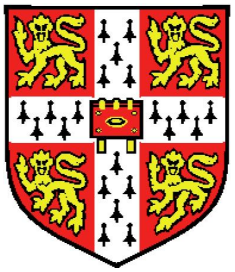
Security architecture and implementation for a TPM-based mobile authentication device



Anders Bentzon

Wolfson College

June 2013

A dissertation submitted for the degree of

Master of Philosophy in Advanced Computer Science

(Option B — Research project)

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Declaration

I, Anders Bentzon of Wolfson College, being a candidate for the M.Phil. in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,995[1] (from page 1, excluding the appendix and bibliography)

Signed:

Date:

[1]Calculated with the command `texcount -brief -inc pico.tex`.

Abstract

Today, passwords are used everywhere to authenticate users. While they are simple for website administrators and software developers to deploy, from a usability perspective, they are becoming increasingly problematic. This is particularly because maintaining adequate security puts an inordinate number of demands on the passwords (difficult to guess, must not be reused, should be changed regularly and so forth), and this translates into an inordinate number of demands on the users.

The Pico has been proposed as an alternative to passwords, that preserves their good qualities while dispensing with the bad. Our main contribution in this dissertation is to take a closer look at the Pico design by implementing a prototype. In the course of this, we survey two industry initiatives that also seek to replace passwords, concluding that, from a design perspective, the Pico is superior, although it suffers from lack of deployability; we investigate using a Trusted Platform Module as tamper-proof storage and conclude that this is feasible and advantageous; we propose an architectural design of the prototype on the Raspberry Pi and provide a working implementation in C, the source code of which is available as open source; and we outline solutions to solving two problems that are currently not addressed by the Pico's design (effective and secure revocation, and storage of non-authentication keys).

We see this dissertation as an incremental step in the development of the Pico and hope that future developers will benefit from its contributions.

Acknowledgements

In writing this dissertation, I wish to thank my supervisor, Dr Frank Stajano, who originally devised the idea behind the Pico. His enthusiasm, encouragement and many insightful comments have been invaluable. I have also benefited a great deal from the helpful remarks and observations of Dr Quentin Stafford-Fraser.

I wish to thank the people at the University of Cambridge Computer Laboratory — students and staff members alike — for contributing to an atmosphere that fosters learning and high-quality research. Any flaws that remain in this dissertation are, of course, entirely my own responsibility.

Furthermore, I wish to acknowledge Tabitha Spence for proofreading a draft of this dissertation, and to thank the open-source community — in particular, the developers of the Linux kernel, of the Raspbian distribution, and of the numerous open-source libraries used in this project.

Finally, the research conducted in the course of my Master's degree has been funded through a grant from the Danish government, for which I am grateful.

Contents

Chapter 1

Introduction

Passwords are a nightmare. If you reuse the same password with different services, you are in serious trouble if someone ever guesses or otherwise obtains it — this is true even if the password is long and complex. If you do the right thing and generate a unique, random password for each service you use, it quickly gets really difficult to manage all these passwords, even using a password 'wallet'.[1] The fundamental problem is that the optimal password — one that is long, complex, hard to guess, unique, and not written down anywhere — is very difficult to remember, since, by definition, such a password must be generated for each service you wish to use that requires a password. This does not scale very well when an average user needs dozens, or maybe even hundreds, of such passwords. Wouldn't it be nice to live in a world without this pain?

That is the objective of this project — providing a contribution that moves us closer to a password-free world. There are many partial alternatives available today,[2] but most are imperfect because they attempt to compensate for the one benefit that passwords inherently possess — they are cheap and very easy to deploy. The system we deal with in this project — the Pico — is a clean-slate solution, intended not to be constrained by these limitations. Imagine we live in a world where it is possible to start from scratch and design the perfect replacement for passwords — how would we do it?

[1] One of the earliest of these is Bruce Schneier's Password Safe from 1999. `http://www.schneier.com/passsafe.html`

[2] We mention some of them in Chapter 2.

1.1 Contributions

The work carried out in this dissertation builds on the Pico, originally proposed by Stajano (2011). Specifically, the contributions of this dissertation are as follows:

- We survey different approaches to solving the online authentication problem without using passwords (Chapter 2), outlining and discussing which requirements may meaningfully be set for such systems.
- We provide a detailed review of the Pico proposal (Stajano, 2011), identifying strengths and weaknesses and suggesting improvements to its protocols (Chapter 3).
- We show that it is feasible *and* advantageous to use a Trusted Platform Module, originally devised for other applications, in hardware security tokens (Section 4.1).
- We further provide a working, albeit limited, implementation (for which the full C code is provided as open source — see Appendix A) on the Raspberry Pi platform with a software-emulated TPM (Section 4.2). The implementation is verified using a comprehensive test bench that also validates the code by running it on a PC platform with a hardware TPM (Section 5.2).
- We take a critical look at the Pico as a concept (Section 5.1), discussing in detail a way to solve to as yet unresolved problems related to the Pico (revocation and storing non-authentication keys — Section 5.3).

1.2 The Pico

The Pico (Stajano, 2011) is designed to make all passwords obsolete. While we will take a more formal look at the design requirements in Chapter 2, we provide a general introduction here.

To accomplish the goal of not requiring passwords, the user carries with him a small token (the Pico) that stores the cryptographic authentication keys he needs to prove his identity to the services he wishes to use.

How does the user log into an account? When the user wishes to authenticate with an account (either a website, a local computer, or maybe even the access-control mechanisms of a door — called the *service*), he scans a QR-code that contains information about the account, and the Pico then opens a connection to

the service and carries out an authentication protocol. The connection is established using short-range radio, such as Bluetooth; in the case of a website, the Pico communicates through the user's local computer (the host computer) to the remote service.

. . . and how does he log out again? The Pico provides *continuous* authentication: as long as the Pico is within range of the host computer, the user remains logged in, but when the Pico comes out of range, it automatically logs out.

How does the user create new accounts with the Pico? This depends on the account type — but generally, by scanning a particular QR-code that instructs the Pico to generate a new set of credentials, and then to communicate with the service and present its new credentials. (Note: we use asymmetric cryptography for this, so the service only learns of the account holder's public key.)

What happens if the Pico is stolen? The Pico is intended to hold *all* the credentials of its owner, so of course this could easily be catastrophic. To prevent a thief from gaining access to these credentials, they are encrypted — and the encryption key is split into shares using threshold cryptography. These shares are distributed to a number of *Picosiblings* that are everyday objects that the user carries around with him (such as shoes, a belt, a pair of glasses, a watch, and so on). If enough of these are within range of the Pico, the encryption key can be assembled and the credentials unlocked; if not, they remain secure, even if the Pico is stolen.

What happens if the Pico is lost? Stajano (2011) envisions using a docking station to recharge the Pico and take backups of the credentials database. This is not investigated in this dissertation.

1.3 Threat model

Formally, we use a (n, k)-threshold scheme: there are n shares (Picosiblings) in total, and k of these are required to assemble the secret. We assume that an attacker cannot gain access to k Picosiblings, and that he cannot capture the Pico while it is unlocked. We assume, however, that the attacker can steal the Pico when it is locked, and that he can disassemble and analyse its hardware and software components. Furthermore, he can not only eavesdrop, but actively participate on the communication channel between the Pico and its Picosiblings and the host computer, as well as between the host computer and the remote service.

However, we do not want to get carried away: at some point, the Pico stops

being the weakest link. If the mafia want access to your credentials, they are likely to use other means than purely technological (see Figure 1.1) — passwords do not protect against this, and neither does the Pico. To quote Roger Needham:

"Whoever thinks his problem can be solved using cryptography, doesn't understand his problem and doesn't understand cryptography."[3]

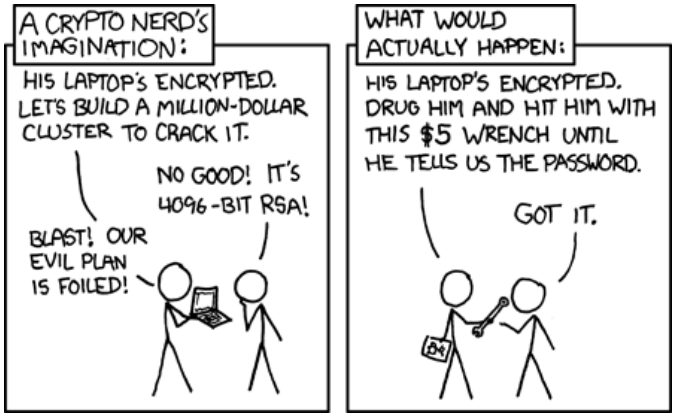


Figure 1.1: Technology does not solve all our security-related problems. `http://xkcd.com/538/`

1.4 Prerequisites

We assume the reader is familiar with the basic concepts of cryptography — symmetric and asymmetric algorithms, hash functions, message authentication codes (MACs) (these topics are covered by Katz and Lindell (2008)) — as well as threshold cryptography (in particular, the secret-sharing scheme of Shamir (1979)).

Furthermore, a superficial understanding of Trusted Computing, as put forward by the Trusted Computing Group (TCG)[4], is helpful. The main component is the Trusted Platform Module (TPM) — a little chip built into many high-end laptops. It is physically separate in hardware from the CPU and implements some of the features of a hardware-security module. In particular, it is resistant to many software-based attacks, and typically provides a degree of tamper resistance to make hardware attacks at least difficult to carry out. An important idea is that of trusted boot — at system start-up, the TPM can be used to ensure that the computer boots a valid operating system.[5]

[3] Attributed by Roger Needham and Butler Lampson to each other, as quoted by Anderson (2008, p. 633).

[4] `http://www.trustedcomputinggroup.org/`

[5] The definition of 'valid' is open to controversy — some take it to mean non-pirated; but it

1.5 Related work

The concept of the Pico was originally devised and presented by Stajano (2011).

Passwords started coming into use when early time-sharing systems made it necessary to authenticate users centrally. Morris and Thompson (1979) describe the early UNIX password implementation and the reasoning behind its design decisions, highlighting issues such as dictionary attacks, salting and algorithm speed that remain relevant today. One of the earliest implementations using one-way functions was the Cambridge Multiple Access System, where the concept was devised by Roger Needham and Mike Guy.[6]

Password reuse across websites may cause a 'domino effect' (Ives et al., 2004), and techniques from natural language processing can make it significantly easier to guess even long passwords (Rao et al., 2012). Bonneau (2012) has investigated the difficulty of guessing human-chosen secrets.

Much research has been conducted on trusted boot; Parno et al. (2010, 2011) present a good survey and coverage of the area. Trusted boot is based on the idea of hash chains as originally put forward by Arbaugh et al. (1997). From a practical perspective, Challener et al. (2007) provide a good introduction to working with TPMs and trusted computing.

The idea of using the visual channel and QR-codes to identify devices was proposed originally by McCune et al. (2005) in the 'Seeing-is-Believing' project.

Threshold cryptography was originally proposed by Shamir (1979). Recently, Peeters (2012) has investigated how to use threshold cryptography in ubiquitous computing with asymmetric cryptography.

Please see the related works sections of each chapter for more references.

can also mean an image that has been verified not to be infected with malware.

[6] This is mentioned by Anderson (2008, page 56) and Bidgoli (2004, page 4). See also Wilkes (1968).

Chapter 2

Architectural design considerations

This chapter considers the Pico and related solutions from a system-architectural point of view. We make the following contributions:

- We compare the Pico with two recent industry initiatives seeking to (partially) replace passwords (Section 2.1).
- We consider fundamental security-related design issues that influence our choice of hardware platform (Section 2.2).

2.1 Systems that replace passwords

In this section, we discuss (in addition to the Pico) a new industry alliance that makes some of the same promises as the Pico does, as well as Google's approach to authentication.

2.1.1 Pico

The Pico is a concept first put forward by Stajano (2011) at the Security Protocols Workshop 2011 in Cambridge. It is intended as a clean-slate solution to solve the authentication problem by replacing *all* passwords with a little token (physical object) that users carry around with them. Using the taxonomy of Stajano, we outline the properties that the Pico should have in Table 2.1[1].

[1] See Stajano (2011, Table 1).

Table 2.1: Properties we wish the Pico to have. See Stajano (2011, Table 1).

MEMORYLESS	The user is not required to memorise secrets.
SCALABLE	Works with hundreds or thousands of different services.
SECURE	At least as secure as passwords, if they are used correctly.
LOSS-RESISTANT	User does not lose his credentials if the token is lost.
THEFT-RESISTANT	A thief cannot impersonate the user by stealing the token.
WORKS-FOR-ALL	Works not only for webpages, but for all passwords.
FROM-ANYWHERE	The system works from any client.
NO-SEARCH	The system selects the correct credentials on its own.
NO-TYPING	The user does not have manually to type in the password.
CONTINUOUS	The authentication occurs throughout a session, not just when the session is initiated.
NO-WEAK	It is not possible to choose a weak password.
NO-REUSE	It is not possible to reuse the same password with different services.
NO-PHISHING	The user cannot be tricked by an adversary impersonating a legitimate service.
NO-EAVESDROPPING	It is not possible to eavesdrop on the communication session.
NO-KEYLOGGING	An adversary does not gain anything from using a keylogger.
NO-SURFING	An adversary does not gain anything from shoulder surfing.
NO-LINKAGE	A user's accounts cannot be linked by colluding services, or even by the same service.

The rationale behind these requirements is that, with few exceptions, we want the Pico to have all the beneficial properties of good passwords (in particular, Loss-resistant and Theft-resistant), but we want to get rid of the bad properties (emphasising properties such as No-weak and No-reuse from a security perspective, and Memoryless and No-typing from a usability perspective). An important, and sadly oft-omitted, property is No-linkage: we do not want different service providers to be able to ascertain that different accounts belong to the same person; even if a user has different accounts with the same service, we do not want the service to know this.[2]

2.1.2 Fast Identity Online Alliance

The Fast Identity Online (FIDO) Alliance was formed in the summer of 2012 by several companies (Lenovo, PayPal, Infineon, Agnitio, Nok Nok Labs and Validity), and was joined in the spring of 2013 by Google.[3] The goal of the project is to establish a single standard for online authentication that provides better usability and security than passwords.

The technical details of the project remain unclear, but an overall summary is provided on the FIDO website.[4] The design consists of several categories of components:

- *FIDO tokens* are physical devices used in the authentication process, capable of generating one-time passwords (OTPs). A token may require proof of user presence — such as a PIN, password or a biometric identifier — to unlock itself. Examples of tokens are USB devices, a Trusted Platform Module built into the computer, or a fingerprint reader.

- The *relying party* is the service (typically a website) that the user wishes to log in to.

- The *validation cache* belongs to the relying party and contains the information necessary to validate the OTPs generated by the FIDO tokens.

- *FIDO repositories* are published by token vendors and contain the token-specific information needed by the validation caches to authenticate the

[2] Obviously, if the user does not take particular steps to prevent this, the service could easily deduce this from looking at IP addresses, browser fingerprints, etc. We only consider the authentication system.

[3] Unless otherwise noted, information in this section stems from `http://www.fidoalliance.org/`.

[4] `http://www.fidoalliance.org/how-it-works.html`

users. It is not clear whether this information also allows for impersonation of the tokens.

- A *browser plug-in* facilitates the communication between the relying party and the user's browser, by informing the relying party of the presence of a FIDO token, and by relaying the encrypted authentication stream.

- *Device-specific modules* are supplied by the token vendors, to provide the browser plug-in with a uniform interface. This ensures that any FIDO token will work for the user, as long as the correct module is installed.

One of the major design goals of the project is to create an open standard that allows different vendors to supply tokens that satisfy varying requirements while adhering to the same architecture. A website may thus define its own security policy, and in this way some websites may choose to allow authentication from tokens not protected by a password, while others may require a fingerprint-protected token.

The FIDO project is ambitious, and its objective of replacing passwords is similar to that of the Pico. Another common design goal is the strong focus on usability. From the user's perspective, the authentication process should be seamless and require little configuration. There are, however, some crucial differences. As the technical details are not yet available, this discussion is based only on the high-level description from the project's website, so these remarks may not apply to the final realisation of the project.

One of the important properties of the Pico is No-linkage, as it is essential in order to guarantee users' privacy. FIDO seems, by design, to forgo privacy by supplying each token with a unique ID. This means that a user cannot create more than one account with a service provider without the provider's being aware of it, and that colluding websites can track users. Additionally, the browser plug-in informs websites about the presence of a FIDO token (presumably in the browser's user-agent string), and FIDO-enabled websites use this information to provide the user with the option to either link his FIDO token to the website, or to log in with his already linked token. From the description of the FIDO project,[5] it is not clear how much information is passed to websites that the user browses, but it seems that 1) *any* website can detect that the user has a FIDO token, and 2) *any FIDO-enabled* website can track the user based on the token's unique ID. (This is necessary in order for the website to determine whether to offer the user the

[5] `http://www.fidoalliance.org/how-it-works.html`, under 'Relying Party / Website'.

option of linking a new token, or authenticating with an existing one.) Clearly, if this is true, it violates common expectations of user privacy.

Additionally, the terminology of the project description is somewhat confusing with regard to two-factor authentication. Swiping a fingerprint to a FIDO token is considered two-factor authentication on the premise that it requires the user to possess both the correct biometric data and the token.[6] However, this fingerprint is checked only by the token, not by the server; if the token is satisfied, it sends a signal to the server to the effect that it has seen a correct fingerprint. Unless the token is verifiably tamper resistant, this does not constitute two-factor authentication since it is sufficient for an adversary to pretend to the server to be a legitimate token.

Furthermore, unsupervised fingerprint scanning can in many cases be fabricated by dummy fingers (Matsumoto et al., 2002; van der Putte and Keuning, 2000). Fundamentally, a system cannot be considered truly two-factor unless both factors are verified by the remote service, and in the case of unsupervised access, biometric identifiers are necessarily problematic since they, by their very nature, are not secret.

In conclusion, while the FIDO initiative is encouraging because it signals that the industry recognises the shortcomings of passwords and is actively working on replacing them, some fundamental weaknesses in its design mean that it is not a good alternative to the Pico project.

2.1.3 Google

Grosse and Upadhyay (2013) and Sachs (2013) discuss the approach taken by Google to replace passwords. The main notion is the concept of *device-centric authorisation* — when strong authentication is initially employed to authorise a specific device (such as a browser or mobile phone) to access an account, subsequent access is granted either automatically, or by using weaker authentication. An example is two-step verification as used by Gmail and other Google services: the user can use one-time passwords generated on a smartphone, or an SMS text message, to authorise a browser initially to log into an account, after which only a password is required.[7]

An important point made by Grosse and Upadhyay is that this approach is

[6] `http://www.fidoalliance.org/how-it-works.html`, under 'Finger Scan Experience'.

[7] The user also has the option of granting authorisation for the current session only, which is useful when a public computer is used.

not always feasible, since many clients only support passwords. To accommodate these, Google introduced application-specific passwords (ASPs). As an example, if a user wants to let an IMAP client access his Gmail account, he can generate an ASP for that particular client. While still a password, it typically needs to be entered into the client only once; therefore, it can be long and contain a lot of entropy. Having unique passwords for each client makes revocation easier, and it potentially allows for better auditing, but it is essentially a work-around.

Grosse and Upadhyay and Sachs consider ASPs a temporary work-around and envision instead a centralised account management model where the user can authenticate with two-step verification and store these credentials centrally in the device. Local apps can then ask this module to authenticate on their behalf with remote services (in essence, the task of authentication is out-sourced from each app to the operating system). This means that the user needs to perform this task only once per account, rather than for each application.

Furthermore, Grosse and Upadhyay (2013) and Sachs (2013) describe how Google plan on adopting dedicated authentication tokens that on the surface seem not unlike the Pico described by Stajano (2011). Indeed, Grosse and Upadhyay outline three requirements that seem similar to those of the Pico: no software installation other than a compliant browser, support by many websites while preserving privacy by not making it possible for them to track and correlate users, and a simple and open framework that does not depend on a trusted third-party. The authors envisage that such tokens can be used to authorise new devices easily (as an alternative to the rather cumbersome two-step verification based on one-time passwords), and develop the idea so that already-authorised devices can then in turn authorise new devices: the user can use his smartphone, which has already been authorised by a dedicated token, to authorise his computer.

The main difference between Google's approach and the Pico is the way clients are authorised to access accounts. In Google's system, strong authentication is required only once, when the device is initially bound to the account, whereas the Pico's CONTINUOUS property binds the device seamlessly to the account only when the user is present. This difference asserts itself in two important ways, usability and trust. Because Google's authorisation only needs to take place once per device, the process does not have to be particularly easy; indeed, Sachs goes so far as to say, "We don't mind making it painful for users to sign into their device if they only have to do it once."[8] The ambition for the Pico, on the other

[8]In an interview about the 5-year roadmap (Sachs, 2013), available at `http://www.zdnet.com/google-unveils-5-year-roadmap-for-strong-authentication-7000015147/`

hand, is to make it easy for the user each time authentication is required. This also has implications for the security: if an authorised smartphone is stolen, the thief gains access to all the accounts stored on it (provided he can circumvent its access-control mechanisms). Google addresses this by making revocation easy, but this may not always be feasible (the user may not notice the phone's absence for a while; and even when he does find out, he may be without internet access, and thus unable to revoke it).[9] Of course, if the Pico *and* enough Picosiblings are stolen, the problem is the same; but all other things being equal, it should be more difficult to lose the Pico along with k Picosiblings, than just one of the many authorised devices users are likely to carry around with them.

This discussion does, however, highlight an issue with the Continuous property that has yet to be addressed. Envisioning that a computer and any sensitive websites that the user has signed into lock themselves when the Pico comes out of range is easy; but how does this work with all the other programs, such as IMAP clients and instant messengers (IMs), that the user also has running? If they all have their credentials stored on the Pico — as indeed they must have, if the Pico is to be a true replacement for passwords — they should be locked when the user leaves to fetch a cup of coffee, but this is often not desirable (he may wish to remain signed into Skype, and to keep receiving e-mails). While application-specific passwords were intended as a work-around for compatibility reasons, they do address this situation neatly, and it is unclear how the current Pico architecture can solve this problem.

2.1.4 Other systems

Many systems that offer to replace passwords have been proposed and marketed. Some well-known alternatives, such as the RSA SecurID and the Yubico Yubikey, are discussed by Stajano (2011), and Bonneau et al. (2012) survey a comprehensive range of products of different categories, including hardware tokens.[10]

It should be mentioned that using biometric identifiers does not automatically solve the remote authentication problem. For one thing, they have severe privacy and anonymity issues and are often impossible to revoke. For another, it is not clear how a remote server can verify a piece of biometric information that has been read on a client computer (this issue is also faced by the FIDO solution, as

(accessed 27/5/2013).

[9] Other measures, such as the 'login approval' mentioned by Sachs (2013), are only really effective against phishing.

[10] Bonneau, Herley, van Oorschot and Stajano (2012)

discussed in Section 2.1.2).

2.2 Generic system requirements

Using threshold cryptography to secure the Pico's master key raises some practical issues. The immediate problem is how the Pico is supposed to prove its identity to the Picosiblings; clearly, we do not want the Picosiblings to reveal their shares to anyone, so each Picosibling needs to be paired to the Pico with a cryptographic key (Stajano (2011) envisions a concept similar to the Resurrecting Duckling (Stajano and Anderson, 2000)). But these keys of course cannot be stored encrypted on the Pico, since they are needed to bootstrap the encryption process, so they have to be stored in cleartext. Still, they need to be reasonably secured on the Pico,[11] since otherwise an attacker could use them to impersonate the Pico to the Picosiblings and thus obtain all the secret shares. We therefore propose to store these keys in tamper-resistant memory, or alternatively, to store them encrypted in regular non-volatile memory, with the decryption key in tamper-proof storage.[12]

However, even if an attacker cannot forcibly obtain the Picosibling authentication keys from tamper-proof memory, he may be able to employ an 'evil maid' attack by replacing the Pico's operating system (or patching it with malware) when the owner leaves the Pico in the office while he is out to lunch; since the OS is not protected with tamper-proof technology,[13] this attack is feasible. When the owner comes back from lunch and uses the Pico, the infected OS gets access to the shares of all Picosiblings within range, and accordingly, the master key as well as the database encrypted with this key.

There are two ways to protect against this attack. The first is to store the OS, along with all its dependencies, in tamper-*evident* read-only memory. This is cheaper than employing full tamper-resistant technology, but it requires that the user physically inspect the Pico before every use in order to confirm that it has not

[11] They should also be stored in a secure way in the Picosiblings, but as we want these devices to be cheap, this is difficult. We do not consider tamper resistance of the Picosiblings in this dissertation. It is worth remarking here that, because of the threshold scheme, losing up to k Picosiblings is not catastrophic.

[12] The degree of tamper resistance required will have to be analysed in future work; a realistic goal is to impede the common thief and his skilled friends, but it is probably not feasible to protect against a determined attacker with access to specialist equipment. We consider defences against both invasive and non-invasive, as well as active and passive, attacks as belonging to this category. In the taxonomy of Abraham et al. (1991), it is probably realistic to protect only against *Class I* attackers (clever outsiders).

[13] Making the *entire* device tamper resistant is difficult, so we envisage only securing the key storage.

been tampered with. From a usability perspective, this solution is problematic; furthermore, it would likely mean that it would be difficult, if not impossible, to update the Pico software after the device has left the production facilities, thus impeding future security or functionality updates. The second, and better, solution is to utilise an approach akin to trusted boot, by which the tamper-proof chip protecting the Picosibling authentication keys verifies that the Pico is running a legitimate version of the OS before releasing the keys. In practice, the Pico software could be signed by a key held by the manufacturer, with the corresponding public key embedded in the security chip; the chip would then refuse to release its secret if the signature check failed. Such an approach would make it possible for the vendor to ship future software upgrades, as long as they are signed with the same key.[14]

Finally, we also require the presence of a secure random number generator in order to be able to generate good key material (not only for the Picosibling keys, but for all new credentials).

2.3 Summary

This chapter consists of two main parts that both deal with replacing passwords from a system-centric point of view. In the first part, we discussed the design requirements of the Pico, and we surveyed two such systems: the FIDO Alliance and Google. The contributions from FIDO[15] and Google (Grosse and Upadhyay, 2013; Sachs, 2013) are very recent, and to our knowledge they have not before been analysed in the literature. In discussing these systems, we highlighted weaknesses and strong points, and we compared these with the Pico.

In the second part of the chapter, we discussed useful design criteria for the Pico. In order to keep the keys used to communicate with the Picosiblings secret, we arrived at three requirements: the Pico's security subsystem has to support trusted boot, it has to be tamper resistant, and it requires a proper random-number generator.

[14] This solution, in essence, bears some resemblance to methods employed in DRM-protected products, and indeed it would also make it impossible for third-party developers and hobbyists to customise the Pico. It remains to be seen if a solution can be found that would encourage and enable third-party developers to freely customise their own Picos, without endangering the security of the normal user running vendor-supplied and signed software.

[15] `http://www.fidoalliance.org/`

2.4 Related work

The importance of usability on security is investigated by Adams and Sasse (1999), who were the first to conclude that usability directly affects security: if users do not understand the reasoning behind security policies or mechanisms, they will seek to circumvent them. This is further emphasised by Whitten and Tygar (1999), who investigate the significance of the user interface in security products. More recently, Cheswick (2012) has argued that passwords, as we use them today, are not working, and that the system needs to be changed.

Chapter 3

Protocols

In this chapter, we describe the protocols used by the Pico. Two categories of protocols exist: the first for communicating with the Picosiblings, and the second for communicating with the (remote) service.

The contributions of this chapter are as follows:

- We reiterate the protocol invented by Stannard and Stajano (2012). We then describe how a known technique, counter resynchronisation, can be used in this context, and why it is sometimes useful not to request the share (Section 3.2).
- Inspired by the idea of Stajano (2011) and the contributions of Stannard (2012), we redesign the protocols used for account creation (Section 3.3) and login (Section 3.4).
- We discuss how to create and maintain a link between an authentication session and an interface, as well as ways to only allow account creation for particular users (Section 3.5).
- Finally, we discuss how the SSL/TLS protocols can be hardened using the Pico in order to prevent some phishing attacks (Section 3.6).

3.1 Notation

In this chapter, we follow the notation established in literature for cryptographic protocols. The steps of a protocol consist of messages between two actors (a sender

and a receiver). Curly brackets denote a cryptographic operation; the key used for the operation is appended in subscript. In case of a symmetric key, the operation is encryption followed by a message authentication code (MAC) of the ciphertext (Encrypt-then-MAC), and the same key is required to decrypt and verify the MAC.[1] If it is an asymmetric key, the operation is either encryption (if the public key is used), or signing (if the private key is used) — the opposite key component must then be used to undo the operation (either decrypt, or verify the signature).

Table 3.1 lists the actors and entities used in the protocols. Note that we enforce a policy on the usage of asymmetric key pairs: an asymmetric key pair can be used for either encryption or for signature operations, but not for both. The protocols must be designed to reflect that.

Table 3.1: Symbols of actors and entities in the protocols of Chapter 3.

S	The service
P	The Pico
s_j	The j'th Picosibling
A	An account on the Pico
K_S, K_S^{-1}	The service public and private keys, respectively (valid for *encrypting*)
K_A, K_A^{-1}	The account public and private keys, respectively (valid for *signing*)
K_{xy}	A symmetric session key between the actors x and y (consists of an encryption component and a MAC component)
N_x	A nonce (generated by x)
ID_A	Identifier for account A
c_j	A synchronisation counter (for Picosibling j)
x_j	The share of the j'th Picosibling
$h()$	A cryptographic hash function

3.2 Picosiblings

The protocol used for communication between the Pico and its Picosiblings was designed by Stannard and Stajano (2012). We briefly introduce it here.

First, the Pico P sends a message to a Picosibling s_j consisting of a random nonce N_j and a synchronisation counter c_j. This (and subsequent messages) are protected with Encrypt-then-MAC (the message is first encrypted, and then a

[1] Of course, the *same* key should never be used for encryption and for MAC — this is just to keep the notation clean; instead, consider that the symmetric key contains two components — an encryption key, and a MAC key.

MAC of the ciphertext is appended), using the key K_{Ps_j}:

$$\text{S1.} \quad P \to s_j : \qquad \{N_j, c_j\}_{K_{Ps_j}} .$$

If the MAC verifies correctly, the Picosibling knows that the request is legitimate, and it responds by sending back the same nonce, the same counter, and its secret share x_j:

$$\text{S2.} \quad s_j \to P : \qquad \{N_j, c_j, x_j\}_{K_{Ps_j}} .$$

Then, both parties renew the key K_{Ps_j} by hashing it. This prevents an adversary who captures the key to decrypt ciphertext he intercepted in the past:

$$\text{S3.} \quad \text{(local)} : \qquad K_{Ps_j} \;:=\; h\left(K_{Ps_j}\right) .$$

Stannard and Stajano (2012) introduce the counters, but they do not describe how to perform the synchronisation. We propose to follow the standard of RFC 4226 for HMAC-based one-time passwords.[2] In this algorithm, if the receiving party cannot immediately verify the MAC (step S2 above), it iterates over steps S2-3 until verification is successful, or until a maximum threshold is reached (in our implementation, 10 iterations). This adds robustness as it allows for communication packages to get dropped and for the sender (the Pico) to get too far ahead, while it never allows the sender (or an adversary) to use old key material.

Furthermore, we add to the message of step S1 a small boolean value that indicates whether the Pico is attempting to request the share x_j, or whether it merely is confirming whether enough valid Picosiblings are present. Why is this useful? Fundamentally, the fewer times the secret is collected, and the less it is available in its collected state in the Pico, the less susceptible it is to interception.[3] If we only collect the secret when it is actually needed (as opposed to the idea of Stannard and Stajano (2012), where it remains in the Pico for as long as the Pico is within range of its Picosiblings), we may at times wish to know if enough Picosiblings are present, without requiring their shares. This is advantageous when creating a new account: we use asymmetric encryption and are thus able to save the new credentials securely without loading the master key, although we only want

[2] `http://www.ietf.org/rfc/rfc4226.txt`, in Appendix E.4 (Resynchronisation counter-based protocol).

[3] We assume in the threat model of Section 1.3 that the Pico cannot be captured when it is unlocked. However, it makes sense to perform any straightforward design changes that increase security beyond this threat model.

to allow account creation when the Pico's owner (identified by his Picosiblings) is present.

3.3 Account creation

The service and the Pico communicate through a trusted[4], out-of-band channel (e.g. the visual channel using QR-codes). First, the Pico sends its public key K_S, along with any other information needed to establish the initial connection:

$$\text{C0.} \quad S \to P: \qquad K_S \qquad \text{(out of band)}.$$

When the Pico's owner wants to establish a new account at the service S, the Pico generates a new asymmetric key pair for that account, K_A and K_A^{-1}, to be used for authenticating the Pico in the future. The Pico sends

$$\text{C1.} \quad P \to S: \qquad \{K_{SP}, N_P\}_{K_S}, \{K_A\}_{K_{SP}}.$$

The first message contains a session key K_{SP} and a nonce N_P encrypted with the service's public key K_S, and the second message, encrypted with the session key, contains the public account key K_A. The symmetric session key allows the Pico and service to communicate using symmetric encryption, which is much cheaper than asymmetric encryption.

The service responds with

$$\text{C2.} \quad S \to P: \qquad \{N_P, N_S\}_{K_{SP}}.$$

The Pico's nonce N_P is an important element. First, it is encrypted with the session key K_{SP}. This confirms to the Pico that the server could decrypt the nonce (and the session key) using the service's private key K_S^{-1}; thus, it verifies that the Pico is talking to the server represented by the public key K_S. In the same packet, the service furthermore sends its own nonce N_S, the significance of which will be explained in the following paragraph.

[4]What is 'trust'? In the context of this project, we emphasise the importance of the initial channel being unrelated to the channel used in the rest of the protocol, so that an adversary has to attack both these channels. Furthermore, the visual channel seems to provide benefits from a usability perspective — requiring the user to physically point a camera at a QR-code implies intent, or, as McCune et al. (2005) call it, *demonstrative identification* (originally coined by Balfanz et al. (2002) in a related context).

Then follows

$$\text{C3.} \quad P \to S: \qquad \left\{ \{N_S\}_{K_A^{-1}} \right\}_{K_{SP}} .$$

The Pico signs the service's nonce and sends the result, encrypted with the session key. This establishes to the server that the Pico sent a valid public key to whose private component it has access. At this time, the server may send back an account ID ID_A, which the account may be referred to in the future:

$$\text{C4.} \quad S \to P: \qquad \{ID_A\}_{K_{SP}} .$$

At the end of this protocol, the following assumptions can be made by the parties:

1. S knows that K_A represents an account stored on a Pico.[5] It does not know any more than that; the account could be owned by anyone.
2. P knows that it has given its public key to the service represented by K_S, which it trusts (by definition).

3.4 Continuous authentication

To initiate the authentication process, the Pico generates a new symmetric session key K_{SP} and encrypts K_{SP} with the service's public key K_S. The result of this encryption is transmitted to the service, along with the account ID ID_A of the Pico owner's account, encrypted with the session key.

$$\text{A1.} \quad P \to S: \qquad \{K_{SP}\}_{K_S}, \{ID_A\}_{K_{SP}} .$$

The server receives this information and verifies that it actually has an account with the specified ID. Note that if the server is not who it claims to be — that is, if it is not in possession of K_S^{-1} — it cannot read K_{SP} and ID_A, and thus the Pico remains anonymous. If, however, the server is legitimate, and the account ID is valid, a nonce is sent back to the Pico.

$$\text{A2.} \quad S \to P: \qquad \{N_S\}_{K_{SP}} .$$

[5] Actually, S doesn't know anything about a Pico; but it knows of a user *somewhere* owning that public key pair. If it is for some reason important to prove to the server that a Pico device is used, the protocol needs to be expanded. Using the Pico's TPM to provide secure remote attestation may be an option.

Note that, if the session key K_{SP} is used purely for encryption (confidentiality), a malicious server could just send some arbitrary noise, which the Pico would decrypt to a random nonce without being any the wiser. However, if K_{SP} is also used to generate an HMAC (integrity), the Pico immediately discovers whether the session key was decrypted by the server, and by extension, whether the server is an imposter.

To prove its knowledge of the account private key, the Pico signs this nonce and sends it back to the server.

A3. $P \to S$: $\left\{ \{N_S\}_{K_A^{-1}} \right\}_{K_{SP}}$.

If the signature verifies correctly, the service knows that it is communicating with the proper owner of that particular account.

To provide continuous authentication, the connection is kept open, and the service periodically sends out a new nonce. It is not necessary for the Pico to sign each nonce, since it has already proven knowledge of its secret key K_A^{-1}. Thus, it is only necessary to reconfirm knowledge of the session key K_{SP} by performing a trivial modification to the nonce, such as:

A4. $S \to P$: $\{N_S\}_{K_{SP}}$ and

A5. $P \to S$: $\{N_S + 1\}_{K_{SP}}$.

3.5 Linked authentication

The protocols described in Sections 3.3 and 3.4 are sufficient in the cases where it is enough to establish the presence of the Pico at a particular place, for example when the Pico is used to unlock a computer workstation. If, on the other hand, the Pico is used to authenticate with a remote service, with the protocols described here being tunnelled through a local host, it is necessary to bind the Pico account and its authentication to a particular user interface (such as an instance of a website). For instance, one can imagine a website supporting the Pico as an authentication token. When the user visits the login page, he is presented with a QR-code containing an identifier for that website (the public key, or a hash of the public key) along with an authorisation nonce N_a.

A0. $S \to P$: K_S, N_a (out of band)

This QR-code is scanned by the Pico, which uses the service's public key to locate the account ID and load the proper account public key pair. Once the protocol of Section 3.4 has been executed, and the Pico has proven that it owns a valid account, it can then send:

$$\text{A4.} \quad P \to S: \qquad \{N_a\}_{K_{SP}}.$$

This creates a link between the authentication channel and the web interface, and the service can subsequently allow access through that particular web client on behalf of the public key pair residing in the Pico; in effect, the Pico's owner authorises access to a particular interface.

This procedure goes some way to defend against a relay attack, as it ensures not only that the Pico is talking to the right server (this was guaranteed with the original protocol), but also that the Pico authorises this authentication for one explicit token of access. Transmitting the QR-code securely seems sufficient. However, an adversary can still circumvent this protocol if he is able to modify the QR-code. For example, he can visit the login page of a service and obtain a QR-code (including an authorisation nonce for his own interface), and then inject this QR-code into the communication path of a user, replacing the legitimate user's authorisation nonce. When the user completes the authentication protocol, access is granted to the attacker's interface, not the user's.[6]

A similar approach may be taken when it is important that only particular users be allowed to create accounts. For instance, a bank may let users authenticate to its website using the Pico, so naturally only the bank's customers should be allowed to create Pico accounts, and these must, from the outset, be linked to a customer. One way to solve this is for the bank to generate a custom QR-code containing not only the bank's public key, but also the account ID, and only then execute the protocol of Section 3.3:

$$\text{C0.} \quad S \to P: \qquad K_S, ID_A \qquad (\text{out of band}, \text{sensitive}).$$

Of course, this information is very sensitive, since it allows anyone to create an account for the provided account ID and impersonate the owner. Also, ID_A should have such a size as to make it infeasible to blindly guess its value (128 bit seems

[6] In a recent real-world example, this was accomplished by installing a Trojan on users' computers (`http://finnaarupnielsen.wordpress.com/2012/03/04/wakeup-call-for-denmark-nemid-under-attack/`). See also Schneier (2005).

more than sufficient).[7] The point, however, is that it does not *remain* sensitive: if the Pico transmits this ID as part of the account creation process, the ID cannot be reused to create another account.

C4. $P \rightarrow S$: $\{ID_A\}_{K_{SP}}$.

3.6 Preventing phishing (TLS spoofing) attacks

Securing the QR-code with TLS may seemingly solve the problem of the relay attack discussed above, since the attacker is then unable to inject his own QR-code into the data stream.[8] However, TLS does not protect the user against a typical phishing attack, whereby the user is tricked into visiting a copy of a legitimate website residing at a false URL, but with a valid CA-signed TLS certificate. Thus, there are two problems with TLS. First, TLS requires that the user pay attention to the URL, and to whether or not TLS is enabled; and second, possessing a valid certificate is no guarantee of legitimacy, as anyone can get a certificate signed for his own URL.

That being said, TLS is in itself not an insecure protocol if these two flaws are corrected. Let us assume that an account has already been created at a remote service. The host computer H is running a client program that communicates with the Pico through a local connection, for example Bluetooth. Furthermore, this client program maintains a list of the accounts currently on the Pico — the account identifier ID_A as well as a user-friendly name and a URL. When the user wants to sign into a remote web service, he selects the account in question from this list, and the client program transmits the information to the Pico:

A0. $H \rightarrow P$: $\{ID_A\}_{K_{HP}}$.

[7] As an example, say a bank has 100 million customer accounts, of which all are open to be linked to a Pico account. Using 128-bit IDs, if an attacker can try one million IDs every second (remember, it is not a matter of raw computing power — these IDs need to be tried against the server, which would certainly detect such flooding), the likelihood of hitting one of these 100 million IDs remains negligible for 10^{20} years. By comparison, the age of the universe is approximately 10^{10} years. Actually, only 98 bits are needed to keep the probability negligible for 10^{10} years, so using 128 bits is very conservative.

[8] This of course does not solve the problem if the attacker uses a Trojan to inject the QR-code. However, if a Trojan is installed on the desktop computer, the user session could be high-jacked anyway. Laurie and Singer (2009) discuss ways in which an external, and presumably secure, device such as the Pico could be used to add some security, but that is not the focus of this project. Hence, we assume that the user's computer is clean.

This step replaces the QR-code in the original authentication protocol, and it simply tells the Pico to initiate authentication for the given account. The authentication proceeds as before, but at the end of the process, the remote service sends an authentication ticket N_a to the Pico along with the public TLS certificate for the website:

A4. $S \to P :$ $\{N_a, K_{S,TLS}\}_{K_{SP}}$.

The Pico simply forwards this information to the host computer:

A5. $P \to H :$ $\{N_a, K_{S,TLS}\}_{K_{HP}}$.

At this point, the client program can open a browser and point it to the right URL (which it presumably knows; alternatively, this can be transmitted in steps A4-5 as well). The browser compares $K_{S,TLS}$ to the certificate actually presented by the website and aborts if they do not match (this prevents a relay attack). If the website is legitimate, the browser can send the ticket N_a to prove its identity. Further requirements may dictate that tickets expire if unused within a very short window, say, 10 seconds.

3.7 Summary

This chapter describes protocols that can be used by the Pico to communicate with its Picosiblings and with the service with which the user wishes to authenticate. We started by summarising the Picosibling protocol (Stannard and Stajano, 2012) and showed how counter resynchronisation could be implemented easily. At the same time, we concluded that the approach originally suggested by Stajano (2011) — keeping each share in memory when the Pico is unlocked, and combining this with an internal counter — is not ideal, inasmuch as it needlessly leaves the secrets susceptible to capture. Instead, we suggested a hybrid approach that adds a new component (a 'pong' without a secret share) to the Picosibling protocol. We then went on to design protocols for creating a new account with a service, and for logging into an existing account.

In the latter part of the chapter, we considered how SSL/TLS can in some cases be used to prevent a relay attack, and we emphasised the importance of creating a strong link between the authentication session and the interface that is authenticated for remote authentication. Further, we remarked on how the Pico

can be used to provide extra protection against typical phishing attacks. While these two contributions are not used in the Pico implementation presented in this dissertation, we hope that they will prove useful in future Pico design iterations.

3.8 Related work

The idea of using an external device to protect against phishing is not new, but was suggested by Parno et al. (2006). The 'phoolproof phishing' mechanism allows a user to authenticate the server using, for example, a mobile phone or a PDA, and then to access the web interface through the normal browser. Password authentication (through the web browser) is still necessary. Also on the subject of using external tokens, Laurie and Singer (2009) argue why it is necessary to use a special-purpose device in conjunction with a commodity operating system to achieve an adequately secured trusted path for use in authentication.

Stannard and Stajano (2012) introduce the protocol of Section 3.2, but without explaining how to do counter resynchronisation, and without mentioning the added value of getting a response from the Picosiblings that does not contain the secret share. In his bachelor's thesis, Tian (2012) uses protocols that achieve the same outcome as those of Sections 3.3 and 3.4, but they require regeneration of a new asymmetric key pair on the Pico each time a new connection is established in order to preserve anonymity; this is very expensive on an embedded device and can be solved by employing simple session keys and symmetric cryptography, as shown in Section 3.4.

Chapter 4

System design and implementation

In this chapter, we describe our work on implementing a prototype of the Pico. Our contributions are as follows:

- We argue that a TPM and the Raspberry Pi are good choices for the Pico's hardware platform (Section 4.1).
- We describe an implementation of a prototype of the Pico of Stajano (2011) on this platform (Section 4.2).

4.1 System design

In this section, we discuss the reasons for the two major design choices of this implementation: using a TPM, and using it in conjunction with the Raspberry Pi.

4.1.1 Why use a TPM?

In Section 2.2, we arrived at three security-related requirements. In order to protect the keys used for communication with the Picosibling, the Pico needs a security chip that 1) supports trusted boot and 2) provides at least some degree of tamper resistance. To generate key material, the Pico needs 3) a cryptographically secure random-number generator.

A number of devices satisfy these requirements; examples are the SAMA5D3 from Atmel, the ST33F1M from STMicroelectronics or Atmel's CryptoAuthentication range of products. However, these are special-purpose hardware devices

that might be difficult to interface. On the other hand, Trusted Platform Modules (TPMs) that follow version 1.2 of the specification issued by the Trusted Computing Group[1] are all compatible, and they are manufactured by many different vendors (such as Atmel, Infineon and STMicroelectronics). As such, a kernel module is included with newer versions of Linux that works for all TIS-compliant TPMs, which significantly simplifies development. Furthermore, using a TPM ensures that we avoid vendor lock-in, which is a great advantage.

One concern is the amount of tamper resistance offered by commodity TPMs. Most TPMs come with modest tamper resistance in the form of protective shielding (Schellekens, 2012), and some even monitor environmental parameters to detect intrusion and fault injection.[2] Considering the budget of the Pico, this degree of tamper resistance seems to be the most realistic option available.[3]

TPMs are primarily designed to counter software-borne threats — they create an isolated environment for sensitive keys and operations that makes it seemingly impossible for malicious software to gain access to this material. Protecting against physical attacks, which requires tamper resistance, is only a secondary requirement. As such, by using a TPM in the Pico, we benefit from the software isolation (which could protect against bugs in the software, or malicious code that might be injected through a vulnerability in, for example, the Pico's network protocol implementation), *and* we get modest tamper resistance.

Finally, in order to comply with the TPM 1.2 specification, all TPMs come with a hardware random-number generator.

4.1.2 Why use the Raspberry Pi?

We choose a hardware platform on its ability to run Linux, since it makes many open-source libraries available (see Table 4.2). Basing the Pico on a stable operating system significantly reduces development time and likely makes for a more robust product with fewer errors.

Several embedded devices are capable of running Linux, but another factor is the availability of relevant hardware interfaces. The Raspberry Pi comes with USB, which is useful for connecting a commodity webcam, it provides an I2C

[1] Officially, the *TCG PC Client Specific TPM Interface Specification* (TIS), it replaced the TPM 1.1b specification (Challener et al., 2007, page 50).

[2] An example is the ST33TPM12 (`http://www.st.com/web/catalog/mmc/FM143/CL1814/SC1522/PF252379`).

[3] Tamper resistance is notoriously difficult to get right, even in highly specialised equipment (Anderson and Kuhn, 1996).

interface that can be used with many TPMs, and it has several general-purpose input/outputs for connecting radios. Furthermore, there is significant expertise at the Computer Laboratory for using the Raspberry Pi, as well as good connections to its manufacturer.

The Pico's RAM is integrated on the same chip as the CPU, which makes the cold-boot attack of Halderman et al. (2009) much more difficult. The attack, however, is still theoretically possible, since the RAM technology (SDRAM) is vulnerable to cold-boot attacks — so it is important not to expose keys in memory unduly.

The main purpose of the TPM is to store the secrets used by the Pico to identify itself to the Picosiblings. Using the TPM ensures that secrets are stored in a reasonably tamper-proof part of memory, and the TPM can guarantee that it only releases the secrets to a processor running a legitimate version of the Pico software using trusted boot technology. However, using trusted boot requires the presence of a special hardware component, the static root of trust for measurement (SRTM); this is a vendor-supplied program placed in read-only memory that is the first to execute, initiating the trusted boot process. Such a component is currently not present in the Raspberry Pi, and it can only be added by the manufacturer of the integrated system-on-chip containing, among other things, the CPU and the memory (Broadcom). If a Raspberry Pi-based Pico goes into production in co-operation with Broadcom, such a core could conceivably be added to the chip.

4.2 Implementation

In this section, we give an overview of the work carried out in implementing the Pico prototype (Figure 4.1). Unfortunately, space constraints make it infeasible to give a detailed description of the entire implementation, and we instead refer to Appendix A for a thorough look at the functions implemented, as well as the source code available at `http://www.cl.cam.ac.uk/~alcb2/pico/`.

4.2.1 High-level product description

The prototype is implemented as a command-line program that interacts with the user using the keyboard, and not using physical hardware buttons as originally proposed by Stajano (2011, Section 3). While this solution is clearly unsatisfactory from a usability perspective, it allows for efficient prototyping of the core Pico

Figure 4.1: The Pico hardware prototype

functions: communicating with the server and Picosiblings, acquiring QR-codes and storing credentials securely.

When the program is started, the Pico tries to unlock itself by seeing if enough Picosiblings are available, and if so, the user is asked to scan a QR-code using the camera. There are two types of QR-codes: one contains information for logging into an existing account, and the other contains information for creating a new account. The steps taken by the program depend on the type of QR-code.

4.2.1.1 Logging into an existing account

In the common case, the user already has an account with a given service, and he simply scans the QR-code presented to him at the front page. The primary information contained in such a QR-code is an identifier for the public key of the service, which allows the Pico to search its database to find the corresponding credentials for that account.

The process is illustrated in Figure 4.2. If such credentials exist, and the user only has one account with the given service, the master key is collected from the Picosiblings (Section 4.2.4.2 for details) and the protocol of Section 3.4 is executed. If more than one set of credentials exist, the user is first asked which one to use[4]; and if none exists, the user is presented with an error message. An attacker cannot use a stolen Pico to determine which accounts a user has by scanning QR-codes, as this requires the Picosiblings.

After the initial login protocol has been executed, the user remains logged in (continuous authentication), as detailed in Section 3.4. Because we have not implemented Bluetooth communication, we cannot use the approach taken by Tian (2012), where the user remains logged in for as long as the radio signal between the Pico and the host computer is strong; instead, to simulate the continuous authentication, the protocols run until the user explicitly cancels it.

4.2.1.2 Creating a new account

The second type of QR-code instructs the Pico to generate a set of credentials for a new account. When such a QR-code is scanned, the Pico first retrieves additional information about the service, including a user-friendly name, and then asks the user to confirm that he wishes to create a new account. Once the user confirms, the protocol of Section 3.3 is executed, and the new credentials are saved locally. Because of the nature of asymmetric cryptography, the key is not required to save the credentials securely, so the Picosiblings are not needed when an account is created; but as mentioned before, for security reasons, k Picosiblings are required before a QR-code can be scanned. Figure 4.3 illustrates the creation of a new account.

4.2.1.3 Resetting the Pico

Alternatively, the program may be invoked with the special argument `reset`. In this case, the user is asked to confirm whether he really wants to clear all the credentials. If that is the case, the whole configuration is erased, and the following components are generated:

- n new Picosiblings, including their encryption keys (see Section 5.2.2 about how these are simulated)
- A new TPM key to protect these encryption keys

[4]How to do this effectively is an unsolved problem; see also Section 5.3.

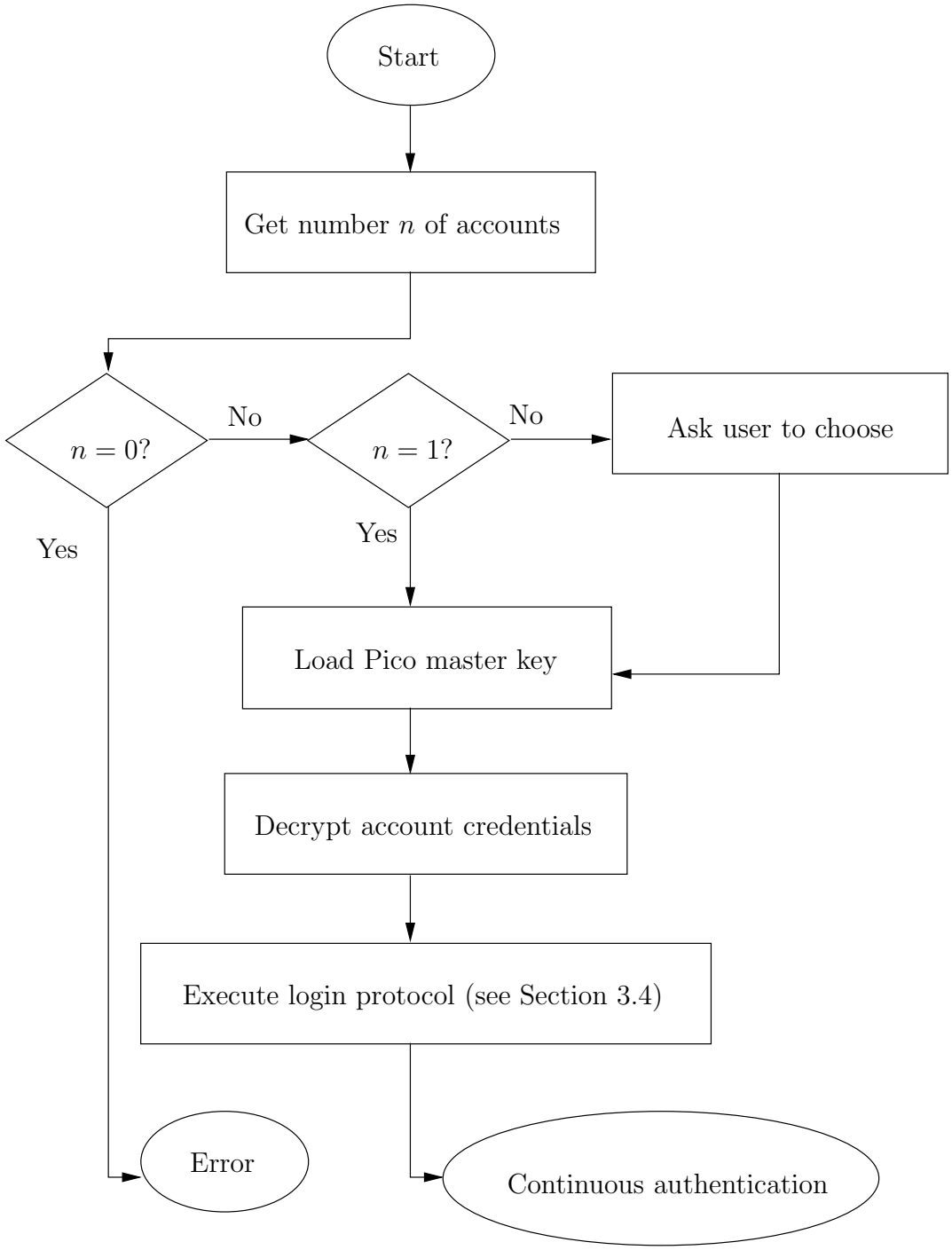


Figure 4.2: Logging into an existing account

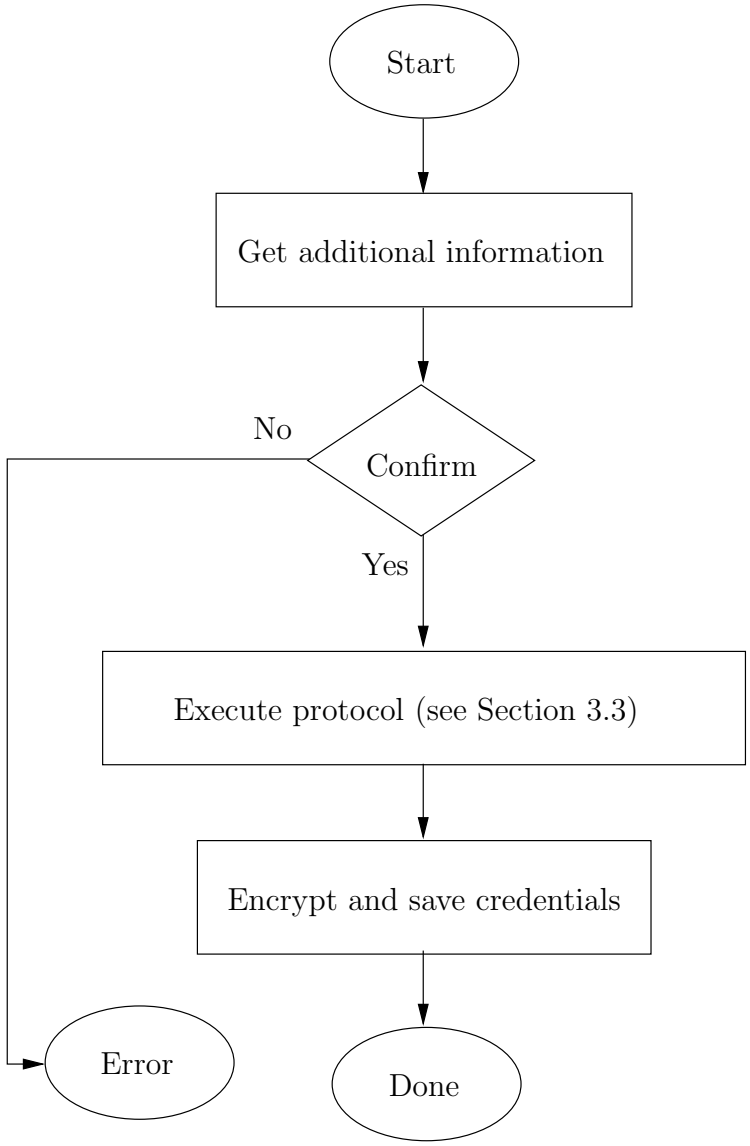


Figure 4.3: Creating a new account

- A random master secret (this secret is split between the Picosiblings)
- A new TPM key for encrypting credentials, encrypted using the master secret above
- A new random salt for hashing the service identifiers (see Section 4.2.1.4)

4.2.1.4 Local authentication database

The Pico stores credentials along with additional information for each account as files in a special subdirectory. Each file contains all information required for one account and is encrypted using the Pico's master key. Since services identify themselves using a hash of their public key in the login-type QR-codes, these files are named using this hash, in order to make it easy to select the correct file. However, for privacy reasons, if the Pico is ever stolen and disassembled, it should be difficult for an attacker to determine which services the owner has accounts with. For this reason, the service's identifier is concatenated with a salt, and the hash of this is used as the file name for each account. When a new login-type QR-code is scanned, the Pico thus needs to concatenate the identifier found in the QR-code with the same salt, hash these together, and see if a file with this name exists. Using a salt thus means that an attacker cannot pre-generate a large list of hashes and services (akin to a rainbow table), but has to do this once for each Pico he captures, which makes the attack much more difficult. A random salt is generated whenever the Pico is reset.

The alternative to this approach would be to encrypt these file names using the master key; this has the advantage of making it impossible (not just difficult) for an attacker to see the accounts on a Pico if he does not have access to the master key on the Picosiblings. The disadvantage is that the master key needs to be loaded earlier, before it has been established whether the account exists in the first place. Whether this trade-off is worth it is an open question.

4.2.2 System architecture

Figure 4.4 shows a high-level diagram of the device hardware (grey items have not been implemented; see Section 4.2.3). The system-on-chip (SoC) is the one supplied with the Raspberry Pi, which is a Broadcom BCM2835 that includes a 700 MHz ARMv11 CPU and 512 MB SDRAM. Non-volatile storage is provided by an SD card (4 GB), and the camera is a low-end USB webcam (Tar-

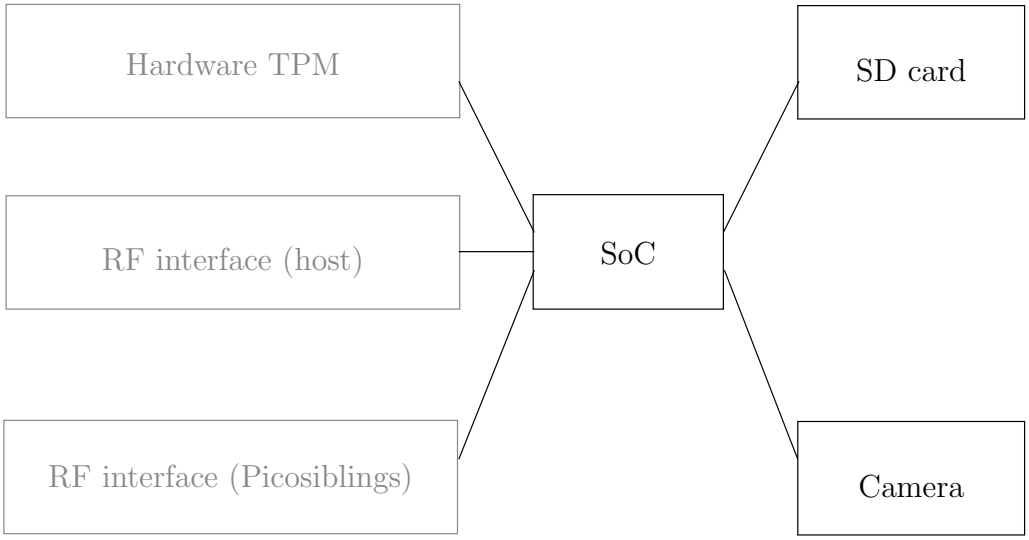


Figure 4.4: Diagram of the hardware components

gus AVC05EU). This camera is interfaced using standard Linux kernel modules through the OpenCV library.

The software is written as a command-line program running under the Raspbian distribution (based on Debian GNU/Linux 7.0) with version 3.6.11 of the Linux kernel. It is written in C and compiled with gcc 4.6.3.

Table 4.1 shows the categories of functionality implemented as well as a description of each; see Appendix A for a detailed reference of the functions implemented.

For a summary of the third-party dependencies of the project, please see Table 4.2. These libraries are all open source, but are licensed under varying terms. However, all licences grant the right to redistribute under certain conditions.

Many of these libraries contain much more functionality than is needed by the Pico. While the present version of the program has been linked against the entirety of the libraries (causing the executable to be larger than 4 MB), in a production version, the libraries could be pruned.

4.2.3 Limitations

Not all of the functionality envisioned by Stajano (2011) is implemented in this project's prototype. The most significant part that is omitted is the Picosiblings. Instead of simulating them using radios, they are simulated using the network interface (which makes it possible to implement and test the protocol of Section 3.2).

Communication with the service is also simulated using the network interface (Section 5.2 provides more details of this and the Picosibling emulation). In ad-

Table 4.1: Categories of functionality in the Pico project.

main unit	Contains the programs' entry point as well as high-level functions for loading the master key by collecting the secret shares from the Picosiblings, and for resetting the configuration.
Pico protocols	Implement the protocols of Chapter 3 for creating new accounts and signing into existing accounts.
Picosibling protocols	Provide high-level functions for communicating with the Picosiblings, primarily for collecting the secret shares.
Camera & QR-codes	Interface the relevant libraries for retrieving and decoding a QR-code from the attached camera.
Crypto operations & key management	Provide high-level functions for managing keys with the TPM and encrypting/decrypting data using asymmetric and symmetric algorithms.

Table 4.2: Third-party libraries used in the project.

IBM libtpm	Provides a low-level interface to any TPM compatible with version 1.2 of the TPM specification. See Challener et al. (2007, chapter 5). `http://ibmswtpm.sourceforge.net/`
OpenSSL	Provides implementations of common cryptographic primitives, including AES in various modes, RSA and hash functions. `http://www.openssl.org/`
GFShare	Provides an implementation of Shamir's secret sharing (Shamir, 1979). `http://www.digital-scurf.org/software/libgfshare`
OpenCV	The Open Source Computer Vision library provides a useful interface for many different webcams. `http://opencv.org/`
ZXing	Provides support for interpreting QR-codes. `http://code.google.com/p/zxing/`
curl	Provides functionality for downloading information through HTTP, which is used by the Pico to retrieve additional information when a new account is created. `http://curl.haxx.se/libcurl/`

dition, tunnelling the communication channel through the host computer to a remote service has not been considered. In the simplified form implemented, the Pico communicates directly with the service through the network interface. In this regard, the Pico can be seen to communicate with a service on the host computer (for example, to provide continuous authentication to the work station, and to lock it when the user leaves the office).

Furthermore, a hardware TPM is not connected to the Raspberry Pi, so a software emulator is used instead; see Section 5.2.3.

4.2.4 Implementation specifics

In this section, we highlight specific implementation points: the key hierarchy, how the Picosiblings are used, and the encryption routines.

4.2.4.1 Key hierarchy

The implementation uses the Advanced Encryption Standard (AES) with a key size of 256 bits for symmetric encryption, in cipher-block-chaining (CBC) mode. In general, a random initialisation vector is used (refer to Section 4.2.4.3 for more information). To maintain the integrity of the ciphertext, an encrypt-then-MAC scheme is used with an SHA-256-based HMAC. A symmetric key thus consists of two components: a 256-bit AES key, and a 256-bit HMAC key.

Asymmetric encryption is performed in the TPM using RSA with an exponent size of 2048 bits. The TPM specification mandates that a usage policy is enforced on TPM keys, so that the same key cannot be used for both signing and encryption (called 'binding'[5]). The padding scheme is PKCS #1 version 1.5.[6]

Please refer to Figure 4.5 for a depiction of the key hierarchy. Secret sharing is used to split a 512-bit secret into n shares distributed amongst the Picosiblings. When combined, these shares form a symmetric encryption key (of a 256-bit AES component and a 256-bit HMAC component, see above), referred to as the *master secret*. The database of credentials is encrypted with an RSA key (for binding) that is generated by the TPM, which we refer to as the (credentials) *master key*. This key is protected in two ways: first, being a TPM key, it is automatically

[5] Challener et al. (2007, chapter 3)

[6] The TPM standard also supports the newer, and more secure, Optimal Asymmetric Encryption Padding (OAEP), but PKCS #1 v. 1.5 was chosen to maintain compatibility with the test bench (in particular, PyCrypto — see Chapter 5). Changing the implementation to using OAEP is trivial.

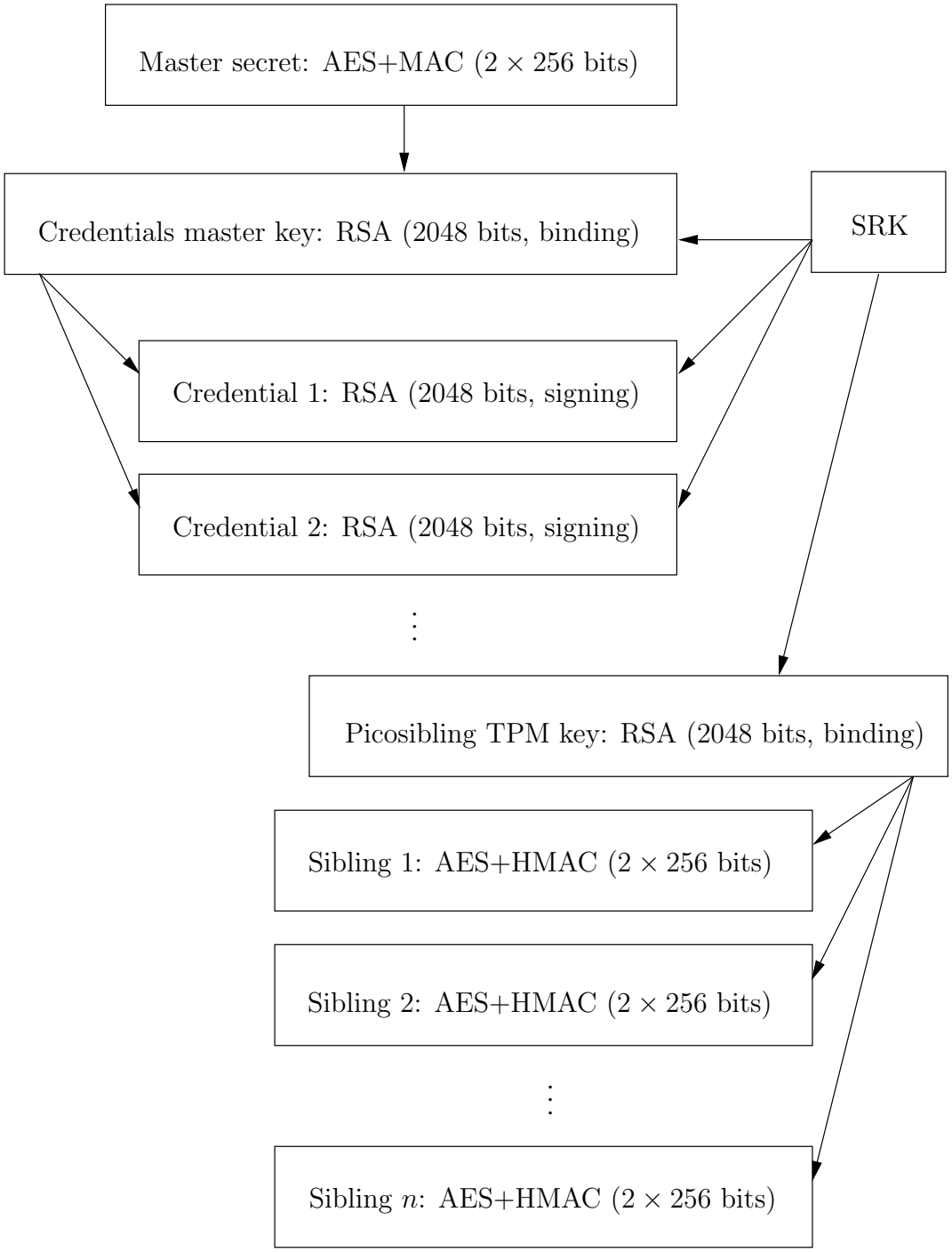


Figure 4.5: Key hierarchy (arrows denote encryption)

encrypted with the storage root key (SRK)[7]; furthermore, when stored in non-volatile memory, it is also encrypted using the master secret so that it can only be decrypted when enough Picosiblings are present. See Section 4.2.4.2 for the process of loading this key.

For each account that the user has with a service, a new TPM key (for signing) is generated. In the TPM key hierarchy, this is derived from the SRK, but when stored outside the TPM, it is further encrypted using the credentials master key.

A symmetric key is generated to preserve confidentiality and integrity for communication between the Pico and each Picosibling. These keys are encrypted using a Picosibling RSA key (policy: binding). Trusted boot (one of the main arguments for choosing a TPM) can ensure that this key is only released to a legitimate Pico OS; however, since no hardware TPM has been connected to the Raspberry Pi in this implementation, this feature is not used. We note that the TPM allows us to further secure not only this Picosibling RSA key, but *all* secret key material used by the Pico.

4.2.4.2 Picosiblings

The Picosiblings are used to split the credentials master key into shares, as described in Section 4.2.4.1. The procedure for loading the credentials for a particular account are as follows:

1. Load the Picosibling RSA key into the TPM
2. Using this key, decrypt the symmetric key used to communicate with each Picosibling present
3. For each of k Picosiblings, use this key to execute the protocol of Section 3.2 to obtain a share
4. Use the secret sharing algorithm to assemble these shares into the master secret, as described by Shamir (1979)
5. The master secret forms a symmetric key, which can be used to decrypt the credentials master key and load it into the TPM
6. This key can now be used to decrypt the key of a particular account, so that it can subsequently be loaded into the TPM

[7]This is unique to each TPM, which means that the key is only valid on that particular device.

In the C code, steps 1, 2 and 5 are performed by the function `load_pico_key()`, step 3 and 4 by `collect_sibling_secret()`, and step 6 by `login()`; please see Appendix A for details.

These functions take extra care to unload keys from the TPM as soon as they are not needed any more. That is, the credentials master key is unloaded immediately after it has been used to decrypt the relevant account key in step 6, and the account key, in turn, is unloaded as soon as it has been used to sign the nonce in the login protocol (step A3 in Section 3.4). Refer to Listing 4.1 for an example — the TPM key is unloaded in line 12, immediately after it has been used to sign the nonce in line 10.

Most of the implementation is agnostic with regard to the Picosibling emulation. To provide proper Picosiblings, only two functions (`retrieve_sibling_share` and `is_sibling_available` need to be reimplemented.

Listing 4.1: Unloading TPM keys (excerpt from `exec_login_protocol()`)

```
1  // Load the TPM key
2  ret = load_private_key(cd->account_private_key_blob,
3          cd->account_private_key_blob_len, TPM_SRK_HANDLE,
4          &priv_key);
5  if (ret != 0) {
6          fprintf(stderr,
7                  "Failed to load account private key\n");
8          return 1;               // indicates failure
9  }
10 ret = read_and_sign_nonce(sock, &session_keys, priv_key);
11 // Unload the TPM key
12 TPM_EvictKey(priv_key);
13 if (ret != 0)
14         return 1;
```

4.2.4.3 Encryption routines

The OpenSSL library, as well as IBM's libtpm, are used for low-level encryption operations. However, these libraries do not provide higher-level functionality, so this has been done as part of the Pico project. Examples of functionality implemented are: perform encrypt-then-MAC with an AES/HMAC key and write to a file descriptor (`write_aes_hmac()`); and encrypt a data buffer using an RSA key and an AES session key, and write the output to a file (`session_encrypt()`).

As an example, the function `write_aes_hmac()` writes a 'packet' to a file descriptor. Note that a file descriptor in UNIX can also be a socket, so this function

is used both when writing to files locally, and when communicating with a remote host (for the same reason, it is necessary to write the length of the packet being sent). It performs these steps:

1. Generate a random initialisation vector (IV, which is 128 bits, the block size of AES-256)
2. Encrypt the plaintext by calling a lower-level encryption routine, using this IV
3. Generate an HMAC of the total packet size, the IV and the ciphertext
4. Write out the total packet size, the IV, the ciphertext and the HMAC

Note that the above procedure ensures that the integrity of not only the ciphertext, but also the packet size and the IV, are covered by the HMAC. This protects against chosen-ciphertext attacks and ensures that the program cannot be used as a decryption oracle (Bellare and Namprempre, 2000; Katz and Lindell, 2008).

4.3 Summary

In the first part of this chapter, we considered hardware options for the system, arguing why the Raspberry Pi is a good choice, and we showed that using a TPM adds benefits beyond tamper resistance that we do not get from other tamper-resistance devices. In the second part, we described an implementation of the Pico, based on the concept given by Stajano (2011) and the requirements, design criteria and protocols considered earlier in this dissertation.

The implementation is written in well-structured code and placed in the public domain in the hope that it can be used in the future.

4.4 Related work

In his PhD dissertation, Schellekens (2012) gives a very thorough review of many different aspects of TPM security. Of interest to the Pico, he shows that TPMs are likely to offer some kind of tamper resistance, but they will be vulnerable to a determined attacker using state-of-the-art reverse engineering (Schellekens, 2012, pages 34f.). Moreover, Schellekens considers how to integrate a TPM directly on the chip, which, among other things, makes it much more difficult to intercept and modify communication between the TPM and the CPU.

SDRAM is vulnerable to so-called cold-boot attacks, because data remain in memory for a while even after power has been turned off (Halderman et al., 2009). This also affects mobile phones (Müller and Spreitzenbarth, 2012). In TRESOR, Müller et al. (2011) show how it is possible to defeat typical cold-boot attacks by storing sensitive keys only in the CPU registers. However, Blass and Robertson (2012) show ways in which this approach can be circumvented. Since the CPU and RAM are on the same chip in the Raspberry Pi, attacking one is presumably as difficult as attacking the other.

Tian (2012) implemented the Pico (without Picosiblings) as an Android app. One of his contributions is to provide continuous authentication in such a way that the user is logged out when the Bluetooth signal between the phone (the Pico) and the host computer weakens.

Chapter 5

Evaluation

In this chapter, we evaluate the Pico — first the concept in general, and then this specific implementation. We furthermore discuss outstanding issues and offer some ideas on how to solve some of these.

The contributions of this chapter are as follows:

- We take a more critical look at the Pico and discuss why it may or may not be successful (Section 5.1).
- We describe the test bench that was used to verify the functionality of the implementation of the previous chapter, and we provide representative outputs of the test runs (Section 5.2). Furthermore, we describe how the TPM-related functionality was tested using a software emulator as well as a hardware TPM.
- We discuss some unresolved questions that have surfaced over the duration of this project, and we take a closer look at a way to solve two of these: revocation, and replacing passwords in a broader sense, for example with reference to encryption passphrases (Section 5.3).

5.1 The concept of the Pico

The Pico was proposed by Stajano (2011) as a clean-slate solution, intended to answer the question, 'if we had the opportunity to redesign our authentication mechanisms, what would the solution be?' In this regard, it is very ambitious: it

seeks to replace *all* passwords,[1] not just for online accounts, and not just related to computers. One could conceive of the Pico employed for access-control mechanisms on doors, or perhaps for the ignition of vehicles. One of the flaws of the solutions discussed in Section 2.1 is that, even if they solve the problem of online authentication, their users will still need to remember passwords for such things as local computer accounts.

The Pico's ambition is one of its strong points, but at the same time, this means it is not backwards compatible. Adopting the Pico requires not only that users acquire the physical token with its affiliated Picosiblings, it also requires that users install software on their local computers, and may entail a major redesign of the authentication engine at the back-end.[2] Furthermore, the benefits of the Pico are not great until a significant part of the whole population is using it — this means that there are limited incentives for initial users. This is particularly problematic because users depend on services to support Pico authentication, but services are unlikely to provide this until a good portion of their user base has adopted the Pico.[3]

One objection to the Pico is to speculate that users really do not want to have to carry around *another* device with them, in addition to everything else that people normally have to carry (wallet, keys, phone and so on).[4] Even worse, what happens if this device is lost or stolen? To counter this objection, Stajano proposes a docking station for backups, and the Picosiblings to ensure that only the rightful owner can use the Pico. However, if we were doubting whether users are willing to carry around the Pico, does it then really make sense to require them *also* to always remember their Picosiblings?

These questions are important, but they cannot be answered with technology. They are related to usability, and a study needs to be conducted in order to determine how users behave when using the Pico for authentication. A related question is whether using QR-codes is a good idea, or if it is merely confusing and annoying to users. For an engineer, it is important to keep in mind that usability is really *the* most important design criterion — having a perfectly secure product

[1] With the exception of super-sensitive credentials for nuclear weapons, etc.

[2] To be fair, Stajano is explicit in mentioning these properties (No-app-changes and No-CLI-changes) as non-goals (Stajano, 2011, page 7); but this does not make the objection any less valid.

[3] Stajano recognises these problems and discusses 'optimisations' by which some of them might be addressed with a less-than-perfect Pico (Stajano, 2011, Section 5).

[4] For an argument of why we cannot simply implement the Pico on a smart phone, see e.g. (Laurie and Singer, 2009, Section 1). In short, a general-purpose operating system contains too many potential weaknesses to make it feasible to secure it.

is of no value if it is so cumbersome to use that nobody wants it (or even worse, if they circumvent its security features).

In the following sections of this chapter, we evaluate — from an engineering perspective — to which degree the prototype designed and implemented in this dissertation was successful; but as is pointed out above, the ultimate success of the Pico is probably not as much a question of technology as it is of usability and market adoption.

5.2 Test environment

As described in Chapter 4, the Pico software is written in C for the the Raspberry Pi. We chose to compile and run the software directly on the Raspberry Pi during the development process, for two reasons:

1. This ensures that the software always works on its intended platform (we do not have to worry about it working in an emulator, but not on the Raspberry Pi).

2. The Pico requires access to low-level hardware connected to this platform, such as a TPM (which was emulated, see Section 5.2.3) and a webcam.

Compiling and running the software on the Raspberry Pi also cuts out the cumbersome steps of configuring and verifying a cross-compiler. The disadvantage of this approach is that, due to the limited resources of the Raspberry Pi, it is infeasible to employ a large test bench, such as unit tests, during the development cycle.

In addition to the Raspberry Pi itself, the test environment consists of a development computer running Python test scripts (see Sections 5.2.1 and 5.2.2). The Raspberry Pi and the host computer use an Ethernet connection to communicate using TCP/IP. Source code editing is performed on the development computer, the files are transferred using SCP, and they are compiled and executed on the Raspberry Pi using an SSH connection.

The software is tested in two different ways: first, code changes are tested as they are committed, with liberal use of debug output; and second, special scripts and software are used to test three major components (communication with the Picosiblings, communication with the service, and usage of the TPM).

5.2.1 Remote service

The test script `dummy_service.py` is a Python script that imitates a remote service. It communicates with the Pico using Ethernet and executes the server side of the protocols of Sections 3.3 and 3.4, allowing the Pico to create new accounts and log into existing accounts. When run for the first time, the script automatically generates the relevant service keys and QR-codes for instructing the Pico to create a new account with that service as well as to log into an existing account with that service.

In a proper deployment, the QR-codes would be displayed by the service with which the Pico authenticates (such as a website). However, the current test scripts are console based and so do not have a user interface that can display QR-codes. The test scripts, therefore, generate QR-codes (using the Python `qrcode` library) and save them to local files for display using a standard image viewer.

5.2.1.1 Creating an account

When a new account is created, the Pico program generates some debug output, which is shown in Listing 5.1. Listing 5.2 shows the corresponding output from the Python test script.[5]

As shown in the listings, the following takes place:

1. The user points the camera at a QR-code that instructs the Pico to create a new account.

2. The Pico communicates with the Picosiblings to unlock itself (see Section 5.2.2 for details).

3. The Pico follows the URL in the QR-code to download additional information about the account.

4. The user is asked to confirm, which he does.

5. The Pico then generates a new RSA key pair for that account and carries out the account creation protocol of Section 3.3.

[5]The `VIDIOC_QUERYMENU: Invalid argument` warning is an unfortunate artefact caused by the OpenCV library, which writes out such messages to `stderr` without making it possible to suppress them. It does not indicate an error condition, as the camera works and reads an image with no problems. This is a known issue; see e.g. `http://www.ozbotz.org/opencv-install-troubleshooting/`.

6. Upon successful completion of the protocol, a new account is created and the credentials are saved locally.

Listing 5.1: Creating a new account: Pico debug output

```
Loading sibling database
Collecting secret now
Using Picosibling (001)
Trying to receive share 1
Counter:        1441772225
Received counter 1441772225
Saving sibling database
Press enter to scan a QR-code, or Ctrl+C to quit
VIDIOC_QUERYMENU: Invalid argument
Retrieving additional account data
Please confirm that you want to create an account, as follows:
 Host: PicoServer9996
 Name: 192.168.137.1:9996
 [yes]
yes
Generating keys for new account at 192.168.137.1:9996...
Connecting...
Sending public key
Receiving nonces...
Something received
Received correct amount of data
Nonces match, loading private key
Private key loaded
Hashed and signed
Received account ID 1682953330
Saving credentials
Saved in db/
    f8216b09ed6e1d40997ad60e80ef9ec4375645dbedec5321759ecfe03e0cc16b
    -000
Press enter to scan a QR-code, or Ctrl+C to quit
```

Listing 5.2: Creating a new account: test script output

```
Ready to accept requests
Request received
Sending data
Creating new account
Waiting to receive session key and nonce
Waiting to receive account public key
Key import successful
```

```
Received signature
Added account with ID 1682953330
Request done
```

5.2.1.2 Logging into an account

From the user's perspective, logging into an account is quite similar to creating a new account, except that the user is not asked to confirm the action. The QR-code contains an identifier for the service offering the login (a hash of its public key), as well as an interface authorisation nonce (see Section 3.5). The test bench uses a default value of all zeros for this, which the Pico correctly sends back.

Listings 5.3 and 5.4 show the output when logging into an account. Once the Pico has been unlocked using the Picosiblings (similar to what happens when a new account is created), the continuous authentication protocol of Section 3.4 is carried out. The user then remains logged into the account, and the client is continuously challenged by the service (as can be seen from the output of the test script) until a key combination is pressed to log out.

Listing 5.3: Logging into an account: Pico debug output

```
Loading sibling database
Collecting secret now
Using Picosibling (001)
Trying to receive share 1
Counter:        1441772228
Received counter 1441772228
Saving sibling database
Press enter to scan a QR-code, or Ctrl+C to quit
VIDIOC_QUERYMENU: Invalid argument
Loading sibling database
Collecting secret now
Using Picosibling (001)
Trying to receive share 1
Counter:        1441772229
Received counter 1441772229
Saving sibling database
Connecting...
Sending account ID 1682953330
Logged in (use Ctrl+C to interrupt)
^CPress enter to scan a QR-code, or Ctrl+C to quit
```

Listing 5.4: Logging into an account: test script output

```
Request received
Sending data
Logging in
Waiting to receive session key
Waiting to receive account ID
Client sent account ID 1682953330
Request verified, user logged in
Received interface ID 00000000000000000000000000000000
Asking for reauthentication
Reauthentication successful
Asking for reauthentication
Reauthentication successful
Asking for reauthentication
Nothing received, aborting
Received invalid response
Request done
```

5.2.2 Picosiblings

The test script `sibling_server.py` emulates Picosiblings by listening for connections on the network interface. Each instance of this script emulates one Picosibling on its own port and carries out the server side of the Picosiblings protocol of Section 3.2, including counter synchronisation if necessary. The k and n parameters, indicating the threshold and the total number of Picosiblings, are set at compile-time. In the test case documented here, $k = 1$ to save space in the output listings, and $n = 10$, which means that only one Picosibling needs to be queried (and it essentially hands over the entire secret); however, the program has also been tested to work with $k > 1$.

When the Pico software is called with the `reset` option (see Section 4.2.1.3), a new master secret is generated and distributed to n Picosiblings. At the same time, new communication keys are generated for these Picosiblings. In order to allow emulation of the Picosiblings, this information is written to files readable by the `sibling_server.py` test script, and these files are transferred to the development computer that runs the test script.

Listing 5.5 shows the output of the Picosibling test script that corresponds to the Pico debug output of Listing 5.3. When a new connection is opened, the script shows the current AES and HMAC keys, along with the current value of the counter. The script also shows how many hashes were necessary to resynchronise the counter.

The output shows that for the first connection, the Pico is not actually interested in the secret share — it merely wants to determine whether enough valid Picosiblings are within range. For the subsequent connection, where the Pico needs to decrypt the credentials, the share is requested and sent. Note also how the communication keys change because they are hashed each time a session is carried out.

Listing 5.5: Picosibling test script output

```
Request received
enc key:
   f106b1f42da71025c56bc19c84a76b1842b118f183f2d2dcca0148c3d4c6f1cc
mac key:
   b3afd8115487d9a190b7629cb3613bfb8c9a3b5423cf58754d42236548b42919
counter: 1441772228
waiting to read
reading done
Had to hash 0 times to synchronise
Received counter 1441772228
Share not requested
Request done
Request received
enc key:
   586dc77a995b7ea50080b1d5b38666b62a44cb3935a074b2aebd8fa03f43c6a2
mac key:
   7a688371d6a098b2a67d7acc9219654e767e0bd54379fd6cee0400859e08ee43
counter: 1441772229
waiting to read
reading done
Had to hash 0 times to synchronise
Received counter 1441772229
share:
   98ae0dc58ba01237046bd729f18f602d0b8c4d24593ae40be087ccb8258b6876
   db1b02905f6feb3be4141fcd06c22c238598a2821312ff43961f63edfca2b940
Request done
```

5.2.3 Trusted Platform Module

To test the Pico's ability to interface a Trusted Platform Module, the TPM Emulator project[6] is used to emulate a TPM in software. This installs a TPM-like

[6]Available at `http://tpm-emulator.berlios.de/`. See also Strasser and Sevnic (2004); Strasser and Stamer (2008).

character device in `/dev/tpm0`, which to the rest of the computer looks like a real TPM. The validity is verified by IBM's libtpm that communicates with this character device — in doing so, no errors are observed, which can be seen to indicate that both the TPM Emulator and libtpm adhere to the TPM 1.2 specification (or to the sceptic, that they both deviate from the standard in exactly the same way).

In order to verify further that the Pico can interface to a TPM, the Pico software was recompiled and run on a business-grade laptop (the Lenovo ThinkPad X230) that features a hardware TPM. After proper support for the TPM was added to the operating system (Ubuntu Linux 12.04), the program executed on this platform without errors using the laptop's hardware TPM.

We thus conclude that, 1) the software works on the Raspberry Pi with a TPM emulator, and 2) it works on a normal laptop with a hardware TPM. We take this as a strong indication that the software would work with a hardware TPM connected to the Raspberry Pi.

5.3 Design contributions

The Pico implementation discussed in this and the previous chapters provides the major features envisaged by Stajano (2011): an embedded platform that allows users to authenticate themselves continuously by scanning QR-codes, and that unlocks itself using threshold cryptography. A great deal of work on the project remains: the radio interfaces were emulated, so they have to be implemented in both hardware and software; a hardware TPM needs to be integrated into the Raspberry Pi; provisions need to be taken for backing up the credentials, such as the docking station suggested by Stajano (2011, Section 4.3); and the hardware platform needs to be refined (using physical buttons, connecting a battery, and so forth). Most importantly, as mentioned at the beginning of this chapter, usability studies need to be conducted so it can be determined whether the current solution satisfies users.

However, the contribution of this dissertation is not simply the Raspberry Pi code. By writing the code we have gained knowledge that allows us to revisit the whole architecture of the Pico with a critical eye, and thus another valuable contribution is this informed critique of the Pico and our suggestions for improvements based on this implementation and design experience. These suggestions can only meaningfully be made after trying to implement the original design – this is where one discovers the conceptual holes. Thus, the following questions can be asked:

- How is the presence-based continuous authentication supposed to work in conjunction with accounts that should remain open (such as e-mail) when the user briefly leaves the computer, taking his Pico with him?
- What is the best way, from a usability perspective, to let the user choose between different credentials for the same account?
- How can we counter the relay attack described in Section 3.5? TLS can protect an end-to-end session between the Pico and the verifier, but this does not help if the local computer is infected with malicious software. Furthermore, how should the authentication protocol of Section 3.4 be changed to let the Pico verify that it is interacting with a TLS page?
- How do we revoke the credentials stored on a Pico if it is lost?
- How do we use the Pico to store not only credentials, but also keys used for whole-disk encryption, e-mail encryption and so forth?

In the following sections, we take an in-depth look at the two last points.

5.3.1 Revocation

In case the Pico is lost or stolen, its credentials need to be revoked in two ways: the device itself should remain locked (even if all the Picosiblings are stolen with it), and the credentials on it should become stale. The first can be addressed by having the Pico ping a server once in a while (this is mentioned by Stajano (2011, Section 4.1)); the user can tell the server not to respond, thus eventually locking the Pico.

The second is somewhat more challenging, since each service provider needs to be notified that the particular credential is revoked, whilst maintaining the anonymity and unlinkability that is one of the Pico's goals. Consider associating each account with a special revocation URL at the service provider; when this URL is accessed, that particular credential is invalidated by the provider. When placed in the docking station, the Pico outputs all these URLs to the station. Note that these are sensitive (they disclose where the Pico owner has accounts, which is in itself sensitive information; and they can be abused in a denial-of-service attack), so they have to be protected somehow. If the Pico is ever lost, a single press of a button on the docking station invokes all these URLs, thus revoking all the credentials.

As for privacy, if all URLs are invoked concurrently, an omnipresent eavesdropper can easily correlate them and conclude where the owner holds accounts. This is true even if they go through anonymising services. If the owner holds two or more accounts with the same service provider, the provider can also draw some conclusions if these accounts are revoked at (almost) the same time. This problem is inherent to the use of revocation URLs, and a good solution — other than separating the revocations in time — has yet to be found.

To protect the revocation URLs, we can associate the docking station with a public key pair.[7] The private key is protected by a strong passphrase kept by the Pico owner somewhere safe (it need only be used in case of revocation and could even be a 128-bit hex string written on a slip of paper). The public key is just stored plainly in the docking station (and perhaps on the Pico). When placed in the docking station, the Pico encrypts the list of revocation URLs using this public key and saves them there, along with the backup of all the credentials.

5.3.2 Using the Pico for non-authentication keys

If the Pico stores non-authentication keys — such as for use with file or e-mail encryption — it must be able to make an informed decision as to whether it should reveal those keys to a host computer wanting to carry out a cryptographic operation. In the ideal case, it should be able to detect whether the host computer is in a trusted state, and only reveal the keys if that is the case. A trusted state might mean that the computer is running a recognised operating system image that is known not to be infected with malicious software. We thus want to arrive at a system that is significantly more secure than the status quo using passwords — first by replacing the password by a more secure key (specifically, no longer requiring the user to enter a password to decrypt his key), and second by making sure we only reveal that key when we are (fairly) certain it is safe to do so.

Trusted boot seems a good way to guarantee the legitimacy of the host computer, but it has a fundamental problem: operating systems come in so many flavours and configurations that it is impractical to consider each of these in order to obtain the image of a trusted variant. For this reason, we will first confine our problem to verifying the system until the boot loader; the technology for doing this is by and large already in place. Then, we will consider possible ways to establish a trusted path once the operating system is booted, which is a more interesting,

[7] Depending on the backup scheme the docking station may already have this, to be used for backing up the credentials.

and as of yet unsolved, problem.

Only being able to guarantee that the boot loader is legitimate might seem insignificant, but it *does* add value in a system protected with full-disk encryption. Consider such a system where the full-disk decryption key is stored on the Pico. When the system boots, the Pico supplies this key to the boot loader. Using trusted boot on the encrypted system ensures that the Pico only reveals the key to the correct boot loader. Therefore, in addition to the added convenience and security of the user's not having to enter a passphrase, we protect against attacks where a malicious boot loader is installed to intercept the key.

It is, however, also desirable to be able to use the Pico for general-purpose cryptographic keys, such as for file and e-mail encryption and online banking signatures.

In a very controlled environment, it is possible to extend the chain of trust approach to the operating system itself and all programs it executes. This is impractical in the general consumer case, since it would require that all programs a user might want to run and install on his computer be certified.[8]

Another approach is to utilise the measures originally associated with digital rights management (DRM). DRM provides a combination of secure hardware, in the form of a TPM and special processor-protected 'curtained memory'[9], and secure software, in which the operating system contains a security kernel which is isolated from the rest of the system and only available to DRM-compatible programs. Note that this TPM needs to reside in the host computer, not the Pico. The security kernel, and by extension all DRM programs, are verified in hardware using the TPM. It is conceivable to imagine a DRM-compliant file encryption program that communicates with the Pico (proving its integrity using a TPM-supplied certificate) and stores the secret key provided by the Pico in the curtained memory region, thus protecting it from the rest of the system. While this solution is imperfect, it may be the right compromise between not revealing Pico keys at all, and giving them out to unverified applications. An implementation of DRM as described above was attempted by Microsoft's *Next-Generation Secure Computing Base* (NGSCB)[10], but it was never included in Windows.

An alternative is a solution based on the principles of Flicker or TrustVisor by McCune et al. (2010, 2008). TrustVisor is a secure hypervisor that uses the security

[8] Certain mobile phones (such as iPhones and Nokia Symbian devices) actually *do* pose such a requirement, and thus the chain-of-trust approach might be applicable in these environments.

[9] Available in Intel's LaGrande/TXT (Grawrock, 2009) and Arm's TrustZone (`http://www.arm.com/products/processors/technologies/trustzone.php`).

[10] `http://www.microsoft.com/resources/ngscb/documents/ngscb_tcb.doc`

features of Intel and AMD processors to allow isolated execution of security-critical software on legacy operating systems. TrustVisor ultimately derives its security from the hardware TPM by using dynamic root of trust measurements[11], and it supports remote attestation, which is very useful in the context of the Pico.

5.4 Summary

In the beginning of this chapter, we took a broader look at the Pico in general, considering its weaker points and how they are likely to influence its ultimate success. Then, we went on to evaluate the implementation presented in Chapter 4 by introducing a thorough test bench that emulates a Pico-enabled service as well as the Picosiblings, and we described how this test bench was used to verify that the implementation performs the actions expected of it. Furthermore, we tested the Pico's compliance with both an emulated software TPM and a hardware TPM, concluding that the Pico works well with these, and is likely to work with a hardware TPM connected to the Raspberry Pi.

Our contribution from the last part of the chapter was to provide an overview of some issues related to the Pico that are still unsolved, as well as an outline of how to solve two of these issues. We hope that the value of this dissertation lies not only in the implementation described in the previous chapter — but also, on a more fundamental level, in the critical look at the Pico design which the research has helped us acquire.

5.5 Related work

The open question of how to use the Pico for non-authentication keys is stated in the original Pico paper (Stajano, 2011, page 14ff.), and Stajano also mentions the need (without outlining a solution) for secure revocation (Stajano, 2011, Section 4.6). The issue of conducting user studies on the impact of the Picosiblings is also mentioned (Stajano, 2011, Section 5.3).

One potential issue with the Pico is the changes it requires to the servers' back-ends. Mao et al. (2011) describe how traditional two-factor authentication can be supported by existing password-based systems by adding just one new authentication server to the legacy environment.

[11] A feature of the TPM 1.2 specification, whereby a special CPU instruction that creates a controlled and attested execution environment, can guarantee untampered execution of a secure loader at any time during normal execution (Challener et al., 2007, page 72).

Chapter 6

Conclusions

This dissertation is intended to provide a prototype of the Pico in order to verify whether the design proposed by Stajano (2011) is feasible. The prototype is conceptual in the sense that it does not provide all the features required of a production-ready model, but it does include the main requirements necessary to evaluate the Pico's feasibility. It is useful here briefly to summarise the work carried out prior to this dissertation: Stajano originally described the Pico design with the intended goal of replacing passwords, outlining the requirements that such a design must meet. This design was subsequently investigated further by Stannard and Stajano (2012), who designed and implemented a protocol for secret sharing using Picosiblings, and by Tian (2012), who implemented the Pico (without Picosiblings) as an Android app. The main contribution of this dissertation is to synthesise these efforts on a dedicated hardware platform — taking the step from having a Pico prototype on a commodity phone, to a prototype on custom hardware[1] — and in addition to investigate whether there are any benefits of using a TPM to unlock the initial secrets required for communication with the Picosiblings. We use this as an opportunity to review the general architecture of the Pico and to contribute with suggestions on how to improve it.

In Chapter 2, we discussed different approaches to replacing passwords at the system level. It is fairly recent that industry has begun moving in this direction, and in particular the FIDO alliance offers a concept that in many ways is similar to that of the Pico. There are, however, some significant differences, especially

[1]The Raspberry Pi is still a commodity platform, but the configuration with a webcam is custom, and its components, including the Broadcom system-on-chip, can conceivably be used in a production prototype.

in terms of the privacy and anonymity offered by the two systems. Based on this survey, we outlined some expectations we may reasonably put to an authentication device from a system architectural perspective, which influence how we choose and design the hardware implementation.

We looked at protocol design in Chapter 3. The Picosibling protocol of Stannard and Stajano (2012) was largely reused, but we further considered how to implement counter resynchronisation in practice as well as the value of not always requesting the Picosiblings' shares. We proposed a redesign of the Pico-server protocol of Tian (2012) that takes into account the speed and power constraints of the device (by not requiring unnecessary generation of asymmetric key pairs), as well as considering the need for linking an authentication session to one particular interface. Furthermore, we discussed employing TLS as a way to mitigate against the relay attack described by Stajano (2011, Sec. 3.2).

In Chapter 4, we discussed our implementation of a Pico authentication token on the Raspberry Pi. We first showed why using a Trusted Platform Module as a tamper-resistant security chip is a good design choice, and we argued that it is beneficial to use the Raspberry Pi as the underlying hardware platform. We then described our software implementation of the Pico, emphasising reuse of code from openly available libraries — and in turn making our own implementation available as open source in the hope that future designs may be based on it.

We performed an evaluation of the Pico concept in Chapter 5, focusing first on some fundamental issues with the clean-slate approach and emphasising the importance of usability studies. Furthermore, we mentioned some unresolved problems, and we outlined in some detail possible solutions to two of these (how to perform revocation securely, and how to use the Pico to replace non-authentication passwords, such as those used for cryptographic keys for email and file encryption). In the same chapter, we also described the test bench that was employed to test our implementation, and we showed how the implementation was tested with both a software-based TPM emulator on the Raspberry Pi, as well as with a hardware TPM in a laptop.

Replacing passwords is a worthy, but not easily achievable goal. In this dissertation, we address the technical problems and argue that Stajano's Pico is feasible. However, one may well consider this the 'easy' part. We have not considered usability, which is an integral part of a final technological product: how will users cope with having to carry the Pico, along with the Picosiblings, around all the time, and are they agreeable to scanning QR-codes when they want to authen-

ticate? Furthermore, for the Pico to be deployed successfully, a significant effort needs to be invested in the server-side software: if we want businesses to adopt the Pico, we'd better make it easy for them to do so in a secure and reliable way. This dissertation does not seek to answer these questions, but it is our hope that, by addressing the technical client-side issues, we advance the state of the art a little bit towards the end goal of a world without passwords.

Appendix A

Source code documentation

The contents of this appendix are automatically generated by DoxyGen[1] by using annotated comments in the source code. The code is openly available at `http://www.cl.cam.ac.uk/~alcb2/pico/` (licensed under a new BSD licence), along with a HTML version of the documentation in this appendix.

We hope that this will serve as a platform on which future developers can build.

Contents

[1] `http://www.doxygen.org/`

A.1 pico.c File Reference

Contains the primary Pico functionality, including high-level functions for collecting the master key, for pairing with the Picosiblings, and the program's main function.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <libgfshare.h>
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/rsa.h>
#include "../libtpm/lib/tpm.h"
#include "../libtpm/lib/tpmutil.h"
#include "../libtpm/lib/tpmfunc.h"
#include "pico.h"
```

Functions

- uint32_t load_pico_key (uint32_t ∗key_handle, BOOL really_load)
- uint32_t make_dummy_siblings (uint32_t sharecount, uint32_t threshold, const char ∗root_dir, sibling_data ∗sd, const key_blob ∗master_secret)
- uint32_t reset_rejoin (void)
- int main (int argc, char ∗argv[])

A.1.1 Detailed Description

Contains the primary Pico functionality, including high-level functions for collecting the master key, for pairing with the Picosiblings, and the program's main function.

A.1.2 Function Documentation

A.1.2.1 uint32_t load_pico_key (uint32_t ∗ *key_handle,* BOOL *really_load*)

Try to load the Pico's master credentials key by retrieving the secret shares stored by the Picosiblings. This function performs the following tasks:

1. Decrypt the file containing keys to communicate with the siblings, using a TPM key
2. Use these keys to communicate with the siblings and collect the Pico master key encryption key
3. Encrypt and save a file containing updated counters and keys for sibling communication
4. Use this KEK to decrypt the master key blob and load it into the TPM. Return the key handle

Parameters

key_handle	Pointer to a uint32_t that will receive the handle of the private key component in the TPM. The caller should unload this with TPM_EvictKey() as soon as it is not required any more.
really_load	If FALSE, no secret is actually collected, and no handle returned in key_handle. This is useful if one wants to check if the Picosiblings are still available, without needing their secrets. (The Picosiblings, in this case, will not reveal their secrets, and thus there is no — not even a remote — risk of leaking the master secret.) Set to TRUE for normal behaviour.

Returns

0 on success, non-zero on failure.

See Also

collect_sibling_secret()

A.1.2.2 int main (int *argc,* char * *argv[]*)

Program entry. If the argument "reset" is passed, reset_rejoin() is used to reconfigure the device. The user can repeatedly scan QR-codes, and the program acts on these by either creating new accounts, or logging into existing accounts.

See Also

reset_rejoin(), create_account(), login()

A.1.2.3 uint32_t make_dummy_siblings (uint32_t *sharecount,* uint32_t *threshold,* const char * *root_dir,* sibling_data * *sd,* const key_blob * *master_secret*)

Split a secret into a number of shares using Shamir's secret sharing and export the shares to a format readable by the Python simulation scripts. This function is used in conjunction with the test bench that simulates actual Picosiblings. A similar function is required when physical Picosiblings are used, but naturally its implementation will differ.

Parameters

sharecount	The total number of shares (Picosiblings) to create.
threshold	The (minimum) number of shares required to assemble the secret.
root_dir	A string containing the name of the directory in which to place the Picosibling data files for the test bench.
sd	A pointer to an array of sibling_data structures (size of sharecount) that contains information about the shares.
master_secret	The secret that should be split into shares.

Returns

0 on success, non-zero on failure.

A.1.2.4 uint32_t reset_rejoin (void)

Generate a new master key, use this to protect a prototype authentication database, and split the key into shares. This function performs the initial steps necessary to set up a working configuration. Three encryption keys are generated:

1. A TPM RSA key for protecting the Picosibling credentials used to retrieve the master secret (this is used to initiate the the trusted boot process and should ideally be sealed using the TPM's PCRs, which is not done in this implementation).
2. A TPM RSA key for protecting the account credentials.
3. A random symmetric key to protect the account TPM key (2). This is split into parts that are distributed to new Picosiblings and henceforth erased. Thus, the Picosiblings are required to assemble (3) in order to decrypt (2).

In addition to the above, a file containing random salt is also created. This salt is used to hash service identifiers, making it harder for an attacker to deduce the services at which a user has accounts, were he ever to obtain physical access to the Pico.

Returns

0 on success, non-zero on failure.

A.2 protocols.c File Reference

Contains functions implementing the protocols for communication between the Pico and the remote service.

```
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <poll.h>
#include <time.h>
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include "pico.h"
```

Functions

- uint32_t create_account (RSA ∗service_pub_key, const char ∗host, const char ∗fname)
- BOOL waitOk (int sock, const key_blob ∗keys, uint32_t num)
- BOOL sendOk (int sock, const key_blob ∗keys, uint32_t num)
- uint32_t exec_create_account_protocol (int sock, account_credential_data ∗cred_data)
- uint32_t login (const char ∗fname, const char ∗interface_id)
- void intHandler (int sig)
- uint32_t exec_login_protocol (int sock, const account_credential_data ∗cd, const RSA ∗serv_pub_key, const char ∗interface_id)

A.2.1 Detailed Description

Contains functions implementing the protocols for communication between the Pico and the remote service.

A.2.2 Function Documentation

A.2.2.1 uint32_t create_account (RSA ∗ *service_pub_key,* const char ∗ *host,* const char ∗ *fname*)

Perform local steps necessary for account creation, and execute the protocol with the service. This function performs the following steps:

1. Generate new key pair for the account
2. Execute the account creation protocol (using exec_create_account_protocol()), try to create the account with the service
3. If successful, save the keys and metadata locally

Parameters

service_pub_key	A pointer to an RSA structure representing the public RSA key component of the service.
host	Host string (currently <IP address>:<port number>) of the remote service.
fname	Name of the file in which the credentials and metadata should be saved.

Returns

0 on success, non-zero on failure.

See Also

exec_create_account_protocol(), login()

A.2.2.2 uint32_t exec_create_account_protocol (int *sock*, account_credential_data * *cred_data*)

Execute the protocol for creating a new account with the remote service.

Parameters

sock	A socket descriptor that is connected with the intended host.
cred_data	A pointer to an account_credential_data structure that holds the public key pair for the new account, and the public key of the service. This is augmented to contain relevant data received during execution of the protocol.

Returns

0 on success, non-zero on failure.

See Also

create_account()

A.2.2.3 uint32_t exec_login_protocol (int *sock*, const account_credential_data * *cd*, const RSA * *serv_pub_key*, const char * *interface_id*)

Execute the login protocol with the server. Once the initial protocol steps have been executed, the connection is kept alive, and the client continuously responds to challenges from the server to indicate its presence. The user can interrupt this by pressing Ctrl+C.

Parameters

sock	A socket descriptor that is connected with the intended host.
cd	A pointer to an account_credential_data structure containing keys and metadata for the account.
serv_pub_key	The service's public RSA key.
interface_id	32-character nonce identifying the interface that the client wishes to unlock.

Returns

0 on success, non-zero on failure.

See Also

login()

A.2.2.4 void intHandler (int *sig*)

Provide a signal handler to catch Ctrl+C while Pico is continuously authenticating. This is a call-back function that is invoked when the Ctrl+C signal is detected. It simply sets a volatile variable keepAuthenticating to FALSE to indicate that the signal was caught.

See Also

keepAuthenticating

A.2.2.5 uint32_t login (const char * *fname,* const char * *interface_id*)

Load credential data for an account given a file name for that account, and log in to the account. This function performs the following steps:

1. Get the master encryption key from the Picosiblings
2. Decrypt the file containing credentials and metadata for the account in question
3. Execute the login protocol using exec_login_protocol()

Parameters

fname	The name of the file from which the account data should be loaded.
interface_id	A 128-bit (32 character) nonce given by the server out of channel (e.g. in a QR-code). This identifies to the server which interface the Pico wants to unlock.

Returns

0 on success, non-zero on failure.

See Also

exec_login_protocol(), create_account()

A.2.2.6 BOOL sendOk (int *sock,* const key_blob * *keys,* uint32_t *num*)

Send an acknowledgement message to the remote host. Certain steps of the protocols mandate sending and receiving acknowledgement signals. These take the form of OK<sequence number>, where sequence number is incremented for each signal and serves to prevent replay attacks.

Parameters

sock	A socket descriptor on which to send the signal.
keys	A key_blob structure containing the encryption and HMAC keys used.
num	The sequence number to send. The remote host will disconnect if this does not match what it expects.

Return values

TRUE	if the acknowledgement signal was sent successfully.
FALSE	if something went wrong.

See Also

waitOk()

A.2.2.7 BOOL waitOk (int *sock,* const key_blob * *keys,* uint32_t *num*)

Wait for an acknowledgement message from the remote host. Certain steps of the protocols mandate sending and receiving acknowledgement signals. These take the form of OK<sequence number>, where sequence number is incremented for each signal and serves to prevent replay attacks.

Parameters

sock	A socket descriptor on which to listen for the signal.
keys	A key_blob structure containing the encryption and HMAC keys used.
num	The sequence number to expect. If the number actually received does not match, the function aborts.

Return values

TRUE	if the acknowledgement signal was received with the expected sequence number.
FALSE	if something went wrong.

See Also

sendOk()

A.3 siblings.c File Reference

Provides high-level functions for dealing with the Picosiblings.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/rsa.h>
#include <libgfshare.h>
#include "../libtpm/lib/tpm.h"
#include "../libtpm/lib/tpmutil.h"
#include "../libtpm/lib/tpmfunc.h"
#include "pico.h"
```

Functions

- sibling_data * get_sibling_data (uint32_t n, sibling_data *sd)
- uint32_t collect_sibling_secret (key_blob *secret, sibling_data *sd, BOOL collect_secret)
- uint32_t get_available_siblings (uint8_t *sharenrs)

A.3.1 Detailed Description

Provides high-level functions for dealing with the Picosiblings. These are generic in the sense that the interface does not depend on the underlying Picosibling implementation.

A.3.2 Function Documentation

A.3.2.1 uint32_t collect_sibling_secret (key_blob * *secret,* sibling_data * *sd,* BOOL *collect_secret*)

Try to get enough shares from the Picosiblings to assemble the master secret.

Parameters

secret	Memory buffer (long enough to store a key_blob structure) where the master secret, if collected, is placed.
sd	Pointer to first element in an array of sibling_data structures representing all the Picosiblings paired with the Pico.
collect_secret	If FALSE, the secret is never actually collected, and nothing is copied into secret, but the function verifies whether enough Picosiblings are present so that it could have been collected. Set to TRUE to collect the secret.

Returns

0 to indicate success (in which case the master secret is stored in secret if collect_secret is TRUE), non-zero to indicate that the secret could not be collected.

A.3.2.2 uint32_t get_available_siblings (uint8_t * *sharenrs*)

Fill out a sharenrs array, return success if at least SIBLING_THRESHOLD siblings are available. This array is used by libgfshare to indicate which shares to interact with. NB: This only confirms that they can be connected to; it doesn't exchange any data with them, and as such they could be illegitimate.

Parameters

sharenrs	Pointer to first element in an array of uint8_t. If a particular share is available (in the sense that a connection to it can be established), the number of that share is written to an element in the array. At most SIBLING_THRESHOLD elements are written.

Return values

0	Success: at least SIBLING_THRESHOLD Picosiblings are available.
1	Failure: not enough Picosiblings are available.

A.3.2.3 sibling_data* get_sibling_data (uint32_t *n,* sibling_data * *sd*)

Search through an array of sibling_data and return a pointer to the item with the given number.

Parameters

n	Total number of Picosiblings.
sd	Pointer to first element in an array of sibling_data structures.

Returns

A pointer to the corresponding sibling_data structure on success, NULL on failure.

A.4 siblings_sock.c File Reference

Defines interface for simulating Picosiblings through the network. The functions implemented here are generic, but their implementation is emulation-specific.

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <unistd.h>
#include <openssl/sha.h>
#include "pico.h"
```

Functions

- uint32_t sibling_connect (int *sock, uint32_t sib_num)
- void print_mem (const uint8_t *, uint32_t, const char *)
- uint32_t retrieve_sibling_share (uint32_t n, uint8_t *buffer, sibling_data *sd, BOOL request_share)
- uint32_t is_sibling_available (uint32_t n, BOOL *response)

A.4.1 Detailed Description

Defines interface for simulating Picosiblings through the network. The functions implemented here are generic, but their implementation is emulation-specific.

A.4.2 Function Documentation

A.4.2.1 uint32_t is_sibling_available (uint32_t *n*, BOOL * *response*)

Check if the n'th Picosibling is available.

Parameters

n	Number of Picosibling whose availability is to be checked.
response	Is set to TRUE if the Picosibling is available, FALSE otherwise.

Returns

0 on success, non-zero on failure.

A.4.2.2 void print_mem (const uint8_t * *data*, uint32_t *size*, const char * *msg*)

Print out the contents of a memory location in hex, with an informative message.

Parameters

data	Pointer to memory location that is to be printed.
size	Number of bytes to print.
msg	Label to print next to the memory.

A.4.2.3 uint32_t retrieve_sibling_share (uint32_t *n*, uint8_t * *buffer*, sibling_data * *sd*, BOOL *request_share*)

Get the secret share of the n'th Picosibling.

Parameters

n	Number of Picosibling whose share is to be retrieved.
buffer	Memory location that receives the share.
sd	Pointer to a sibling_data structure containing information (including keys) for communicating with the Picosiblings.
request_share	If TRUE, the share is actually requested. If FALSE, the main part of the protocol is carried out (to verify the presence of the Picosibling), but the share is not requested.

Returns

0 on success, non-zero on failure.

A.4.2.4 uint32_t sibling_connect (int * *sock*, uint32_t *sib_num*)

Try to create a connection to a Picosibling.

Parameters

sock	Pointer to an int that receives a socket descriptor if the connection is successful.
sib_num	Number of the Picosibling to which a connection is to be established.

Returns

0 on success, non-zero on failure.

A.5 qrcam.cpp File Reference

Provides a wrapper for the OpenCV (webcam) and ZXing (QR-codes) libraries, allowing them to be interfaced from C.

```
#include <opencv/highgui.h>
#include <opencv/cv.h>
#include <zxing/qrcode/QRCodeReader.h>
#include <zxing/common/HybridBinarizer.h>
#include <zxing/common/GreyscaleLuminanceSource.h>
#include <zxing/DecodeHints.h>
#include <zxing/Exception.h>
#include <iostream>
#include <stdio.h>
#include <unistd.h>
```

Functions

- int decode_QR_from_cam (unsigned char *data, unsigned int *len, unsigned int num_tries)
- int qr_decode (IplImage *frame, unsigned char *data, unsigned int *len)

A.5.1 Detailed Description

Provides a wrapper for the OpenCV (webcam) and ZXing (QR-codes) libraries, allowing them to be interfaced from C.

A.5.2 Function Documentation

A.5.2.1 int decode_QR_from_cam (unsigned char * *data,* unsigned int * *len,* unsigned int *num_tries*)

Read an image from a webcam and try to decode it as a QR-code, returning the decoded data. The first webcam found on the system is used. This function is callable by C code.

Parameters

data	Buffer to receive QR-code contents.
len	Pointer to an int that contains, at call time, the length of the data buffer. This is set to the length of the QR-code data returned in data. If the buffer is not long enough, the data is truncated, and len is set to the size required.
num_tries	The number of times the function should try to decode the webcam stream before giving up.

Returns

0 on success, non-zero on failure.

See Also

qr_decode()

A.5.2.2 int qr_decode (IplImage * *frame,* unsigned char * *data,* unsigned int * *len*)

Decode a QR-code from an image.

Parameters

frame	A handle to an OpenCV image object (of type Intel Image Processing Library).
data	Buffer to receive QR-code contents.
len	Pointer to an int that contains, at call time, the length of the data buffer. This is set to the length of the QR-code data returned in data. If the buffer is not long enough, the data is truncated, and len is set to the size required.

Return values

0	Success.
1	Failure.

See Also

decode_QR_from_cam()

A.6 key_management.c File Reference

Contains functions for creating, storing and loading RSA public and private keys in conjunction with a TPM.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/rsa.h>
#include "../libtpm/lib/tpm.h"
#include "../libtpm/lib/tpmutil.h"
#include "../libtpm/lib/tpmfunc.h"
#include "pico.h"
```

Functions

- void _prepare_keydata (keydata ∗k)
- void prepare_key_storage (keydata ∗k)
- void prepare_key_binding (keydata ∗k)
- void prepare_key_signing (keydata ∗k)
- uint32_t create_tpm_key (const keydata ∗k, uint32_t parent_handle, uint8_t ∗priv_key_buffer, uint32_t ∗priv_key_len, RSA ∗∗pub_key)
- uint32_t create_tpm_key_file (const keydata ∗k, uint32_t parent_handle, const char ∗pub_key_file, const char ∗priv_key_file, const key_blob ∗wrap_key)
- uint32_t save_public_key (const RSA ∗rsa, const char ∗file_name)
- uint32_t load_public_key_file (const char ∗key_file, RSA ∗∗rsa)
- uint32_t load_private_key_file (const char ∗key_file, uint32_t parent_handle, uint32_t ∗key_handle)
- uint32_t load_wrapped_private_key_file (const char ∗key_file, uint32_t parent_handle, const key_blob ∗wrap_key, uint32_t ∗key_handle)
- uint32_t load_private_key (const uint8_t ∗blob, uint32_t blob_len, uint32_t parent_handle, uint32_t ∗key_handle)

A.6.1 Detailed Description

Contains functions for creating, storing and loading RSA public and private keys in conjunction with a TPM.

A.6.2 Function Documentation

A.6.2.1 void _prepare_keydata (keydata ∗ *k*)

Prepare a key data structure for default operation (no authorisation data etc.); do not fill in the type (signing/binding/storage) field. This function is internally used by the other functions that prepare key data.

Parameters

k	Memory location large enough to hold a keydata structure.

See Also

prepare_key_storage(), prepare_key_binding(), prepare_key_signing()

A.6.2.2 uint32_t create_tpm_key (const keydata * *k*, uint32_t *parent_handle*, uint8_t * *priv_key_buffer*, uint32_t * *priv_key_len*, RSA ** *pub_key*)

Create an RSA key pair and provide the caller with a blob of the private (wrapped) key and a handle to the public key.

Parameters

k	Contains information about the key type (e.g. signing or binding) and can be filled out by prepare_key_storage(), prepare_key_binding() or prepare_key_signing().
parent_handle	Handle of the parent TPM key, which must be loaded into the TPM (use TPM_SRK_HANDLE for the storage root key).
priv_key_buffer	Receives the wrapped private key.
priv_key_len	Specifies, at call time, the length of the buffer. The function fails if this is less than MAX_KEY_BLOB_LEN. The function updates this to contain the actual length of the blob copied into priv_key_buffer.
pub_key	Is set to the address of the RSA structure that provides a handle for the corresponding public key. This must be free'd by the caller using RSA_free().

Returns

0 on success, non-zero on failure.

See Also

prepare_key_storage(), prepare_key_binding(), prepare_key_signing(), create_tpm_key_file()

A.6.2.3 uint32_t create_tpm_key_file (const keydata * *k*, uint32_t *parent_handle*, const char * *pub_key_file*, const char * *priv_key_file*, const key_blob * *wrap_key*)

This is similar to create_tpm_key(), but instead of providing the caller with in-memory representations of the key pair, it saves them to two files (one for each of the public and private keys). The caller may optionally specify a key for encrypting the private key blob before it is saved. Note that the private key blob is already encrypted (wrapped) by the TPM, so specifying a key adds an additional layer of encryption.

Parameters

k	Contains information about the key type (e.g. signing or binding) and can be filled out by prepare_key_storage(), prepare_key_binding() or prepare_key_signing().
parent_handle	Handle of the parent TPM key, which must be loaded into the TPM (use TPM_SRK_HANDLE for the storage root key).
pub_key_file	Name of the file in which the public key should be saved. This is done by save_public_key(), in PEM format.
priv_key_file	Name of the file in which the private key blob should be saved.
wrap_key	May be NULL or point to a key_blob structure. If non-NULL, the private key is encrypted using the AES key component, and a HMAC of the cipher text is added using the MAC key component.

Returns

0 on success, non-zero on failure.

See Also

prepare_key_storage(), prepare_key_binding(), prepare_key_signing(), create_tpm_key(), save_public_key()

A.6.2.4 uint32_t load_private_key (const uint8_t * *blob*, uint32_t *blob_len*, uint32_t *parent_handle*, uint32_t * *key_handle*)

Load a private key from a memory blob into the TPM. This can later be unloaded using TPM_EvictKey().

Parameters

blob	Memory location of the private key blob.
blob_len	Length (in bytes) of the private key blob.
parent_handle	Handle of parent key (use TPM_SRK_HANDLE for the storage root key).
key_handle	Pointer to memory location to which the new TPM key handle is saved.

Returns

0 on success, non-zero on failure.

See Also

TPM_EvictKey(), create_tpm_key_file(), load_private_key_file(), load_wrapped_private_key_file()

A.6.2.5 uint32_t load_private_key_file (const char * *key_file*, uint32_t *parent_handle*, uint32_t * *key_handle*)

Load a private key from a file into the TPM, returning a handle to the TPM key. This can later be unloaded using TPM_EvictKey().

Parameters

key_file	Name of file from which to read the private key blob (created by e.g. create_tpm_key_file()).
parent_handle	Handle of parent key (use TPM_SRK_HANDLE for the storage root key).
key_handle	Pointer to memory location to which the new TPM key handle is saved.

Returns

0 on success, non-zero on failure.

See Also

TPM_EvictKey(), create_tpm_key_file(), load_wrapped_private_key_file(), load_private_key()

A.6.2.6 uint32_t load_public_key_file (const char * *key_file*, RSA ** *rsa*)

Load a public key in PEM format from a file and return it as an OpenSSL RSA structure.

Parameters

key_file	Name of file from which to read the public key (PEM format).
rsa	Address of pointer that will be updated to point to an RSA structure representing the public key. Must be free'd by the caller using RSA_free().

Returns

0 on success, non-zero on failure.

See Also

save_public_key(), load_private_key_file()

A.6.2.7 uint32_t load_wrapped_private_key_file (const char * *key_file,* uint32_t *parent_handle,* const key_blob * *wrap_key,* uint32_t * *key_handle*)

Load a private key into the TPM from a file that has been encrypted with a wrap key by create_tpm_key_file(). This can later be unloaded using TPM_EvictKey().

Parameters

key_file	Name of file from which to read the private key blob (created by e.g. create_tpm_key_file()).
parent_handle	Handle of parent key (use TPM_SRK_HANDLE for the storage root key).
wrap_key	Pointer to a key_blob structure containing the AES and HMAC keys to verify and decrypt the file.
key_handle	Pointer to memory location to which the new TPM key handle is saved.

Returns

0 on success, non-zero on failure.

See Also

TPM_EvictKey(), create_tpm_key_file(), load_private_key_file(), load_private_key()

A.6.2.8 void prepare_key_binding (keydata * *k*)

Prepare a keydata structure to specify a TPM key pair used for binding (encryption). This is used by create_tpm_key() when a new key pair is generated by the TPM.

Parameters

k	Memory location large enough to hold a keydata structure.

See Also

create_tpm_key(), create_tpm_key_file(), prepare_key_storage(), prepare_key_signing()

A.6.2.9 void prepare_key_signing (keydata * *k*)

Prepare a keydata structure to specify a TPM key pair used for signing. This is used by create_tpm_key() when a new key pair is generated by the TPM.

Parameters

k	Memory location large enough to hold a keydata structure.

See Also

create_tpm_key(), create_tpm_key_file(), prepare_key_binding(), prepare_key_storage()

A.6.2.10 void prepare_key_storage (keydata * *k*)

Prepare a keydata structure to specify a TPM key pair used for storage (key encryption). This is used by create_tpm_key() when a new key pair is generated by the TPM.

Parameters

k	Memory location large enough to hold a keydata structure.

See Also

create_tpm_key(), create_tpm_key_file(), prepare_key_binding(), prepare_key_signing()

A.6.2.11 uint32_t save_public_key (const RSA * *rsa,* const char * *file_name*)

Convert an OpenSSL RSA structure to a PEM string and save it to a file.

Parameters

rsa	A pointer to an OpenSSL RSA structure of a public key.
file_name	Name of the file in which the PEM data should be written.

Returns

0 on success, non-zero on failure.

See Also

load_public_key_file()

A.7 crypto_primitives.c File Reference

Simple functions for performing encryption using a TPM and OpenSSL.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/rsa.h>
#include <openssl/hmac.h>
#include "../libtpm/lib/tpm.h"
#include "../libtpm/lib/tpmutil.h"
#include "../libtpm/lib/tpmfunc.h"
#include "pico.h"
```

Functions

- void get_random (uint8_t ∗buffer, uint32_t len)
- uint8_t ∗ aes_encrypt (const uint8_t ∗data, uint32_t ∗len, const uint8_t ∗key, const uint8_t ∗iv)
- uint8_t ∗ aes_decrypt (const uint8_t ∗data, uint32_t ∗len, const uint8_t ∗key, const uint8_t ∗iv)
- uint32_t aes_save_encrypt_file (const char ∗file_name, const uint8_t ∗data, uint32_t data_size, const key_blob ∗keys)
- uint32_t aes_load_decrypt_file (const char ∗file_name, uint8_t ∗∗data, uint32_t ∗data_size, const key_blob ∗keys)
- uint32_t rsa_encrypt (const uint8_t ∗data, uint32_t data_len, uint8_t ∗∗buffer, uint32_t ∗buf_len, const RSA ∗pub_key)
- uint32_t rsa_save_encrypt_file (const char ∗file_name, const uint8_t ∗data, uint32_t data_size, const char ∗pub_key_file)
- uint32_t rsa_decrypt (const uint8_t ∗data, uint32_t data_len, uint8_t ∗∗buffer, uint32_t ∗buf_len, uint32_t key_handle)
- uint32_t rsa_load_decrypt_file (const char ∗file_name, uint8_t ∗∗data, uint32_t ∗data_size, uint32_t key_handle)
- uint32_t session_encrypt (const uint8_t ∗data, uint32_t data_size, const char ∗file_name, const char ∗pub_key_file)
- uint32_t session_decrypt (uint8_t ∗∗data, uint32_t ∗data_size, const char ∗file_name, const char ∗priv_key_file, uint32_t parent_handle)
- uint32_t session_decrypt_handle (uint8_t ∗∗data, uint32_t ∗data_size, const char ∗file_name, uint32_t priv_key_handle)
- uint32_t write_rsa (int fd, const RSA ∗key, const uint8_t ∗buffer, uint32_t len)
- uint32_t read_rsa (int fd, uint32_t key_handle, uint8_t ∗buffer, uint32_t len, uint32_t buf_max_len)
- uint32_t write_aes_hmac (int fd, const uint8_t ∗data, uint32_t data_len, const key_blob ∗keys)
- uint32_t read_aes_hmac (int fd, uint8_t ∗buffer, uint32_t len, const key_blob ∗keys)
- void pico_sha256 (const uint8_t ∗data, uint32_t data_len, uint8_t ∗digest)

A.7.1 Detailed Description

Simple functions for performing encryption using a TPM and OpenSSL. Anders Bentzon, April 2013. alcb2@cam.ac.uk

A.7.2 Function Documentation

A.7.2.1 uint8_t* aes_decrypt (const uint8_t * *data*, uint32_t * *len*, const uint8_t * *key*, const uint8_t * *iv*)

Perform AES-256-CBC decryption with optional IV.

Parameters

data	Buffer containing cipher text to decrypt.
len	Length of the cipher text buffer.
key	The 32-byte key with which to perform the operation.
iv	If non-NULL, a pointer to a memory location containing a 16-byte initialisation vector.

Returns

On success, a pointer to a buffer containing the plaintext, which the caller must free. NULL on failure.

See Also

aes_encrypt()

A.7.2.2 uint8_t* aes_encrypt (const uint8_t * *data*, uint32_t * *len*, const uint8_t * *key*, const uint8_t * *iv*)

Perform AES-256-CBC encryption with optional IV.

Parameters

data	Buffer containing plaintext to encrypt.
len	Length of the plaintext buffer.
key	The 32-byte key with which to perform the operation.
iv	If non-NULL, a pointer to a memory location containing a 16-byte initialisation vector.

Returns

On success, a pointer to a buffer containing the cipher text, which the caller must free. NULL on failure.

See Also

aes_decrypt()

A.7.2.3 uint32_t aes_load_decrypt_file (const char * *file_name*, uint8_t ** *data*, uint32_t * *data_size*, const key_blob * *keys*)

Read a file containing AES cipher text and an HMAC, verify the HMAC and, if successful, decrypt the data into a buffer.

Parameters

file_name	Name of the file to read.
data	Buffer that will receive the plaintext.
data_size	Length (in bytes) of the plaintext buffer. This is updated to contain the actual plaintext length.
keys	Pointer to a key_blob structure containing the AES and HMAC keys.

Returns

0 on success, non-zero on failure.

See Also

aes_save_encrypt_file(), read_aes_hmac()

A.7.2.4 uint32_t aes_save_encrypt_file (const char * *file_name,* const uint8_t * *data,* uint32_t *data_size,* const key_blob * *keys*)

Encrypt a buffer using AES and attach an HMAC of the cipher text, saving the result to a file.

Parameters

file_name	Name of the file to create.
data	Buffer containing plaintext to encrypt.
data_size	Length (in bytes) of plaintext buffer.
keys	Pointer to a key_blob structure containing the AES and HMAC keys.

Returns

0 on success, non-zero on failure.

See Also

aes_load_decrypt_file(), write_aes_hmac()

A.7.2.5 void get_random (uint8_t * *buffer,* uint32_t *len*)

Acquire random data from the TPM. Exits the program on error.

Parameters

buffer	Memory location to receive the random data.
len	Number of random bytes to retrieve.

A.7.2.6 void pico_sha256 (const uint8_t * *data,* uint32_t *data_len,* uint8_t * *digest*)

Use SHA-256 to obtain a message digest. The digest buffer is expected to accommodate SHA256_BLOCK_BYTES (32 bytes).

Parameters

data	The date to hash.
data_len	Length (in bytes) of the data buffer.
digest	Address of a buffer into which the digest is written. This must be at least 32 bytes long.

A.7.2.7 uint32_t read_aes_hmac (int *fd*, uint8_t * *buffer*, uint32_t *len*, const key_blob * *keys*)

Read a packet encrypted using write_aes_hmac() from a file descriptor. The HMAC is verified first, and decryption is only attempted if the cipher text verifies correctly.

Parameters

fd	File descriptor from which the packet is read.
buffer	Buffer in which the resultant plaintext is placed.
len	Length of the buffer. If this is not long enough, the plaintext is truncated.
keys	Pointer to a key_blob structure containing the AES and HMAC keys to use.

Returns

On success, the number of bytes placed in buffer. 0 on failure.

See Also

write_aes_hmac()

A.7.2.8 uint32_t read_rsa (int *fd*, uint32_t *key_handle*, uint8_t * *buffer*, uint32_t *len*, uint32_t *buf_max_len*)

Read cipher text from a file descriptor and decrypt using RSA.

Parameters

fd	File descriptor from which cipher text is read.
key_handle	A handle to a TPM key that is used for decryption.
buffer	Buffer in which to place the resultant plaintext.
len	Number of cipher text bytes to read from the file descriptor.
buf_max_len	The length of the buffer. No more than buf_max_len bytes is placed there.

Returns

On success, the number of bytes copied into the buffer. If this is buf_max_len, this indicates that the buffer was not large enough and data was lost. 0 on outright failure.

See Also

read_rsa()

A.7.2.9 uint32_t rsa_decrypt (const uint8_t * *data*, uint32_t *data_len*, uint8_t ** *buffer*, uint32_t * *buf_len*, uint32_t *key_handle*)

Perform a decryption operation using RSA/PKCSv15.

Parameters

data	Buffer of cipher text to decrypt.
data_len	Length (in bytes) of cipher text buffer.
buffer	Address of pointer that will receive the address of the plaintext. This must be free'd by the caller.
buf_len	Pointer to an int that receives the length of the plaintext.
key_handle	Handle of the private key to use. This must already have been loaded into the TPM.

Returns

0 on success, non-zero on failure.

See Also

rsa_encrypt(), load_private_key()

A.7.2.10 uint32_t rsa_encrypt (const uint8_t * *data*, uint32_t *data_len*, uint8_t ** *buffer*, uint32_t * *buf_len*, const RSA * *pub_key*)

Perform an encryption operation using RSA/PKCSv15.

Parameters

data	Buffer of plaintext to encrypt.
data_len	Length (in bytes) of plaintext buffer. Cannot be larger than the key modulus (normally 256 bytes).
buffer	Address of pointer that will receive the address of the cipher text. This must be free'd by the caller.
buf_len	Pointer to an int that receives the length of the cipher text.
pub_key	Pointer to an OpenSSL RSA structure containing the public key to use.

Returns

0 on success, non-zero on failure.

See Also

rsa_decrypt()

A.7.2.11 uint32_t rsa_load_decrypt_file (const char * *file_name*, uint8_t ** *data*, uint32_t * *data_size*, uint32_t *key_handle*)

Perform an decryption operation using RSA/PKCSv15, loading the cipher text from a file.

Parameters

file_name	Name of file containing cipher text.
data	Address of pointer that will receive the address of the plaintext. This must be free'd by the caller.
data_size	Pointer to an int that receives the length of the plaintext.
key_handle	Handle of the private key to use. This must already have been loaded into the TPM.

Returns

0 on success, non-zero on failure.

See Also

rsa_save_encrypt_file(), load_private_key()

A.7.2.12 uint32_t rsa_save_encrypt_file (const char * *file_name,* const uint8_t * *data,* uint32_t *data_size,* const char * *pub_key_file*)

Perform an encryption operation using RSA/PKCSv15, saving the cipher text to a file.

Parameters

file_name	Name of file where cipher text is written.
data	Buffer of plaintext to encrypt.
data_size	Length (in bytes) of plaintext buffer. Cannot be larger than the key modulus (normally 256 bytes).
pub_key_file	Name of file containing the public key in PEM format.

Returns

0 on success, non-zero on failure.

See Also

rsa_load_decrypt_file()

A.7.2.13 uint32_t session_decrypt (uint8_t ** *data,* uint32_t * *data_size,* const char * *file_name,* const char * *priv_key_file,* uint32_t *parent_handle*)

Decrypt a file generated by session_encrypt(), loading the private key from a file.

Parameters

data	Address of pointer that will be set to the plaintext buffer. This must be free'd by the caller.
data_size	Receives the length (in bytes) of the plaintext buffer.
file_name	Name of file containing the cipher text.
priv_key_file	Name of file containing the private key to use.
parent_handle	Handle of the parent key which must already be resident in the TPM (use TPM_SRK_HANDLE for the storage root key).

Returns

0 on success, non-zero on failure.

See Also

session_encrypt(), load_private_key_file()

A.7.2.14 uint32_t session_decrypt_handle (uint8_t ** *data*, uint32_t * *data_size*, const char * *file_name*, uint32_t *priv_key_handle*)

Decrypt a file generated by session_encrypt(), using a private key already in the TPM.

Parameters

data	Address of pointer that will be set to the plaintext buffer. This must be free'd by the caller.
data_size	Receives the length (in bytes) of the plaintext buffer.
file_name	Name of file containing the cipher text.
priv_key_handle	Handle of private key to use.

Returns

0 on success, non-zero on failure.

See Also

session_encrypt(), session_decrypt()

A.7.2.15 uint32_t session_encrypt (const uint8_t * *data*, uint32_t *data_size*, const char * *file_name*, const char * *pub_key_file*)

Encrypt arbitrary-length data using AES/RSA and a random session key.

Parameters

data	Buffer containing the plaintext.
data_size	Length (in bytes) of the plaintext buffer.
file_name	Name of file where cipher text is saved.
pub_key_file	Name of file containing the public key to use.

Returns

0 on success, non-zero on failure.

See Also

session_decrypt()

A.7.2.16 uint32_t write_aes_hmac (int *fd*, const uint8_t * *data*, uint32_t *data_len*, const key_blob * *keys*)

Encrypt using AES-256-CBC and write the cipher text, including an HMAC of the same, to a file descriptor. This function writes a whole packet of information in a pre-defined format to the file descriptor, as follows:

1. 4 bytes in network order: the length of the packet (including these 4 bytes).
2. 16 bytes: The initialisation vector used in the encryption.
3. The cipher text.

4. 32 bytes: A HMAC (SHA-256) of the cipher text.

Parameters

fd	The file descriptor to which the packet is written.
data	The plaintext to encrypt.
data_len	The length (in bytes) of the plaintext.
keys	A pointer to a key_blob structure containing the AES and HMAC keys to use.

Returns

On success, the number of bytes written. 0 on failure.

See Also

read_aes_hmac()

A.7.2.17 uint32_t write_rsa (int *fd*, const RSA * *key*, const uint8_t * *buffer*, uint32_t *len*)

Encrypt to an RSA public key and write to a file descriptor.

Parameters

fd	File descriptor to which cipher text is written.
key	A pointer to an OpenSSL RSA structure containing the public key to be used.
buffer	The plaintext to encrypt.
len	Length (in bytes) of the plaintext buffer.

Returns

On success, the number of bytes written. 0 on failure.

See Also

read_rsa()

A.8 aes_openssl.c File Reference

Simple functions to interface OpenSSL's AES API (based on code by Saju Pillai saju.pillai@gmail.com).

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/evp.h>
#include <openssl/aes.h>
```

Macros

- #define uint8_t unsigned char

Functions

- uint8_t * aes_encrypt_ex (EVP_CIPHER_CTX *e, const uint8_t *plaintext, int *len)
- uint8_t * aes_decrypt_ex (EVP_CIPHER_CTX *e, const uint8_t *ciphertext, int *len)

A.8.1 Detailed Description

Simple functions to interface OpenSSL's AES API (based on code by Saju Pillai saju.pillai@gmail.com). Original URL: http://saju.net.in/code/misc/openssl_aes.c.txt Based on code written by Saju Pillai (saju.pillai@gmail.com). The author has placed this code in the public domain.

A.8.2 Macro Definition Documentation

A.8.2.1 #define uint8_t unsigned char

A.8.3 Function Documentation

A.8.3.1 uint8_t* aes_decrypt_ex (EVP_CIPHER_CTX * *e,* const uint8_t * *ciphertext,* int * *len*)

Decrypt data using AES. The padding scheme used is PKCS#7.

Parameters

e	An EVP_CIPHER_CTX structure initialised with the key and mode to be used.
ciphertext	A pointer to the buffer with the cipher text.
len	A pointer to an int containing the length of the cipher text. This is updated to contain the length of the resultant plaintext.

Returns

On success, a pointer to a buffer containing the plaintext, which must be free'd by the caller. NULL on failure.

See Also

aes_encrypt_ex()

A.8.3.2 uint8_t* aes_encrypt_ex (EVP_CIPHER_CTX * *e*, const uint8_t * *plaintext*, int * *len*)

Encrypt data using AES. The padding scheme used is PKCS#7.

Parameters

e	An EVP_CIPHER_CTX structure initialised with the key and mode to be used.
plaintext	A pointer to the buffer with the plaintext.
len	A pointer to an int containing the length of the plaintext. This is updated to contain the length of the resultant cipher text.

Returns

On success, a pointer to a buffer containing the cipher text, which must be free'd by the caller. NULL on failure.

See Also

aes_decrypt_ex()

A.9 pico_util.c File Reference

Contains miscellaneous utility functions for the Pico project.

```
#include <openssl/rsa.h>
#include <openssl/pem.h>
#include <sys/socket.h>
#include <poll.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <curl/curl.h>
#include <sys/stat.h>
#include "pico.h"
```

Data Structures

- struct curl_data

Functions

- void bytes_to_hex (const uint8_t *byte_array, uint32_t num_bytes, char *buffer)
- void hex_to_bytes (const char *hex_buffer, uint32_t num_characters, uint8_t *byte_array)
- void print_mem (const uint8_t *data, uint32_t size, const char *msg)
- uint8_t * DER_encode_RSA_public (RSA *rsa, uint32_t *len)
- RSA * DER_decode_RSA_public (const uint8_t *buffer, uint32_t len)
- RSA * PEM_decode_RSA_public (const uint8_t *buffer, uint32_t len)
- uint32_t generate_credentials (account_credential_data *cd, uint32_t parent_handle)
- int connect_sock (const char *host)
- uint32_t read_and_sign_nonce (int sock, const key_blob *session_keys, uint32_t priv_handle)
- uint32_t hash_and_sign (const uint8_t *data, uint32_t data_len, uint8_t *signature, uint32_t priv_handle)
- size_t pico_write_callback (char *ptr, size_t size, size_t nmemb, void *userdata)
- uint32_t get_http_data (const char *url, char *buffer, uint32_t buf_len)
- const char * get_line (const char *buffer, uint32_t *len)
- uint32_t write_all (int fd, const uint8_t *buffer, uint32_t len_total)
- uint32_t read_all (int fd, uint8_t *buffer, uint32_t bytes)
- uint32_t get_QR_info (pico_qr_format *pico_qr)
- uint32_t get_create_info (pico_qr_format *pico_qr, char *service_name, char *service_addr, RSA **pub_key, uint8_t *pem_digest)
- uint32_t get_account_file_name (BOOL create, const uint8_t *hex, char *file_name)

A.9.1 Detailed Description

Contains miscellaneous utility functions for the Pico project.

A.9.2 Function Documentation

A.9.2.1 void bytes_to_hex (const uint8_t * *byte_array,* uint32_t *num_bytes,* char * *buffer*)

Convert an array of bytes to hexadecimal representation.

Parameters

byte_array	Array of bytes that is to be converted.
num_bytes	Length (in bytes) of array.
buffer	Destination buffer that receives hexadecimal number. The function writes 2*len+1 characters to this buffer (including '\0').

See Also

hex_to_bytes()

A.9.2.2 int connect_sock (const char * *host*)

Connect to a host and return a socket handle which must be closed with close().

Parameters

host	The host to connect to. Must be either an IP address or a domain name, and a port number, separated by a colon.

Returns

On success, a descriptor for a socket. 0 on failure.

A.9.2.3 RSA* DER_decode_RSA_public (const uint8_t * *buffer,* uint32_t *len*)

Convert a DER representation of a public key to an RSA key handle.

Parameters

buffer	Memory location holding DER representation to convert.
len	Length of DER data in buffer.

Returns

On success, a pointer to an OpenSSL RSA structure containing a key handle. NULL on failure.

See Also

DER_encode_RSA_public()

A.9.2.4 uint8_t* DER_encode_RSA_public (RSA * *rsa,* uint32_t * *len*)

Encode an RSA key handle in DER. Caller must free resultant buffer.

Parameters

rsa	OpenSSL handle to an RSA key.
len	Receives length of the buffer allocated to hold DER data.

Returns

On success, a pointer to a buffer holding the DER representation. This must be free'd by the caller. NULL on failure.

See Also

DER_decode_RSA_public()

A.9.2.5 uint32_t generate_credentials (account_credential_data * *cd,* uint32_t *parent_handle*)

Generate an RSA key pair and fill out an account_credential_data structure with them.

Parameters

cd	Pointer to the location of the account_credential_data to receive the key data (the public key is DER-encoded, the private is a blob encrypted by the TPM).
parent_handle	TPM handle to the parent key (e.g., TPM_SRK_HANDLE).

Returns

0 on success, non-zero on failure.

A.9.2.6 uint32_t get_account_file_name (BOOL *create,* const uint8_t * *hex,* char * *file_name*)

Given a 256-bit hexadecimal representation of the server's public key, return a local file name for storing account data for that server. To preserve anonymity, the base of the file name is hashed with a salt. In case an account already exists, a sequence number is incremented, and the user asked which one he wants to use.

Parameters

create	If FALSE, it is an error if no such account exists, while the function asks the user to choose if more than one exists. If TRUE, the function increments a pointer appended to the file name in order to create a new unique file name.
hex	256-bit hexadecimal representation that is hashed with a salt and used as base for the file name.
file_name	Receives the file name found using the algorithm described above.

Returns

0 on success, non-zero on failure.

A.9.2.7 uint32_t get_create_info (pico_qr_format * *pico_qr,* char * *service_name,* char * *service_addr,* RSA ** *pub_key,* uint8_t * *pem_digest*)

Given data from a QR-code containing a URL and a hash, retrieve the content at the URL and verify that it hashes to the same value. The content retrieved includes a user-friendly name of the service in question,

its address (IP:port), a handle to its public RSA key as well as a hash of the server's PEM representation of its public key.

Parameters

pico_qr	Pointer to a pico_qr_format containing data from a QR-code.
service_name	Buffer that receives human-readable name of the service.
service_addr	Buffer that receives the address (IP:port) of the service.
pub_key	This pointer is changed to point to an OpenSSL RSA structure representing the public key of the service. Must be free'd by the caller.
pem_digest	This buffer receives a hash of the server-supplied PEM representation of its public key.

Returns

0 on success, non-zero on failure.

See Also

get_QR_info()

A.9.2.8 uint32_t get_http_data (const char * *url*, char * *buffer*, uint32_t *buf_len*)

Retrieve the resource at the given URL.

Parameters

url	URL of resource to retrieve.
buffer	Buffer to hold the downloaded data.
buf_len	Length of buffer.

Returns

0 on success, non-zero on failure.

A.9.2.9 const char* get_line (const char * *buffer*, uint32_t * *len*)

Given a multi-line data buffer, get the length of the current line.

Parameters

buffer	Multi-line buffer that is to be scanned.
len	Receives length of the current line.

Returns

A pointer to the start of the next line in the buffer (or NULL if this was the last).

A.9.2.10 uint32_t get_QR_info (pico_qr_format * *pico_qr*)

Read a QR code from the camera and retrieve its information. The following information is retrieved:

1. Command (login or create new account)

2. The RSA public key
3. A digest of the public key

Parameters

pico_qr	Pointer to a pico_qr_format structure into which the retrieved data is copied.

Returns

0 on success, non-zero on failure.

A.9.2.11 uint32_t hash_and_sign (const uint8_t * *data*, uint32_t *data_len*, uint8_t * *signature*, uint32_t *priv_handle*)

Hash a data buffer using SHA-1 and sign the digest using RSA.

Parameters

data	The data to be signed.
data_len	Length of the data buffer.
signature	Buffer that receives the signature. Must be able to hold RSA_SIGNATURE_BYTES bytes (for a key exponent size of 2048 bits, this is 256 bytes).
priv_handle	TPM handle of the private key to use for signing (must be loaded in the TPM).

Returns

0 on success, non-zero on failure.

See Also

read_and_sign_nonce()

A.9.2.12 void hex_to_bytes (const char * *hex_buffer*, uint32_t *num_characters*, uint8_t * *byte_array*)

Convert a string of hexadecimal characters to an array of bytes. Fails silently on error (if string contains an odd number of characters, or invalid characters).

Parameters

hex_buffer	Hexadecimal representation of data.
num_characters	Number of hexadecimal characters to convert (must be even).
byte_array	Receives the converted binary data.

See Also

bytes_to_hex()

A.9.2.13 RSA* PEM_decode_RSA_public (const uint8_t * *buffer*, uint32_t *len*)

Convert a PEM character buffer of a public key to an RSA key handle.

Parameters

buffer	Memory location holding PEM representation to convert.
len	Length of PEM data in buffer.

Returns

On success, a pointer to an OpenSSL RSA structure containing a key handle. NULL on failure.

A.9.2.14 size_t pico_write_callback (char * *ptr*, size_t *size*, size_t *nmemb*, void * *userdata*)

Callback function for cURL used in get_http_data.

Parameters

ptr	Pointer to data received by cURL.
size	Size representation. Data in ptr is size*nmemb.
nmemb	Size representation. Data in ptr is size*nmemb.
userdata	Pointer to Pico-relevant data (a structure of type curl_data).

See Also

`http://curl.haxx.se/libcurl/c/curl_easy_setopt.html`

A.9.2.15 void print_mem (const uint8_t * *data*, uint32_t *size*, const char * *msg*)

Print out the contents of a memory location in hex, with an informative message.

Parameters

data	Pointer to memory location that is to be printed.
size	Number of bytes to print.
msg	Label to print next to the memory.

A.9.2.16 uint32_t read_all (int *fd*, uint8_t * *buffer*, uint32_t *bytes*)

Loop until an exact number of bytes has been read from a file descriptor. Times out if reading is blocking for more than RW_TIMEOUT_SECONDS seconds. Returns the number of bytes actually read.

Parameters

fd	File descriptor to which to write (typically a socket).
buffer	Memory location to receive the data.
bytes	Number of bytes to read.

Returns

Number of bytes actually read (may be less than len_total only if the operation timed out).

See Also

write_all()

A.9.2.17 uint32_t read_and_sign_nonce (int *sock,* const key_blob * *session_keys,* uint32_t *priv_handle*)

Read a 128-bit nonce from a socket, hash and sign it, and write it back, encrypted with AES using the session key.

Parameters

sock	Descriptor of the socket on which to read and sign the nonce.
session_keys	Pointer to a key_blob structure containing the AES and HMAC keys to use.
priv_handle	TPM handle to the private key (must be loaded in the TPM) that is used for signing the nonce.

Returns

0 on success, non-zero on failure.

See Also

hash_and_sign()

A.9.2.18 uint32_t write_all (int *fd,* const uint8_t * *buffer,* uint32_t *len_total*)

Loop until the entire buffer has been written to a file descriptor. Since normal sockets do not guarantee that all data are really written out, this function keeps writing until this condition has been met, or a time-out (of duration RW_TIMEOUT_SECONDS) has been reached.

Parameters

fd	File descriptor to which to write (typically a socket).
buffer	Data to write to the file descriptor.
len_total	Length (in bytes) of data to write.

Returns

Number of bytes actually written (may be less than len_total only if the operation timed out).

See Also

read_all()

References

D. Abraham, G. Dolan, G. Double and J. Stevens (1991). Transaction security system. *IBM Systems Journal*, 30(2):206–229. (See p. 14.)

A. Adams and M. Sasse (1999). Users are not the enemy. *Communications of the ACM*, 42(12). (See p. 16.)

R. Anderson (2008). *Security Engineering — A Guide to Building Dependable Distributed Systems*. Wiley, second edn. (See pp. 4, 5.)

R. Anderson and M. Kuhn (1996). Tamper resistance—a cautionary note. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, vol. 2, pp. 1–11. (See p. 28.)

W. Arbaugh, D. Farber and J. Smith (1997). A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, pp. 65–71. IEEE. (See p. 5.)

D. Balfanz, D. Smetters, P. Stewart and H. Wong (2002). Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the 9th Annual Network and Distributed System Security Symposium (NDSS)*, pp. 7–19. (See p. 20.)

M. Bellare and C. Namprempre (2000). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—ASIACRYPT 2000*, pp. 531–545. Springer. (See p. 41.)

H. Bidgoli (2004). *The Internet Encyclopedia*, vol. 3. Wiley. (See p. 5.)

E. Blass and W. Robertson (2012). TRESOR-HUNT: Attacking CPU-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference—ACSAC'12*, pp. 71–78. ACM. (See p. 42.)

J. Bonneau (2012). *Guessing human-chosen secrets*. Ph.D. thesis, University of Cambridge. (See p. 5.)

J. Bonneau, C. Herley, P. van Oorschot and F. Stajano (2012). The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pp. 553–567. IEEE. (See p. 13, 13.)

D. Challener, K. Yoder, R. Catherman, D. Safford and L. Van Doorn (2007). *A practical guide to trusted computing*. IBM Press. (See pp. 5, 28, 36, 37, 55.)

W. Cheswick (2012). Rethinking passwords. *Queue*, 10(12):50. (See p. 16.)

D. Grawrock (2009). *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press. (See p. 54.)

E. Grosse and M. Upadhyay (2013). Authentication at scale. *IEEE Security and Privacy*, 11:15–22. (See pp. 11, 11, 12, 12, 12, 15.)

J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum and E. Felten (2009). Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98. (See pp. 29, 42.)

B. Ives, K. Walsh and H. Schneider (2004). The domino effect of password reuse. *Communications of the ACM*, 47(4):75–78. (See p. 5.)

J. Katz and Y. Lindell (2008). *Introduction to modern cryptography*. Chapman & Hall. (See pp. 4, 41.)

B. Laurie and A. Singer (2009). Choose the red pill and the blue pill: a position paper. In *Proceedings of the 2008 Workshop on New Security Paradigms*, pp. 127–133. ACM. (See pp. 24, 26, 44.)

Z. Mao, D. Florêncio and C. Herley (2011). Painless migration from passwords to two factor authentication. In *2011 IEEE International Workshop on Information Forensics and Security (WIFS)*, pp. 1–6. IEEE. (See p. 55.)

T. Matsumoto, H. Matsumoto, K. Yamada and S. Hoshino (2002). Impact of artificial "gummy" fingers on fingerprint systems. In *Electronic Imaging 2002*, pp. 275–289. International Society for Optics and Photonics. (See p. 11.)

J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor and A. Perrig (2010). TrustVisor: Efficient TCB reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*, pp. 143–158. IEEE. (See p. 54.)

J. McCune, B. Parno, A. Perrig, M. Reiter and H. Isozaki (2008). Flicker: An execution infrastructure for TCB minimization. *SIGOPS Operating Systems Review*, 42(4):315–328. (See p. 54.)

J. McCune, A. Perrig and M. Reiter (2005). Seeing-is-believing: Using camera phones for human-verifiable authentication. In *2005 IEEE Symposium on Security and Privacy*, pp. 110–124. (See pp. 5, 20.)

R. Morris and K. Thompson (1979). Password security: a case history. *Communications of the ACM*, 22(11). (See p. 5.)

T. Müller, F. Freiling and A. Dewald (2011). TRESOR runs encryption securely outside RAM. In *Proceedings of the 20th USENIX conference on Security*, pp. 17–17. USENIX Association. (See p. 42.)

T. Müller and M. Spreitzenbarth (2012). Frost: Forensic recovery of scrambled telephones. Tech. rep., University of Erlangen. (See p. 42.)

B. Parno, C. Kuo and A. Perrig (2006). Phoolproof phishing prevention. In G. Crescenzo and A. Rubin (editors) *Financial Cryptography and Data Security*, vol. 4107 of *Lecture Notes in Computer Science*, pp. 1–19. Springer Berlin Heidelberg. (See p. 26.)

B. Parno, J. McCune and A. Perrig (2010). Bootstrapping trust in commodity computers. In *2010 IEEE Symposium on Security and Privacy*, pp. 414–429. IEEE. (See p. 5.)

B. Parno, J. McCune and A. Perrig (2011). *Bootstrapping Trust in Modern Computers*. Springer. (See p. 5.)

R. Peeters (2012). *Security Architecture for Things That Think*. Ph.D. thesis, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium. (See p. 5.)

T. van der Putte and J. Keuning (2000). Biometrical fingerprint recognition: Don't get your fingers burned. In *Smart Card Research and Advanced Applications*, pp. 289–303. Springer. (See p. 11.)

A. Rao, B. Jha and G. Kini (2012). Effect of grammar on security of long passwords. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*. (See p. 5.)

E. Sachs (2013). Stronger consumer authentication — 5 year report. Public draft, last visited: 27/5/2013. (See pp. 11, 12, 12, 12, 13, 15.)

D. Schellekens (2012). *Design and Analysis of Trusted Computing Platforms*. Ph.D. thesis, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium. (See pp. 28, 41, 41, 41.)

B. Schneier (2005). Two-factor authentication: too little, too late. *Communications of the ACM*, 48(4):136. (See p. 23.)

A. Shamir (1979). How to share a secret. *Communications of the ACM*, 22(11):612–613. (See pp. 4, 5, 36, 39.)

F. Stajano (2011). Pico: No more passwords! In *Proceedings of the Security Protocols Workshop*. Author's preprent, rev. 61 of 2011-08-31. (See pp. 2, 2, 2, 3, 5, 7, 7, 7, 8, 12, 13, 14, 17, 25, 27, 29, 35, 41, 43, 44, 44, 44, 44, 44, 51, 51, 52, 55, 55, 55, 55, 57, 57, 58, 58.)

F. Stajano and R. Anderson (2000). The resurrecting duckling: Security issues for ad-hoc wireless networks. In B. Christianson, B. Crispo, J. Malcolm and M. Roe (editors) *Security Protocols*, vol. 1796 of *Lecture Notes in Computer Science*, pp. 172–182. Springer Berlin Heidelberg. (See p. 14.)

O. Stannard (2012). *Picosiblings*. Bachelor's thesis, University of Cambridge Computer Laboratory. (See p. 17.)

O. Stannard and F. Stajano (2012). Am I in good company? A privacy-protecting protocol for cooperating ubiquitous computing devices. *Security Protocols Workshop*. (See pp. 17, 18, 19, 19, 25, 26, 57, 58.)

M. Strasser and P. Sevnic (2004). A software-based TPM emulator for Linux. *Semesterarbeit, ETH Zurich*. (See p. 50.)

M. Strasser and H. Stamer (2008). A software-based trusted platform module emulator. In P. Lipp, A. Sadeghi and K. Koch (editors) *Trusted Computing — Challenges and Applications*, vol. 4968 of *Lecture Notes in Computer Science*, pp. 33–47. Springer Berlin Heidelberg. (See p. 50.)

B. Tian (2012). *Pico: a security token to replace passwords*. Bachelor's thesis, University of Cambridge Computer Laboratory. (See pp. 26, 31, 42, 57, 58.)

A. Whitten and J. Tygar (1999). Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, vol. 99. McGraw-Hill. (See p. 16.)

M. Wilkes (1968). *Time-sharing computer systems*. Elsevier, New York. (See p. 5.)