Shatter: Using Threshold Cryptography to Protect Single Users with Multiple Devices

Erinn Atwater and Urs Hengartner
Cheriton School of Computer Science
University of Waterloo
{erinn.atwater, urs.hengartner}@uwaterloo.ca

ABSTRACT

The average computer user is no longer restricted to one device. They may have several devices and expect their applications to work on all of them. A challenge arises when these applications need the cryptographic private key of the devices' owner. Here the device owner typically has to manage keys manually with a "keychain" app, which leads to private keys being transferred insecurely between devices – or even to other people. Even with intuitive synchronization mechanisms, theft and malware still pose a major risk to keys. Phones and watches are frequently removed or set down, and a single compromised device leads to the loss of the owner's private key, a catastrophic failure that can be quite difficult to recover from.

We introduce Shatter, an open-source framework that runs on desktops, Android, and Android Wear, and performs key distribution on a user's behalf. Shatter uses threshold cryptography to turn the security weakness of having multiple devices into a strength. Apps that delegate cryptographic operations to Shatter have their keys compromised only when a threshold number of devices are compromised by the same attacker. We demonstrate how our framework operates with two popular Android apps (protecting identity keys for a messaging app, and encryption keys for a note-taking app) in a backwards-compatible manner: only Shatter users need to move to a Shatter-aware version of the app. Shatter has minimal impact on app performance, with signatures and decryption being calculated in 0.5s and security proofs in 14s.

1. INTRODUCTION

With breaches of personal data becoming a daily occurrence, and in the wake of the Snowden revelations of mass government spying [18], consumer interest in strong cryptography to protect their data and transactions is at an all-time high. In particular, end-to-end cryptography allows users to obtain strong privacy and authentication properties for their communications without the need to trust any intermediate parties. However, there are numerous unsolved challenges standing in the way of incorporating such protections into traditional user-facing products [7, 16, 32]. Whitten and Tygar [40] showed in their seminal paper "Why Johnny Can't Encrypt" that major usability flaws prevent users from widely adopting PGP email encryption software. A significant aspect of this is requiring users to interact manually with a keychain; the concept of managing keys is not an intuitive one for most non-technical users.

Using end-to-end encryption necessitates the creation and handling of long-term identity keys for each user, with both a private component that must be handled with care, and a public component that must be distributed to and verified by others. Usability literature has widely revealed that the average computer user is not comfortable managing such keys manually, and necessitated placing trust in third parties instead (such as the public key infrastructure, or intermediate message-routing services). These key management problems include distributing one's public keys (and only the public keys) to communication peers, as well as protecting private keys from adversaries while also enabling access to them from any location the user wishes to use their software. While the former is being addressed by other researchers [25], we are interested in tackling the latter.

The most recognizable form of this problem is posed by users wishing to use the same software on multiple different devices. For example, it is common for users to own at least a smartphone and a personal computer; other common devices include tablets, work computers, smartwatches, a variety of Internet of Things devices, and access to various cloud computing services. In order for a user to authenticate themselves with a consistent identity across all devices, or read their encrypted messages from any location, they need to share access to secret keys that are intended to remain secure and moved around as little as possible. If any single device with access to these keys is stolen or compromised by malware, the data on every device becomes vulnerable and the key itself must be revoked (a costly process). The keys can be partially protected—for example, with a password entered by the user—but they are still vulnerable to attacks such as pulling them from memory while in use, phishing, or offline password cracking. Although practitioners have developed some partial solutions to this problem [1, 19, 35], we believe a more promising approach lies in using threshold cryptography to distribute cryptographic operations across the user's very devices that are creating the problem in the first place.

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 the author(s) 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.

WiSec'16, July 18–20, 2016, Darmstadt, Germany.

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4270-4/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2939918.2939932

In this scenario, the user is able to perform actions requiring private keys from any of their devices, but the theft of a single device (or more, up to some user-defined threshold number) does not result in the loss of the private key. For example, identity authentication mechanisms in communication platforms such as instant messaging and email typically rely on digital signatures generated from the user's private key. If a single copy of that private key is stolen, the user must generate a new key for themselves, inform everyone they communicate with that the old key has been compromised, and somehow securely distribute the new key to everyone. If the signature were generated using threshold cryptography, however, then no individual device contains the private key any longer – it contains a *share* of the key. Even if a device is compromised by sophisticated malware that initiates these threshold signature operations on behalf of the user, the user will notice the unsolicited requests on her uncompromised devices and simply reject them. Furthermore, even if the user's devices automatically acquiesced to such requests (for the sake of convenience), the attacker would not actually gain the private key *itself*, and the requests generated by the attacker must still show up in the logs of the other devices. We are not simply distributing cryptographic *keys* across devices – we are distributing the cryptographic operations themselves. In this case, the attacker would obtain digital signatures for some number of messages, but still not possess the private key to generate more after the user has noticed the compromise (which is easily visible in the logs of all other, uncompromised devices).

Our work makes the following contributions:

- We show that using threshold cryptography is a viable way of allowing users to distribute private keys and cryptographic operations amongst their many devices, while leveraging what was once a vulnerability into protection against device theft.
- We provide Shatter, the first (to our knowledge) free and open-source [1] cross-platform framework for easily managing private keys on multiple devices, and providing threshold cryptographic protections to applications using strong cryptography.
- We demonstrate how Shatter can be used to protect identity keys in the ChatSecure instant messaging app, and encryption keys in the OmniNotes note-taking app.
- We show that Shatter-compatible apps can be used with existing communication platforms without requiring non-Shatter users to adopt our new software wholesale (or even being made aware that it exists), facilitating easy adoption.

We define our version of the multi-device problem in Section 2 as well as how the use of threshold cryptography addresses it. Section 3 describes Shatter, our framework for providing protections of threshold cryptography to end-user applications. We have integrated Shatter with two example Android applications, and this integration is described in Section 4. A performance evaluation is provided in Section 5. Related work is discussed in Section 7, and Section 8 concludes.

[1] https://crysp.uwaterloo.ca/software/shatter/

2. PROBLEM SETTING

This section describes the multi-device problem as we seek to address it. Section 2.1 describes the problem itself, and Section 2.2 enumerates the properties our ideal solution would have. Section 2.3 describes how threshold cryptography can be employed as a solution, and Section 2.4 walks through a real-world example. Section 2.5 defines the threat model we use for this paper.

2.1 The multi-device problem

As discussed above, it has become common for users to own and even carry several different computing devices at a time. Cryptographic keys that are traditionally stored on a user's sole computer are now needed from each of these devices. In order for a user to authenticate themselves with a consistent identity across all devices, or read their encrypted messages from any location, they need to share access to secret keys that are intended to remain secure and moved around as little as possible. The keys can be partially protected with e.g. a password entered by the user, but they are still vulnerable to attacks such as pulling them from memory while in use, phishing, or offline password cracking.

The simplest approach to solving this problem is simply to sync private keys between devices, protected during transport with, for example, the user's password for logging into the software. However, this solution engenders a new set of problems. For example, if a user transfers their private key to their smartphone, and the phone is subsequently stolen or lost, the user's key has been compromised for all the devices they were using it on. They are required to revoke the lost key, generate a new one, update all their remaining devices with the new one, and communicate the revocation and new key to all parties they are in contact with. Protecting the keychain with a password, as mentioned above, does not change this scenario in the event that the keychain is somehow compromised.

Another approach is to use multiple keys, such as the case with "manual" threshold schemes[2]: a different identity key is generated for each of the user's devices, and remote parties must be made aware of all of them (or a hierarchical scheme could be used to issue keys from a central dealer). This approach also has its problems. For one, it exposes details of how a user uses their individual devices to the recipient (such as whether they are currently sending an email from their phone, instead of their workstation). For another, a stolen device still results in a stolen key that must be revoked. It also places a burden on the communication partner: they must now run software that is compatible with the multi-key scheme chosen by the sender, in order to encrypt each message to all the user's devices simultaneously (or to verify more complex conditions such as "require at least three signatures on all messages", as discussed below). This is a significant barrier to real-world deployment, as it necessitates all members of a communication system to upgrade and adopt the system. Group signature schemes, where one device generates a signature on behalf of all the devices in the group [3], can solve several of these issues, but not all of them; primarily, they are still vulnerable to the problem of single stolen devices.

[2] http://crypto.stackexchange.com/q/15520

2.2 Goals

To motivate our own work, we must first consider other various potential solutions to the multi-device problem. Below, we enumerate various approaches to performing digital signature operations for the same user on multiple devices. Encryption, depending on implementation, tends to have the same single/multiple device cooperation properties, and requires remote parties to encrypt for multiple keys in schemes where a single key is not somehow distributed amongst devices.

Per-device keys The user has an independently generated key on each of their devices.

Key sync The user generates a single key and copies it to their other devices.

Manual thresholding As per-device keys, but the user also embeds a "policy" in each signature, instructing verifiers to look for multiple signatures (from the user's other devices) on each message.

Personal PKI As 'key sync', but the user also has a single "master" key, stored on one device, which signs the keys on each other device. This can also be used in combination with 'manual thresholding'.

Group signatures Group signature schemes allow any one member of a group to sign messages on behalf of the entire group. These schemes present a single public key to the world, but each member has a unique "share" of the private key.

Secret-shared keys As 'key sync', but a threshold secret sharing scheme is used to protect the private key. Whenever it is required for a cryptographic operation, a user-defined threshold number t of their n devices ($t \leq n$) work together to recover the original private key.

Threshold cryptography As 'secret-shared keys', but the private key is not regenerated for normal operations (signature generation). Instead, the cryptographic operation itself is distributed amongst the t user devices. This is our approach, as described above.

Figure 1 summarizes the properties provided by each of these schemes, which are defined as follows:

Backwards compatibility The scheme can be used to communicate with people who are using unmodified software. Schemes are listed as "potentially" having this property if a modified version of the original cryptography algorithm is required.

Weak theft resistance If only $0 < x < t$ devices are compromised, the long-lived private keys remain uncompromised and do not need to be revoked (as long as devices do not automatically participate in requests from other devices).

Strong theft resistance If only $0 < x < t$ devices are compromised, the long-lived private keys remain uncompromised and do not need to be revoked (even if devices do automatically participate in requests from other devices). For example, in the secret-shared keys scheme, a single request is all the attacker needs to perform in order to recover the private key.

Only one active device The scheme does not require multiple devices to be powered on and in communication with each other in order to perform a necessary operation (initial enrolment and revocation do not count for this purpose).

Device anonymity The remote party cannot distinguish which of the user's devices were used to perform an operation.

Single public key Remote parties only see a single public key representing the user, and do not need to consider others when performing signature verification or encryption.

No master device / CA All of the devices are treated equally; there is no device that acts as a single point of failure.

2.3 Proposed solution

This work proposes using threshold cryptography to aid in solving the problems with sharing a single cryptographic identity across multiple devices. Threshold cryptosystems are usually presented as $(t, n)-$threshold algorithms, where n parties are initially enrolled in a system, but any subset of size $t < n$ can subsequently work together to perform the corresponding operation (typically decryption, or creation of a digital signature).

Traditionally, threshold cryptography schemes have been designed with organizations in mind: the authority to perform some action (such as "launch missiles") is split amongst high-ranking officials, and several of them must act in unison in order to carry out said action. The rising prevalence of individual users possessing multiple computing devices for their own personal use, however, leads to an opportunity to adapt such schemes for the single-user setting. Using threshold schemes to distribute secrets, or the ability to perform signing/decryption operations with a combination of devices working in unison, solves many of the problems with the schemes enumerated above. Theft of a (single) device no longer necessitates revoking the key; the user simply has to use a coalition of the remaining devices in their possession to recover the original secret key, and they can generate new shares without distributing one to the affected device. It can also be implemented in a manner entirely application-agnostic and invisible to the communication partner (as long as threshold versions of the cryptographic algorithms used in the original application exist). Corresponding parties do not need to upgrade their software, and indeed do not even need to be aware that the user is generating signatures or performing decryption in a distributed manner. The drawback to this approach is that it now requires multiple devices (a user-configurable threshold) to be powered on and accessible to each other, be it by proximity or over the Internet. This inconvenience can be mitigated by providing many forms of interconnection between devices, and creating software that runs on a wide variety of platforms. In this way, the user simply has to set the threshold number t to something that is convenient for them and their particular set of devices (which can be as low as only two devices), many of which are left powered on at all times anyway.

Going forward, we are interested in the applications of both threshold signature and threshold encryption algorithms. Signatures are frequently used as components to provide

	Backwards compatibility	Weak theft resistance	Strong theft resistance	Only one active device	Device anonymity	Single public key	No master device / CA
Per-device keys	●	◐	-	●	-	-	●
Key sync	●	-	-	●	●	●	●
Manual thresholding	-	●	●	-	-	-	●
Personal PKI, thresholding	-	◐	-	-	◐	●	-
Personal PKI, no thresholding	-	◐	-	●	◐	●	-
Group signatures	◐	-	-	●	●	●	●
Secret-shared keys	●	●	-	-	●	●	●
Threshold cryptography	◐	●	●	-	●	●	●

● = provides property; ◐ = could provide property, depending on implementation; - = does not provide property

Figure 1: Attributes provided by various solutions or proposed solutions to the multi-device problem

authentication to security protocols, including email (e.g., PGP and S/MIME), instant messaging (e.g., OTR and TextSecure), and authentication itself (e.g., FIDO and SQRL). Encryption is used to provide confidentiality to user communications (e.g., email), backend communications (e.g., software updates), and user data-at-rest (e.g., password managers). Each of these applications would interact with threshold versions of their cryptography in (sometimes subtly) different ways, posing their own questions, all of which must be answered by the framework we define.

2.4 Walked-through example

To illustrate how this solution would work in practice, let us consider the example of a user Alice using an instant messaging app to talk to her friend, Bob. Alice has three devices: a smartphone, a laptop she uses for school, and a desktop at her home. Alice's first step is to install Shatter on each of her devices. After installation is complete, the three devices discover each other through her home network and ask if she would like to add them all to her personal set of devices. She agrees, and decides to set the threshold t to two. Upon doing so, one of the devices generates a new public/private keypair and splits the private key into three shares. It distributes a share to each of Alice's devices, keeping one for itself, and then erases the original private key from its memory. Finally, Alice installs a Shatter-compatible version of the IM app onto her smartphone.

Later, Alice is traveling and has only her phone with her. Bob, who is using a normal version of the IM app and has never heard of Shatter, tries to initiate a secure conversation with her. The IM app uses digital signatures for authentication, so it sends a request for a signature to the Shatter process running on the phone. Shatter prompts Alice to accept the request via a notification on her phone, and also connects to her desktop and laptop over the Internet to request participation. Although Alice has left her laptop running at home, she has it configured to show a prompt requiring her to approve incoming requests. Her desktop, however, is configured to automatically accept all requests, and so the Shatter process on the phone communicates with the desktop to generate a threshold signature and returns it to the IM app. On Bob's end, he sees that a secure connection has been established without knowing anything unusual happened, and proceeds to chat with Alice.

After their conversation, Alice sets her phone down on a table in a coffee shop and accidentally leaves it behind. When she gets home, she realizes it is gone. If someone at the coffee shop were to pick up her phone and try to initiate a new conversation with Bob, it would only be possible because Alice has configured her desktop to automatically accept requests, and Alice would be able to look at the request log on her other two devices to see that it had happened. Even if the thief were aware that Shatter was installed on the phone and accessed its control panel, they would be unable to obtain Alice's private key – it is not stored on the phone, it is not regenerated in the process of creating a threshold signature, and neither the desktop nor the laptop will agree to regenerate the key without explicit user authorization.

At home, Alice is able to see whether or not her phone was used to perform any privileged operations. She then accesses the Shatter control panel on her desktop, and initiates a revocation of the smartphone. The revocation request appears on her laptop, which she accepts, and the laptop sends its secret share to the desktop, where they are recombined to recover the original private key. The desktop then creates a new set of secret shares for only the desktop and laptop. It sends a courtesy message to Shatter running on the smartphone (if it is able to reach it over the Internet) alerting it to the revocation. It is not imperative that this notification reaches the smartphone, however – the desktop and laptop will no longer accept signature requests from the smartphone. Indeed, even if the same thief were to then steal Alice's laptop, now possessing two of her devices, he would still be unable to recover her private key. Despite having two of Alice's three devices in such a situation, Alice was able to revoke the phone's keys first, essentially starting a new epoch the moment she did so.

Alice then purchases a new phone, installs Shatter and the IM app on it, and enrols it with her desktop and laptop using a similar recover-and-reshare process as used for revocation. This process does not change her public key, and she is able

to initiate new secure conversations with Bob without him having any way of knowing she had lost a key share.

2.5 Threat model

Our threat model is primarily restricted to less than some user-specified threshold t of that user's n devices being compromised by colluding adversaries. "Compromise" for our purposes is defined to include physical theft, privileged malware running on the device, and remote code execution. In the event that $x \geq t$ devices are compromised, private keys are able to be regenerated by the adversary and should be considered lost by the user.

In this threat model, two considerations must be made for damage mitigation and recovery in the event of device compromise. The first is in the case of data recovery by the user: in the event that $t \leq x \leq n - t$, any data protected by the private key should be considered compromised, but it is possible for the user to recover that data as well. However, for $x > n - t$, the user will no longer be able to recover the private data. For example, if an application is designed to protect a user's personal notes with threshold encryption, both the user and the adversary would have access to the notes with $t \leq x \leq n-t$; with $x > n-t$ only the adversary has access to the notes. In our threat model, another consideration is for the harm possible by $x < t$ devices being compromised when some of the user's other devices automatically participate in threshold operations. In this instance, the adversary is limited to performing the pre-designed cryptographic operations but cannot obtain the long-lived private keys of the user. This creates a window of opportunity wherein the adversary can cause harm by imitating the user (such as accessing plaintext or impersonating the user), but results in a user-auditable log of operations performed. It also does not preclude the user from subsequently revoking the lost device and continuing to use the same private keys.

We also consider the case of using untrusted cloud providers as participants in the user's threshold scheme. Such a cloud provider would accept threshold participation requests from the user's devices prerequisite on the user authentication with the service through some separate mechanism (which may be persistent, e.g. "stay logged in" cookies). Third-party cloud services should be barred from initiating requests (by having the user's devices ignore requests generated by cloud services), and should require re-authentication from the user before participating in key recovery operations. The user should not add more than one third-party service to their group of threshold participants, as collusion between two or more such services cannot be prevented and would compromise the scheme as described above. However, cloud services acting as part of the user's device group provide an always-on party that can participate in threshold cryptographic operations, and can even allow users with only a single internet-connected device (e.g., a smartphone) to gain the protections afforded by our solution by setting $t = n = 2$.

3. SHATTER ARCHITECTURE

Shatter's primary component is a library that implements various threshold cryptography protocols, as well as providing convenience functions and platform-specific operations for easily incorporating the library into client applications. It contains a daemon that can be run by client applications, although this daemon is intended to be run by a dedicated Shatter app, with client apps in turn communicating with the dedicated app to have it perform operations on their behalf. This software pattern allows users to use multiple client applications while only having to maintain their device configuration in a single location. Figure 2 illustrates this organization and how the components interact with each other. The following sections give an overview of the architecture and role of the individual components themselves.

3.1 Crytographic algorithms

The first threshold algorithm we have implemented is a (t, n)-Threshold-DSA signature scheme proposed by Gennaro et al. [14]. DSA was chosen as it provides the authentication in most OTR (Off-The-Record messaging) protocol implementations, which we later aim to provide threshold cryptography support to. OTR uses end-to-end cryptography to give instant messaging protocols authentication, encryption, and deniability [6]. The Gennaro scheme consists of six rounds that can be performed in parallel by all of the participating devices, each either broadcasting their result for the round when complete or, in the case of the last two rounds, sending it to the initiating device. This means that the algorithm runs in time independent of the values of t or n. Some of the interim calculations in the protocol are performed under additively homomorphic encryption, in order to hide secret information from other participants. There is some choice available as to precisely which encryption scheme is used.

The second threshold algorithm implemented is the Damgaard-Jurik (t, n)-threshold version of Paillier's additively homomorphic encryption algorithm [10]. This algorithm provides the properties required of the encryption algorithm used as a component of the Gennaro signature algorithm. For convenience, we also make the algorithm available for general-purpose use to other Shatter applications. For example, one of our example applications in Section 4 uses it to encrypt the contents of a user's personal notes.

3.2 libShatter

The library provides the actual implementation of the threshold cryptography protocols, as well as classes for facilitating network communication between clients, loading and storing configurations, and a variety of other functions useful to client applications. It is implemented in Java (targetting versions 1.7+), and is thus easily portable to all major desktop platforms. It also compiles to an Android app library, allowing it to be easily imported and used in any Android app.

libShatter consists of the following major components:

ThresholdAgent: Daemon that performs the core operations of a threshold cryptography operation, as well as device management. The ThresholdAgent daemon is responsible for managing network connections (via delegation to the NetworkAgent, below), incoming packets, and any received or pending requests. Client applications are intended to instantiate a ThresholdAgent thread and register a callback handler with it in order to implement any operations they require. For example, a client application would register a ConfigEventListener to take over loading/storing any configuration changes required by the daemon (such as changing keys or enrolled devices). It would register a SignatureEventHandler in order to be notified of new requests for distributed operations (to which it should respond by ei-

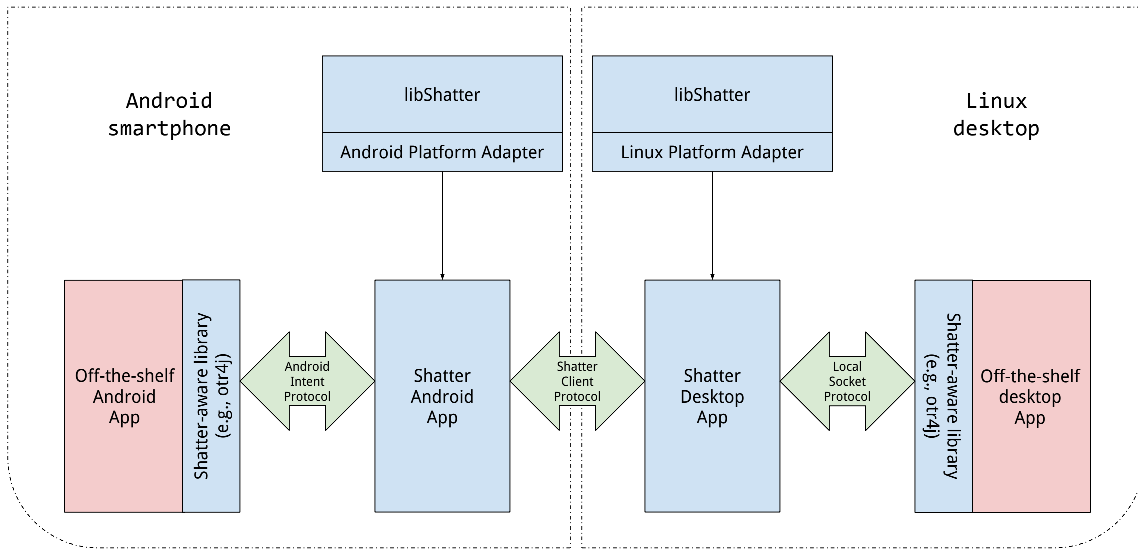

Figure 2: High-level architecture of the Shatter library. Android Wear devices run a Shatter client similar to the Android implementation (left), while Windows and OSX machines use an analogous setup to the linux example (right).

ther agreeing or declining to participate), progress updates, and the result of completed threshold operations.

NetworkAgent: Daemon that manages connections to the user's other devices, and performs all network communication between them. Currently, the NetworkAgent finds other devices by periodically broadcasting a multicast message advertising its presence to other devices on the local network, and keeping track of recent advertisements from other like devices. These advertisements contain the device identity (for determining if it is part of the user's personal set of devices), as well as instructions for which address and port it listens for messages on. It is also responsible for providing transport layer encryption, and authenticating messages received from other devices. Currently, this is done using client-authenticating TLS.

The NetworkAgent listens on this port for incoming connections, reads a single JSON-formatted message from the initiating party, and closes the connection. The JSON message is then passed back to the ThresholdAgent for processing. Although incurring some additonal overhead by not using long-lasting connections, this pattern means that the application is not weak in the presence of poor-quality network conditions or roaming users, which is frequently the case in mobile environments. Finally, while the NetworkAgent currently operates only via local network connections, it is architected in such a way that new network modalities can easily be dropped-in. For example, future network adapters might allow connecting devices over Bluetooth, or via a third-party or user-hosted cloud server as a proxy to allow easy firewall piercing and long-distance roaming (we are currently developing both of these options). The Multicast classes can be replaced with any program that advertises and finds other active devices; the LocalServer class with anything that receives JSON-formatted messages; and finally a PacketSendHelper with anything that allows sending JSON-formatted messages to an address compatible with the two Server classes.

Platform Adapters: Package containing adapters for platform-specific operations. This includes a set of convenience classes for manipulating and storing persistent configuration information with a uniform interface on various platforms. There is also an implementation of several BroadcastReceivers to be used by Android apps. Android apps simply need to add this receiver to their manifest files, and it will listen for Shatter-specific broadcast intents from other apps. It also registers event handlers with the ThresholdAgent when instantiated, allowing it to communicate the result of threshold operations back to the app that made the original request.

Crypto: Package containing the actual implementations of cryptographic protocols (although not the network protocols, which are done by ThresholdAgent). These operations are called by ThresholdAgent in order to perform threshold DSA operations, as well as (currently) distributed and non-distributed Paillier operations. It also contains miscellaneous convenience utilities, such as a central class for generating secure random numbers. For non-threshold operations (such as DSA verification and storing DSA keys/parameters), it imports SpongyCastle as a dependency (SpongyCastle is a full version of the BouncyCastle cryptography library, with its namespaces changed in order not to conflict with the pared-down version of BouncyCastle included with Android).

3.3 Shatter desktop app

This is the reference implementation for a Shatter client application. It is implemented in Java as an interactive command-line application, and allows making all possible calls to ThresholdAgent, displaying and responding to requests, and maintaining configurations using flat files. The source code for the application shows how a developer need only implement a thin UI that binds to the various exposed API functions: it primarily consists of a switch statement driving a menu displaying the available API calls, and several methods designed to handle the callbacks required

for it to act as both a ThresholdEventListener and ConfigEventListener.

3.4 Shatter Android app

In addition to replicating the basic driver functionality of the desktop application, the Android app acts as the host service for the ThresholdAgent for all other apps on the device. This provides a central device management interface for the user, who may have multiple distributed apps installed on their phone. Apps that want to perform distributed operations communicate with the central app via intents. The app itself communicates with any other Shatter clients (desktop or Android) to perform requested operations, and communicates the result back to the requestor via intents.

3.5 Device group management

Any number of Shatter-compatible apps can connect to a single instance of a Shatter client application, giving the user the convenience of only have to manage their devices from one place. The Shatter client takes care of group formation, new device enrolment, and old/lost device revocation. Initial formation of a group involves showing the user all discovered devices, asking which ones should be included in the group, generating a set of key shares for the group, and asking the user to securely pair each of the devices in order to receive those shares. Currently, the user has two choices for secure pairing: scanning a QR code, or verifying a displayed code on both screens (both of which contain a cryptographic hash of the public key for the central device, allowing the devices to authenticate each other from then on). For both revocation and enrolment, the user must gather at least the threshold number of the devices in their group together. The process is initiated on one device, which then sends the request to the other devices in the group. The user must accept this request (Shatter clients should never accept these requests automatically) on each other device, which causes them to send their shares directly to the initiating device. The shares are then combined to recover the original private key, which can then be used to generate a new set of shares (plus or minus the targeted device).

4. MOBILE APPLICATIONS

To demonstrate the feasibility of adding support for our library to end-user applications, we added Shatter support to two open-source Android apps: one which provides off-the-record messaging with strong participant identity verification, and one which allows a user to write and store arbitrary plaintext notes on their device. Screenshots of the Shatter client and Shatter-compatible apps are presented in Figure 3.

4.1 ChatSecure

We added Shatter support to ChatSecure[3], an app by The Guardian Project that allows the user to connect to arbitrary Jabber servers, and optionally to use Off-the-Record (OTR) messaging over them. In the OTR protocol, parties in a conversation authenticate each other once (at the beginning of the conversation) by way of DSA signatures. The implementation of OTR used by ChatSecure is called **otr4j**, which is originally developed and maintained by Jitsi, an open-source videochat and IM application.

We replaced calls to sign() in otr4j with a request for a distributed DSA signature from the Shatter app. This required a slight refactoring of the library – it originally expected signature operations to complete near-instantaneously, and thus simply blocked on calls to the method. This was replaced with a request+callback pattern, so the encryption resumes once the user has assembled the requisite number of other devices and permitted them to generate a signature. The refactoring was accomplished by adding new message types to the otr4j incoming message handler, and putting the OTR engine into a "waiting for callback" state in the meantime. Thus even though calls to the sign() method may be made on the main thread, they will return immediately without causing the UI to hang.

4.2 OmniNotes

We also added Shatter support to OmniNotes v5.1.3[4], the most popular open-source Android note-taking application at the time of this writing. OmniNotes supports very basic (password-keyed) encryption on a per-note basis, which we improve by generating strong random keys and protecting them with a threshold encryption scheme.

OmniNotes' default encryption behaviour is to display a prompt for the user to enter their password whenever opening a "locked" note. We removed the text field from this prompt and replaced it with a call to Shatter asking to decrypt the symmetric encryption key for the current note (encrypted keys are stored prepended to encrypted notes, as is typical with hybrid encryption). This effectively turns the password prompt into a loading screen for the note, while the Shatter client sends requests to the user's other devices requesting their participation. When the decryption is completed, the callback closes the loading screen and supplies the decrypted text of the note to be displayed.

We note that, if the user's other devices are accessible and configured to automatically participate in decryption requests, the final experience is actually *easier* with Shatter. Whereas previously the user had to stop, remember, and type their password, they now have to wait only a second or two for a loading screen (see Section 5). It also provides significantly stronger encryption for the user's notes. Previously, all notes were protected with a single key derived from a password that generally needs to be simple enough for users to remember. With Shatter support, each note has a different, randomly-generated 128-bit key.

5. EVALUATION

This section provides an evaluation of the algorithms currently implemented by Shatter. It should be noted that performance is generally particular to the cryptographic algorithms being used, and not the implementation itself.

5.1 Procedure

We ran our threshold signature and decryption algorithms on a desktop running Ubuntu 14.04 with an AMD FX-6100 3.3GHz 6-core processor, Nexus 5 smartphones running Android 5.1.1, and a Moto 360 smartwatch running Android Wear v5.0.1. These devices can be combined in any permu-

[3]https://chatsecure.org/

[4]https://play.google.com/store/apps/details?id=it.feio.android.omninotes

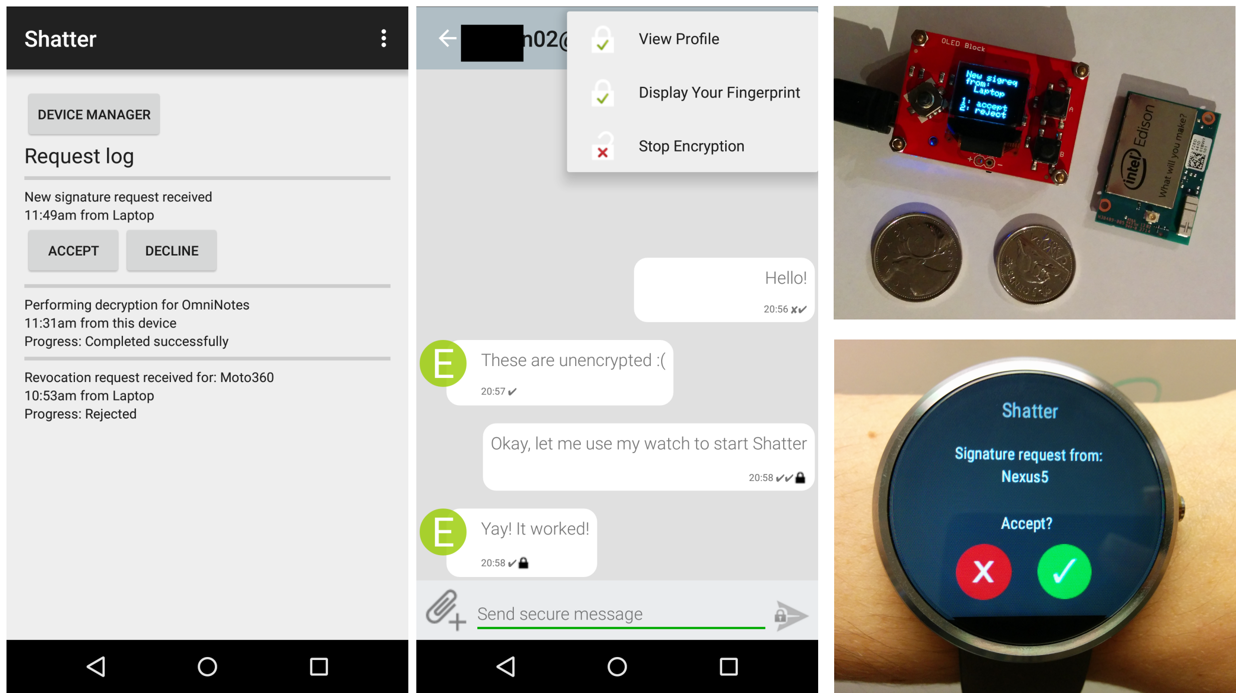

Figure 3: The Shatter client app for Android (left); Shatter-aware ChatSecure conversation with a non-Shatter aware remote participant (middle); Shatter client running on an Intel Edison with OLED display (top-right); Shatter client running on a Moto 360 (bottom-right).

tation; we restrict our evaluation to all-desktop, all-phone, and all-unique (that is, phone-watch-desktop) configurations with $n = t = 3$. As mentioned above, the performance of both algorithms is independent of the number of devices participating. Although the desktop clients are run on the same machine, in practice each instance runs on a single core and so the performance slowdown on our 6-core machine is negligible. We provide numbers both with and without checking of the zero-knowledge proofs that the algorithms require in order to detect malicious parties, but not in order to proceed with functionality. If an app is able and willing to provide latent detection of adversaries, it can get away with deferring the checking of proofs until the actual result has been computed, and the user will experience a shorter interruption.

Computation times are averaged across 20 runs of the algorithm. These times include some periods of network transmission, but this is performed over a local network and is negligible. The timer is started once a threshold number of devices have agreed to participate in a threshold operation, and stopped once the result has been calculated and delivered to the application requesting it. We use DSA parameters of $L = 1024, N = 160$, chosen to correspond to ChatSecure's DSA parameters, with the Paillier parameters being derived from the DSA parameters to preclude needing to use a block cipher mode of encryption.

5.2 Results & observations

Table 1 shows the computation time needed to perform threshold signing and decryption operations. Both signatures and decryption can be calculated in a quarter of a second on all devices. If proofs cannot be deferred, decryption needs several seconds extra while signatures need up to fourteen seconds extra.

We believe the delays are acceptable if proofs do not need to be verified immediately. In the event that the app does require up-front verification, the longer delay for signatures is mitigated by the fact that signatures tend to be performed only for infrequent operations. They only need to be calculated once at the beginning of a new OTR session with a friend, and this is done immediately before the user typically sends a "hello" message and waits for the recipient to notice it. Another common use of signatures is on outgoing emails; in this scenario, users are already accustomed to waiting a short period of time for an email to be sent (and indeed, Gmail currently suggests adding an artificial delay of ten seconds or more in order to provide "undo" functionality to sending mail [8]). Decryption only needs to be performed in OmniNotes when a note is first opened and, as we note in Section 4.2, this can actually result in a *shorter* delay for the user. Encryption is performed when closing/saving notes, and does not need to be distributed since Damgaard-Jurik is a public-key encryption algorithm.

As discussed in Section 3.1, we implemented a (t, n)-Threshold-DSA signature scheme designed by Gennaro et al. because it was necessary to find a threshold equivalent of the DSA algorithm in order to be backwards compatible with existing OTR apps. In analyzing the performance of our implementation, we found the homomorphic encryption and decryption operations dominate the computation time. As we outline in Section 3, the major calculations involved in this particular threshold signature scheme are performed under homomorphic encryption. This dominating factor, as well as the ratio of encryption to decryption time, is ap-

Device, role	No proofs	Proofs
Signature, 3 desktops	0.243 (0.01)	10.8 (1.09)
Signature, 3 phones	0.509 (0.01)	12.0 (1.09)
Signature, all unique	0.543 (0.02)	13.2 (1.11)
Decryption, 3 desktops	0.118 (0.01)	0.822 (0.10)
Decryption, 3 phones	0.249 (0.01)	1.941 (0.11)
Decryption, all unique	0.261 (0.01)	2.479 (0.11)

Table 1: Performance numbers, broken down by device combination ("all unique" means phone-watch-desktop). Average times are given in seconds, with standard deviations in parentheses. "No proofs" refers to the time to complete *before checking* the deferrable zero-knowledge proofs (see text for discussion).

proximately consistent with the analysis in Paillier's original paper [27].

Gennaro et al.'s scheme also requires several rounds wherein a partial computation by one device is broadcast to the other participants, and the broadcast from all other participants must be received before continuing with the computation. This results in t messages being sent in parallel a total of six times in sequence, with most messages being 4kB and the largest being 16kB (including transport encryption). The scheme also *uses* the Damgaard-Jurik threshold encryption scheme twice as a sub-operation, and therefore its computation time can never be lower than that of the encryption scheme by itself. For our threshold decryption scheme, however, backwards-compatibility with other communication partners is not a concern and so we can make use of any algorithm we wish. We chose to re-use our implementation for convenience and to demonstrate the utility of our crypto library; in the future, we will investigate using more appropriate algorithms for apps like OmniNotes.

6. LIMITATIONS

Shatter's protections only apply within the limitations of the threat model defined in Section 2.5. If an adversary compromises more than the user-defined threshold t of that user's devices, they are able to regenerate any private keys protected with threshold cryptography. Shatter does not allow the user to set $t = 1$, which would be essentially the same as not using Shatter at all. We do not consider application security vulnerabilities in our implementation to be in scope; however, we note that even an exploit allowing device compromise still only results in a *share* of a private key being stolen, and the attacker will still need to compromise t or more devices in order to defeat Shatter.

Our ability to provide backwards compatibility with remote communication partners is limited to apps using protocols for which there is a threshold version of any cryptographic algorithms relying on private keys available. In our current implementation, these are the DSA/ECDSA signature scheme and the Paillier homomorphic encryption scheme. Threshold cryptography is an active area of research, however, and threshold versions of many popular algorithms have already been published.

Shatter also introduces a requirement for t of the user's devices to be online and accessible to each other (e.g. over the internet) whenever the user wishes to perform an operation relying on their private keys. The user is expected to pick a value of t that is realistic for their normal usage (e.g., it would be annoying to pick $t = 3$ when one of the user's devices is a laptop that they typically leave off). We help mitigate this inconvenience by supporting as many communications protocols as possible, making it easy to add new ones, and implementing new ones ourselves as we are able to. We also aim to support as many platforms as possible, with our current Java implementation easily running on Windows, Linux, Mac OSX, Android, and Android Wear (with an iOS implementation currently under development).

7. RELATED WORK

The idea of using threshold cryptography between a single user's cooperating devices has been proposed before. In 1979, Blakley et al. systematized the set of threats posed against storage of sensitive private keys [5]. While one of the authors later worked on using threshold cryptography to help defend against some of these threats [4], it was two decades later before Desmedt et al. proposed using threshold cryptography to protect keys stored on "things that think" (contemporarily, Internet of Things devices) [11]. Desmedt et al. did not go so far as to offer comprehensive guidelines on how this should be implemented, however; they merely suggested that a set of standards should be developed to allow a wide variety of different devices and different applications to interoperate. These recommendations built on earlier work outlining the threat model addressed by threshold cryptography schemes in general [20], discussing how it might be adapted to the personal device scenario. The earlier work remains relevant, however, in that it contains suggestions and schemes that are applicable to our work (such as methods of identifying misbehaving devices, and key rotation schemes that reduce the damage done in the event of a compromised share). Papers building on Desmedt et al.'s work have investigated protocols for securing the underlying communication between devices, focusing on such goals as adding and removing devices from the threshold group securely [29] or preventing eavesdroppers from tracking the user via their device signatures [38]. More recent work has focused on applications (as do we), using threshold schemes to protect keys used for unlocking a device [39] or evaluating the usability of various pairing schemes [22].

The key difference between the work in this section and our proposed work is a lack of focus on real-world deployment issues. All of the aforementioned papers focus on using ordinary secret sharing to protect a secret key, and thus lack desirable properties as discussed in Section 2.1. They have no thought given to backwards compatibility or theft protection against malware, which requires modifying applications to perform distributed cryptographic operations and, in some cases, creating new cryptographic operations themselves. With one exception [22], they also tend to ignore the usability outcomes of their proposals, whereas we are interested in facilitating a smooth experience for both users of our software and the developers themselves.

The thread of work by Peeters et al. resembles our work most closely, in that they are also interested in applications of threshold things that think. In 2008, they presented an adaptation of Shoup's threshold RSA encryption algorithm [33] intended to be run on things that think [30] (including a proof-of-concept implementation). In a 2012 PhD thesis, Peeters discussed some of the ideas outlined herein, and presented proposals for storing secret shares on

devices without secure storage [28, 34]. The thesis also discussed methods of securely pairing devices [29], and presented a proposal using the location information available on devices with a GPS [31, 36]. Stajano presented the Pico/Picosiblings system in 2011 [37]. Picosiblings are IoT devices intended to be worn by a user, which use threshold cryptography to authenticate the user to remote services. This application is similar to proposals like FIDO / SQRL, but using threshold cryptography to offer theft-resistance. Although Stajano does not explicitly consider Picosiblings as a solution to the multi-device problem (only one device, the Pico, is intended to permit user interaction), it is trivial to envision how such a solution might be incorporated into the existing work. Finally, Gennaro et al. present an algorithm and software for distributing Bitcoin wallets across multiple devices with threshold cryptography [14]. Their work is discussed extensively in Section 3, as our current work implements their proposed threshold-DSA algorithm.

Other researchers have proposed solutions to the multi-device problem with alternative methods to threshold cryptography. For example, Sinclair and Smith described a method of sharing PKI credentials (signing keys) to mobile devices, reducing the impact of stolen / compromised devices [35]. Their work essentially uses personal PKI to issue "proxy" (short-lived) certificates from a master certificate on a central workstation. As we discussed in Section 2, this approach does not solve problems such as having to revoke stolen signing keys, and reveals details of the user's device usage to correspondents. Some researchers propose using multiple devices for novel two-factor authentication mechanisms, such as proximity detection [9, 21] or using a mobile phone to authenticate on an untrusted (public) terminal [26]. In the latter scheme, a trusted third party server is used to establish what is essentially a VNC connection via an untrusted terminal, while authenticating to it using the phone's internet connection (and issuing temporary credentials to the untrusted device). While not directly addressing the problems we are facing, this work contains interesting and useful discussion of the design issues that arise with such two-factor authentication schemes on mobile devices.

Farb et al. [12] considered making personal key management and distribution easier for smartphone users. While we are working toward similar goals of user-friendliness and easy adoption, they did not consider the case of users with multiple devices. Other researchers have proposed protocols for *multiple* users in proximity to communicate anonymously [17, 23, 24], and we are investigating using similar schemes for the single-user setting in order to decrease potential side-channel privacy invasions. Geambasu et al. [13] presented Keypad, a filesystem which stores per-file keys on a remote server. This provides some of the properties of our OmniNotes app, such as remotely revoking access from lost devices and creating a guaranteed audit log, but is not as flexible in terms of combining multiple devices (which may be local, thus not necessitating an internet connection).

There has been some non-academic work on sharing private keys between a user's personal devices, but these rarely incorporate threshold cryptography and tend to rely on simply encrypting the private key and uploading it to a third-party synchronization service. Such is the case with Whiteout.io, which generates a long random password for the user and requires them to write it down and manually enter it into their devices during configuration [19]). Apple iMessage, a widely-deployed instant messaging platform, uses another of the approaches discussed in Section 2.1. Users have a separate keypair generated for each of their devices, and key distribution is done by sending all of the user's public keys from Apple's central server (and encryption by encrypting to *all* of the user's devices) [2]. Gil summarized common community approaches to this problem in 2014 [15].

8. CONCLUSION

In this work, we presented Shatter, a free and open-source library and set of applications for providing threshold cryptographic protections to desktop and Android apps. As opposed to making users copy private keys to each of their multiple devices, introducing a significant weakness to device theft and compromise, we instead leverage this multitude of devices to provide significant protection against a variety of adversaries. Shatter allows app developers to easily perform distributed signature and encryption operations, by communicating with a central application that takes care of enrolment and revocation of the user's other devices. We showed that Shatter has acceptable overhead in the context of two real-world applications—ChatSecure and OmniNotes—and that threshold cryptography is, indeed, a feasible method of distributing and protecting "keys" across a user's different devices.

Shatter and its source code are available now, and are under active development by our lab. Ongoing work includes adding support for more cryptographic algorithms, improving the user experience to make it desirable by everyday device owners, and adding support for new technologies such as Bluetooth LE and new Internet of Things platforms.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments. This work is supported by funding from the Natural Sciences and Engineering Research Council of Canada, the Ontario Research Fund, and a Google Focused Research Award.

10. REFERENCES

[1] Overview of projects working on next-generation secure email. https://github.com/OpenTechFund/secure-email. Accessed Feb 2015.

[2] Apple Inc. iOS security. https://www.apple.com/business/docs/iOS_Security_Guide.pdf, June 2015.

[3] M. Bellare, D. Micciancio, and B. Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *Advances in Cryptology – Eurocrypt 2003*, pages 614–629. Springer, 2003.

[4] B. Blakley, G. Blakley, A. H. Chan, and J. L. Massey. Threshold schemes with disenrollment. In *Advances in Cryptology – CRYPTO'92*, pages 540–548. Springer, 1993.

[5] G. R. Blakley. Safeguarding cryptographic keys. In *the National Computer Conference*, volume 48, pages 313–317, 1979.

[6] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *2004 ACM Workshop on Privacy in the Electronic Society*, pages 77–84. ACM, 2004.

[7] P. Bright and D. Goodin. Encrypted e-mail: How much annoyance will you tolerate to keep the NSA away? *Ars Technica*, June 2013.

[8] A. Chowdhry. Gmail's 'Undo send' option officially rolls out. *Forbes*, June 2015.

[9] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *8th Annual International Conference on Mobile Computing and Networking*, pages 1–11. ACM, 2002.

[10] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Public Key Cryptography*, pages 119–136. Springer, 2001.

[11] Y. Desmedt, M. Burmester, R. Safavi-Naini, and H. Wang. Threshold things that think (T4): Security requirements to cope with theft of handheld/handless internet devices. In *Symposium on Requirements Engineering for Information Security*, 2001.

[12] M. Farb, Y.-H. Lin, T. H.-J. Kim, J. McCune, and A. Perrig. Safeslinger: Easy-to-use and secure public-key exchange. In *19th Annual International Conference on Mobile Computing & Networking*, pages 417–428. ACM, 2013.

[13] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. In *Sixth Conference on Computer Systems*, EuroSys '11, pages 1–16, New York, NY, USA, 2011. ACM.

[14] R. Gennaro, S. Goldfeder, and A. Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security. In *14th International Conference on Applied Cryptography and Network Security*. Springer, 2016.

[15] D. L. Gil. Multiple devices and key synchronization. https://github.com/coruus/zero-one/blob/master/multidevice-keysync.markdown, 2014.

[16] M. Green. The daunting challenge of secure e-mail. *The New Yorker*, November 2013.

[17] B. Greenstein, D. McCoy, J. Pang, T. Kohno, S. Seshan, and D. Wetherall. Improving wireless privacy with an identifier-free link layer protocol. In *Sixth International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 40–53, New York, NY, USA, 2008. ACM.

[18] G. Greenwald, E. MacAskill, and L. Poitras. Edward Snowden: The whistleblower behind the NSA surveillance revelations. *The Guardian*, 2013.

[19] T. Hase. Secure PGP key sync – a proposal. https://blog.whiteout.io/2014/07/07/secure-pgp-key-sync-a-proposal/, 2014.

[20] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances in Cryptology – CRYPT0'95*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer Berlin Heidelberg, 1995.

[21] A. Kalamandeen, A. Scannell, E. de Lara, A. Sheth, and A. LaMarca. Ensemble: Cooperative proximity-based authentication. In *8th International Conference on Mobile Systems, Applications, and Services*, pages 331–344. ACM, 2010.

[22] F. M. A. Krause. Designing secure & usable Picosiblings. https://www.cl.cam.ac.uk/~fms27/papers/2014-Krause-picosiblings.pdf, 2014. Masters thesis.

[23] M. Lentz, V. Erdélyi, P. Aditya, E. Shi, P. Druschel, and B. Bhattacharjee. SDDR: Light-weight, secure mobile encounters. In *23rd USENIX Security Symposium*, pages 925–940, 2014.

[24] Y.-H. Lin, A. Studer, H.-C. Hsiao, J. M. McCune, K.-H. Wang, M. Krohn, P.-L. Lin, A. Perrig, H.-M. Sun, and B.-Y. Yang. Spate: Small-group PKI-less authenticated trust establishment. In *7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 1–14, New York, NY, USA, 2009. ACM.

[25] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium*, pages 383–398, Aug. 2015.

[26] A. Oprea, D. Balfanz, G. Durfee, and D. Smetters. Securing a remote terminal application with a mobile trusted device. In *20th Annual Computer Security Applications Conference*, pages 438–447, Dec 2004.

[27] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology – EUROCRYPT'99*, pages 223–238. Springer, 1999.

[28] R. Peeters. Security architecture for things that think. http://www.cosic.esat.kuleuven.be/publications/thesis-202.pdf, 2012. Ph.D. thesis.

[29] R. Peeters, M. Kohlweiss, and B. Preneel. Threshold things that think: Authorisation for resharing. In J. Camenisch and D. Kesdogan, editors, *iNetSec 2009 – Open Research Problems in Network Security*, volume 309 of *IFIP Advances in Information and Communication Technology*, pages 111–124. Springer Berlin Heidelberg, 2009.

[30] R. Peeters, S. Nikova, and B. Preneel. Practical RSA threshold decryption for things that think. In *3rd Benelux Workshop on Information and System Security*, 2008.

[31] R. Peeters, D. Singelée, and B. Preneel. Threshold-based location-aware access control. *Mobile and Handheld Computing Solutions for Organizations and End-Users*, pages 20–36, 2013.

[32] S. Sheng, L. Broderick, C. Koranda, and J. Hyland. Why Johnny still can't encrypt: Evaluating the usability of email encryption software. In *2006 Symposium On Usable Privacy and Security - Poster Session*, 2006.

[33] V. Shoup. Practical threshold signatures. In *19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag.

[34] K. Simoens, R. Peeters, and B. Preneel. Increased resilience in threshold cryptography: sharing a secret with devices that cannot store shares. In *Pairing-Based Cryptography-Pairing 2010*, pages 116–135. Springer, 2010.

[35] S. Sinclair and S. Smith. PorKI: Making user PKI safe on machines of heterogeneous trustworthiness. In *21st*

Annual Computer Security Applications Conference, pages 10 pp.–430, Dec 2005.

[36] D. Singelee, R. Peeters, and B. Preneel. Toward more secure and reliable access control. *IEEE Pervasive Computing*, (3):76–83, 2012.

[37] F. Stajano. Pico: No more passwords! In *Security Protocols XIX*, volume 7114 of *Lecture Notes in Computer Science*, pages 49–81. Springer Berlin Heidelberg, 2011.

[38] O. Stannard and F. Stajano. Am I in good company? A privacy-protecting protocol for cooperating ubiquitous computing devices. In *Security Protocols XX*, volume 7622 of *Lecture Notes in Computer Science*, pages 223–230. Springer Berlin Heidelberg, 2012.

[39] Q. Staórd-Fraser, G. Jenkinson, F. Stajano, M. Spencer, C. Warrington, and J. Payne. To have and have not: Variations on secret sharing to model user presence. In *2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pages 1313–1320. ACM, 2014.

[40] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *8th USENIX Security Symposium*, SSYM'99, pages 14–14, Berkeley, CA, USA, 1999.