweight on the client to minimize overhead and is full-fledged on the trusted node to retain precision. Pebbles [57] uses tainting to infer the application level semantic from the OS level, and develops security tools based on the semantics. Pebbles can be complementary to *TinMan* to automatically identify more types of *cor*.

Computation Offloading: There are many systems designed for computing offloading, including Hera-JVM [39], MAUI [17], CloneCloud [15], Cuckoo [34], JESSICA2 [69], and COMET [25]. These systems are designed to improve performance and lower energy usage by offloading parts of a mobile application to a more powerful remote compute resource. None of these systems leverage offloading to address security issues like *TinMan*. ECOS [24] presents an enterprise-centric offloading system that addresses the security needs of mobile applications. It selectively encrypts offloading communication to protect the transmission of private data. *TinMan* has different goals and focuses on protecting *cor* from device theft.

Cloud-based Security Services for Mobile Devices: The wide adoption of mobile devices also stimulates lots of protection systems [14, 23, 33, 35, 38, 46, 51, 60, 61], among which the cloud is increasingly being used to offer security as a service. Keypad [23] and CleanOS [61] are two typical systems that leverage the cloud as a backend service for encryption and storage. Mackenzie et al. [38] present a cloud-based authentication system with capture-resilient cryptography. Some researchers try to offer antivirus service by cloud, such as SmartSiren [14], CloudAV [46], and ThinAV [33]. Similarly, Paranoid Android [50] uses record and replay to synchronize states between the phone and cloud and deployed security enforcement on the cloud. *TinMan* focuses on protecting *cor* and could be considered as complementary to these work. It can also leverage existing work like [64, 65, 67] to enhance the security of the trusted node.

Memory-less Computing: Memory-less computing [22, 26, 42, 43, 48, 49, 56, 58, 62], also known as CPU-bound computing, removes plaintext of critical data from RAM and puts it only in CPU (i.e., cache or registers) during computation to defend against physical attacks. These solutions are effective to protect keys for local RSA encryption/decryption, but it also relies on a huge TCB including the entire OS. Architectural support for private data protection [36, 37, 53–55, 59] is a strong way to protect both confidentiality and integrity of data even under physical attacks, by only placing plaintext data in on-chip CPU cache and encrypting the data when being stored to the main memory. However, there is no such commodity processor available so far.

Secure Execution: PrivExec [47] provides secure execution as an OS service and allow applications to execute in a private execution. *TinMan* could be considered as a similar method that provides secure access to *cor* as a service at the granularity of arbitrary code instead of an entire application. DARKLY [32] is another system that leverages a similar approach to operating secure critical data (e.g., camera raw data). An application can operate such data through some API defined by trusted libraries, which run in an environment isolated with the application. *TinMan* also separates the code of an application and runs all the code accessing *cor* on the trusted node. *TinMan* differs with those systems in both the threat model and the implementation.

8. Conclusion and Future Work

This paper presents *TinMan*, a system aiming at protecting Confidential Record (*cor*) on mobile devices. *TinMan* places all *cor*s only on a trusted node and puts corresponding placeholders on the mobile device. By combining taint tracking and on demand offloading using distributed shared memory, *TinMan* ensures no exposure of plaintext of *cor*s on mobile devices and thus can protect *cor*s from rootkit malware and even sophisticated physical attacks. Experimental evaluation confirms the security and efficiency of *TinMan*.

In the future, we'll further reduce the overhead caused by offloading. On the trusted node side, we plan to deploy more sophisticated static and dynamic analysis technologies to monitor *cor* access, and leverage massive knowledge and statistical analysis to detect anomaly behavior to further enhance the security of critical data access.

9. Acknowledgements

We thank our shepherd Rüdiger Kapitza and the anonymous reviewers for their insightful comments. This work is supported by a research grant from Huawei Technologies, Inc., China National Natural Science Foundation (No. 61303011), the Program for New Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), a research grant from Intel and the Singapore NRF (CREATE E2S2).

References

[1] Alipay. http://www.alipay.com.

[2] Beast attack on client-side ssl. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3389.

[3] Cve-2013-6271. http://www.cvedetails.com/cve/CVE-2013-6271/.

[4] Bitlocker drive encryption technical overview. http://technet.microsoft.com/en-us/library/cc766200%28WS.10%29.aspx, 2009.

[5] Lookout mobile security. lost and found: The challenges of finding your lost or stolen phone. blog.mylookout.com/blog/2011/07/12/lost-and-found-the-challenges-of-finding-your-lost-or-stolen-phone, 2011.

[6] Break out a hammer: You'll never believe the data 'wiped' smartphones store. http://www.wired.com/gadgetlab/2013/04/smartphone-data-trail/all/, 2013.

[7] Critical app flaw bypasses screen lock on up to 100 million android phones. http://arstechnica.com/security/2013/04/crital-app-flaw-bypasses-screen-lock-on-up-to-100-million-android-phones/, 2013.

[8] Samsung galaxy s iii has a lockscreen bug; security can be easily bypassed. http://www.brighthand.com/default.asp?newsID=19867&news=Samsung-Galaxy-S-III-Unlock-Screen-Bug-Lets-Security-Be-Bypassed, 2013.

[9] Skype for android lockscreen bypass. http://seclists.org/fulldisclosure/2013/Jul/6, 2013.

[10] Xperia z security flaw exposed as lock screen bypassed. http://www.xperiablog.net/2013/03/25/xperia-z-security-flaw-exposed-as-lock-screen-bypassed/, 2013.

[11] T. Cannon and S. Bradford. Into the droid: Gaining access to android user data, 2012.

[12] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *ACSAC*, pages 463–475. IEEE, 2007.

[13] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *35th International Symposium on Computer Architecture, 2008. ISCA'08.*, pages 401–412. IEEE, 2008.

[14] J. Cheng, S. H. Wong, H. Yang, and S. Lu. Smartsiren: virus detection and alert for smartphones. In *MobiSys*, pages 258–271, 2007.

[15] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *EuroSys*, pages 301–314, 2011.

[16] F. C. Commission. Announcement of new initiatives to combat smartphone and data theft. www.fcc.gov/document/announcement-new-initiatives-combat-smartphone-and-data-theft, 2012.

[17] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *MobiSys*, pages 49–62. ACM, 2010.

[18] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The Tangled Web of Password Reuse. In *NDSS*, pages 23–26, 2014.

[19] S. M. Diesburg and A.-I. A. Wang. A survey of confidential data storage and deletion methods. *ACM Computing Surveys (CSUR)*, 43(1):2, 2010.

[20] W. Enck, P. Gilbert, 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. In *OSDI*, pages 1–6, 2010.

[21] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication, 1999.

[22] B. Garmany and T. Müller. Prime: private rsa infrastructure for memory-less encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 149–158. ACM, 2013.

[23] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: an auditing file system for theft-prone devices. In *EuroSys*, pages 1–16, 2011.

[24] A. Gember, C. Dragga, and A. Akella. Ecos: practical mobile application offloading for enterprises. In *Hot-ICE*, 2012.

[25] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: code offload by migrating execution transparently. In *OSDI*, pages 93–106, 2012.

[26] L. Guan, J. Lin, B. Luo, and J. Jing. Copker: Computing with private keys without ram. 2014.

[27] P. Gutmann. Data remanence in semiconductor devices. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, pages 4–4. USENIX Association, 2001.

[28] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *ACSAC*, pages 9–pp. IEEE, 2005.

[29] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[30] G. Inc. Android honeycomb encryption. http://source.android.com/tech/encryption/android crypto implementation.html.

[31] P. Institute. The lost smartphone problem. http://www.mcafee.com/us/resources/reports/rp-ponemon-lost-smartphone-problem.pdf, 2011.

[32] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *IEEE Symposium on Security and Privacy*, 2007.

[33] C. Jarabek, D. Barrera, and J. Aycock. ThinAV: truly lightweight mobile cloud-based anti-malware. In *ACSAC*, pages 209–218, 2012.

[34] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *MobiCASE*, pages 59–79. Springer, 2012.

[35] 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 *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 8. ACM, 2014.

[36] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proc. SOSP*, 2003.

[37] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. ASPLOS*, pages 168–177, 2000.

[38] P. MacKenzie and M. K. Reiter. Networked cryptographic devices resilient to capture. In *IEEE Symposium on Security and Privacy*, pages 12–25. IEEE, 2001.

[39] R. McIlroy and J. Sventek. Hera-jvm: Abstracting processor heterogeneity behind a virtual machine. In *HotOS*, 2009.

[40] Microsoft. 2003. encrypting file system in windows xp and windows server 2003. http://technet.microsoft.com/en-us/library/bb457065.aspx.

[41] M. Milian. U.s. government, military to get secure android phones. http://www.cnn.com/2012/02/03/tech/mobile/government-android-phones/index.html, 2012.

[42] T. Müller, A. Dewald, and F. C. Freiling. Aesse: a cold-boot resistant implementation of aes. In *Proceedings of the Third European Workshop on System Security*, pages 42–47. ACM, 2010.

[43] T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, pages 17–17, 2011.

[44] T. Müller and M. Spreitzenbarth. Frost. In *Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[45] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

[46] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *USENIX Security*, pages 91–106, 2008.

[47] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. Privexec: Private execution as an operating system service. In *IEEE Symposium on Security and Privacy*, 2007.

[48] J. PABEL. Frozencache mitigating cold-boot attacks for full-disk-encryption software. In *27th Chaos Communication Congress (Berlin, Germany*, 2010.

[49] T. P. Parker and S. Xu. A method for safekeeping cryptographic keys from memory disclosure attacks. In *Trusted Systems*, pages 39–59. Springer, 2010.

[50] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *ACSAC*, pages 347–356, 2010.

[51] K. P. Puttaswamy, C. Kruegel, and B. Y. Zhao. Silverline: toward data confidentiality in storage-intensive cloud applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 10. ACM, 2011.

[52] J. Reardon, S. Capkun, and D. A. Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *USENIX Security Symposium*, pages 333–348, 2012.

[53] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proc. MICRO*, pages 183–196, 2007.

[54] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proc. PACT*, pages 123–134, 2004.

[55] W. Shi, H.-h. S. Lee, and C. Lu. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation College of Computing. In *Proc. ISCA*, 2005.

[56] P. Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 73–82. ACM, 2011.

[57] R. Spahn, J. Bell, M. Z. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. 2014.

[58] P. Stewin. A primitive for revealing stealthy peripheral-based attacks on the computing platforms main memory. In *Research in Attacks, Intrusions, and Defenses*, pages 1–20. Springer, 2013.

[59] G. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proc. Supercomputing*, 2003.

[60] C. Tan, H. Li, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Precrime to the rescue: defeating mobile malware one-step ahead. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 5. ACM, 2014.

[61] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *OSDI*, 2012.

[62] A. Vasudevan, J. McCune, N. Qu, L. Van Doorn, and A. Perrig. Requirements for an Integrity-Protected Hypervisor on the x86 Hardware Virtualized Architecture. In *Proc. Trust and Trustworthy Computing*, pages 141–165, 2010.

[63] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *FAST*, volume 11, pages 8–8, 2011.

[64] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and

physical attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture, HPCA'13.*, pages 246–257. IEEE, 2013.

[65] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proc. DSN*, 2012.

[66] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278. USENIX Association, 2006.

[67] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor : Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. SOSP*, pages 203–216, 2011.

[68] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[69] W. Zhu, C.-L. Wang, and F. C. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Cluster Computing*, pages 381–388. IEEE, 2002.