TRUSTED EXECUTION: APPLICATIONS AND VERIFICATION

by

IAN GILBERT BATTEN

A thesis submitted to the University of Birmingham for the degree of DOCTOR OF PHILOSOPHY

School of Computer Science
University of Birmingham
March 2016

Abstract

Useful security properties arise from sealing data to specific units of code. Modern processors featuring Intel's TXT and AMD's SVM achieve this by a process of measured and trusted execution. Only code which has the correct measurement can access the data, and this code runs in an environment trusted from observation and interference.

We discuss the history of attempts to provide security for hardware platforms, and review the literature in the field. We propose some applications which would benefit from use of trusted execution, and discuss functionality enabled by trusted execution. We present in more detail a novel variation on Diffie-Hellman key exchange which removes some reliance on random number generation.

We present a modelling language with primitives for trusted execution, along with its semantics. We characterise an attacker who has access to all the capabilities of the hardware. In order to achieve automatic analysis of systems using trusted execution without attempting to search a potentially infinite state space, we define transformations that reduce the number of times the attacker needs to use trusted execution to a pre-determined bound. Given reasonable assumptions we prove the soundness of the transformation: no secrecy attacks are lost by applying it. We then describe using the StatVerif extensions to ProVerif to model the bounded invocations of trusted execution. We show the analysis of realistic systems, for which we provide case studies.

Acknowledgements

I would first and foremost like to thank my wife Sue and daughters Sarah and Ruth, and my parents Geof and Margaret, for their support in my move from industry to academia.

In the School of Computer Science at the University of Birmingham, my supervisor Professor Mark Ryan welcomed me and gave me invaluable support and guidance, and Professor Achim Jung extended the kind invitation to return to the school. Dr Shiwei Xu was generous with his time and ideas and provided deep insights into ProVerif. Within the school I received valuable support from Mr Bob Hendley, who was both my undergraduate tutor and more than thirty years later chaired my viva. Drs Chris Novakovic, Peter Lewis and Catherine Harris, amongst many others, welcomed me as a fellow postgraduate and refrained from pointing out that I was twice their age.

At my former employer Fujitsu Telecommunications Europe, my various line managers Andy Stevenson, Tony Oppenheim, Marc Curtis and Phil Chater supported my decision to move, and my erstwhile colleagues Neil Prentice and Simon Parsons provided much appreciated injections of hardware and advice.

Again at FTEL Dr Nigel Garnham and at the University of Birmingham Dr Rosamund Moon convinced me that doing a PhD after a previous career was possible.

And last but not least, I'd like to thank Jenny Marris of the Open University who as my tutor on AA100 Arts Foundation made a valiant attempt to improve my writing skills; the failings in the writing are, of course, all my own.

Contents

List of Figures

Chapter 1

Introduction, motivation and background

1.1 The problem

The correct behaviour of a computer program is dependent on the correct behaviour of the platform it runs on. An attacker who can control or observe the operation of the platform can also control or observe the execution of a program running on the platform. The attacker can either change the way the program behaves, or can observe its inner workings and obtain information that permits further attacks. If the correct operation of the program is the basis of some security properties, the program can appear to execute correctly but the security properties may not be maintained.

This problem is of increasing interest when we consider the increasing use of cloud technologies. Many of the protections that have been used in the past — physical control of computers, computers being dedicated to one task, close control of software on computers — have been supplanted by the idea that programs run on remote computers, controlled by third parties, with several layers of software between the executing program and the physical machine. An attacker has many more opportunities to subvert correct operation.

We would like to be able to execute programs on a platform, such that the programs maintain their security properties in the face of a strong attacker, while making the minimum set of assumptions about the operation of the platform.

Hardware technologies — Intel TXT [1] and AMD SVM [2] — have emerged that claim to provide the means to put a processor into a specific mode of operation which restricts the power of an attacker while a particular program is being run. Further, it is claimed that encrypted data can be securely linked to these programs, such that only the precise unmodified program, executing in the restricted mode, can have visibility of the decrypted form of the data. That the program cannot be modified while retaining access to the data prevents the program itself from being subverted; that the platform is placed into a restricted state removes the ability of the attacker to use the platform's facilities to observe or control the running program.

I refer generically to these technologies, which differ mostly in the details of implementation, as "trusted execution". This encompasses both the ability to confirm that a particular program is running, and also the ability to bind that program to a particular piece of data.

These facilities are complex, and their architecture relies on a range of pre-existing technologies (particularly the Trusted Platform Module [3]) as well as additional features in the processor. This thesis sets out to:

- Describe why these features are useful and timely, having the capacity to solve useful problems;
- Provide a history of past attempts to provide these, or similar, properties, setting the new developments in a broader context;
- Suggest a variation on an established cryptographic primitive, Diffie-Hellman key exchange, as an example of the sort of program that is made possible by having the facility to perform computation that cannot be observed by an attacker using data that is visible only to the program;
- Describe a verification of the architecture of the trusted execution facilities when used to execute sensitive pieces of code, and describe a toolchain built on a proven verification tool which permits those pieces of code to be verified when executing as part of a trusted execution system;
- In support of the verification, describe and prove a set of transformations and abstractions which make the verification possible;
- Discuss developments and refinements of the trusted execution facilities, as proposed in newer processor designs.
- Propose some future lines of research.

1.2 The current position

Drawing from other lines of research in the department, it was decided to use a symbolic model for verification. The attraction lay in being able to build models which span both the architecture of trusted execution and applications running on top of it, so that a single verification can show that an application is making correct use of the trusted facilities.

This thesis therefore revolves around the use of StatVerif [4], a state of the art verification tool which is a derivative of the well-established tool ProVerif [5].

ProVerif is a symbolic verification tool for cryptographic protocols. It makes the assumption that the underlying cryptographic primitives are sound, or could be replaced with other cryptography that is. Verifications carried out with ProVerif start from the assumption of symmetric ciphers that cannot be broken other than by infeasible exhaustive search, asymmetric algorithms of similar strength and hash functions which do not have attacks better than brute force. Verifications produced by ProVerif assure the protocol up to an attacker who can break cryptographic primitives.

Building and executing a ProVerif model of a protocol or system will have one of three results:

- It terminates without finding an attack, indicating that the protocol being verified maintains the tested security properties because all possible attacks have been excluded;

- It terminates by finding an attack, which is a sequence of actions and messages that the attackers can use to break one of the tested security properties;
- It fails to terminate, which no assurance of security while not giving details of any potential attack. It can be left to run longer, or on faster hardware, and a skilled user may be able to guess with some measure of confidence as to whether it is "making progress" towards termination or likely to run indefinitely. However, this is a unsatisfactory position.

ProVerif's abstractions are unable to deal with changes in state in the model. They are therefore unsuitable for modelling TPMs, which are inherently stateful. There is support for protocols with "phases" (once a phase has been completed, the model moves to the next phase which has different behaviour) but this is not general enough to handle a TPM.

This limitation was attacked by Ryan *et al* in their StatVerif tool, which builds on ProVerif to provide a general purpose cell which can hold state. It is this mechanism which allows us to model a TPM in a natural way.

However, StatVerif, like ProVerif, does not guarantee termination. Indeed, non-termination is a common situation that the developer of a model faces. Problems which provide the opportunity for the search space to continue to expand will often not terminate, and one can see this clearly in the operation of a TPM.

As we will see later, a TPM contains a register file, which can only (once the device has been reset) be operated on by the action of *extension*, which incorporates new data by hashing. This can be repeated an arbitrary number of times, and StatVerif as it stands has no way to exclude the possibility that repeatedly performing the operation will not eventually yield a situation useful to the attacker. This reflects the possibility that extending a register of a TPM will result in a useful configuration.

Intuitively, this is not feasible: if the hash functions used for extension have properties we can reasonably assume, then the chances are negligible that an attacker will succeed in repeatedly extending the register, even with chosen input and having it take on a desired value. And in terms of a symbolic verification tool, where the hashes are ideal, it can never succeed. It is worth briefly exploring why this is true.

A concrete hash function has a chance of collision: given two distinct inputs to a hash function, there is a small but non-zero chance that the outputs will be identical. Hash functions are constructed with the intent of making finding such a pair of inputs, or (more usefully to an attacker) finding a pair given one initial value, computationally infeasible. The risk remains, but we usually discount it; an attacker who has sufficient resources and time to compute clashing inputs to hash functions is outside the threat model we are considering.[1]

The symbolic model assumes idealised hash functions, which never clash. So even if we were to consider the risk of the attacker being able to generate two dissimilar inputs which produce equivalent outputs from a concrete hash function, this situation would not arise in the symbolic model: we have already discounted that possibility by opting to working in the symbolic model.

[1] In fact, such an attacker would not waste their time attempting to extend registers to get a particular value; they would be able to create a malicious program which compromised security but had the same "measurement", a hash, as the legitimate program the defender was attempting to execute. The entire trusted execution model cannot withstand such an attacker, as it depends on measurements uniquely identifying programs.

Therefore, by restricting the number of successive hashes that an attacker can perform, we impose a reasonable restriction in both the concrete and symbolic worlds. In the concrete world, we are accepting the negligible risk of a very strong attacker who can compute clashing hashes; in the symbolic world, we are using a property of hashes which is true under our assumptions.

Part of the work of this thesis is to place bounds on the search space StatVerif needs to cover, proving that failure to find an attack within the limited search space means that a wider search would not find an attack.

1.3 Contribution

This thesis provides the following contributions:

- A model of a simplified TPM and associated processor facilities for trusted execution. This uses a state-of-the-art verification tool, StatVerif.
- A set of transformations, with associated proofs of correctness, which permit the use of StatVerif to check the security properties of programs running within a trusted execution environment. The transformations address issues of termination which are described in Chapter 5.
- A set of tools which permit these transformations to be carried out automatically.
- A set of tools which allow models to be written in a compact language, TXML, from which StatVerif code can be generated. This both simplifies the process of modelling and removes some sources of errors in the construction of the models.
- A proposed exploitation of trusted execution to permit the exchange of keys in cryptographic protocols without long-term reliance on random number generators, which present both practical problems and problems of verification.

1.4 Research questions addressed

1. **Can we verify the security properties of real applications which use trusted execution as part of those properties?**

   For trusted execution to be useful, it must have applications that solve real problems. And to get useful benefits from the verification of the trusted execution architecture, it must be possible to verify not only the trusted execution itself but also the applications that use trusted execution, and crucially the combination of the two. The work in this thesis provides a toolchain that takes models of security-sensitive protocols that use trusted execution, and verifies the whole model in a single step.

2. **As a foundation for the above, can we verify the architecture and operation of trusted execution in isolation?**

Although it is important to verify that more complex protocols are correct, we should also be confident that the underlying architecture is sound. We therefore construct a minimal test case which we show to be secure, as well as verifying more complex applications.

3. **Is it possible to use state of the art tools to search potentially infinite spaces, without needing to modify (and therefore invalidate the proofs of correctness for) their existing operation?**

   We need to make very specific changes to the searching conducted by our chosen tools. In principle, the tool could be modified just for us, and it might be possible to construct a proof of the correctness of those modifications for some more general case. However, our chosen course is to prove the correctness of our abstraction and transformation, while leaving the supporting tool unchanged.

4. **Are there applications enabled by trusted execution which would not otherwise be feasible, that have useful or novel security properties?**

   As a demonstration, we outline a method of key-exchange which avoids some concerns with the quality, availability and assurance of strong random number generators, in particular in embedded devices, but which hinges crucially on the use of trusted execution.

In summary: does trusted execution work correctly, and what can we use it for?

1.5 Scope and limitations

This thesis contains a verification of the architecture of a trusted execution mechanism using a symbolic verification tool and a symbolic model. It does not, and cannot, verify concrete implementations of real silicon. Symbolic verification tools address simplified and idealised models, but give strong results about those models. That a symbolic model of an architecture has been verified means that it is possible to build correct silicon, but does not provide guidance or techniques for doing so.

The same applies, *mutatis mutandis*, to the verification of applications that use trusted execution. The verification covers the high-level algorithms and security architectures, showing that the use of trusted execution is conceptually correct, but provides no assurance that any concrete implementation on a real computer will have those security properties. As might be deduced from the literature review, I had intentions towards providing tooling to produce executable code from TXML models, but this work was not eventually completed.

Limitations in the method we use mean that we can study confidentiality of protocols, but we cannot study freshness. Our abstraction assumes that when a piece of code is executed under trusted execution, it always returns the same results for the same inputs. A common technique when using trusted execution is to pass a nonce (a random number, a sequence number or some similar distinguishing identifier) into function calls, so that this can be incorporated into the result to ensure that the returned value is the result of computation that has been freshly carried out. This provides protection against a variety of replay attacks. The abstraction that is used does not permit this. This is discussed at more length in the conclusions.

1.6 Prior publication

Chapter 5 is in part based on a paper presented by the present author at the 8th International Symposium on Trustworthy Global Computing, TGC 2013, in September 2013 [6]. This paper listed as authors Dr Shiwei Xu, myself and Professor Mark Ryan.

Dr Xu provided the initial suggestion to use StatVerif to analyse the operation of the TPM and trusted execution. Dr Xu and I, with supervision from Professor Ryan, coded by hand StatVerif models of trusted execution, which encountered serious problems with termination. I proposed and implemented the TXML method which allowed us to transform the problem into one which could be made to terminate, using the SUBR abstraction described in the paper, and we collectively worked on the proof of that method's correctness. Dr Xu and I specified the transformations that would be necessary to the Horn clauses to take advantage of our simplifications, and I then wrote the surrounding tool-chain which automated the transformation and verification, as well as working on the transformations to ensure that they did, in fact, terminate. I showed that TXML could be used as a vehicle for writing models as well as for proving that the transformations were correct, and re-worked the manually constructed models to instead start from TXML.

An initial draft of the paper was jointly written by myself and Dr Xu, but he returned to China at the point of its first (unsuccessful) submission to a conference and played no further part in its production. I then rewrote the paper to better suit the venues it seemed appropriate to, and it was eventually accepted by TGC 2013.

We agreed at an early stage to give Dr Xu first author credit because it was a requirement of his sponsors that he publish a paper as first author. The paper as finally published was approximately 80% my work, and a substantial amount of other material relating to my tooling and modelling work, which was not part of the paper as published, has been added to the chapter.

1.7 Structure of thesis

After some background to the concept of trusted execution and a survey of previous work, I give a brief history of the concept of using hardware assurance to provide a basis for security properties. I describe a set of problems that would benefit from the availability of trusted execution. These range from the generation of key material for secure communication, through the signing of certificates for secure websites, to a mechanism for performing password-based authentication without exposing the password itself outside a trusted execution container. These applications cover both client- and server-side functionality. I then present a formal verification of the architecture of one trusted execution technology, Intel's TXT, with a tool-chain which allows algorithms to be verified for security within the TXT framework. I then survey newer technologies which have arisen during the time this research has been carried out, and propose some future directions for further research.

Chapter 2

Background to trusted execution

2.1 What is trusted execution?

The concept of Trusted Execution covers a range of technologies which allow the execution of code to take place in a known, trusted and trust-worthy environment. The execution takes place free from external influence and observation. Applications also have the option of binding execution to pieces of confidential data, such that the confidential data can only be seen by authorised pieces of code. Facilities for confidentiality and integrity of data, and integrity of code, are provided. A piece of code can therefore be securely linked to some secret data, such as a key, which is not available to other code running on the same platform, with the basis for that guarantee being a fundamental hardware property of the system.

This thesis explains why this technology is useful and timely, describes some applications which it would enable, verifies the architecture of one implementation, describes a tool-chain which permits the verification of applications running within that implementation, summarises some technology developments which have emerged in parallel, and considers why it is that although useful and timely, the technologies in question have largely been rejected (or, perhaps, ignored) by the market.

2.2 What problems does trusted execution address?

Trusted execution sets out to address the issue of assuring the execution of security-critical code. Modern operating systems and hardware platforms are complex, and even if an application is itself secure in its design and implementation, the operating system and hardware platform will offer a variety of avenues for an attacker to attempt to subvert execution.

Consider the case of a small application which embeds a key that is used to decrypt some confidential data, operate on it, and then encrypt the results. It is very difficult to make the binary file which contains the key unreadable to a well-resourced and determined attacker. They can either use facilities that are offered for the very purpose (stopping a running program, examining its data and modifying either the flow of control or the data in use is the main purpose of debugging tools) or misuse other features (some device drivers offer the ability to read arbitrary regions of memory for performance reasons, which can then be used to examine running programs), depending on the level of access they have to the running system. And that is only using official interfaces: the range of attacks using bugs and unintended features in operating systems is large and worrying.

So someone wanting to run code that needs secrets, or which needs to run correctly, or both, is left with a very difficult problem: no matter how much work they put into the design and implementation of their application, they will have to confront the security, or lack of security, of the platform the application runs on. Other than in the most well-resourced environments which are willing to accept the limitations of specialised platforms that have been assured to a similar standard (such as the platforms used for cryptographic services at and above TOP SECRET), the application developer has to accept the risks associated with running on a large, complex, un-assured (and arguably un-assurable) general purpose operating system.

Trusted execution sets out to offer a more restricted environment, hosted on the same hardware, but temporarily isolated from the main system by a set of hardware protections that it is reasonable to accept as correct for the application's purposes; attackers who have the ability to discover, or perhaps even indeed inject, faults into commodity processors are outside the scope of most assurance activities. Applications, or more precisely the security-sensitive parts of applications, are then able to run isolated from the operating system, using a contained subset of the hardware facilities, with code and data tightly bound together such that the data cannot be accessed by other pieces of code.

2.3 Security considerations for personal devices

It is tempting to think that the device in your hand is easier to secure, or more trustworthy, than devices outside your physical control. Debates about moving systems into the cloud tend to start from the presumption that this reflects an overall reduction in security, with the main issue being the securing of the cloud component of the resulting system.

There is no particular reason for this to be true: each device involved in the overall system is a computer, running some software, subject to approximately the same threats as any other computer running some software. Trust can be subverted by a variety of means including malware and physical intrusion.

Personal devices are at least as susceptible to these attacks as any other computer. Any vector which permits a malware attack on a cloud device also applies to a personal computer, but the personal computer will also almost certainly have some sort of web browser which "fetch" material. And unless people have their personal device implanted sub-cutaneously, physical intrusion is at least as likely (or unlikely) when the device is in a house, hotel room or train as in a data centre.

So what assumptions are made about devices in the user's physical possession?

The first is that devices under the physical control of the user are secure against, or at least less vulnerable to, malware. This hardly seems worth raising only to demolish, but I have heard it expressed repeatedly. The idea that their personal laptop or phone, which they use all the time, should be as susceptible to attack as any other device seems to be emotionally worrying. There is little technical justification for this position — at best, a device that is in constant use may exhibit symptoms of infection that the user will notice — and many reasons to suspect that it is not true. For example, a primary vector for malware distribution is via compromised or malicious websites exploiting bugs or user error in web browsers: a personal device will be more prone to this than a device running a more restricted workload.

The second is an assumption of physical trustworthiness: that the hardware has not been tampered with, or if it has the user will detect that it has happened. Leaving aside that attacks conventionally thought of as physical may in fact be executed by malware (for example, a keylogger is as likely to be a malware attack as an actual piece of hardware), very few personal devices are physically secure. Even the simplest of precautions, such as tamper-evident seals on screws or access panels, are rarely taken. Even when they are, the threat model is not useful to the user: the tamper-evidence is there so that a manufacturer can deny responsibility for a repair based on the device having been interfered with, not so that the owner can determine that the device has been opened.

Once the case has been opened, adding additional devices is not difficult, especially as today the most common structure of a laptop is for the internal devices — the keyboard, the mouse, the camera, the slot for inserting memory cards — to be located on an internal USB bus to which can be added (with suitable jumpers) additional USB devices. If someone believes that their equipment is subject to physical attack (a travelling senior manager who has to leave their laptop in places where competitors have access to it, for example) then resources in a large, relatively secure data centre are a substantially more difficult target.

2.4 Security considerations in operating systems

The history of operating systems that are used on the Internet is worth recalling. Implicit in these operating systems are a set of assumptions about security threats and security models which, I will argue, are no longer valid.

The dominant operating system amongst computers providing services on the Internet is Unix [7]. For the purposes of this discussion, Unix encompasses both "real" Unix, which traces its lineage back to work done at Bell Labs under Ken Thompson and Dennis Richie (largely Oracle's "Solaris", but also such systems as HP's HP-UX and IBM's AIX), and Unix-like operating systems such as Linux, which share interfaces and architectures but little or no actual code. Unix and Linux are essentially interchangeable from the perspective of an application: it is rare to find an application which will run on one but not the other, and when this arises it is usually straightforward to resolve. Henceforth, when I say "Unix" I include "Linux" unless explicitly stated.

The dominance of Unix is such that not only does it power machines that are obviously Unix machines (such Unix workstations as still exist, mainstream cloud services, laptops and desktops of ostentatiously geeky mien). It is also the core of not only Apple's "OSX" desktop and laptop product (hidden away, although in fact shipped with every traditional Unix tool imaginable) but also their iOS operating system used to power iPhone, iPad, iPod and Apple TV.[1] And the other major player in the portable device space, Android, is similarly Unix-based (in this case, a heavily modified version of Linux).

The Unix security model is relatively straightforward. The core of the operating system, the *kernel*, runs in supervisor mode on the processor. It manages a set of tasks, known as

[1] There is some debate over the precise Unix-nature of iOS and OSX, because of the use of the Mach microkernel underneath a Unix kernel derived from FreeBSD. That does not detract from the operating system API and security model being a very pure, and standards-compliant, Unix.

processes which run in user mode. Each user mode process has a "user id" or uid, conventionally a small integer associated with a name, and a similar "group id" or gid, representing some organisational structure amongst uids. Historically uids were either 0, representing an all-powerful "root", or greater than 0, representing unprivileged users. By convention, small positive integers are reserved for system services, and flesh and blood users start at 100, 500 or some similar value, depending on the version of Unix being used.

The kernel mediates access to files and resources based on the uid and gid of the process; resources have an associated user and group, and a bit masks indicates the access read, write and other access available for processes with the same uid, the same gid, or neither.[2]

This mechanism is not sufficient to allow shared access to resources like (to take an example of the era) serial lines. Suppose a serial line is connected to a shared computer, and multiple users want to use the line to access some external piece of hardware. If the serial line is directly accessible to all, then nothing stops multiple users from interfering with each other. If the serial line is owned by, and only accessible by, one user, then other users cannot access it at all.

This problem is solved with "set userid", usually known as "setuid": a program can be set up such that when it runs, it assumes some other user's credentials rather than those of the user that invoked it. For historical reasons, serial lines on Unix machines are traditionally owned by the user "uucp"[3] and programs that grant access to those serial lines are therefore "setuid uucp". When they are invoked, they use a locking mechanism to ensure that they are granting exclusive access to the line, and then use the capability of now running as user "uucp" to read and write the serial data. A similar, less-used, mechanism exists for "setgid" [10].

In summary, the Unix security model consists of a privileged operating system, unprivileged users, and a mechanism by which users can obtain additional privileges in a controlled way. Other operating systems have used different mechanisms for controlling privilege, but the setuid technique has withstood the test of forty years, albeit with concerns about practical implementations [11, 12].

This security model stemmed from two, interconnected, problems of the time, that are no longer true: that computers are expensive, and that availability is more important than any individual's data.

Initially, the Unix security model was intended to deal with a group of users of a shared system who trust each other but need some isolation so as to share resources fairly when developing software, and need some protection of the core operating system to prevent the system being accidentally rendered unavailable. In other words, its original target audience of

[2] There are more sophisticated access control mechanisms on specific versions of Unix, ranging from access control lists to the various compartmentalisation projects such as the NSA's "Flask" [8] which inspired the Linux SELinux subsystem, but because of the lack of standardisation applications rarely make full use of the facilities. Sadly, the components of the system which are probably the best tested and examined, the core operating system components, will make effective use of SELinux, OSX "Sandbox" or Solaris "RBAC", while custom applications that are installed will tend to demand, and be granted, rather more wide-ranging capabilities.

[3] "Unix to Unix CoPy": the uucp suite [9] is an early networking package originally targeted at serial lines and modems, which permits file transfer and remote execution. Although now almost completely obsolete, even in versions that run over more modern network infrastructure, it was hugely influential and enabled the rise of email on systems that had previously not been powerful enough or well connected enough to connect to the nascent Internet. Strangely, a complete uucp implementation ships with the 2015 version of Apple's OSX; it would be fascinating to know if anyone is using it.

researchers at Bell Labs, or its later audiences of groups of computer science postgraduates.

An attacker able to masquerade as a particular user is able to destroy all that user's data, but is not able to destroy other users' data or bring the system down. If your system is being used by thirty people, this is not an unreasonable model: the activities of the other twenty-nine are not affected by the activities of the thirtieth. Computers are expensive, shared resources, and the correct functioning of the operating system for the majority is more important than the interests of any one individual; the operating system's primary security task is to isolate users from each other and keep them away from critical system resources.

Until the 1990s, most system services (usually implemented as "daemons", long-running processes that are started at boot time and communicate with users via the network and local inter-process communication mechanisms) ran as root, because it simplified design and implementation. A few services ran with lower privileges ("uucp", "lp" or "daemon"). But system services were largely able to access each other's data, and services which had little need to access privileged resources nonetheless ran as root. The Morris Worm [13] of 1988 proved how foolish this was: an attacker was able to use a crude buffer-overrun to attack a common system service, and from that was rapidly able to take over machines and propagate the attack. Both "finger", a now-obsolete service that tells you what people in your building are doing, and "sendmail", the standard mailer, ran as root in order to access user directories and to bind to privileged ports.

There has been a subsequent drive to run system daemons with as little privilege as possible, with a separate user for each subsystem. Where necessary a small additional helper program encapsulates privileged operations. A simple cloud server running a web server and a mailer might have processes owned by ten or more separate users, with both the web server and the mail system using multiple distinct uids to isolate possible contagion caused by a malware attack.

However, all of these improvements derive from the original assumption: that the main purpose of the security model is to protect the core operating system from possibly hostile users. When the machine being defended is a DEC VAX 11/750 running 4.2bsd in a 1986 computer science department machine room[4] and providing both time-sharing and wider departmental services, this makes perfect sense: one student cannot prevent the other students from working; the mail server keeps running even if the researchers misbehave. But this model is much less relevant today, when a user will normally have sole control over a machine: whereas in the past the core operating system was taken to be more valuable than any user's data, today it is precisely the opposite. The operating system can be re-installed in minutes, while the user's data — both its integrity from change and deletion, and its confidentiality — is far more important. The machine has a very different focus: it needs to protect the user's data, and run the user's software correctly. Substantial changes to the Unix security model have been made in Android [14], whose focus shifts from protecting multiple users of the same platform from each other, to isolating distinct programs run by the same user so that they cannot be used by an attacker to compromise the security properties of other programs run by the same user.

[4] Immediately prior to the arrival of the Sun workstation, a medium sized VAX was the computer science department machine of choice.

2.5 Security considerations in the cloud

Consider a simple cloud system consisting of one user terminal and one server or virtual service located outside the user's direct control, processing some sensitive data. An address book application for a business, perhaps.

The security challenges faced by both devices are similar: an attacker can physically steal the device and attempt to extract or use data stored on it, an attacker can attempt to run their own software on the device, an attacker can attempt to mis-use already installed software for purposes disadvantageous to the legitimate user, an attacker can attempt to obtain the credentials of a legitimate user. For a system which is networked (and at the time of writing the existence of personal devices that are not networked can for practical purposes be discounted), all these threats are similar in nature whether the sensitive data resides local to the user or on the cloud-based system.

On the one hand, the cloud-based system will be located in a secure data centre, protected by appropriate firewalls and possibly intrusion detection systems, focused on delivering one service with a software stack that is under change management. In some, over time increasingly many, cases, the system will be subject to a formal security management policy.

On the other, the personal device will be running a wide range of applications with limited segregation between them, exposed to a wide range of threats on possibly hostile and certainly public networks, with little or no formal security management. The user will be, in the general case, perfectly willing to install software from untrustworthy software on a whim, and will be poorly equipped to understand the problems that this may cause, or notice and deal with problems if they actually arise.

There are, of course, some additional risks facing the system that is using cloud services. One is that the data will have to be moved over a public network, where it is subject to interception and possible modification. A common protection applied to data in transit over public networks is encryption; this requires that the parties exchanging data share secret information, known as a key. As we will see, the generation of keys can be problematic. Another is that data is at risk of compromise by the owner or operator of the cloud service, but a reasonable model of their posture is "malicious but cautious" [15]: they might be assumed to have the motive and the capability to misuse data, but they have a substantial reputational investment in not being caught doing so, and therefore will balance the value of misusing the data against the cost of getting caught. The relationship between the operator and the user will be based on a contract, rather than ownership, and that contract may be hard to enforce (the contract probably includes a venue clause which is convenient for the operator, in most cases less convenient for the user). But in most cases, the operator will benefit more from a reputation of behaving appropriately than from the compromising of any individual user.

This is further complicated by the uncertainties in the relationship between customers and cloud providers. In many cases, there is no contractual relationship at all, or not one that could realistically be enforced. The user is often not paying the provider (in money, least: the value for the provider comes from advertising, brand awareness or some other "soft" payment) and even if there is a financial relationship, the venue in the terms and conditions is often California (for the typical Silicon Valley start-up) or Delaware (for the more legally astute American company that has taken good advice). For a small user outside those states, their chances of bringing any effective legal action against the provider are low [16, 17, 18, 19].

When larger customers are involved, there may be a realistic chance of legal action, as companies who purchase cloud services at scale will ensure there is a mutually agreed venue for any dispute. However, this does not really provide much comfort; unless the contract is drawn so as to include consequential losses or liquidated damages, the most that a provider would be liable for is the refund of payments made. The reality is that a cloud customer has to follow the old poker dictum that one should trust the dealer, but always cut the deck:[5] they should select a cloud provider who has a reputation for honesty, but also take precautions so as to reduce their risk exposure to that provider.

This is difficult enough when the cloud provider is offering just a bare operating system to run the customers' applications on, but becomes much more complex when the customer is purchasing a service using the provider's applications, which themselves may be hosted by third parties It is a common pattern now for applications offered by start-ups to obtain their processing power from one cloud provider, use a database hosted by second, with backups directed to storage provided by a third. This is not surprising, or of itself undesirable; one would expect cloud providers to specialise in different services, and for a large application to take the "best in class" service from a variety of providers. It does, however, complicate liability and security. Not only can the cloud provider access the customers' data, but many of the measures taken to protect the data from third parties can be compromised, either maliciously or negligently, by the provider.

It is impractical, in most cases, to assure systems of this complexity to any meaningful extent. It might be possible if a large enterprise with extensive resources and access to a full spectrum of IT security skills were negotiating a contract of sufficient scale that the cloud providers was willing to provide extensive logical and physical access to their assets, but the assurance would have a limited lifetime; without continuous audit, ongoing development by cloud providers will invalidate the assurance.

2.6 A brief history of hardware security

There have been many attempts to address security issues that arise from attackers having physical access to computer platforms; the intent is to ensure that the stakeholders in the system have some control over, or at least knowledge of, its design and implementation. The threats range from attackers who are able to influence the design of a computing device at the point at which it is built, to attackers able to modify devices in service, to attackers able to extract data in ways which bypass protections claimed by operating systems and other system software, to attackers able to deny access to or restrict the use of the system. It is worth examining the history of these projects, and looking at the wide variety of threat models that are implied.

What is follows is a brief history of some of the issues surrounding hardware assurance.

[5] In most card games, the randomness of a deck of cards is a requirement for honest play, and the cards are shuffled by the person that deals the cards. Although the dealer has to be trusted by the other players, they can reduce their exposure to a dishonest dealer by cutting the deck: dividing the deck into (usually) two parts and reversing them. Although not as effective as a full shuffle, it also offers fewer opportunities for malpractice, so is a means for someone to protect themselves to some degree against a dealer who is able to order the deck in their favour, without facing the accusation that they themselves have ordered the deck in their own favour.

My contention is when computers first started to be used the physical and design properties were not considered as risks, but the way in which computers were designed and used meant that a certain degree of security was inherent. As computers became a commercial product, from an industry predominantly based in the USA, other countries saw that there was a risk in this domination of the supply chain by one country, but the risk they were most concerned about was cessation of supply. The actions of the UK, in particular, show that there was a willingness to fund computer development even if the results were not strictly commercially viable, in order to maintain a capability to return to home-production of computers should it be necessary.

Following this period, as computers became much more widespread and embedded into our society, risks that we now see as computer security risks became more obvious, and the idea arose of building tamper-resistant and auditable modules to handle the most sensitive elements of a system. These modules usually had a fairly narrow range of functionality. Over time, this developed into the idea of having more general purpose hardware elements to secure systems without needing to build specific hardware for the particular task at hand.

2.6.1 Before the PC era

When computers were large, expensive and rare, they were housed in secure facilities and tended by large numbers of operators, system programmers and other staff. This large body of people, without whom the computer could not function, performed tasks that are today mostly the province of operating systems. Operators scheduled and loaded programs, often known as jobs, and ensured that the correct punched cards, tapes and disks were mounted. System programmers provided additional meta-data for jobs, which made sure the appropriate resources were available.

For computers that processed confidential data, all the staff with access to the machine would have been trusted, just as if they had been handling the data on paper kept in filing cabinets. The computers were rarely networked, and even if they were, the networking would connect the computers to facilities with similar security requirements. The computers that were not owned by governments were owned by large businesses, and the computer replaced pre-existing paper systems. The security arrangements around the computer mirrored the security arrangements around the physical records the computer replaced. It does not appear that at the time much consideration was given to what we would today recognise as “computer security”, but in large part that was because the issue did not arise: the computers were *de facto* airgapped from other computers, the buildings they were contained in were physically secure, the people who were using and controlling the machines were trusted employees.

However, what clearly was of concern was the supply chain for the computers themselves. A national computer industry was seen as a strategic requirement. The precise details may not have been widely known to the public — we now know that the main applications included cryptanalysis and the simulation of conditions in nuclear weapons — but there was no attempt to keep secret the fact that computers were being used in large quantities by the government and defence establishment.

The major western powers initially attempted to have their own computer design and manufacturing facilities, but while large American manufacturers were able to operate profitably, the European powers rapidly found that their indigenous computer industry was unable survive without heavy subsidy. Apart from manufacturing American designs under license, very few European design operations survived. The only manufacturer that managed to sustain a large, long-lived run of original mainframe designs was the UK's ICL.

International Computers Limited (ambitiously named, as in practice its business was heavily dominated by UK customers subject to pressure from the government to "Buy British") was formed by the nationalisation of ICT and English Electric. ICL inherited from English Electric the System 4, which was license-built version of the American RCA Spectra, which was in turn a compatible clone of the IBM 360 series [20]. But it also inherited from Ferranti, via ICT, the 1900 series [21], which was a completely self-designed mainframe. This continued to be built into the late 1970s; its successor, the 2900 series, continued into the 1990s. By then ICL had been acquired by Fujitsu; the requirement for an indigenous design facility seemingly no longer substantial enough as to justify the government subsidy. Close working with government clearly has long term benefits, as Fujitsu remains to this day one of the largest suppliers of IT services to the UK government.

The 1900 and to a greater extent the 2900 series were heavily influenced by research projects at Manchester University [22], and contained many innovative features taken from academic projects. Manchester actually built mainframe computers, culminating in the MU5, which was the progenitor of the 2900 series [23, 24]. It was able to do this because of the opaque funding arrangement between the government, ICL and the university which made parts of the computer science department in effect a national computer design centre.

The French Groupe Bull, the other large national company, rapidly ceased to do its own mainframe design and license-built and integrated designs from General Electric; when GE sold its computer interests to Honeywell, Groupe Bull followed. By the 1970s Groupe Bull's main business lay in installing large Honeywell systems [25]. It too managed to capture the government market in its home country: Groupe Bull still supplies super-computers to French nuclear research establishments, albeit now built from American components.

Other "national" companies, such as Siemens in Germany and Olivetti in Italy, either never entered or were unable to remain in the mainframe market, and moved into making mini-computers and desktop systems. Their governments were not, it appears, willing to underwrite the massive investment required to produce a national mainframe design.

The demand to have a national computer industry seems, in retrospect, to have been greatest in two countries, France and the UK, which were both nuclear powers and maintained a substantial cryptanalytic capability. It is interesting to read an 1971 CIA report on the sale of ICL computers to the USSR [26] to see the sensitivities, and why the UK might have felt that it was necessary to maintain an independent capability: it is clear that the Americans were willing impose very tight restrictions on the sale of American equipment to other nuclear powers, including a requirement that all the operators and system programmers be American and that American auditors have the right to examine all software run on the machine. Relations between the UK and the USA over cryptanalytic capabilities were always close, but France had never been part of the UKUSA or the "Five Eyes" (UK, USA, New Zealand, Canada, Australia) [27] sharing agreements. The UK had various political tensions with the USA over weapons data — Britain was forced to design its own fusion weapons, as the American government refused to release information, and the thermonuclear weapons tested by the British in 1958 were home-grown, although later there was a much closer relationship. France has always done its own weapons research and design, with little or no American or British input.

Although the decision to retain a computer design and manufacturing capability is of a piece with other industrial policy decisions of the period, it is hard to avoid the conclusion that it was in part about ensuring continuous access to modern, high-performance computing. This may have been a vain hope — the ICL 1900 series was under-powered by the standards of the time, and although it had many innovative features the 2900 was certainly not a fast scientific computer. Although both ranges were heavily used in academia (there was substantial pressure on universities to "Buy British") there were national facilities for specialist tasks shared between universities containing high-powered scientific computers that were in large part American. But had the American government, as it had done in 1946, excluded Britain from atomic weapon co-operation and started to restrict access to computers deemed too powerful or insisted on supervising their use in a way which compromised British interests, then the UK had a capability that could rapidly have been brought to a higher level. There was a value to having an independent design capability, not subject to oversight by a potentially unfriendly power.

Perhaps there was a concern that computers bought from other countries might have been in some sense untrustworthy, but the behaviour of purchasers implies it was not a significant concern: that Russia was purchasing equipment from ICL and CDC to use for applications in nuclear power (and in the 1970s, nuclear power was inseparable from weapons research) implies that they did not see those computers as a possible channel for espionage.

However, the desire for local designs was not just driven by a concern that access to modern equipment from elsewhere might be cut off. There were also local requirements which other countries were not willing to provide.

British telephone networks evolved in parallel to American networks, using somewhat different signalling protocols. In reality, the differences were not substantial enough to prevent harmonisation given the will to do so. But there was little incentive: telephone exchanges were large, complex, fragile mechanical systems, and there was no reason to ship these across the Atlantic. The divergence continued when electronic switching, and then digital transmission, were introduced, and by the 1970s the standards involved both different signalling technologies and different speech paths.[6] Again, however, it would not have been impossible to harmonise the standards, nor would it have been a major surrender of British sovereignty to adopt the American standards, but Britain took the decision to build its own equipment.

In part this came from a tradition of vertical integration that affected all large British concerns. The General Post Office and its successors had maintained a network of factories around the country making everything from telephone handsets to the red and white tents placed over manhole covers to prevent water getting into tunnels, just as each railway company not only designed and specified but physically built the vast majority of its own stock. But by the 1970s other industries had moved away from building all their own equipment. The pre-nationalisation railways, for example, had built almost all of their own stock, a practice that continued after nationalisation in 1947; however by the 1970s there was far more emphasis

[6] An American T1 is 1.544Kbps, consisting of 8000 frames of 193 bits each; 8 bits for each of the 24 speech channels, plus 1 one for management. A European E1 is 2.048Kbps, consisting of 8000 frames of 256 bits each; 8 bits for each of the 30 channels, but 16 bits of management. Because the E1 has more capacity for management traffic, all 64Kbps of each channel is usable; all the signalling travels in the management. The American 1 bit per frame is insufficient to handle the signalling, so an additional bit is taken from each speech path, leaving only 56Kbps usable. Similar differences exist between other standard units of transmission.

on operating the factories as commercial concerns, willing to sell to anyone.

But the government was willing to underwrite the design and manufacture of complex, sophisticated telephone exchange for which it must have been apparent that the export market was small. The companies involved were either nationalised (Post Office Telecommunications, later BT[7]) or were heavily involved in defence electronics (Plessey and GEC). It was only later that the reason for this became apparent.

The American AT&T company produced a range of digital telephone exchanges, culminating in the 5ESS (the fifth "electronic switching system") [28]. The UK used a variety of justifications, including the protection of UK jobs and technological know-how, exchange control and the aforementioned differences in standards — as might be imagined, elaborate claims were made within the industry to "prove" the benefits of the European specifications — in order to underwrite the design and manufacture of a British-only telephone exchange. The successor to the partially electronic TXE3 and the increasingly electronic TXE4 and TXE4A (TXE standing for Telephone eXchange Electronic) was dubbed System X. It was developed by the nationalised Post Office, later British Telecommunications, in conjunction with companies such as STC, GEC and Plessey, which had substantial connections to the defence establishment [29, 30]. It was the first all-digital exchange to be used in the UK: incoming calls were digitised upon arrival and then switched entirely digitally. It was also, a fact rather less well publicised at the time, far more capable in terms of interception and government control than any other comparable switch that could be purchased on the open market.

The System X is still widely used in the UK telephone network. It has the capability to intercept any voice call, giving no indication of the intercept to either of the parties to the call, and deliver the speech to any other point on the network. The AT&T equivalent offers this service, although not (at the time of its launch, at least) with the same ease of use. But System X also supports the euphemistically named "government preference working", which allows the operator to rapidly switch off service to users by a predetermined preference level, starting with the general users, then emergency services, and finally leaving only government services able to make calls. This was seen as a vital facility for civil defence, and was only ever provided on British-built exchanges. System X's export sales were negligible, in part because of the cost and complexity of these government-mandated facilities, but it is perhaps no accident that one of the few overseas installations outside the former Empire was in the USSR, where a System X exchange was interfaced to the Soviet network to provide service to hotels frequented by European business visitors to Moscow.[8]

Here, the motivation for a locally designed piece of hardware was to be able to add features that other vendors would not provide, and maintain some semblance of secrecy so that users of the system were not aware of these capabilities.

[7] The "Post Office Factories Division" was effectively part of the defence establishment. As late as 1990 they were being used to produce EMP- and fallout-proof communications equipment to enable RAF bases to operate in a post-nuclear Britain under the code-name BOXER. Many of the staff held appropriate clearances, and the sites had appropriate security facilities, so this work could be undertaken without much in the way of additional cost.

[8] Source: retired System X senior staff.

2.6.2 After PCs arrive

As computers became smaller and more affordable, they started to be used in environments which were not as physically secure. Rather than being operated by large groups of staff for whom it was their sole responsibility, they were operated as part of the office equipment by less trained, and perhaps less trusted, staff in less secure environments. Computers, in the form of Automated Teller Machines ("ATMs"), started to directly handle cash, which made them much more immediate targets for fraud: criminals could now get value from computers which was usable and untraceable. And from very low-technology beginnings, computers started to be networked to destinations other than just to other buildings of the same organisation, which opened up the scope for attack far more widely. In parallel with these changes, computers became more homogeneous: they were built by a smaller range of manufacturers, running a smaller range of software, sourced from a smaller range of countries.

When computers were in secure rooms, tended by specialist staff, with networking only linking them to similarly secure spaces, the distinction between data stored on computers and data stored as physical records was less important: both were secured by a physical boundary and the staff that guarded that boundary. Stealing from a filing cabinet and stealing from a computer were tasks of similar challenge. Even an attacker who was able to make changes to the design of the hardware would face a substantial problem in exfiltration: they would need to find some way of collecting the products of their attack. That changed with computers in office areas, or connected to networks, or equipped with cash dispensers.

It is surprising how little attention was paid to assurance of hardware. By this time, the pool of manufacturers was diminishing, and computers were powered either by commodity processors manufactured by a small number of semi-conductor companies (Intel, Motorola, to a limited extent National Semiconductor) or processors designed by individual computer manufacturers and fabricated for them by others (for example Sun's SPARC, fabricated by TI and Fujitsu, SGI's MIPS, fabricated by a variety of companies, IBM's various products culminating in the PowerPC range, fabricated often by Motorola). None of these products were examined formally, and even if they had been formalised as designs, it was impossible to trace this through fabrication. The customer was taking the correct operation of the processor on faith. There was research into the use of formal methods to construct processor designs, but no mainstream processors were ever designed using such techniques.

More prosaically, techniques which would make the hardware harder to attack were developed, but did not become mainstream. For example, disk encryption (either from within the operating system, or using special hardware in the disk controller) was available from specialised after-market vendors, but never penetrated outside the defence and government market. In part this was because of a critical lack of sufficiently high-performance ciphers: DES was too slow and insufficiently accredited, AES was in the future, and various ciphers approved by security agencies were themselves classified and not available for general use[9]

[9] One software-based product, used by the UK government, had a complex architecture in which a classified encryption algorithm, which was both accredited for encrypting a disk's worth of data under one key and fast enough to be make the system usable, was itself encrypted using triple DES, which met neither criterion but was regarded as secure enough to encrypt a few kilobytes of object code. To mount the disk, the operating system had to use key material to decrypt a part of the device driver, and then load other key material into that decrypted section in order to access the disk. Unsurprisingly, especially given the developers of the software were not cleared to see the encryption algorithm, the software was not robust enough for use outside very specialised environments.

But more fundamentally, customers did not really see the issue as requiring attention.

Several Unix vendors, notably Sun Microsystems and SGI, continued to ship workstations capable of operating with either no attached disk or a small disk containing only the operating system, so that all user data could be stored on a file-server in a more secure environment while the workstation was on a user's desk. Specialised file server vendors (Auspex, and later Network Appliances) arose to sell the other side of that equation. But outside universities (for whom the main benefit was cost, not security) and government agencies, this architecture failed to gain much traction: it was a poor fit to common procurement cycles, and required well-engineered networking that many companies did not have.

Paradoxically, although the processors' instruction sets became simpler with the arrival of RISC technologies, the relationship between source code and object code became more complex. The concept of the RISC processor is for silicon to run a simple instruction set "well" (reliably, quickly, consistently) with the complexity moved instead into the compiler and its tool-chain. This has many benefits, especially as computers become faster and the additional effort required to compile code is repaid by simpler hardware and faster execution on many machines. It is, however, often achieved by using long instruction pipelines; this substantially complicates the task of someone wishing to verify correct operation, because the complex instruction scheduling hugely complicates the number of states the processor might be in when executing a particular instruction. Together with the reliance on complex compilers the relationship between source code and actual execution is complex, and for large applications it is very difficult to, for example, confirm that secret data is not left visible in registers.

2.7 Overview of TPM

The Trusted Platform Module is designed to provide a root of trust for a computer system. With the help of a TPM, an observer can be shown a proof that a computer is in some particular configuration. It is for the observer to then verify that the configuration has any required properties. Additionally, the TPM provides mechanisms to store data that can only be retrieved when the system is in a specific configuration.

The TPM is a small microprocessor which has some additional functionality to help it perform its task. As well as the standard features of a microprocessor, it contains shielded key material, a random number generator, and a set of shielded Platform Configuration Registers, or PCRs, which are used to encode the configuration of the host platform.

Each PCR is a 160 bit register, a size chosen to match the output of the SHA1 hash function. The PCRs are protected from external manipulation by the design of the hardware.

The PCRs do not function as registers in a standard computer, which can be assigned arbitrary values are operated on with a range of arithmetic and logical functions. A PCR can only be manipulated in one of two ways:

- At power-on, the PCR is initialised to a value of all-ones, and is then set to all-zeros as the TPM is initialised.
- Thereafter, the PCR can only be *extended*. Extension consists of taking the current value of the PCR, appending some additional data, taking the SHA1 hash of the concatenation and replacing the PCR's value with the output of the SHA1 function.

$$\text{PCR}_{n+1} := \text{SHA1}(\text{PCR}_n || \text{data})$$

The TPM also contains some key material, some of it immutable, information about its own manufacturing, and a random number generator to support the generation of further keys. The random number generator is not standardised over different manufacturers, and concerns have been raised [31] about its quality and the ease with which generators from different manufacturers can be distinguished.

The TPM 1.2, which is our main focus of attention, does not expose any symmetric cryptography, in large part because to do so would have greatly complicated export of devices at the time the TPM was being designed. Rather it uses asymmetric cryptography, specifically the Rivest-Shamir-Adelman (RSA) algorithm [32, 33], to perform all encryption tasks. This restricts its performance (RSA is slower than symmetric ciphers on texts of any significant size, more so when run on a low-powered processor) to the point that it is no longer of concern when exported.

The TPM does support conventional secure hashing, specifically the SHA-1 hash algorithm.

The keys stored within the TPM comprise an endorsement key (EK), which can be used to sign objects to indicate that the TPM has generated or otherwise verified them, and a storage root key (SRK) which is used to encrypt other keys and data. The SRK can be regenerated as part of the process of taking ownership of a TPM; once this has happened, objects encrypted with any previous SRK can no longer be decrypted. The process of taking ownership of the TPM resets the SRK and provides a fresh set of authentication data with which subsequent accesses to the TPM are validated.

Other than the EK and the SRK, keys generated by the TPM are not stored on the TPM itself; rather, a composite object consisting of the unencrypted public part of the key and the private part of the key encrypted with the SRK are exported. It is the responsibility of the generator of the key to store this object, and to provide it to the TPM for loading into one of the small number of key registers when required. In situations such as sealing, the asymmetric nature of this encryption is not taken advantage of: the encryption and the decryption are performed by the TPM.

2.7.1 Booting

From power-on, a system's configuration is recorded in one or more PCRs by an alternating sequence of *measurement* and extension. Measurement uses already running code to produce a measurement, typically the result of hashing some larger amount of data. That measurement is then extended into one of the PCRs of the system. The PCR is tightly bound to the sequence of measurements that produced it. Even if the attacker has the power to perform arbitrary extensions, it is not computationally feasible to find a piece of data which, when applied to a PCR as an extension, produces a chosen value.

The system will need to contain a small piece of code in some form that is regarded as immutable (non-programmable ROM soldered to the main board, or a more robust encapsulation technique appropriate to the threat model) which performs the initial measurement of the BIOS and other low-level resources. Thereafter, each step in the process of booting the machine measures the next piece of code, extends an appropriate PCR with the measurement, and hands control onwards. Thus at each stage, the PCRs contain a representation of the state of the machine at that point. Someone wishing to confirm that a system is in a desired state can perform the same measurements of known-good versions of the code and hardware involved, and confirm that the value is correct.

2.7.2 Attestation

If the PCRs represent the state of the system, how can this be used by an observer?

Attestation provides a proof of the state of the PCRs. In outline, a challenge is made to the TPM, which includes a nonce. That nonce is packaged with the current values of the TPM, and the result is signed using a secret key known only to the TPM. The verifying party is given the public key, which is either signed by the manufacturer of the TPM or has a signature that can be traced to the manufacturer via some intermediate keys. The verifying party is thus told that the TPM's PCRs are in a particular state and that the TPM was manufactured by a known manufacturer (who is in this model assumed to be trusted), and the veracity of these claims is proved by the signature.

Attestation is of little use within a single platform. If the attacker controls the platform, then they can control the results shown to the user as a result of an attestation. That a program was able to unseal some sealed data is evidence that it is in the correct state, and a computation that can only have been made with that unsealed data (for example, a correct cryptographic operation using a sealed key) provides local assurance. Attestation becomes useful when multiple systems are networked, because one system can demand an attestation of another. If the attacker controls both systems then the results are not trustworthy; any situation where the attacker controls all the resources is problematic. But if one system is taken to be secure, that can be used to verify the software loaded on to the others.

2.7.3 Sealing and binding

The TPM can also be used to ensure that data is tied either to the platform, or to the platform in a particular state. The more general case is sealing. A piece of data is encrypted by the TPM, using a key known only to the TPM. Within the cipher-text is stored a set of PCR values, specified at the time of encryption, and a unique value private to the TPM called the TPM Proof. When the TPM is asked to decrypt the data, the result of the decryption is only released from the protection of the TPM if the PCRs specified are in the correct state and the TPM Proof has been correctly matched. This can be used to ensure that the decrypted data is only available to a platform in the correct configuration; the TPM Proof ensures the decryption can only take place on the same TPM.

If no PCR values are specified, any application can decrypt the data; nonetheless, the TPM Proof still has to match, which means that the data is securely tied to the platform.

2.8 Summary of chapter

This chapter has given the background to the concept of trusted execution, and set it in a historic context of attempts to control software execution by managing the design or manufacture of the hardware upon which it runs. We have seen the basic structure of the Trusted Platform Module, and how it controls the booting of a system and provides facilities for managing secret data used by the system.

Chapter 3

Prior work and literature survey

In this review, I intend to look at current work in these areas:

1. The effects on existing security architectures of a move into the cloud;
2. The use of hardware features mainly intended to support virtualisation in order to provide isolated or otherwise secured execution environments;
3. The use of virtualisation software to provide additional assurance about the execution environment of running programs;
4. Other software techniques that can be used to ensure that execution fulfils specified properties;
5. Formal analysis of the Trusted Platform Module chip which is at the heart of much of the other work;
6. Ways to bring the benefits of TPM to systems where the use of the actual hardware modules may not be practical;
7. Some work on trusted database engines.

Some of the work I refer to assumes that the programs which the data owner wishes to run are correct, in the sense of performing the task that the data owner intends and none other. With the exception of work on tainting and dataflow, which will be addressed in Section 3.4, the protection mechanisms are designed to prevent or detect the compromise of running code by external agents. Given the prevalence of frontal attacks on flaws in (or, less judgmentally, unintended characteristics of) running code — buffer overruns, timing attacks, replay attacks, other protocol attacks — and the apparent rarity of attacks on execution environments — modifying running code, modifying processor and device features — this emphasis may perhaps be seen as attacking the more tractable problems rather than the larger ones. However, there is a possible explanation: most of the attacks on execution environments are products of running in a virtualised environment, and as cloud systems tend to go hand in hand with virtualisation, it is reasonable for people to research the impact of new risks caused by the new context, even if ultimately those risks turn out to be smaller than the ones we already face. It is also worth noting that attacks on the execution environment are easily carried out by insiders with access to the hardware platform a virtualised system resides on, and those insiders (who are, of course, outsiders from the perspective of the owner of the data) are a key new risk in multi-tenant cloud provision.

3.1 Cloud security concerns

The cornerstone of much of the policy and governance of information security in recent years has been the ISO 27000 series of Information Security Management System standards [34]. Last revised in 2013, they set out a clear framework for assessing risk, implementing controls to mitigate and reduce that risk, and putting in place clear governance to ensure those controls are both present and effective. However, they very clearly date from an era in which physical control of computers was almost a *sine qua non* of security: large sections of the ISO 27002 code of practice [35] address physical, environmental and personnel issues. Unfortunately, a cloud infrastructure places these in the hands of third parties. Anyone with experience of an ISO 27001 process will likewise know that walking around data centre buildings looking at door locks, CCTV and waste disposal is a significant part of external audit activities. Although the standard does address third-party contractors, outsourcing and the use of external data-centres, it is clear from the context and the tone of the standard that you are expected to be running your primary infrastructure in facilities you control.

A group at Warwick University, working with a representative of a major contractor to UK government and large enterprises, have studied the implications of cloud systems to traditional information security policies [36]. They conclude, unsurprisingly, that although the basic management and governance processes are relevant to many different architectures, the detailed codes of practice and sample controls are less relevant to cloud than they are to systems physically controlled by the data owner. Going further, they also call into question the use of more rigorous assurance methodologies such as the CESG Tailored Assurance Service [37], on the grounds that these processes likewise rely heavily on physical and personnel security. They go on to say that "with high assurance techniques costly and time-consuming, the primary use of code analysis for flaws may take on an increased importance, especially given the previously outlined testing restrictions.".

The same paper also addresses an issue we shall return to later on. Virtualisation can, amongst other things, provide some protection against the risks of sharing infrastructure with other tenants, and also the some of the risks associated with "insiders" within the cloud provider being able to reach inside the data owners' virtual machines. However, although hypervisors have been brought to market which offer some measure of security accreditation, including Common Criteria [38, 39], it is notoriously difficult to deploy products in the configuration which matches that used to obtain the certification [40]. As the Common Criteria on offer are at a relatively low level and the products' main focus is on functionality and performance for the more general market, it is not clear how much benefit this brings to a cloud environment. An EAL 4+ hypervisor is not without advantages: it might mean, for example, that within a secure data centre the same hardware platform could be shared between different security domains at the same classification. But it is a long way from there to believing that it offers substantial benefits for a similar configuration in a cloud environment.

However, this is not to say that virtualisation cannot offer benefits for security when physical control is not available. We shall now turn to some research projects which use virtualisation in innovative ways to deliver specific security properties. Glott *et al* [41] consider at a high level the various attacks that cloud systems may be prone to which are different either in nature or scale to systems more tightly controlled by data owners. Their main concern is around multi-tenancy: that cloud systems bring previously unrelated customers into close proximity, possibly without their knowledge or control. A customer purchasing a cloud service from Amazon cannot, without perhaps the power they might obtain from spending millions of dollars per annum, specify with whom they will share infrastructure. And even were Amazon willing to write a contract that provided such guarantees, it is not obvious how it would either be enforced or audited. They also write about insider risks from cloud administration (a group of people outside the enterprise to which business systems have previously not been exposed) and the new reliance that an enterprise's systems will have on the cloud management platform, an additional component which introduces its own risks and unreliabilities. They point out that there are various ways that Trusted Computing Group technologies could address some of these issues, but (writing in 2011) they conclude that "run-time attestation solution still remains an open and challenging problem".

Ristenpart *et al* [42] make the concerns of Gott *et al* concrete. In a paper filled with practical details, they describe attacks on Amazon's EC2 infrastructure which allow the attacker to map and understand the physical systems underpinning the virtual services, cause new virtual machines to be instantiated in parts of the infrastructure that are beneficial to the attacker, then mount side-channel attacks to extract information from target virtual machines. Although the information they extract appears somewhat technical — processor utilisation, network rates, cache occupancy — it would certain qualify as a covert channel [43]. Although the mapping and placement techniques they describe can be mitigated, the basic contention that a general purpose operating system under the control of an attacker can measure things about the aggregate use of the platform on which it resides seems rather harder to deal with.

3.2 Hardware isolation

Both AMD (with Secure Virtualisation Mode, SVM) and Intel (with Trusted Execution Technology, TXT) have equipped some of their processors with capabilities generically called trusted execution. These have been used by several researchers, notably McCune at CMU, to provide secure execution environments for security modules. His work is based around the SKINIT and SENTER instructions, which provide a means to measure a piece of code, place the computer into secure state (bypassing memory management, interrupts, etc), set configuration registers to reflect the unique condition of the SKINIT/SENTER instruction have caused the code to be executed, and commence execution. This functionality is intended to be used to allow virtual machines to be launched securely on machines that have not previously been running a trusted operating system, so-called "secure late-launch" virtualisation. This uses dynamic measurement to check that a code being executed has not been modified: "measurement" refers to a hash of the code that is loaded, and "dynamic" refers to the fact that this is happening on a running machine at an arbitrary point in time, rather than only during boot.

The first of his sequence of papers is that on Flicker [44]. This system uses the infrastructure for late-launch virtualisation, but does not actually use it to launch virtualisation. Instead, the protection offered to the process of measuring and initialising a virtual guest operating system is used to directly execute a piece of application logic (a PAL). The intent is that this will be a small component of a larger application, but one which fulfils a distinguishable security function. By using hardware features which provide for measured code to execute without possibility of interruption, the PAL can unseal private key material with confidence that no changes have been made to its code. The result of running that code can be returned to its invoker with an "attestation" — a cryptographic proof from the TPM that the platform was in a particular configuration at the point the attestation was made, which in this case demonstrates that the PAL was running unaltered.

It is the unsealing operation that is at the heart of a PAL's security. The Flicker framework can be used to execute any PAL, and if the model of the attacker presumes that they have "root" or equivalent access to the operating system then the attacker can run a PAL of their choice. An attacker less capable than that is not interesting, as such an attacker can be foiled by use of the operating system's standard access control mechanisms. However, the measurement of the PAL is extended into a platform configuration record, which has been previously reset to zero. Sealed data is encrypted and linked to a specified set of PCRs, such that decryption only succeeds if those PCRs are in the state that was specified at the point of encryption. An attacker can run the PAL of their choice, but cannot unseal anything owned by other PALs. Conversely, a piece of data can be sealed, secure in the knowledge that only a specified piece of code can unseal it. Provided the PAL unseals some item of data essential to its execution, it cannot be modified: a modified version would have a different measurement, and thus would be unable to perform the unsealing.

However, this capability comes at a price. Firstly, the PAL runs to completion, with no provision for any sort of multi-tasking. The PAL, and nothing but the PAL, runs. This applies even if the machine has multiple processor cores; during the execution of the PAL, all cores other than the first are put to sleep. While the PAL executes, the machine will freeze so far as both the operating system and any users are concerned. Although this may be only a few hundred milliseconds, repeatedly locking out the clock interrupt for that period will cause issues with timekeeping, network throughput and disk scheduling. For a platform being used by multiple tenants, perhaps even with virtualisation, this is completely unacceptable.

Secondly, the state the machine is in during execution of the PAL is necessarily very restricted. The environment is akin to programming a micro-processor in the 1980s: as well as there being no access to operating system or library facilities (other than those statically linked to the PAL), customary protections such as read-only executable segments and non-executable stacks are entirely absent. So any failure of software to defend against buffer overruns, for example, will be magnified by the extremely basic execution environment.

Flicker proves the basic concept of isolating a portion of the system logic to improve its security. The same team produced a more sophisticated follow-up with TrustVisor [45], which avoids some of the failings of Flicker. Rather than using the facilities for secure virtualisation to execute a PAL, those same facilities are used to launch a hypervisor as intended. However, that hypervisor is not sufficient to support a general purpose guest operating system, with the hypervisor providing emulation of a complete system's screen, keyboard, networking, disk and other hardware. Rather, the hypervisor — totalling less than eight thousand lines of code, and therefore amenable to some measure of systematic inspection — only provides sufficient functionality to run modules akin to Flicker's PALs.

The critical distinction between TrustVisor and Flicker is that Flicker modules execute as though they were a virtual machine being initialised — a rare process where performance is not necessarily the first requirement — while TrustVisor modules execute as though they are a running guest operating system under a hypervisor, invoked by a trap or hypercall. Flicker modules have limited capability to maintain state from one invocation to the next, because they have no access to stable storage outside the TPM and their memory image is cleared each time. As any protocol involving more than one phase will involve more than one invocation, this is a major constraint. TrustVisor modules, by contrast, persist for as long as they wish, handling multiple invocations within the same context, and therefore able to store state in system memory (which may be a mixed benefit: TrustVisor modules are responsible for clearing their own security-sensitive side-effects).

There is a downside to this, however. Flicker has the problem that each time a PAL is invoked, it starts from the same state each time. But by starting from this single initial state, and executing with the rest of the system stopped, the opportunities are limited for an attacker to manipulate the PAL's state. As the attacker cannot execute a single instruction on the CPU, it can only manipulate the PAL's image by using debugging equipment, hardware modifications and other attacks that would not normally be considered practical against anything but the most exotic data. TrustVisor, and other experimental systems that rely on virtualisation, only have the same protections that standard virtualisation hypervisors offer. Although the secure hypervisor can make arbitrary checks on the PAL that is run, an attacker who is able to attack the hypervisor itself can do so with results that persist until the machine is reset, and can probably attack the persistent PAL with similar effects.

There is a literature covering techniques that would be effective against TrustVisor, or any other system which uses virtualisation as its basis. Duflot *et al* [46] examined the behaviour of the Advanced Configuration and Power Interface, a component which most Intel-based systems of recent vintage have within the chip set. This triggers an interrupt under certain conditions, or when requested by the processor, which places the machine into a state where all memory is accessible without limitations imposed by virtualisation or virtual memory translation. Any attacker able to install an interrupt handler would be able to run code which accesses the entire memory image, and TrustVisor as constructed appears vulnerable to this attack. Xu *et al* [47] show that the properties of the Level 2 cache in common processors can be used to exfiltrate data: they set an upper bound on the available bandwidth, but this is still sufficient to allow an attacker to extract valuable data, such as encryption keys, and this work has been developed further by Wu *et al* [48].

TrustVisor also still has the limitation that when a trusted module is running on one processor, all other processors are quiesced. This is identified in the paper as being an area for future study. Another area for future study, which appears substantially more challenging, is the fact that TrustVisor uses the system's virtualisation capabilities itself, and therefore cannot be used on a hardware platform which needs virtualisation to, for example, support multiple guest operating systems. There can only be one hypervisor, and that hypervisor is TrustVisor. Recursive virtualisation is not currently available on any mainstream hardware platform. This means that an environment which needs sophisticated features from a virtualisation product — migration, redundancy, thin provisioning — cannot also use TrustVisor, as the virtualisation layer of the hardware has already been claimed.

The team at CMU that produced TrustVisor have continued to work on trusted hypervisors, but their work has moved away from using TPMs as a root of trust. Their most recent work on “extensible and modular hypervisor frameworks” [49] focuses on small TCBs which are verified using the CBMC model-checker which is capable of working on assertions about a restricted subset of the C programming language [50].

3.3 Security benefits of virtualisation

One of the advantages of virtualisation is that it allows code to run as part of the hypervisor, which is ideally outside the influence of the code running within a guest operating system. This permits the hypervisor to monitor the guest, without the guest being able to tamper with or hide from the monitoring. This is a powerful technique for detecting changes to guest operating systems. Slightly confusingly, this technique of using a hypervisor to monitor the behaviour of guest operating systems is referred to as virtual machine introspection — the word introspection might make one think that the guest operating systems are looking at their own configuration, when in fact the hypervisor is examining the guests.

A recent example of work in this field was done at IBM’s Thomas Watson Research and Zurich Research organisations [51], building on prior work by multiple groups elsewhere (the paper includes an excellent survey of this). The IBM system functions without detailed knowledge of the internals of the operating systems whose health it is monitoring: specifically, it does not require Windows source code or knowledge derived from it in order to monitor the integrity of a Windows guest operating system.

It operates by first examining the memory image of a clean installation of the operating system — a significant practical problem, as it requires re-evaluation each time there is a change made, including adding new applications or applying patches to any part of the system — and building from that a set of valid pages that may be executed. Using knowledge that does not require source code, the operating system is identified and key data structures are examined and measured. Subsequent changes are then monitored and where necessary flagged as attacks.

This offers a clear benefit: a variety of root-kit type attacks will be detected. It is not explicit in the paper what action would then be taken, because it is not the detection technologies’ responsibility, but spotting a root-kit contamination is the first step to whatever remediation is seen as the best response. This detection comes as a very beneficial side-effect of the heavy use of virtualisation in cloud environments, but is not of itself something unique to the cloud: anyone, including a standalone user of a laptop, is perfectly able to boot an operating system under a hypervisor and monitor it for intrusion. And attacks leading to root-kit contamination are neither new, nor unique to the cloud. So although very interesting, and offering a sound means to protect a running operating system from some categories of attack, this paper does not address many of the security problems that are new in cloud environments. Specifically, as the detection technology exists in the cloud management domain, it provides no protection at all against attacks by cloud insiders; indeed, because it relies on the ability of the cloud management layer to “reach into” the guest operating systems, it might even be seen to legitimise access that might be used by malicious insiders, and therefore make identifying such misuse harder.

3.4 Other software techniques

So far, we have considered two uses of virtualisation technology: running small parts of an application in a restricted environment to improve assurance, and running general purpose operating systems under the control of hypervisors that can enforce particular properties. These techniques are of wider use than just the cloud; in both cases, they defend against a general class of attacks. It is interesting, therefore, to consider work which has been done to provide protection against errors in the translation of security-sensitive code from design to implementation.

A wide range of techniques are surveyed by Sabelfeld and Myers [52]. Their review is in large part concerned with non-interference, which is given a process-calculus treatment by Ryan and Schneider [53]. Non-interference is a strong property: a program running at a "low" security level must be unable to influence or derive information from the execution of a program running at a "high" level. Sabelfeld and Myers' review therefore does not address other properties which might be of interest, particular isolation between multiple programs running at the same level.

The PhD thesis of S. K. Nair of the University of Amsterdam [54] describes the use of a modified Java Virtual Machine to trace classes of data (for example key material, plain text, cipher text) through execution, and to impose policies on how those classes of data may interact. It uses trusted execution to ensure the use of the correct JVM and to provide remote attestation. Although a complete JVM is a substantial piece of software to include within a trusted computing base, and the dataflow analysis is unable to see beyond the scope of the JVM (it cannot, for example, control data once it has been passed into underlying operating system facilities or external libraries) this is a fascinating piece of work: code can be annotated to describe the policies that should apply to the flow of information, and this is then enforced by the JVM. It is worth clarifying that although Java is "compiled" with the `javac` compiler or its equivalent, this compilation does not translate source code into object code for a specific target processor; rather, the output of the compiler is a generic byte-code that is then either interpreted or, in some cases, further compiled into object code by a platform-specific virtual machine. This allows Java execution to be controlled more tightly than some other languages, because at no point is a self-contained object file executed without supervision.

Nair's work builds on that of Haldar and collaborators who added mandatory access control [55] and tainting [56] to the JVM. Their initial work also used Trusted Computing and remote attestation to the state of the Virtual Machine, which they described as "Semantic remote attestation" [57]. Their work, at least as described in the papers, provided a mechanism for dataflow tracing, for which Nair then developed a richer semantic in his work.

These pieces of work rely on the fundamental integrity of the Java Virtual Machine: they do not set out to model the semantics of the entire JVM, not do they substantially re-use previous work in this area. Java is a complex language to model, amongst other reasons because it contains a built-in threading facility. The JVM has been analysed formally, at least for a simplified version, in [58]. But in the dataflow and tainting projects, the JVM is assumed to behave as expected, and additional facilities are then added to improve the integrity of specific properties. Java source code can have meta-data added to specify required properties, and those properties are enforced dynamically by the augmented JVM. The correctness of the program in other ways, either at source level or in terms of its execution by the JVM, is not proved.

Nair's work also relates to that of Giambiagi [59], although perhaps surprisingly neither Giambiagi's 2001 thesis nor his work with Dam is referenced in Nair's later work. Giambiagi builds a theoretical model which allows mobile code to carry with it a proof of its correctness for a property he defined (in an earlier paper co-authored with Dam [60]) as "admissibility". This is a weaker version of non-interference, which permits (it might be better had they chosen the word "permissibility") data to be down-graded to a state where it no longer has requirements for being processed securely (for example, by encryption, where the cipher-text is not sensitive). He develops a theory to reason about systems with this property, and a practical system which allows proofs to be verified by systems that run the code.

These pieces of work rely on the code that is being executed being byte-code intended to be interpreted by an underlying virtual machine. The virtual machine acts as a trusted part of the system, controlling the execution of the byte code but itself assumed to be inviolate. Even accepting that the Java Virtual Machine is, indeed, immune to attack from executing Java code, the vast majority of code which is exposed to attacks from outsiders contains at least some portion that is executing directly on the hardware platform.

The two main means of protecting executing code are to (a) confirm that the code that is loaded for execution is indeed code that is acceptable to execute and (b) ensure that code once loaded is then not changed. These approaches assume that the best way to demonstrate that a piece of code has particular properties is to confirm the identity of the code. If a system can demonstrate that it is running a particular set of bytes then that is assumed to indicate that the program is correct: verifying that the code that is executing actually has the security properties that it claims to have is a distinct problem.

Modern hardware has features to prevent code being altered once it has been loaded into memory [61, 62]. This is generically known as "W $\oplus$ X", as memory pages can either be writable or executable, but not both. This is enabled with the NX ("no execute") bit on AMD processors and the analogous XD ("Execute Disable") bit on Intel processors, or the XN ("Execute Never") bit on ARM processors since Version 6. This feature is used by a range of operating system features that have varying names but equivalent functionality: code, once loaded and made executable, cannot be modified.

If that follows some code measurement or code-signing strategy to ensure the right bytes have been loaded, correct execution — in the sense of the execution of the expected instructions, rather than any statement about their functional correctness — should be ensured. Attacks which rely on being able to load a piece of code into memory and then use a buffer over-run or other "stack smashing" [63] technique to execute the code are substantially harder when the attacker cannot install the code they wish to run.

Unfortunately, whenever defences are built to defend against one problem, attackers find new weaknesses. In this case, attackers have discovered that they can often find within legitimate code small sequences of instructions, culminating in a return op-code, which perform a task useful to the attacker. By chaining these small fragments together, invoked via traditional stack-smashing techniques, the attacker is able to make progress. This attack is known as return-oriented programming [64, 65]. Work has been done to analyse and limit this attack, for example that by Davi *et al*. It is interesting to look at it in order to see the benefits that flow from running security-sensitive code within a virtual machine interpreter.

Davi's first piece of work [66] uses code instrumentation from dtrace [67], a powerful tracing and analysis tool that was originally developed as part of Solaris but has spread to other Unix-like operating systems, to examine the flow through object code. Using dtrace, his tool is able to spot instances where execution leaves a specific function without having ever invoked it: a sure sign that a sub-sequence of the function's code is being used for nefarious purposes. Apart from the small overhead of dtrace (single-digit numbers of instructions on each function call) this does not have a major performance impact, and on the face of it offers a strong degree of protection.

However, it transpires that possible attacks arise not only from using instructions that are also used by legitimate execution; attacks also arise from instructions that can be obtained if the text segment is examined as an array of bytes rather than as a sequence of instructions. By treating each byte as potentially the first byte of an instruction, rather than only looking at the instructions on their intended alignment, an attacker is able to give themselves a wider range of choices to execute. Davi's later paper [68] addresses this issue, and uses a more sophisticated instrumentation scheme to detect the execution of unaligned instructions which dtrace would not be able to see. This comes with the penalty of a two-fold performance degradation. Unfortunately, unless a security-critical region is run in some isolated container, the penalty applies to the execution of all code loaded into the reachable address-space of a process, not the execution of the security-critical portions of it, so will be paid in many cases for the entire execution.

The consequence of Davi's work to date is that an attacker who is able to perform a stack smashing attack will be weakened, but not prevented, by W$\oplus$X protection on executable text. He proposes mechanisms for protecting against the attacks that circumvent W$\oplus$X protection, one of which has a minimal performance overhead but deals only with a subset of the attacks, and another which imposes at worst a two-fold penalty (considerably less than the overhead of a virtual machine interpreter, for example) which deals with what at the moment appears to be the full spectrum of attacks. Note that the attacker has more reasons to use "unaligned" instructions that merely responding to Davi's 2009 paper: they also provide the attacker with a richer set of options. However, his work is focused on providing a mechanism to prevent an attacker from executing their own code, or at least their own choice from existing code, in the context of the target program; it does not provide any assurance that the legitimate execution of the code provides the correct properties.

Another area that has been the subject of extensive work is Property Based Attestation [69, 70, 71]. The intent is to allow attestations to carry statements about the properties that code can be assumed to deliver rather than simply statements about a secure hash of the object code that is running. The concept is outlined in Sadeghi's paper, and more concrete protocols and systems are advanced in later work by Kuhn, Chen and others. It is fair to say that the papers focus far more on the protocols and reasoning associated with properties than with what those properties themselves might be.

A somewhat related approach was proposed for commercial systems, known generically as posture analysis or network admission control. Examples included Cisco's Network Admission Control and Microsoft's Network Access Protection. A summary of some developments was published as RFC5209 [72]. The intent was to ensure that only systems running the correct combination of software patches, virus pattern files and other security-related connected to your corporate network, while systems not meeting those standards are instead connected to a remediation network which provides only access to the resources needed to bring the systems up to date.

Property-based attestation involves a system being able to assess the security properties that it itself can provide, and then using a secure infrastructure (usually a TPM) to provide cryptographic objects that attest to these properties. For it to be meaningful, the assessment must be of a similar integrity to the attestation.

In commercially fielded network admission control systems it was not: the agent that provided the assessment was just another program, as subject to attack as the things it was assessing. This still provided some practical benefits: the threat being addressed was machines that were currently uninfected but whose out-of-date software made them vulnerable. But it certainly did not generate strong, testable security properties. The Trusted Computing Group's Trusted Network Connect [73, 74] attempted to bridge this gap by using TPM attestation to report the state of Platform Configuration Registers, but that presumes a mechanism to reliably summarise all the attributes that are of interest into PCRs.

Flicker could at first sight address this issue, but as with other systems leveraging the TXT framework there would be practical problems. The Flicker Paper [44] talks about rootkit detection, where a PAL is invoked to check the running kernel for unauthorised changes. One might think this could be extended to checking for software patches and virus detector updates: a small, trusted piece of code which could confirm the protections in place within the larger system and then attest to that to a remote management station would be very popular with network administrators.

However, Flicker and TrustVisor have a fundamental difficulty: because they are running in isolated execution environments, they can neither interact with the file-system of other operating systems nor can they reliably read the memory image of running processes. Even if it were feasible to search the physical memory of the machine looking for the required pages, as Flicker could in principle do, a PAL does not have access to virtual memory translation tables, and even worse would not have the ability to page in data that is not in RAM. Looking for rootkits in a kernel is practical because on Linux and traditional Unix the kernel is not page-able;[1] in other operating systems such as Windows this is not true [78]. In any event the information required to analyse patch levels and virus scanners in depth is not usually available to the operating system kernel.

3.5 Formal analysis of TPM-based systems

Delaune *et al* [79] have studied a real-world application of the TPM, albeit with some simplifications. They consider Bitlocker, a Microsoft full-disk encryption product [80].

Bitlocker when used in conjunction with a TPM-equipped computer uses the TPM Platform Configuration Registers both to ensure that the correct software is run at each stage during boot and to protect the secret keys required to decrypt the contents of the disk from access by anything other than that correct software. This therefore provides a two-fold benefit: there is a chain of trust running from the pre-BIOS through to the launch of the main operating system

[1] Apart from the use of overlays in order to run Seventh Edition Unix [75] on a pdp11/34, Unix has traditionally neither paged nor swapped the kernel. There are exceptions, however. For complex historical reasons [76], Apple's OSX is founded on the Mach micro-kernel [77] and here even parts of the traditional kernel functionality may be paged out.

kernel which ensures that none of these components have been tampered with — the evidence of this is that the disk was able to be decrypted in order to load each successive stage — and the disk is also encrypted such that removing it and placing it in another machine will not work (to within the limits of the bulk cryptography, in this case by default 128-bit AES in Cipher Block Chaining mode). In this form Bitlocker is, with caveats, accredited for the protection of RESTRICTED assets without the use of external keying material [81], a major achievement for an off-the-shelf cryptographic product.

The authors analyse the steps that the Windows boot procedure makes when using Bitlocker, and how those steps alter the state of the TPM's PCRs in order to eventually unlock the keying material that is required. From a theoretical perspective, they provide proofs that although an attacker may have the power to extend the PCR an arbitrary number of times, that can be modelled as a "small" (ie, tractable) set of extensions without loss of generality. This makes practical the modelling of protocols which use platform registers and extension to protect key material, whether in a genuine hardware TPM or in a software analogue. These results are extremely important, because although the specification of the TPM is open and widely distributed, it was not developed from a formal model nor has a formal model been produced retrospectively. Delaune *et al* [79] does not model all of the TPM, but it does model some important functionality that I intend to rely on.

From a practical perspective, they provide strong evidence that the broad approach of Bitlocker is sound. Their paper makes some simplifications of the details of Bitlocker's operations, although those appear not to affect the regular operation of it, and of course it analyses the protocol and methods that Bitlocker uses, rather than any actual implementation: their proof is of the soundness of the approach, not of any particular piece of executable code. They show that Bitlocker as described can be implemented correctly to give the claimed security properties, not that some particular version of Bitlocker within a particular operating system release actually has these properties.

A. H. Lin's MEng dissertation [82] was supervised by Ronald Rivest of MIT, and uses the Otter theorem prover [83] and the Alloy modelling language [84] to examine the properties of the TPM. A key part of his work, in chapter 8, is his consideration of the implications of the TPM specification not being formally stated. He has a model of the TPM functionality in Alloy, which he has shown meets its stated objective (of course, as in all such models, there are simplifications, and the fidelity of the model to the reality of either the TPM specification or any particular TPM implementation is difficult to prove). He then makes small changes to his model to reflect what he sees as plausible errors that a chip designer might make, were their reading of the specification to be less than perfect. He shows that the security properties of the TPM are critically dependent on these areas of the specification. He does not say how many candidate changes he considered to find these examples, so we do not know from this work whether the whole specification is similarly brittle, or if these are the three examples he found after months of work.

One is at this point reminded of the incident in which a well-meaning programmer made changes to the OpenSSL entropy collection code in Debian Linux, thinking that their changes improved code quality by removing some compiler warnings. It transpired that the warnings were around code which gathered entropy from deliberately unpredictable places, and the result was that for several years Debian Linux key generation (and that of other Linux distributions which had taken the same changes) had less than sixteen bits of randomness. The security warning is bland enough [85], but the underlying bug report gives more of a sense of the scale of the problem [86]. Here a failure to understand the implications of some quite complex code was combined with a well-intentioned desire to improve quality and reduce maintenance requirements, to devastating effect. Lin's work serves as a similar warning to anyone implementing the TPM specification, either in hardware or, as we shall see below, software: the TPM specification contains things that may not at first sight appear necessary, but which may be critically important to the overall security of an implementation.

There have been other formal analyses of TPM and dynamic measurement. Gurgens *et al* [87] describe an analysis of the TPM API using a finite state automata, but the model fragment given does not appear to consider PCR state and the analysis in the paper is predominantly informal. Coker *et al* [88] focus on the analysis of TPM APIs for remote attestation, but their SAL[89] model is not yet publicly available.

Much of the previous formal work on dynamic measurement is based around model checking of the TPM and its associated processor facilities, rather than applications built on the facilities. Millen *et al* [90] use computation tree logic and the SMV model checker [91] to model the roles and trust relationships in a dynamic measurement system. The results confirm that the SENTER instruction is capable of distinguishing between correct and modified code.

Datta *et al* [92] propose a logic for reasoning about secure systems. Their work is extensive, and includes a modelling language. It focuses on the correctness of attestation protocols used by trusted computing and the integrity of code, rather than confidentiality properties.

Fournet [93] reasons in the cryptographic model about the security of sealed data. They construct an information-flow model which functions for distributed systems with varying levels of trust.

Arapinis, Ritter and Ryan [4] extend the process language of ProVerif to allow the modelling of global state in their StatVerif language. Their work allows the description of the TPM and dynamic measurement in a process language with state, and the automatic generation of Horn clauses from this model. However, ProVerif will not terminate on these clauses, because it attempts to search through an infinite sequence of possible extensions and resets.

To assist termination, Delaune *et al* [94] show a first-order model based on Horn clauses. This model focuses on PCR state and related API commands. They place an upper bound on the number of times a PCR needs to be extended between two resets. They show that if there is an attack using an unlimited number of extensions, then there is also an attack which requires some bounded number of PCR extensions. They also show that this upper bound is small enough to be tractable using ProVerif. Their model solves the non-termination problem caused by PCR extension; however, in some applications further termination problems are caused by multiple PCR resets as the result of multiple invocations of dynamic measurement. Bounding the number of PCR extensions does not solve the problem caused by multiple PCR resets, which this thesis characterises and solves.

3.6 Virtual TPMs

A TPM only has the capacity to service one chain of trust per platform configuration register. There is no equivalent to "virtual memory", in which every process gets its own address space ranging from zero to some defined maximum, which is then mapped onto system resources in a manner that is concealed from the process. If your process regards PCR17 as containing the state in which you are interested, then no other user of the hardware platform can share it. Although we can imagine a broker which shares the limited supply of PCRs out amongst a small number of processes, and remaps their conception of the PCR number, it would not in general work correctly. PCR values are used to seal storage, and the ability to unseal a piece of storage is one of the main ways to confirm that a specified PCR has the correct value: if the PCRs against which the storage was sealed have the wrong value, the unsealing will fail.[2] There is no mechanism to re-map the PCR set used in a sealing: the process attempting the unseal operation must have exclusive access to the precise set of PCRs that were used to perform the sealing.

One solution to this problem is to implement the functionality of the TPM in software, replicating the register file (and, if necessary, other parts of the TPM's storage) for each process that requires it. This re-uses the design of the TPM, which has been widely studied, and re-uses the API, which means that code that runs standalone on a system where it has access to a real TPM and run essentially unmodified making reference to a virtual TPM. The new risk, of course, is that the software implementation introduces new bugs, and as we have seen Lin's work [82] provides a powerful warning of this.

We can look at two papers that have considered implementing software TPMs. One we have considered already: McCune's paper on Trustvisor [45]. In order to be able to offer TPM-type functionality to a guest "operating system" (as TrustVisor operates as a hypervisor, any application module running under its control is, so far as the hardware is concerned, using the functionality intended for guest operating systems), Trustvisor has to emulate a TPM, or at least sufficient of a TPM for the purposes of the guest. He refers to this as the μTPM, and his Table 1 shows a very informative breakdown of the proportion of the code involved in doing this. Assuming (as seems reasonable) that the RSA implementation is mostly required in order to provide a virtual TPM, rather than to interact with the key infrastructure on the real TPM, approximately three thousand lines of code are required to emulate a TPM: nearly 50% of the code in the project. He does not describe how much of the library functionality is associated with the TPM emulation rather than either the hypervisor or the services extended to the guest, but it is reasonable to assume that at least a small amount is used by, even if it is not exclusive to, the TPM emulation. We can see that in a system designed to minimise the trusted computing base, and which places heavy reliance on a hardware TPM which is then emulated by a software implementation, at least half of the code is required to provide that emulation. As we have seen from Lin's work, the specification is subtle and the consequences of mistakes may be serious, so the proportion of the trusted computing base which constitutes the μTPM has to be a matter of concern.

Berger *et al* [95] adopt a similar strategy in order to make TPM functionality available in guest operating systems in a conventional virtualisation context. They provide a software implementation of the TPM, but which provides a split view. The lower half of the register file (which would typically contain the chain of trust for the BIOS and its immediate successors) is made available through the virtual TPM from the underlying hardware TPM, so those values are common to all running virtual machines (and read only as a consequence). The upper

[2] It will not simply yield the wrong result; because the TPM Proof value is folded into the sealing, the TPM itself can distinguish the case of unsealing with incorrect values, because the TPM Proof will not be correctly recovered.

half of the register file is available to each guest operating system as a set of read-write (or, more accurately, read-extend) registers, and those are therefore distinct amongst the multiple guests.

This work pre-dates the availability of TXT-type trusted execution, and therefore only considers the case of a 16-PCR TPM. Later TPMs add registers 17 through 24 to support dynamic roots of trust, but it is not clear what this functionality would mean within a guest operating system, which would not have access to the additional hardware features in the processor that are also required. It may well be, therefore, that for a guest operating system, the TPM 1.1 specification with 16 PCR registers is sufficient. There is no reason why a software implementation of a TPM cannot have any arbitrary number of PCRs, subject only to limits of memory and usefulness; however, the special semantics of PCR17 (set to zero during the execution of certain instructions, extended with the measurement of specific memory regions) will be very different, and embody substantially less trust, within a guest operating system.

The work also uses Xen para-virtualisation, rather than VMware-style hardware virtualisation. Xen [96] requires a small amount of change to the low-level components of the guest operating system, to simplify the task of the hypervisor and permit operation with good performance on hardware which has no virtualisation support. The price paid for this is that the operating system needs to be aware of the difference between real devices and virtual ones, and have appropriate drivers and other modifications. Hardware virtualisation in the manner of VMWare [97] is able to provide an unmodified operating system with the resources it expects to see without change, but also requires hardware support to achieve performance that is not substantially slower than native operation.

As a TPM is another device on the bus, emulating the non-TXT parts of the TPM's capability under a hardware hypervisor would be similar to providing an Ethernet adaptor or a CD-ROM drive. But the mechanism for sharing the device between multiple guest operating systems cannot rely on the behaviour of the guest operating systems allowing for the fact that it is sharing the resource; the guest operating system has to be presented with a virtual device which is equivalent in functionality when driven by software written only for the real device.

However, Berger *et al*'s work relies on para-virtualisation, because the virtual TPM made available to guest operating system is not identical, in functionality or command-set, to the real hardware. Although later versions of Xen introduced the ability to run the Windows operating system unmodified, they did this by introducing full hardware virtualisation, and the para-virtualised version of Windows XP referred to in the original Xen paper was never released; the vTPM does not, therefore, unable a virtual platform to support multiple Windows instances each secured with Bitlocker [98].

3.7 Trusted databases

It is interesting to look at work which permits a client to be given specified security properties for access to structured remote storage. One piece of work provide an SQL interface to the storage; another attempts to provide additional security properties for more general web services, which could include WebDAV file access.

TrustedDB [99] shows the difficulty of encrypting databases. An IBM 4764 secure coprocessor (a more recent iteration of the IBM 4758 [100]) is used to execute part of the resolution of each database query, so that sensitive components of both queries and database tables are only decrypted within the secure boundary. The SCP offers strong guarantees against physical tampering; however, the software that is being run is complex, and the decomposition of queries and storage into "sensitive" (and therefore processed within the SCP) and the rest (which can be processed on the far cheaper cycles available on the host process) is opaque to the client. The SCP has a mechanism to attest to the software that it is running, known as Outbound Authentication, which is analogous to the chain of trust and attestation in a TPM-based system. The SCP provides a far richer set of services than a TPM — it is a PowerPC processor, running embedded Linux — and yet the authors have to produce a scheme to allow the SCP to page using memory resources from the host computer in order to have enough computing power available to run their minimal secure portion. One cannot help thinking that once a system requires a Linux kernel and external paging support, it would almost be easier to place a tamper-proof boundary around a standard x86 platform. It is not at all clear that the additional assurance that is available for the SCP's Linux kernel provides much additional assurance for the overall solution, given the complexity of fitting the SCP into the overall system.

SafeWS [101] considers the practical security problems that web services face, and provides a useful response to the argument that TPM, TXT and other Trusted Computing concepts are excessively complex or address unrealistic threats. The authors are attempting to produce a minimised TCB (although they do not use the term) which can monitor and mitigate threats to a web server. Without the facilities of a TPM, they are rapidly involved in attempting to obfuscate a private key to conceal it within a binary. No matter how this is done, to an attacker with access to the binary and the matching source code (a reasonable assumption, especially given the strength of modern forensic tools in recovering comprehensible source code from binaries, a process documented in [102]) the recovery of the key is a relatively straightforward task.

They also claim to perform TXT-style measurement of both running binaries and those binaries' precursors or invokers. This is problematic: it is a long-standing issue on Unix that you cannot portably determine the name of the file that is currently executing within a process. environment, and doing it for another process is essentially impossible, particularly from user-space. It is possible to execute a file and then unlink it so it is not accessible from the filesystem, and this is explicitly permitted by the semantics of Unix This technique is routinely used to permit the updating of long-running system daemons, which continue to run the old, unlinked binary until the system is rebooted. In this case, a program is started with some name indicating the use of a particular file, that file is replacement with a different version, but the original version continues to execute.

Although well-intentioned, and containing some interesting ideas, this piece of work shows the difficulties involved in attempting to separate trusted from untrusted code without the benefit of a hardware facility to enforce the distinction. Once the attacker has to be assumed able to observe and alter the execution of the entire code-path, the defender's task expands to include some tasks which are insoluble on conventional Unix-type operating systems.

3.8 Summary of literature review

From these papers, we can see that there has been extensive work done to provide mechanisms for executing trusted pieces of software. These divide into three categories:

- Formal verifications of aspects of trusted systems using hardware roots of trust, but not encompassing applications running on these platforms;
- Systems that permit the running of applications with the help of a hardware root of trust, but lacking a formal verification;
- Systems that attempt to run trusted code without the help of a hardware root of trust, and which therefore encounter a range of problems when confronted by a sufficiently strong attacker.

Chapter 4

Applications for trusted execution

In this chapter we consider some applications for trusted execution.

These have been chosen from two sources:

- McCune's Flicker paper [44] contains a set of case studies of sample applications. Given this thesis sets out to verify the facilities used by Flicker, it seems worthwhile to verify the case studies' security properties as well.
- A novel variation on Diffie-Hellman key exchange which takes advantage of trusted execution to avoid ongoing reliance on random number generators. This construction is believed to be new; it is enabled by the properties of trusted execution, and offers some advantages over existing constructions. The bulk of this chapter focuses on the background, justification and mechanisms of this.

4.1 Certification Authorities

A critical component of the Internet's infrastructure is the certification authority. Their role is simple, but if not performed correctly leads to substantial failures of security.

A certification authority (CA), exists to confirm the provenance of public keys, contained in certificate signing requests, by using a small number of well-known keys. When the CA signs the request, it binds the name contained within the request to the key, and confirms (to a standard appropriate to the nature of the signature) that the key is genuinely owned by the entity identified by the name and other information inside the request.

The CA makes these signature to attest to its agreement with the claim that the certificate's subject (a domain, a person, a company) is truthfully represented. A user who is willing to trust the certification authority's judgement, and has a copy of the CA's public key that they similarly trust, can verify that the signature is genuine. If it is, they will then trust that the certificate belongs to the subject. In practice, users do not make judgements about CAs: a bundle of CAs with their public keys is provided with either the operating system or web browser that they are using, and it is rare for users to remove CAs from this bundle. Users may add additional CAs to, for example, accept keys signed by their employer.

There are many ways that CAs can fail, including document forgery, bribery and corruption by criminals (or, indeed, by state actors), malware attacks on their infrastructure and deliberate malfeasance by their principals. Most of these attacks are not "technical", in that they attack the human processes which lead up to the decision to sign the certificate, but they usually result in the same outcome: the CA's private key is used to sign a certificate that people relying on that CA would prefer was not signed. Research has been done into techniques which make it much harder to sign certificates covertly, so that the wider community can audit the behaviour of a CA [103].

One of the core protections that CAs should employ is the use of a hardware security module, to contain the private key that is used to make signatures, so that the key cannot be copied.

The functionality of HSMs is rather specific, however: their objective is to ensure the confidentiality of the key, and they are designed therefore to permit the formation of signatures on inputs without revealing the key. They may provide some audit facilities available to the owner, but there is neither a guarantee that the input is a legitimate thing to sign, nor any way for a recipient of a signature to verify that it was performed using an HSM, rather than using some other, less secure, key. Typically, the only assurance that a CA is actually using an HSM to store its private key is audit reports by human auditors, as the HSM itself does not provide any evidence of its use that can be externally verified.

Trusted execution offers some interesting alternative architectures. Rather that using a physical HSM, whose functionality is limited and whose security is difficult to audit, keys can be protected by sealing them against code which performs signing but also updates other structures which feed into transparency or other governance infrastructures. It would be possible, therefore, to use trusted execution to link the use of a key to the updating of external audit logs, and provide proof that as part of a certificate that the key was stored in such a system and the certificate generated by the linked code.

4.2 Password authentication

A common strategy for authenticating users is that a hash of a password is stored by the server [104, 105, 106] The client presents a password, which is hashed using the same algorithm, and the outputs compared. If the hashes match, it is assumed that the passwords matched. This strategy pre-dates the arrival of standardised secure hashes: for example, the early Unix mechanism described in [105] uses a somewhat *ad hoc* variation on the DES Data Encryption Standard [107] to form a one-way encryption.

A problem with this method is that the plaintext password is exposed to the full range of attacks on the server software, prior being hashed. Even if the transmission of the password from the client to the server takes place via an encrypted channel, the hashing is performed by the server which therefore has access to the plaintext.

Alternative methods for password-based authentication exist. Examples include CRAM-MD5 [108] and DIGEST-MD5 [109]. Their operation tends to follow a similar pattern:

1. The server generates a nonce and sends it to the client. This is to prevent a replay attack and need be neither random nor secret: a timestamp or sequence number is sufficient.

2. The client combines the nonce with the user's password (the details of this are specific to the particular protocol) and applies a secure hash function to the resulting string (again,

the details vary, and there will be iterations of the hash function, specific encodings of the result and other elements specific to the protocol). The result is sent to the server.

3. The server is able to perform the same computation, as it has both the plaintext and the nonce. If the result of its computation matches what it was sent by the client, the authentication succeeds. An observer is not able to deduce the password (assuming reasonable properties for the secure hash function) and is not able to reply the authentication (assuming nonces are not re-used).

Unfortunately, although this does improve matters if the opponent under consideration is only able to observe network traffic, it greatly increases the capabilities of an opponent able to observe activity on the server. Not only is the plaintext of the user's password exposed during the computation, a copy of it must be stored on a long-term basis in order to be available to hash with the nonce. An attacker able to compromise the authentication server can obtain a full set of passwords in plaintext.

As we will see later, it is possible to use trusted execution to attack this problem without requiring the storage of plain-text.

4.3 Random number generators

A key component in a system which makes use of cryptographic primitives is its ability to generate random numbers. Many cryptographic protocols require random numbers, and the security of individual runs of the protocol and of long-term keys depends on those random numbers being generated with the correct properties.

"Correct" is here something of a flexible term, as the requirements vary: sometimes the requirement is that a future random number not be predictable from some previously disclosed information, sometimes it is that a past random number not be discoverable from subsequently disclosed information, while in some cases all that is required is that the number not have previously been used by the participants.

There are also differing requirements for the distribution of the random numbers: it is important when generating session keys that no random value is more likely than any other, while in other cases where the nonce is visible to the attacker it does not matter if the random numbers are not evenly distributed.

These requirements, taken together, make the engineering of random number generators a challenging task [110, 111, 112]. There are examples of failure going back several decades [113, 114, 115]. Nonetheless random number generators will usually set out to provide random numbers that are usable for multiple purposes. In order to satisfy all requirements, a random number generator will need to have the following properties, although it must be noted that much of the earlier literature on random number generators focused on their suitability for use in Monte-Carlo methods of simulation rather than for cryptographic purposes; see, for example [116].

Unpredictability : given the complete history of all random numbers the system has previously generated, predictions of the next number are no better than chance.

Non-Retrodictability : given the subsequent history of all random numbers generated after a particular point, earlier numbers cannot be retrodicted with a probability better than chance.

Homogeneity : each possible output of the random number generator has the same probability of being generated by any one invocation.

Some protocols require that the same random number not be used twice. The requirement that the same output not be generated twice is usually not explicitly implemented: the assumption is made that for a suitably large output, the chance of duplication is negligible. In protocols where the use of the same nonce twice permits the recovery of long-term keys (for example, Elliptic Curve Digital Signature Algorithm [117]) this can present significant difficulties; it was the re-use of nonces (albeit in an extreme scenario where no attempt was made to generate a fresh nonce for each signature and the same one was used each time) that led to the recovery of the signing key for the Sony PlayStation 3 [118]. And as we will discuss, the assumption that the chance of duplicated output is small is a weak assumption in many embedded systems.

There are two fundamental classes of random number generators suitable for use in cryptographic protocols: those which use physical processes to generate randomness, and those which use cryptographic algorithms to produce a stream of data from some initial seeding material which is itself random.

Hardware Random Number Generators use some physical process that is not predictable, and generate numbers from this. Examples include thermal noise in resistors and zener breakdown noise in reverse-biased diodes or transistors, the jitter in simple oscillators and the intervals between neutron emissions in radioactive sources. These are often seen as being the gold standard of RNGs.

The basic mode of operation is straightforward. A physical process is set up which generates either unpredictable intervals between events or unpredictable counts of events in a fixed time period (the difference is usually one of implementation). This can be done directly from a process which generates discrete events, as in the case of radioactive decay, or indirectly by generating noise and sampling it in some way. This data is then either filtered to remove systematic bias and used directly as randomness, or is used as the seeding material for a pseudo random number generator.

Cryptographically Secure Pseudo Random Number Generators take an initial secret, and then generate a sequence of outputs which have the properties of random numbers, subject to the secret remaining secret and the intractability of the functions that the algorithm is based on. For example, the Blum-Blum-Shub algorithm [119] relies on the hardness of finding square roots in a finite field, while Fortuna [120] uses a block cipher chosen by the implementer, typically AES [121], and therefore relies on the security of that algorithm.

Although using a strong encryption algorithm or a computationally hard problem with a secret input will generate random bits, the disclosure of the initial secret will permit the calculation of all past and future values. Even this degree of security depends critically upon the cipher's resistance to a known-plain-text attack or the hardness of the problem in the face of increasing attacker capability.

In order to provide some protection against this, the usual construction (used in various form in the `/dev/random` implementations in Unix and Unix-alikes, and formalised in algorithms such as Fortuna) is to have a pool of bytes from which random data is extracted by hashing, with the pool itself evolving via further hash operations involving the injection of "entropy" from the environment. That "entropy" may take many forms, including hardware random number generators. The assumption is that stirring together a large range of entropy sources, of varying quality and varying assurance of quality, will produce a result which is "better" in some sense than any of the individual sources.

There is also an assumption that cryptographic primitives are strong; in this case, that hash functions are strong against pre-image attacks and produce an output which is indistinguishable from random noise. The hope is that the size of the "entropy pool" would make attacks infeasible, even if they could be performed with a probability of success that is substantially better than chance.

Pseudo random number generators which operate independent of assumptions about underlying cryptographic primitives have been promoted by agencies that might be expected to have a deep understanding of those primitives. Although later controversy finally ruined the already tarnished reputation of the Dual Elliptic Curve Deterministic Random Bit Generator [122] the basic premise of using a system which does not depend on other cryptographic primitives resisting attack seems attractive. Unfortunately, it had been conjectured [123, 124] that the construction, although sound mathematically, was vulnerable to an attack where the organisation constructing the constants for a given implementation did so in a particular way which made it easy to predict values. This concern, coupled with scepticism about the motives of the construction's main promoters, made it unattractive when it was originally standardised. Later revelations from Edward Snowden have tended to confirm the original doubts [125].

Although hardware random number generators are available on modern processors, they are still not widely used. Even on relatively common platforms which have such facilities, operating systems have been slow to provide access to them; for example, Apple had early access to Intel's "Haswell" processors which contain a hardware random number generator [126], but at the time of writing the OSX operating system does not make use of it. Even when such devices are available, most commonly either as part of the processor or via the fitment of a TPM, the hardware random number generators are used as additional sources of "entropy" for the system PRNG, rather than as standalone devices supply unmediated randomness (see, for example, the architecture of the Solaris cryptographic framework [127] or the necessity to use external user-space helpers to inject TPM randomness into the Linux kernel [128]).

What, then, is this "entropy"? The intent is that by stirring a continuous stream of unpredictable events into the state of the PRNG, the three properties set out at the start of this chapter can be delivered [129]. If the entropy is unpredictable, the state of the pool and therefore the random numbers extracted from it are unpredictable [130]. If the added entropy is not known to an observer, even an attacker who is able to obtain the complete state of the PRNG at some point in time will not be able to use it to recover later random numbers; although the primary defence against a stolen copy of the pool allowing the retrodiction of earlier random numbers is the hash functions' resistance to a pre-image attack, that the pool's state has also been modified by the addition of entropy does no harm.

However, unpredictable events that are neither observable or controllable by an attacker are not easy to obtain.

The Linux kernel is a typical example of a modern operating system likely to be used for secure applications [131]. Similar observations can be made of the Solaris and OSX kernels, the source of both of which is now readily available. Based on an analysis of the object code of its random number generator there is little reason to believe that Windows is substantially different [132].

There are four functions which provide data to be added to the Linux random pool, each with notable problems.

- `add_device_randomness()` incorporates device identifiers and other static data, which it is hoped vary from machine to machine even if they do not vary from boot to boot. It is likely that most of this data is available to an attacker (although there is one, rather anomalous call to this function from `posix_cpu_timers_exit()`, which may be more unpredictable).

- `add_input_randomness()` uses the timing of input events from directly connected keyboards and other "Human Interface Device" class peripherals such as mice. These are unlikely to exist in the typical server or embedded system, and even if they are connected will not be used significantly.

- `add_disk_randomness()` feeds timing information from the disk subsystem back into the pool. When `/dev/random` was first proposed, this was one of the main sources of randomness in a computer system: RAM was in short enough supply that most systems would page regularly, and disk seek times were unpredictable because of air turbulence within the disk enclosure [133]. Today, neither is true: RAM is plentiful enough that systems providing a constant service may not need to make frequent seeks over any substantial distance, and more importantly systems are using flash (solid state) storage which delivers predictable, constant seek times. By happenstance, `/dev/random` arose when the presence of a hard drive was almost inevitable: unlike Solaris and other enterprise Unixes of the 1980s, Linux was very rarely used in diskless environments, while solid state disks and plentiful RAM were in the future.

- `add_interrupt_randomness()` relies on the unpredictability of the precise time at which interrupts are delivered to the operating system or, if the hardware supports it, the value of a free-running high-speed timer at the time the interrupt is raised. This implicitly captures the arrival time of network packets, although those can be observed or controlled by an attacker. On any given system, it will require detailed analysis to determine whether the timestamps on these events relate to free running clocks (which may have many tens of bits of jitter if they are a poor quality oscillator subject to fluctuations in temperature) or to clocks conditioned by external references, which will be several orders of magnitude more stable. In virtualised environments the fidelity of the simulation of high-resolution timers is poor: timers are quantised more coarsely than would be the case in a real system.

Another source of data for the entropy pool is the previous state of the entropy pool. On Linux, although not always on other Unix derivatives, its state is saved at several points during boot and shutdown, so that it can be used to initialise the pool the next time the machine is booted. This has multiple problems:

- Many embedded systems have no re-writable disk, and boot from a fixed image. That fixed image may be re-writable, but not by the running operating system. The system boots from the same image each time, unless an update has been applied which, typically, replaces the entire image. They may be a separate piece of flash memory that holds configuration data, but this will be small and not suitable for storing the entropy pool.

  This means that entropy cannot be saved from boot to boot, so the sole source of entropy becomes the arrival of network packets during the current session of the operating system, together with any effects from temperature fluctuation. In some cases there may be a per-host secret available[1] but in general, a particular embedded system will behave very similarly to another instance of the same device, and an attacker able to observe or control the arrival time of packets (for example, by playing network trace back into a captive device) may be able to reproduce the entropy pool from which subsequent random numbers are generated. The open literature does not have convincing accounts of such attacks, but it would seem foolish to believe that research into them is not being conducted either by intelligence agencies or the non-academic security research community.

  Because of the way the standard Linux initialisation scripts are structured, the assumption is made that the restored state is different on each boot, even when either it is not present at all or is a fixed value. The random pool is re-seeded from the saved state early in boot, and random numbers, in some cases used for security-sensitive purposes (the portion of the ssh host key that guarantees forward secrecy, for example) are generated from the pool without waiting for further entropy to be aggregated.

- Even if the pool can be saved, the saved pool is updated infrequently. The usual approach is to add the saved entropy pool to the embryonic pool resulting from the early stages of boot, and then immediately save it back out again in the hope that it will be somewhat different. Given that most embedded devices lack a battery-backed real-time clock, this is a somewhat vain endeavour: the only thing that will differ from boot to boot will be the temperature-induced frequency variation in the master oscillator, as time stamps will all start from the same, predictable time. The pool is then not re-written until the system is shut down in an orderly fashion (ie, by the execution of the system "stop" scripts). In practice, most embedded devices run until they either crash or are power-cycled, and orderly reboots are rare (indeed, it is common for embedded systems to not expose any mechanism for performing an orderly shutdown). Adding a periodic

[1] Each controller in Fujitsu's ADSL product contains a kilobyte of data taken from an RNG, embedded in the system identity chip at the time of manufacture. This was specified "just in case": the idea being that if at some later stage there was a need to derive fresh keys, there would be means to do so using information not visible to an outside observer. Network control elements have trouble obtaining any sort of useful entropy even though they are managing a large number of network interfaces, because the "management plane" does not have direct sight of the individual packet arrival times. Source: the author specified it.

re-writing of the saved state would greatly improve matters, but does not seem to be common: it would be interesting to investigate why this is the case.[2]

So, to summarise, “some, if not many, RNGs are of unknown quality, and the likelihood is that the quality is lower rather than higher than desired.” When communicating with a remote system, it is very difficult, if not impossible, to determine the quality of the random number generation, and this poor random number generation may place you at a risk of harm. Let us take a brief diversion to an historic problem, and see the problems this issue can cause.

4.3.1 The TCP ISN problem

TCP [134, 135] makes use of “sequence numbers” to identify each byte in a stream of data. For a single connection, there are two sequence numbers, one for each direction. The initial sequence number in each direction is determined by the transmitter when the session is established, and the *n*th byte transmitted thereafter is identified as the initial sequence number plus *n* (with the minor complication that some protocol operations are considered to consume one byte in the sequence space).

The sequence numbers are agreed as follows:

1. The client sends a packet called a “SYN”, requesting a new connection (“SYN” is the flag in the header which indicates the desire to SYNchronise the sequence numbers). The packet contains *inter alia* the initial sequence number (“ISN”) the client will use, a “port number” on the destination machine to identify the service being requested, and a port number on the source machine to disambiguate multiple connections between the same two machines accessing the same service.

2. Assuming the server intends to accept the connection, the server acknowledges the SYN, and sends its own ISN (a “SYN-ACK”).

3. The client in turn acknowledges the server’s SYN, and the two parties then are able to communicate using the agreed sequence numbers (“ACK”).

This “three-way handshake”, “SYN SYN-ACK ACK” is common to all TCP sessions prior to data being transmitted.[3]

[2] One can speculate. For systems with flash memory, early flash devices were slow and had limited numbers of re-write cycles. A manufacturer of long-lived embedded devices might be reluctant to compromise the life and performance of a device for what they would see as a completely theoretical risk.

[3] In fact, the complete TCP state machine permits other sequences of packets to establish connections, and permits the piggy-backing of data onto some parts of this. In reality, the widely deployed TCP implementations do not support these: current TCP stacks either trace their origins to the Berkeley TCP/IP stack as distributed in BSD Unix [136] or implement a similar interface, and the API provided by the Berkeley stack does not permit some of the more esoteric connection establishment mechanisms. There has been recent discussion about “split handshakes” [137] where the SYN-ACK packet is split into a SYN and an ACK, but the scenarios where these packets arise are highly unusual. An implementation that attempted to, for example, send data with the third packet of the handshake would probably encounter difficulty in inter-working, and the packets would be targets for interception by intervening stateful firewalls or intrusion prevention systems. Similarly, tearing down a connection with the standard API requires the exchange of four packets in most situations, although the state diagram indicates it should be possible in three. The Berkeley “socket” API only supports a subset of the possible transitions, and after thirty years the protocol has come to be defined as much by the practical implementations as the original state diagram. Documentation for long-dead TCP/IP implementations such as that for TOPS-20 [138] shows similar limitations.

A problem with the generation of ISNs was noted in 1985, in the infancy of TCP [139, 140]. At the time, the nascent Internet was only populated by large, expensive computers, and the assumption was that those computers were in the control of honest administrators. This gave rise to a crude remote login protocol called "rlogin". Rlogin operated as follows:

1. The client runs a program which is "set uid". The Unix "set uid" mechanism permits a privileged user to install a program which, when executed, assumes the privileges of the owner rather then the invoker. The Unix kernel prevents non-privileged programs from using local ports numbered below 1024. `rlogin`, running setuid to root, binds its source port to 513, something an unprivileged program cannot do.

2. The client then connects to the `rlogind` server on a remote system, and simply sends the username of the user that invoked it.

3. The server knows the name of the client system (by looking up its IP address) and has been told the name of the user by a trusted party (`rlogin` on a trusted machine). It looks up the system/username pair, and if there is a match grants access.

Sequence space attacks allow this mechanism to be subverted. A weak attacker with the ability to inject forged packets into the network, but who lacks the ability to intercept packets other than those legitimately sent to them, can construct a SYN whose source address is forged to be one of the trusted systems and send it to the server they are intending to attack. The server responds with a SYN-ACK which the attacker cannot see.

The attacker does not know the server's chosen ISN, and therefore cannot immediately forge an acknowledgement of the SYN-ACK to establish a connection. However, if the attacker can guess the sequence number the server chose, it can still complete the TCP handshake and then send commands to be executed. It cannot see the responses to those commands, but may well not care. It might, for example, add its own identity to the access database, or maliciously delete all the files, or email a sensitive document.

Given the network performance of the era, the ISN was difficult to guess: the attacker had a 1 in 2^{32} chance of success, and would therefore on average require 2^{31} attempts. Today, any attempt to perform such a brute-force attack would be detected and prevented by even the most basic firewall. However, if the ISN of a connection is predictable given the ISN of previous connections, the situation is rather different: the attacker can make a normal connection from a source address they control, which in the normal course of events reveals an ISN, and then immediately follow it with an attempt to forge a connection from elsewhere.

In early TCP stacks, the ISN was not chosen in any way securely, and if an attacker could observe one TCP connection, they would be able to guess with a high probability the ISN of the next connection. They could make a connection from an untrusted machine to some public service (machines of the era would have supported `discard`, for example, which accepted a connection and discarded all data sent over it), observe the ISN used by that connection, and then use that to forge an rlogin request appearing to come from a trusted machine.

Similar attacks exist against the use of the source IP numbers for other access control mechanisms (for example, "permit unauthenticated email from local users, but demand authentication when they are remote"). There are a variety of mitigations [141], and it has long been standard practice for boundary firewalls to reject incoming packets whose source addresses appear to be local [142]. But the general problem still needs to be solved.

Later TCP stacks used a variety of techniques to make the ISN less predictable, but the results were very mixed. Research work by Michal Zalewski [143, 144] at the turn of the century showed that most of the then-deployed operating systems, including specialised networking OSes, had sequence numbers that were somewhat or entirely predictable. It is also worth noting that his report, in terms, describes the so-called "Kaminsky attack" [145] on DNS request identifiers that caused major concern nearly ten years later. A full solution requires careful use of random numbers at various points in the networking stack, and those random numbers have to be unpredictable to the attacker.

The problem, for a cautious user of network services, is that it is very difficult to determine the quality of the TCP stack with which one is communicating. For any individual user of a system, the ISNs of successive connections may appear to be random. However, an attacker who can observe many connections may well be able to predict sequence numbers, and use that ability to construct an attack whose real victim (in terms of information loss, perhaps) is a user, rather than an administrator, of the system. No matter how carefully a client constructs their initial sequence number, an undetectably weak server may compromise the client's security.

4.3.2 Diffie-Hellman security

Asymmetric encryption allows two parties to communicate without previously having shared a key in secret. However, it is slow and usually suffers from large amounts of "cipher-text expansion", where the encoded data is substantially larger than the plain-text. Therefore, a common technique is to use asymmetric encryption to agree a session key, which is then used with a symmetric algorithm which is faster (hardware implementations of symmetric block ciphers can run at the speed of the fastest networks) and does not suffer from expansion (the worst case is normally padding to an even multiple of a block, where a block is at most a few hundred bytes).

There are two common mechanisms by which two parties can use asymmetric cryptography to agree a session key.

Rivest-Shamir-Adelman (RSA) Encryption [32] Alice publishes a public key, while keeping the corresponding private key secret. Bob generates a random session key, encrypts it using Alice's public key and sends the result to Alice. Alice decrypts it, using the private key only she knows. Alice and Bob now share a session key, while no observer can calculate it. Correct use of the RSA algorithm is prone to error and requires some care to obtain all the security properties that might be assumed. There is a variety of attacks available, some of which are summarised in [146].

Diffie-Hellman (DH) Encryption [147] Alice and Bob agree, in public, the parameters of a multiplicative group. Each generates a random number which they keep secret, raises an agreed element within the group to that number and sends the result to the other party. By raising the value they receive to their own secret number, each party obtains the agreed element raised to the power of the product of the two initial secrets, while no observer can repeat the calculation. The shared secret is not usually suitable for use as a key directly, but can be deterministically processed to make it so [148, 149].

(Although most commonly described as operations in a cyclic group given by integers modulo a large prime, there exist analogues of each of these protocols using elliptic curves [150]. This does not materially affect the issues that I now describe).

In the case of using RSA, the generation of the session key itself is entirely in the hands of Bob. At the end of the protocol, the value Bob generated is known to both parties. If Bob is concerned about the quality of the key generation they can improve the quality or size of the key and have that reflected in the key used to encrypt subsequent exchanges. Alice's random number generation does not enter into the generation of the key; it may well be used to randomise the asymmetric encryption, but the ability to predict these values will not in general provide the attacker with an advantage.[4]

Unfortunately, using RSA in this manner has a major risk: if Alice's private key leaks or is otherwise compromised, all past communications are also compromised. An attacker who has the private key can decode the key establishment phase of any past intercepted session, recover the session key, and then decode the entire session.

This makes using DH attractive. Both parties generate a random number, and at the end of the exchange they have agreed a shared secret derived from both of those random numbers, which they can then convert into key material. Provided that both parties securely discard both the generated key and initial secret random number, subsequent compromise of the systems does not permit the attacker to decrypt past communications, a property known as perfect forward secrecy.

Unfortunately, an attacker only needs to be able to predict one of the secrets and combine it with the public information in order to learn the shared secret. During a DH exchange, each party performs a calculation involving their secret and information that is public to any observer; at no stage does either party learn or require the other's initial random number.

This presents problems for Bob, who is a careful participant, when talking to Alice, who is a widely used service, when Bob is concerned about Eve intercepting the communication. In this situation, perhaps Alice is a webmail service, Bob is a user with privacy concerns, Eve is a government agency.

On the one hand, Bob cannot be sure that Alice is generating random numbers honestly or competently (dishonest implies malice; incompetence can happen for generally benign reasons). Bob cannot even obtain the random numbers himself to make a cursory inspection that they are not, for example, simple timestamps or counters. Tests for randomness exist [151], but they will not distinguish between a pseudo-random number generator that is entirely deterministic once its initial seed values are set, and one which is using some external source of entropy. For example, a stream of timestamps could be hashed using a secure hash function, and the output would pass tests for randomness. As equivalence to a random oracle is a desired property for a secure hash function, failure to pass such tests would make it unacceptable as a secure hash [152]. But as in this scenario Bob does not have access to the random numbers Alice generates, even such tests are not available.

Alternatively, Bob may not be able to verify that his own device generates high quality

[4] An attacker able to guess the nonces used in RSA encryption can perform an offline attack to check guessed values for the plain-text, by encrypting the guess with the same nonce and the public key, and seeing if the result matches the cipher-text. However, this is equivalent to guessing the session key and checking the guess by decrypting a packet from the session, and as asymmetric encryption is substantially slower than symmetric encryption, it is this latter method than an attacker would use.

random numbers; no matter how much faith he has in Alice's generation of random numbers, the secrecy of the session key depends on Eve being unable to predict the random numbers that both Alice and Bob generate, and the failure of either of them to perform this task correctly fatally compromises the protocol.

4.3.3 A rôle for trusted execution

Trusted execution provides a mechanism to prove that a result was computed with a specific piece of code, using data that is linked to that code. Whether by attestation or by the use of a secret sealed to that code, or both, the client can be confident of the instructions that were executed to produce the result, and if there are security properties associated with that code can be confident that those properties hold for a particular instance of execution.

However, this still does not address the issue of random numbers. Trusted execution will, if it is delivered via a TPM, have access to random numbers from the TPM itself. However, there are concerns as to whether these are universally of high quality [31]. The random number generator is one of the areas in which the manufacturer of a TPM is free to use their own design, so the problem of assurance is compounded by needing to distinguish between each manufacturer and each iteration of that manufacturer's design.

Manufacturers are in any event reluctant to describe in detail the structure of the random number generators embedded in hardware, and even if they do, actually confirming that the silicon matches the design is a major undertaking. This might be within the capabilities of a major national facility with a large budget, but the complete reverse engineering of a chip design is a major undertaking even for such an organisation. It is not enough to confirm, for example, that a suitable hardware random number generator is fitted to the TPM: you would also need to confirm that the hardware random number generator is the source of random numbers actually used by the rest of the chip.

The same issue applies to other hardware random number generators: their properties are unknown and difficult to evaluate. As an example, there is an undocumented hardware RNG provided on the Broadcom BCM2835 used on the popular Raspberry Pi: Broadcom are only willing to release details of the RNG to other manufacturers and integrators under a non-disclosure agreement, and even with those details confirming that the silicon as supplied matches that description would be a major engineering task of limited value. Broadcom supply a device driver for Linux which permits access to the hardware RNG, but it is entirely opaque: bytes are read from a memory location, and the presence in the driver of comments such as `the initial numbers generated are "less random" so will be discarded` (relating to code controlled by the `RNG_WARMUP_COUNT` constant) and `double speed, less random mode` hardly fill the user with confidence.

TPMs and other hardware random number generators are therefore commonly used as an additional source of entropy for an operating system's existing facilities. The pragmatic reason for this is simple: applications deal with only one source of randomness, and do not need to be altered or reconfigured to cope with different platform capabilities. But there is a reasonable security argument as well. If the hardware RNG is sound, the quality of the operating system's random numbers is improved. If the hardware RNG is unsound in some way, no benefit accrues, but the system RNG is not made any weaker than it already was.[5] Software like `rngd`

[5] In principle, a defective random number generator which supplied endless, high bandwidth zeroes could dilute

exists to read data from a random number generator, perform some checking to ensure that the data passes minimal tests of randomness, and inject it into the operating system RNG.

Unfortunately, if the TPM random number generator is used simply as a source of entropy to seed a PRNG, the linkage between the TPM and the actual random numbers used by applications is weak, and any attestation by the TPM of its operation would be of very limited value. To prove to a remote challenger that the system RNG is, in fact, being seeded with the output of a hardware random number generator is extremely difficult, and begs the question of how the hardware random number generator would be verified.

In the next section, we describe a mechanism to decouple session key generation from TPM random number generators, while still maintaining the benefits of trusted execution.

4.3.4 Session key generation

In the previous chapter, we verified the operation of pieces of application logic, depending only on the TPM and a small set of facilities embedded in the processor. The particular example we used was based on the Flicker system, but the protocol that follows is not bound to that particular mechanism: any means of providing an oracle which two parties trust, and are able to prove to another part is actually in use, is sufficient.

Operation

- Assume a generator g in a multiplicative group of a finite field, or an elliptic curve group. Given g^x it is infeasible to compute x.
- Assume an oracle which has an associated secret x_a. The oracle takes an input y and returns y^{x_a} and a proof that the computation was performed by the oracle. The code for the oracle is public; x_a is secret to the oracle alone (this can be implemented using Flicker, Trustvisor, etc).

Alice is a server. Multiple clients Bob$_i$ wish to communicate with Alice. At least one Bob$_i$ is unwilling to trust the ability of Alice to generate random numbers that cannot be guessed by an attacker.

Alice uses the oracle once to compute g^{x_a}, which is public. Alice then generates distinct session keys for successive sessions with various Bob$_i$ as follows:

1. Alice chooses a new public n; a monotonically increasing timestamp or counter is sufficient.
2. Alice computes $y_a := g^{x_a}.g^n$ (which is equal to g^{x_a+n}), and sends it to Bob$_i$. This does not require use of the oracle as Alice already has g^{x_a}. In a conventional Diffie-Hellman exchange, Alice generates a random number which has to be kept secret until the exchange has been completed, and then discarded securely. In this case, there is no such random number: at this point, the only secret material that Alice has is the sealed x_a.

the randomness pool, driving out other sources. It is the responsibility of the designer of the aggregation functions to ensure that the failure of one entropy source to deliver credible data cannot cause this failure.

3. Alice receives y_b from Bob$_i$. For a key private to the two parties to be generated this must be the result of Bob$_i$ computing $y_b := g^{x_b}$ for some private x_b.

4. Alice computes key $K_a := {y_b}^{x_a}.{y_b}^n$, using the oracle to obtain ${y_b}^{x_a}$. Bob$_i$ computes $K_b := {y_a}^{x_b}$. $K_a = K_b$, because both are $g^{(x_a+n).x_b}$. The key is fresh because $x_a + n$ is previously unused. Secrecy depends only on the secrecy of x_a and x_b.

Advantages

- The oracle is only invoked once per exchange, and contains no mutable state. This has performance benefits (trusted execution mechanisms are often not fast) and also allows us to reason about its correctness using the techniques of the previous chapter.
- No Bob has to trust Alice's random numbers, only the secrecy of x_a (for which we have proofs) and their own x_b. The protocol is also backwards-compatible with standard DH where the random numbers are trusted.
- Session keys are still fresh, provided n is not re-used with any given x_a.
- Alice does not need to maintain the secrecy of initial random contributions to the protocol. g^{x_a}, g^n and n are all public, so Alice only needs to maintain the secrecy of the actual session key, rather than the components which go into its construction.

4.3.5 Attestation

In order for Bob to trust that Alice is generating keys honestly, it is necessary for Alice to prove to Bob that the shared key was generated by the operation of this protocol, rather than by some other mechanism which is not trustworthy. There is nothing we have so far described that prevents Alice from claiming to use this hardware-anchored mechanism, but in fact either performing a standard Diffie-Hellmen exchange with some other source of random numbers, or indeed using a timestamp or other value that is known to the attacker. On the assumption that the oracle is anchored to a TPM, this involves the use of attestation.

We have not as yet discussed TPM attestation. Attestation forms a signature over a set of PCR values in such a way that a verifying party can confirm that the PCR values are those that are expected, and that the signature has been made by a genuine TPM. The verifying party specifies a set of PCRs that are of interest, and supplies that information along with a nonce which ensures that the attestation is fresh. The TPM takes the specified PCR values and the nonce and forms a signature, which is returned to the verifying party for checking.

All TPMs have an Endorsement Key which is used to recognise a genuine TPM. Conceptually, attestation would work correctly if the EK were used to sign attestations: it is a key, unique to the TPM, whose private part is protected by the TPM and whose public part is signed to show that it belongs to a TPM. However, in the design of the TPM, it was recognised that if the same EK were used for all attestations, any concept of anonymity or unlinkability in the use of a platform with a TPM would be lost. Any application which needed to check the PCR values would need to request an attestation, and all the attestations from a single TPM would be signed with the EK. This would permit the linking of each and every use of the platform for any purpose, simply by looking at the public EK that was used to sign an attestation.

The TPM therefore forms the signatures on attestations using an Attestation Identity Key, or AIK, of which there may be many. An AIK is generated by the TPM, which then uses either a Privacy CA or Direct Anonymous Attestation to have that key signed as proof that it was generated by a device which has an EK. In the simpler case of using a Privacy CA, the Privacy CA receives the AIK signed by the EK, which it can verify as coming from a genuine TPM, and the Privacy CA then re-signs the AIK using its own key which is trusted by verifiers. A TPM can have multiple AIKs which cannot be linked either to each other or to the EK. Direct Anonymous Attestation is a more complex scheme which allows a similar outcome without the use of a Privacy CA.

For the purposes of our example, this additional complexity is not required: we are considering Bob as interacting with a public service that is not attempting to run multiple services while concealing that they share a common platform, and that the AIK certificate will be, indeed should be, well known. For consistency with published specifications we will notate the signature as being made with an AIK, but use of the EK would not reveal additional information. We will therefore not consider in detail the process by which an AIK is generated from a TPM containing an EK.

Forming an attestation: waiting for the request

The conventional use of attestation would permit the verifying party to confirm that a key had been generated using a specific PAL as an oracle. It would proceed as follows:

1. Alice and Bob generate a shared key, as previously described, with one modification. Prior to extending PCR17 with the fixed public constant that renders it unusable for any unsealing, the PAL extends it with a measurement (hash) of the key that was generated.

2. Bob sends Alice a nonce and a request for signed attestation to the value of PCR17, which reflects the execution of the PAL.

3. Bob will receive a signature over the nonce and the PCR value, the PCR value in turn being the successive extensions of the initial value, a measurement of the code, a measurement of the generated key and the fixed public constant. Bob verifies the signature, using a certificate for the AIK, and then verifies that the PCR value is as expected.

Unfortunately, this mechanism has several problems. It is slow, and will greatly reduce the throughput Alice is able to sustain. It requires that the PCR value remain unchanged until Bob has requested a nonce. As our intent is that the mechanism be backwards-compatible with clients that wish to interact with a standard Diffie-Hellman mechanism, this involves Alice waiting until she is certain that Bob is not going to make such a request before proceeding with the next client's request. That will reduce the number of connections the oracle can establish: the oracle cannot be used again until the timeout expires.

Forming an attestation: always doing it

Rather than Bob supplying a nonce for the purpose of forming the attestation, we can instead rely on data that has already been exchanged. As soon as the shared key has been calculated, Alice can immediately perform a `TPM_Quote` operation using Bob's contribution g^{x_b} as a nonce.

This can be requested from the TPM immediately the result is returned from the execution of the PAL. The attestation can be stored as part of the session state, and is securely linked to the session key, so Bob can at any time request it and confirm that session key was generated by the PAL.

Unfortunately, this mechanism also has a performance problem. `TPM_Quote` operations are not fast, as they involve substantial RSA operations. Although we have now removed the requirement to give Bob time to request an attestation, we may well in fact be worse off: if, as is likely, only a minority of sessions are checked in this way, the time and resources used by calculating an attestation for every session may be greater than the problems associated with introducing a small pause after each calculation before the next session can be initiated. Whether the TPM is sat idle so as not to disturb its PCRs or calculating a signature that no-one will use is irrelevant: in neither case can it do the useful work of calculating the next session key.

Forming an attestation: generate it on demand

If, as is likely, only a small proportion of connections have their credentials verified, it would be better to only generate on demand the attestation that proves that the key was generated using the correct mechanism. However, by that time the PCR values will almost certainly have been destroyed, either by the generation of the key for another session or by the use of the TPM for some other purpose.

However, we take advantage of the fact that the calculation performed by the PAL is deterministic: for the same input, it always gives the same output. The values n and g^{x_b} are assumed to be available to the attacker; the former because we have stated that Alice does not keep it secret, the latter because it is passed over a public channel. There is therefore no harm in storing in the session control block data already known to a potential attacker.

In this variation, Bob can at any time request an attestation, supplying a nonce of his choice. Alice then recomputes the session key, using the saved data from the protocol exchange, and immediately performs a `TPM_Quote` using the supplied nonce.

The price that is paid for this is that the complete session key computation has to be repeated, including the unsealing of the private component x_a, prior to the forming of the attestation. It would require experience gathered from real-world usage to determine which solution was best.

Forming an attestation: nonces

The standard mechanism for obtaining an attestation requires a nonce. That nonce is supplied by the verifying party and included in the object that the signature made by the TPM is formed over. This ensures that the signature is fresh (dates from after the request was made), which matters in some common scenarios for using attestations. For example, an attestation can be used to verify that a platform is in a particular configuration at, or after, a particular point in time. Without the nonce, an attacker can pre-generate an attestation, then alter the configuration of the platform. If challenged the pre-generated attestation would convince the verifying party, were it not for the nonce. The nonce, therefore, needs to be unpredictable and drawn from a range sufficiently large that an attacker cannot simply generate one attestation against each possible value. These are not minor requirements.

However, this does not apply in the case of using a PAL to generate a session key. The PCR is extended with the hash of the key that has just been generated. The attestation is therefore over a hash which includes a measurement of the key, and the key is known to the verifying party. The attestation cannot be calculated until the key has been generated, and the key cannot be generated until the initial contribution from Bob has been sent. Therefore, provided that the initial contribution is unpredictable, the measurement of the key ensures that the attestation can relate only to a run of the PAL that generated that particular key. Hence no nonce is required, or, alternatively, any nonce that is used can be known to the attacker in advance.

Forming an attestation: summary

- Although in general a nonce is required to form a trustworthy attestation, in this case one is not necessary. Any arbitrary data can be used, if the API requires it, but it is not required.

- The attestation can either be formed each time the protocol is run, if most of all of the runs culminate in a request for an attestation, or alternatively can be generated on demand by re-running the key generation with the same inputs. The attestation proves that the key was generated by the PAL, rather than by some other means, and therefore proves that it was generated by an entity that knows x_a.

4.3.6 Generating the initial secret

We have left until last a question that has lain unanswered throughout this section: where does x_a come from, and how does Bob know it was generated fairly? At some point, a random number generator must be used, either within the TPM or from somewhere else. The details of this will depend upon the assurance requirements of the users.

A random number generator will be improved by discarding substantial quantities of generated randomness both prior and subsequent to the generation of one random value. An attacker would have to predict or retrodict the state through a much longer sequence of states. Several such random numbers can be generated, without being revealed outside the TPM, and then combined in order to form the required secret. Unless the TPM's random number generator is fatally flawed, in which case its other properties (including sealing, as that relies on the generation of keys) must be suspect as well, this process will suffice; as it only needs to be executed once, the volume of randomness discarded on each side of the selection of the secret can be vast.

Alternatively, an external RNG can be used. The TPM is used to generate a key pair, and some external random process — perhaps one capable of being independently audited, or one which is more trusted by the user community — can be used to generate an initial secret, which is encrypted using the public key and passed to a trusted execution module which decrypts the randomness and seals it to the oracle. This has weaknesses in the chain of custody: the external RNG could reveal its selected number. But it permits the use of an RNG of a higher quality, or more assured quality, than that embedded in the TPM. This is not a complete solution, as the key will be protected by TPM's sealing mechanism, which in turn depends on the TPM RNG; a threat analysis of the problems raised by this will depend on the precise application and assurance sought.

4.3.7 Security properties

Diffie-Hellman key exchange requires only that the initial secrets, used as the exponent, be secret and unpredictable by the attacker; it does not place requirements on the relationship between successive secrets. This is not a well-explored area, however, and there may be weaknesses which mean that using a simple counter is not sufficient. If that proved to be the case, then a weak random number generator with a secret offset might resolve it.

4.4 Summary of chapter

We presented three applications for trusted execution. Two are well-established mechanisms taken from existing literature which we intend to verify. One is a novel variation on an existing algorithm which is enabled by trusted execution.

Chapter 5

An approach to trusted execution and its modelling

5.1 Background

5.1.1 Trusted computing

Hardware primitives

A TPM can provide evidence that a system is running in a particular configuration through the use of platform configuration registers (PCRs). The TPM only allows the PCRs to be reset to initial values by privileged instructions or at system reset. They can however be extended at any time. Extension involves hashing the current value of the PCR with another value; a PCR containing value p is extended with x by concatenating p with x, hashing the result and making this the new value of the PCR.

When a software module is loaded, its measurement, conventionally consisting of a secure hash of the code, can be extended into a PCR. A sequence of such extensions will result in a PCR value that is unique to the set of modules so measured, and the order in which they were measured, starting from an initial measure of trusted code loaded at boot-time.

Secret information can be sealed against a set of PCR values. The TPM encrypts the information using a key that the TPM control, and the TPM will perform the matching decryption if and if only the PCRs are in a specific state.

The TPM will also provide an attestation to the current state of the PCRs, by providing a signature over the PCR values using a signing key which is private to the TPM, but whose matching public part is widely available.

That the machine has loaded and executed the expected code can therefore be demonstrated in two ways: an attestation provides direct evidence of the state of the PCRs, while unsealing may provide indirect evidence by (for example) permitting the software to unseal and use a particular piece of key material.

Although in principle this provides a means to verify the integrity of code as it is loaded, only the loading of an exact sequence of modules will produce a particular PCR. Any change to any one of the modules will produce a different value. So in practice, this limits the measurement of the whole configuration to embedded or other strongly change-controlled systems: the process of booting a general purpose operating system from initial power-on to user login has too many variable elements, and changes too frequently. Additionally, because code is measured at the point of loading, an attacker who can access memory (for example, any attacker who is able to obtain supervisor-mode access to the operating system) can overwrite already-measured code without altering the PCR value.

To address these issues, version 1.2 of the TPM specification introduced the concept of a dynamic root of trust. Rather than tracing execution back to power-on, a privileged instruction can be used to reset PCRs in a new bank (17–22), initialised at power-on to all-ones (u_1), directly to all-zeroes (u_0). These *resetable* PCRs are then available to be extended with measurements from the reset onwards, rather than from power-on. This functionality is the basis for dynamic measurement technology, such as Intel's Trusted Execution Technology (TXT) [153] and AMD's Secure Virtual Machine (SVM) [154].

As the Intel and AMD technologies operate in a similar fashion, for simplicity we only consider the AMD SVM technology; the differences do not affect our argument in any material way. SVM provides a privileged command SKINIT, which creates a protected environment in which code can execute free from external influences. The unit of code to be executed with this protection is called a Secure Loader Block (SLB). When the SKINIT instruction is executed, interrupts and DMA are disabled to prevent access to the SLB and the resetable PCRs are reset. The SLB is sent to the TPM for measurement (using a hash function) and the result is extended into PCR17. The SLB is then executed.

Flicker architecture

Flicker [44] provides a mechanism for executing pieces of code with specific guarantees of privacy and integrity by making use of dynamic measurement. It uses Intel TXT or AMD SVM technology to measure and protect an SLB, which consists of initialisation and clean-up (SLB Core) and application functionality (Pieces of Application Logic, PAL); the measurement allows the SLB to unseal data which is private, while the protection allows execution using the results of that unsealing to happen without interference.

Flicker provides the framework for this mechanism to be used to execute PALs. A standard SLB template is provided (although that can be modified for specific purposes) which makes writing a PAL easier. Appropriate kernel services are made available to enable execution.

The standard SLB template provides some additional functionality. It provides a standard mechanism for passing arguments in to the PAL and passing out results. After execution of the PAL has completed, but before the SLB exits, the SLB measures the inputs and the outputs of the PAL and extends their value into PCR17, and then extends a fixed public constant into PCR17. One effect of this is to leave the PCRs in a state which is of no use to an attacker who is attempting to unseal confidential data; the extension with the fixed public constant leaves a value against which no data will have been sealed. It also provides a verifiable chain to prove the execution: PCR17 is left as the result of successively extending an initial u_0 with the measurements of the SLB, any inputs, any outputs and finally the fixed public constant. A later TPM_Quote operation on PCR17 provides an attestation as a verifiable link between the inputs, the outputs and the SLB for a verifier.

$$
\begin{array}{rcl}
\mathsf{sdec}(x, \mathsf{senc}(x, y)) & \rightarrow & y \\
\mathsf{adec}(x, \mathsf{aenc}(\mathsf{pk}(x), y, z)) & \rightarrow & z \\
\mathsf{fst2}(\mathsf{tuple2}(x, y)) & \rightarrow & x \\
\mathsf{snd2}(\mathsf{tuple2}(x, y)) & \rightarrow & y \\
\mathsf{fst3}(\mathsf{tuple3}(x, y, z)) & \rightarrow & x \\
\mathsf{snd3}(\mathsf{tuple3}(x, y, z)) & \rightarrow & y \\
\mathsf{trd3}(\mathsf{tuple3}(x, y, z)) & \rightarrow & z
\end{array}
$$

Figure 5.1: Additional Constructors and Reductions

5.1.2 Verification tools

ProVerif

ProVerif is an automated theorem proving tool, which accepts both Horn clauses [155] and ProVerif process calculus [156] language as its input. ProVerif translates its input into Horn clauses if necessary, and then determines whether specified queries are reachable from that initial knowledge. The ProVerif process calculus language is similar to the applied pi calculus [157] and is reviewed in the first part of Figure 5.2; the details of the syntax can be found in [156].

ProVerif will check security properties by assuming that an arbitrary adversary process is run in parallel with the supplied processes. ProVerif allows processes to send terms built over a signature including names and variables: these terms model the messages that are exchanged during a protocol. Cryptographic operations are modelled by constructors and reductions such as $\mathsf{h}(x, y)$ for the hash result of the concatenation of x and y and $\mathsf{md5}(x)$ for the md5 value of x. Other constructors and reductions that we use are shown in Figure 5.1. The first two reductions model symmetric and asymmetric decryption of messages in the usual way. The last five reductions model the tuple actions: they allow us to store related items in 2- and 3-element tuples, and extract the first, second and (where relevant) third items.

StatVerif

StatVerif [4] extends the ProVerif calculus language with global state. In order to model the global state, the final five processes in Figure 5.2 have been added. The process $[s \mapsto M]$ represents a state cell s that has the initial value M. Process read s as x; P reads the value of the cell s (calling it x); process $s := M$; P assigns M to s; lock; P obtains exclusive access to the state cells and unlock; P releases the lock on the state cells. In each case, the process continues as P. As a part of StatVerif calculus syntax, some minor additional restrictions are required; details are available in [4].

The application StatTrans is used to translate StatVerif process calculus into corresponding Horn clauses which are then verified using ProVerif.

Horn clauses

Both ProVerif and StatVerif work by translating the process language into Horn clauses [158, 159]. Like ProVerif, StatVerif uses two special predicates att and mess to represent the knowl-

$M, N ::=$	terms
x, y, z	variables
a, b, c, k, s	names
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
0	nil
out(M, N); P	output
in(M, x); P	input
$P \mid Q$	parallel composition
$!P$	replication
new a; P	restriction
let $x = g(M_1, \ldots, M_n)$ in $P$else$Q$	destructor application
if $M = N$ then P else Q	conditional
$[s \mapsto M]$	state
read s as x; P	read
$s := M$; P	assign
lock; P	begin locked section
unlock; P	end locked section

The terms and the processes up to and including the conditional are from ProVerif [156]; the remaining processes are StatVerif [4] additions.

Figure 5.2: The StatVerif calculus

edge of the attacker and the messages exchanged between processes. Unlike ProVerif, the numbers of the parameters to the two predicates depend on how many state cells are defined. In this paper, we only consider the case of one state cell; models with more than one state cells can be transformed into models with one cell by tupling. With one state cell, StatVerif translates process language into Horn clauses with the binary predicate att and the ternary predicate mess. The fact att(M_1, M_2) means that the attacker has the value M_2 in the state M_1. The fact mess(M_1, M_2, M_3) means that the message M_3 has appeared on the channel M_2 in the state M_1.

5.2 A model of trusted execution

In this section, we model the functionality of dynamic measurement and TPM. We introduce a modelling language, TXML, along with its semantics. We describe a theorem that allows us to bound the length of traces such that models can be verified with available tools. To illustrate our model, we introduce a simple Flicker-based decryption oracle. We then formally define our model, using this decryption oracle as a running example.

5.2.1 Simplifications and abstractions

We simplify both Flicker and the TPM, but in ways which grant additional powers to the attacker. If the attacker cannot compromise our simplified system, he also cannot compromise the full system. We omit TPM authdata, normally required to authenticate access to the TPM. This is a specific case of our assumption that the attacker has access to all the data and facilities of the computer. We assume that all keys required by the TPM have been created and are permanently loaded in the TPM. For simplicity, we consider a TPM with only PCR17, as it is sufficient to model the facilities we are using. Finally, as we are analysing the secrecy properties of the system rather than the correctness of attestations, we omit the SLB's extension of its input and output into PCR17.

5.2.2 An introductory example

Our introductory example is a decryption oracle implemented as an SLB. Any object supplied to the SLB is decrypted using symKey and returned to the user. The intent is that symKey is never revealed outside the oracle. In order to prevent the oracle being used to decrypt arbitrary cipher-texts, a check is made for the presence of a pre-arranged tag. Our concern is the privacy of the decryption key in the face of a powerful attacker, so the tag is made available to the attacker.

We represent the oracle SLB as slbD which receives as input the sealed symKey and an object which is to be treated as cipher-text. slbD attempts to unseal symKey. If that succeeds it uses symKey to decrypt the cipher-text. If that succeeds, and the tag is found in the plain-text, the plain-text is output. Finally, the fixed public constant fpc is extended into PCR17 to revoke access to the secrets.

Assuming the correctness of the TPM unsealing function, symKey can only be unsealed when PCR17 value is $\mathsf{h}(\mathsf{u}_0, \mathsf{slbD})$. Due to the operation of dynamic measurement, PCR17 can be set to $\mathsf{h}(\mathsf{u}_0, \mathsf{slbD})$ only by the execution of slbD with an SKINIT instruction. slbD itself uses, but does not output, symKey. Consequently, symKey is not exposed outside the decryption oracle.

5.2.3 Trusted Execution Modelling Language

We present a formal model of trusted execution. This models a machine equipped with a simplified TPM which offers sealing, unsealing, resetting, reading and extension of the PCR. The machine also allows users to execute programs in the protected execution environment provided by dynamic measurement.

We introduce the syntax and semantics of a language, TXML[1]. We then model the decryption oracle example and define transformations from one strategy, which is used to model a list of commands in TXML, to another strategy. Finally, we prove some interesting properties of the transformations under certain conditions.

Syntax

Suppose sets N of names including 0, 1, $\mathsf{sk}_{\mathsf{Srk}}$, tpmPf, $\ldots$; V of variables with typical elements x, y, z, $\ldots$. The letters u, v, w, $\ldots$ range over V $\cup$ N. Typical constructor function symbols, including at least $\mathsf{h}/2$, $\mathsf{senc}/2$, $\mathsf{aenc}/3$, $\mathsf{pk}/1$ and $\mathsf{measure}/1$, are represented as f. These represent respectively the SHA1 hash function, symmetric and asymmetric encryption, the derivation of

[1] Trusted eXecution Modelling Language.

statement ::=
$x := \mathsf{f}(u_1, .., u_n) \mid x := \mathsf{g}(u_1, .., u_n) \mid$
$x := \mathsf{seal}(u, v) \mid x := \mathsf{unseal}(u) \mid$
$\mathsf{extend}(u) \mid \mathsf{reset} \mid$
$\mathsf{check}\ u = v \mid \mathsf{skip}$
command ::=
statement |
$x := \mathsf{SKINIT}\{\text{list(statement)}; \mathsf{rtn}\, u\}$
$x := \mathsf{SUBR}\{\text{list(statement)}; \mathsf{rtn}\, u\}$
program ::=
list(command)

Figure 5.3: Syntax of TXML

a public key from a private key, and the measurement with SHA1 of a fragment of program text. Typical destructor function symbols, including at least sdec/2 and adec/2, are represented as g. These represent respectively symmetric and asymmetric decryption. We define terms t_1,t_2,... over V ∪ N in the usual way. The rewrite rules $t_1 \to t_2$ where t_1,t_2 are over variables include at least $\mathsf{sdec}(x, \mathsf{senc}(x, y)) \to y$ and $\mathsf{adec}(x, \mathsf{aenc}(x, y, z)) \to z$, linking associated encryption and decryption operations.

The syntax of TXML, used to describe programs running on the machine, is shown in Figure 5.3 and described as follows:

- $x := \mathsf{f}(u_1, .., u_n)$ and $x := \mathsf{g}(u_1, .., u_n)$ are applications of constructors and destructors.
- $x := \mathsf{seal}(u, v)$, $x := \mathsf{unseal}(u)$, $\mathsf{extend}(u)$ and reset are the actions of sealing and unsealing data, extending a PCR and resetting the TPM.
- check $u = v$ confirms the equality of two terms.
- skip is the null action.
- $x := \mathsf{SKINIT}\{\text{list(statement)}; \mathsf{rtn}\, u\}$ is a list of statements that are executed protected by SKINIT, which return u.
- $x := \mathsf{SUBR}\{\text{list(statement)}; \mathsf{rtn}\, u\}$ is analogous to $x := \mathsf{SKINIT}\{\ldots; \mathsf{rtn}\, u\}$. However, it does not modify the PCR value. SUBR is not available to the programmer or the attacker, and cannot appear in the definition of an SLB; it is present in TXML as a technical convenience that we use during transformation.

Semantics

We will be using TXML as the basis for our proof that we can bound the number of SKINITs used by the attacker. We therefore need to define its semantics. We consider configurations (K, p), where the attacker's knowledge base K : V → ground terms is a partial function and p is a ground term. We assume K is extended to V ∪ N, as the identity function on N. The initial configuration is $(\mathsf{K}_{init}, 1)$. Transitions between configurations are labelled by programs. We

$$(\mathsf{K}, p) \xrightarrow{\mathsf{skip}} (\mathsf{K}, p)$$

$$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{f}(u_1,..,u_n)} (\mathsf{K}[x \to \mathsf{f}(\mathsf{K}(u_1), .., \mathsf{K}(u_n))], p)$$

$$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{g}(u_1,..,u_n)} (\mathsf{K}[x \to t], p)$$

if $\mathsf{g}(t_1, .., t_n) \to t_{n+1}$ is a reduc
and $\mathsf{K}(u_i) = t_i\sigma$ and $t = t_{n+1}\sigma$

$$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{seal}(u,v)}$$

$$(\mathsf{K}[x \to \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, \mathsf{K}(u), \mathsf{K}(v)))], p)$$

if $\mathsf{K}(u) \neq \bot$

$$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{unseal}(u)} (\mathsf{K}[x \to t], p)$$

if $p \neq \bot$and$\mathsf{K}(u) = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, p, t))$

$$(\mathsf{K}, p) \xrightarrow{\mathsf{extend}(u)} (\mathsf{K}, \mathsf{h}(p, \mathsf{K}(u)))$$

if $p \neq \bot$

$$(\mathsf{K}, \bot) \xrightarrow{\mathsf{extend}(u)} (\mathsf{K}, \bot)$$

$$(\mathsf{K}, p) \xrightarrow{\mathsf{reset}} (\mathsf{K}, 1)$$

$$(\mathsf{K}, p) \xrightarrow{\mathsf{check}\ u=v} (\mathsf{K}, p)$$

if $\mathsf{K}(u) = \mathsf{K}(v)$

$$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{SKINIT}\{\mathsf{L};\mathsf{rtn}\, u\}} (\mathsf{K}[x \to \mathsf{K}'(u)], p')$$

if $(\mathsf{K}, \mathsf{h}(0, \mathsf{measure}(\mathsf{L}))) \overset{\mathsf{L}}{\Rightarrow} (\mathsf{K}', p')$

$$(\mathsf{K}, p) \xrightarrow{x:=\mathsf{SUBR}\{\mathsf{L};\mathsf{rtn}\, u\}} (\mathsf{K}[x \to \mathsf{K}'(u)], p)$$

if $(\mathsf{K}, \mathsf{h}(0, \mathsf{measure}(\mathsf{L}))) \overset{\mathsf{L}}{\Rightarrow} (\mathsf{K}', p')$

Figure 5.4: The relation $\xrightarrow{C}$

assume side conditions that $\mathsf{K}(x)$ is defined and r is non-deterministically chosen whenever we write $\mathsf{K}(x)$ and r. We also assume an injective function $\mathsf{measure} : \mathsf{TXML}^* \to N$ taking a sequence of TXML commands and returning a name.

$(\mathsf{K}, p) \xrightarrow{C} (\mathsf{K}', p')$ means that when the knowledge base is K and the PCR value is p, and the attacker performs command C, his new knowledge base will be K', and the new PCR value will be p'. The relations $\xrightarrow{C}$, which relates to individual commands, and $\overset{S}{\Rightarrow}$, which relates to sequences of commands, are defined in Figure 5.4.

Figure 5.4 shows the semantics of each command in TXML. Let S be a TXML program. The relation $\overset{S}{\Rightarrow}$ is defined as $(\mathsf{K}, p) \overset{\emptyset}{\Rightarrow} (\mathsf{K}, p)$ in the case that S is the null program. In other cases, $(\mathsf{K}, p) \overset{C;S}{\Longrightarrow} (\mathsf{K}', p')$ if $(\mathsf{K}, p) \xrightarrow{C} (\mathsf{K}'', p'')$ and $(\mathsf{K}'', p'') \overset{S}{\Rightarrow} (\mathsf{K}', p')$.

To clarify some of the more complex rules:

- A sealed blob consists of the encryption of (tpmPf, p, t), where tpmPf is a constant known only to the TPM, p is the PCR state which must be current for an unsealing operation to succeed, and t is the data which has been sealed. The encryption is done with a public key, whose private part is available only within the TPM. The rules for unseal add the secret t to the attacker's knowledge base if the required PCR value in the sealed blob matches the current PCR value.

- In the rule for $\mathsf{SKINIT}\{L; \mathsf{rtn}\ u\}$, we first compute the effect of L when run from knowledge base K and a PCR value reflecting the measurement of L.
- SUBR is similar to SKINIT, except that the final PCR value is p rather than p'. As mentioned, SUBR is not used in source TXML programs; we use it in our transformation.

Modelling the introductory example

Two objects are supplied to the decryption oracle: a sealed blob containing the symmetric key pre-sealed against PCR17 with the value of u_0 extended with the measurement of the decryption oracle's program, and some message encrypted with the symmetric key. Therefore, the initial knowledge base is:

$$
\begin{aligned}
&\mathsf{K}_{\mathit{init\_DO}} = \{\\
&\qquad x_{\mathit{sdata}} = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r},\\
&\qquad\quad (\mathsf{tpmPf}, \mathsf{h}(0, \mathsf{measure}(\mathsf{slbD})), \mathsf{symKey})),\\
&\qquad x_{\mathit{EncBlob}} = \mathsf{senc}(\mathsf{symKey}, \mathsf{message})\\
&\}
\end{aligned}
$$

where slbD is the program:

```
result := SKINIT {
  xSymKey := unseal(xSBlob);
  reveal (xSymKey);
  xMessage := sdec(xSymKey, xEncBlob);
  xTag := fst2(xMessage);
  check xTag = theTag;
  xText := snd2(xMessage);
  extend(fpc);
  rtn xMessage;
}.
```

The security property we are checking is the secrecy of the symmetric key symKey. To attempt to obtain the key, the attacker can adopt any strategy. The desired security property is therefore that there is no TXML program S, knowledge base K', PCR values p, p' and variable x such that $(K_{\mathsf{init\_DO}}, p) \overset{\mathsf{S}}{\Rightarrow} (\mathsf{K}', p')$ and $\mathsf{K}'(x) = \mathsf{symKey}$.

Transformations of strategies

The security property above requires reasoning over all possible TXML programs S. We show that it is sufficient to consider only programs involving a bounded number of SKINITs and resets. To achieve this, we transform any strategy S into a new strategy S' that is equivalent to S and has a number of SKINITs and resets which is bounded by a value derivable from the initial knowledge base. This allows us to use automatic tools to search for strategies that achieve a certain goal and model at most this number of SKINITs and resets; if none are found, we may conclude that there are no longer strategies that achieve the goal.

The transformation performs two changes. First, unseal operations that are not necessary are replaced by an equivalent assignment. An unseal operation is not necessary if there is another variable in the knowledge base that has the value of the unsealed item. Second, operations that reset the PCR, namely, reset and SKINIT, are removed if there is no necessary unseal operation between the current point and the next reset or SKINIT. This reflects the fact that PCR values are required to be correct only in order for unseals to work. The transformation uses configurations (K, p) as before, but with the extension that p may take the special value $\perp$ which signifies that any value will do; the PCR value is not needed. The transformation is such that $p = \perp$ if and only if there is no unseal between the current position in the program and the next reset or SKINIT.

The result of this transformation does not weaken the attacker. The attacker can perform computation with the defender SLB using any number of inputs. The attacker can run arbitrary code with the PCR in the state left by the execution of the defender SLB. The attacker can attempt to unseal data sealed by the defender. The attacker already knows the contents of sealed data sealed by the attacker. If the attacker wishes to obtain more results from the defender SLB, these are available to him from the transformed strategy.

The transformations we use are shown in Figure 5.5. Given a knowledge base $\mathsf{K}_{\mathsf{init}}$, a strategy S is transformed to S' if $(\mathsf{K}_{\mathit{init}}, 1) \overset{S}{\underset{S'}{\Rightarrow}} (\mathsf{K}, \perp)$ for some K, where the relation $\overset{S}{\underset{S'}{\Rightarrow}}$ (S and S' are strategies) is defined from $\xrightarrow[C']{C}$ above as follows: $(\mathsf{K}, p) \overset{\emptyset}{\underset{\emptyset}{\Rightarrow}} (\mathsf{K}, p)$ where $\emptyset$ is the empty strategy; $(\mathsf{K}, p) \overset{C;S}{\underset{C';S'}{\Longrightarrow}} (\mathsf{K}'', p'')$ if there exists (K', p') such that $(\mathsf{K}, p) \xrightarrow[C']{C} (\mathsf{K}', p')$ and $(\mathsf{K}', p') \overset{S}{\underset{S'}{\Rightarrow}} (\mathsf{K}'', p'')$.

In the definition of $\xrightarrow[C]{C'}$, the truth or falsity of $p = \perp$ enforces a global constraint on the way the transformation works, and makes the transformation deterministic. The two rules for each of SKINIT and reset appear to be non-deterministic, but in fact only one of them may be chosen; which one is chosen depends on whether there is an unseal before the next SKINIT or reset, as expressed by the rule for unseal which requires $p \neq \perp$. The apparent free choice of whether $q = p$ or $q = \perp$ in the rule for unseal is similarly constrained by the remainder of the program S being transformed.

SUBR is introduced into S$'$ by the second rule for SKINIT, which is invoked if and only if there is no unseal between now and the next SKINIT or reset, as indicated by the $\perp$ value on the right hand side.

Bounding the number of SKINITs and resets

The ability to seal data against arbitrary PCR values is very flexible, and can lead to situations more complex than the simple sealing of one piece of data against one set of PCR values. A piece of data sealed against one set of PCR values can contain another piece of data sealed against a disjoint set of values. A piece of data may be sealed against an SLB which will only release the decrypted version once some other conditions are met. These conditions might include the decrementing of a counter, or transition through some state machine.

We define data as boundedly sealed if there is a finite bound to the number of SKINITs required to extract it.

The relation $(\mathsf{K},p) \xRightarrow[S']{S} (\mathsf{K}',p')$ indicates that when in configuration (K,p) the execution of command C yields the new configuration (K',p') and adds the command C' to the transformed strategy.

$$(\mathsf{K},p) \xrightarrow[\mathsf{skip}]{\mathsf{skip}} (\mathsf{K},p)$$

$$(\mathsf{K},p) \xrightarrow[x:=\mathsf{f}(u_1,..,u_n)]{x:=\mathsf{f}(u_1,..,u_n)} (\mathsf{K}[x \to \mathsf{f}(\mathsf{K}(u_1),..,\mathsf{K}(u_n))],p)$$

$$(\mathsf{K},p) \xrightarrow[x:=\mathsf{g}(u_1,..,u_n)]{x:=\mathsf{g}(u_1,..,u_n)} (\mathsf{K}[x \to t],p)$$

if $\mathsf{g}(t_1,..,t_n) \to t_{n+1}$is a reduc
and $\mathsf{K}(u_i) = t_i\sigma$ and $t = t_{n+1}\sigma$

$$(\mathsf{K},p) \xrightarrow[x:=\mathsf{seal}(u,v)]{x:=\mathsf{seal}(u,v)}$$

$$(\mathsf{K}[x \to \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}),\mathsf{r},(\mathsf{tpmPf},\mathsf{K}(u),\mathsf{K}(v)))],p)$$

$$(\mathsf{K},p) \xrightarrow[x:=y]{x:=\mathsf{unseal}(u)} (\mathsf{K}[x \to \mathsf{K}(y)],p)$$

if $\mathsf{K}(u) = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}),\mathsf{r},(\mathsf{tpmPf},p',\mathsf{K}(y)))$

$$(\mathsf{K},p) \xrightarrow[x:=\mathsf{unseal}(u)]{x:=\mathsf{unseal}(u)} (\mathsf{K}[x \to t],q)$$

otherwise, if $\mathsf{K}(u) = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}),\mathsf{r},(\mathsf{tpmPf},p,t))$
and $p \neq \perp$ and $(q = p$ or $q = \perp)$

$$(\mathsf{K},p) \xrightarrow[\mathsf{extend}(u)]{\mathsf{extend}(u)} (K,\mathsf{h}(p,\mathsf{K}(u))) \text{ if } p \neq \perp$$

$$(\mathsf{K},\perp) \xrightarrow[\mathsf{extend}(u)]{\mathsf{extend}(u)} (K,\perp)$$

$$(\mathsf{K},\perp) \xrightarrow[\mathsf{reset}]{\mathsf{reset}} (\mathsf{K},1)$$

$$(\mathsf{K},\perp) \xrightarrow[\mathsf{skip}]{\mathsf{reset}} (\mathsf{K},\perp)$$

$$(\mathsf{K},p) \xrightarrow[\mathsf{check}\ u=v]{\mathsf{check}\ u=v} (\mathsf{K},p) \quad \text{if } \mathsf{K}(u) = \mathsf{K}(v)$$

Suppose $(\mathsf{K},\mathsf{h}(0,\mathsf{measure}(\mathsf{P}))) \xrightarrow[\mathsf{P}']{\mathsf{P}} (\mathsf{K}',p')$. If P is a defender SLB:

$$(\mathsf{K},\perp) \xrightarrow[x:=\mathsf{SKINIT}\{\mathsf{P};\mathsf{rtn}\,u\}]{x:=\mathsf{SKINIT}\{\mathsf{P};\mathsf{rtn}\,u\}} (\mathsf{K}[x \to \mathsf{K}'(u)],p')$$

$$(\mathsf{K},\perp) \xrightarrow[x:=\mathsf{SUBR}\{\mathsf{P};\mathsf{rtn}\,u\}]{x:=\mathsf{SKINIT}\{\mathsf{P};\mathsf{rtn}\,u\}} (\mathsf{K}[x \to \mathsf{K}'(u)],\perp)$$

Suppose P is an attacker SLB:

$$(\mathsf{K},p) \xrightarrow[x:=\mathsf{SUBR}\{\mathsf{P'};\mathsf{rtn}\,u\}]{x:=\mathsf{SKINIT}\{\mathsf{P};\mathsf{rtn}\,u\}} (\mathsf{K}[x \to \mathsf{K}'(u)],\perp)$$

Figure 5.5: Transformation of strategies

Definition 1 *1. A piece of sealed data B is sealed against program P if*

$$B = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), r, (\mathsf{tpmPf}, p, t))$$

and

$$p = \mathsf{h}(\ldots \mathsf{h}(\mathsf{h}(0, \mathsf{measure}(P)), t_2), \ldots t_n).$$

2. A piece of sealed data B produces B' if B is sealed to a program P and P can output a sealed blob

$$B' = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), r', (\mathsf{tpmPf}, p', t'))$$

containing data t'. We then define

$$\bar{B} = \{B' \mid B \textit{ produces* } B'\}$$

where produces is the reflexive transitive closure of produces.*

Sealed data B is boundedly sealed if $\bar{B}$ is finite.

Theorem 1 Suppose K is a knowledge base with sealed blobs, B_1, $B_2 \ldots B_n$, all of which are boundedly sealed. Let $m = |\bar{B_1}| + |\bar{B_2}| + \ldots + |\bar{B_n}|$. Let S be any strategy, and suppose that $(\mathsf{K}, \perp) \underset{S'}{\overset{S}{\Rightarrow}} (\mathsf{K}', \perp)$. We have the following properties:

1. $(K, \perp) \overset{S}{\Rightarrow} (K', p)$ implies $\exists S'.(K, \perp) \underset{S'}{\overset{S}{\Rightarrow}} (K', \perp)$.

2. S' simulates S; that is, $(K, \perp) \underset{S'}{\overset{S}{\Rightarrow}} (K', \perp)$ implies $\exists q.(K, \perp) \overset{S'}{\Rightarrow} (K', q)$.

3. S' uses only the data present in S that is, every name in S' is in S.

4. The number of SKINITs plus the number of resets in S' is at most m.

The theorem is proven below in Section 5.3

This theorem enables us to undertake practical verification of systems that use trusted execution. Without the theorem, we would need to consider attacker strategies of unbounded length, or accept weak results based on a significantly weakened attacker who can only use strategies of a fixed length. Our theorem removes these restrictions.

Although we can bound the number of SKINIT operations that need to be considered in terms of the effect on the PCR state, this theorem does not bound the number of times the attacker can use the SKINIT operation to run the defender's SLB. A typical SLB will perform some unsealing, and will leave the PCR is some new state, but these are not the core function, which is usually to perform some computation on the results of the unsealing and return the result. We cannot limit the attacker's ability to perform those computations without severely limiting their capability.

This is the basis for our introduction of the SUBR operation. Although it does not modify the PCR state (because we have shown that we can bound the number of such modifications we need to consider) it returns the same result as the corresponding SKINIT. So in the typical case where data is simply sealed so that we need only consider one SKINIT, the attacker can nonetheless make unlimited use of the SUBR to get results from the defender's SLB.

5.3 Proof of Theorem 1

Proof of part 1 of the theorem

Part 1 is proved by induction on the strategy S.

We distinguish two kinds of SLB P that can appear in an SKINIT of S. If an SLB P within SKINIT in S is such that there is a blob $\mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), r, (\mathsf{tpmPf}, p, t))$ in K where $p = \mathsf{h}(\ldots \mathsf{h}(\mathsf{h}(0, t_1), t_2), \ldots t_n)$ and $t_1 = \mathsf{measure}(P)$, we say it is a defender's SLB. That is because the defender of the secret has chosen to seal the secret against this P, possibly with further extensions. Any other P in SKINIT in S is said to be an attacker's SLB. That's because this P is one that the attacker has chosen himself.

We first prove the following lemma about commands C.

Lemma 1 *Suppose* K *is a knowledge base.*

If $(\mathsf{K}, p) \xrightarrow[C']{C} (\mathsf{K}', p')$ *then* $(p \neq \bot$ *and* $(\mathsf{K}, p) \xrightarrow{C'} (\mathsf{K}', p'))$ *or* $(p = \bot$ *and* $\forall q \exists q'.(\mathsf{K}, p) \xrightarrow{C'} (\mathsf{K}', q'))$.

Proof. Consider each case of C. The cases

$$\begin{array}{l} \mathsf{skip} \\ x := \mathsf{f}(u_1, .., u_n) \\ x := \mathsf{g}(u_1, .., u_n) \\ x := \mathsf{seal}(u, v) \\ x := \mathsf{extend}(u) \end{array}$$

are all straightforward. The remaining cases

$$\begin{array}{l} x := \mathsf{unseal}(u) \\ x := \mathsf{SKINIT}\{P; \mathsf{rtn}\, u\} \\ \mathsf{reset} \end{array}$$

require more detailed argument.

Case C is $x := \mathsf{unseal}(u)$.

If C' is unseal, *then $p \neq \bot$, so $(\mathsf{K}, p) \xrightarrow{C'} (\mathsf{K}', p')$. If C' is the assignment, then $p = \bot$, so the result holds easily.*

Case C is reset.

If C' is reset, *then $p' = 1$, and the result holds. If C' is* skip, *then $p' = \bot$, and again the result holds.*

Case C is $x := \mathsf{SKINIT}\{P; \mathsf{rtn}\, u\}$.

Suppose $(\mathsf{K}, \mathsf{h}(0, \mathsf{measure}(P))) \xrightarrow[P']{P} (\mathsf{K}'', p'')$. Instantiating the hypothesis of the lemma to the case at hand, we have $(\mathsf{K}, \bot) \xrightarrow[C']{\mathsf{SKINIT}\{P; \mathsf{rtn}\, u\}} (\mathsf{K}', p')$ and we need to prove that $\forall q \exists q'.(\mathsf{K}, q) \xrightarrow{C'} (\mathsf{K}', q')$.

Suppose P is a defender SLB. If C' is SKINIT, then q can take any value, and $p' = q' = p''$. If C' is SUBR, then $p' = \bot$, q can take any value, and $q' = q$.

Suppose P is a attacker SLB. Then P' is run outside of the SKINIT. By definition of attacker SLB, all the seals against P were produced by the attacker; therefore, there are no seals in P', and P' will produce the same knowledge as the SKINIT. Since $p' = \bot$, the result holds again.

We now prove that this lemma holds for strategies, not just commands.

Lemma 2 *Suppose* K *is a knowledge base. If* $(\mathsf{K}, \bot) \underset{S'}{\overset{S}{\Rightarrow}} (\mathsf{K}', \bot)$ *then* $\forall q \exists q'.(K, q) \overset{S'}{\Rightarrow} (K', q')$*.*

Proof. By induction on the strategy S. Base case: $S = \emptyset$. The result holds immediately. Inductive case: $S = C; S_1$. The result holds by simple reasoning and use of Lemma 1 for C.

The proof of part 1 of the theorem now follows, since it is a special case of Lemma 2.

Proof of part 2 of theorem

Part 2 of the theorem is readily proved by inspection of the transformation.

Proof of part 3 of theorem

Part 3 follows from the facts that:

- at most m plain-text distinct sealed blobs can be produced from the initial data;
- the transformed strategy S' runs at most one SKINIT for each blob sealed to a PCR value rooted in 0 (other invocations are run as SUBRs);
- the transformed strategy S' runs at most one reset for each sealed blob rooted in 1 (other resets are transformed into skips).

5.4 Compiling TXML to StatVerif

TXML serves two purposes. Firstly, we use it to model trusted code that is run under an SKINIT, so that we can generate StatVerif models in a consistent way. Secondly, we reason about the capabilities of an attacker who is able to use the capabilities expressible in TXML. TXML. TXML therefore provides the basis for our proof that we can bound the number of extensions, resets and SKINITs.

We have constructed a compiler for TXML which generates StatVerif code. This allows us to take an SKINIT described in TXML and obtain equivalent StatVerif code which can then be supplied to StatTrans and ProVerif for verification. We can also construct deliberately weakened versions of the same SKINITs together with an attack expressed in TXML.

5.4.1 The actions of the compiler

The compiler, which we call txml2statverif, operates on TXML source to generate StatVerif. The full description can be found in Section 5.4. The TXML is translated into the equivalent StatVerif in a natural manner. Indeed, the testing process for the compiler includes checking it generates byte-for-byte equivalent versions of manual translations.

When an SKINIT is encountered, two separate processes are generated: one includes a preamble and postamble which manipulates the PCR register and other state information; the other simply runs the TXML included in the SKINIT and returns the result. These correspond respectively to the SKINIT and SUBR forms used in our theorem.

5.4.2 Experiments with the compiler

For each of our case studies, txml2statverif has been used to show that the TXML can be compiled into StatVerif and processed by ProVerif to yield results equivalent to the hand-written versions of the same models. Optimisations to the StatVerif code can be applied automatically to improve termination; the details of these are discussed below in Section 5.5.2. An additional operation, "reveal", has been introduced into TXML to allow a deliberately weakened version of a model to be produced. This statement outputs its arguments on a public channel when requested; it allows deliberately weakened models to be analysed to confirm that the tools find an attack.

5.5 Tooling support

TXML serves two purposes. Firstly, it offers a means to describe the actions that will taken during trusted execution, and transform that into a list of actions that were taken. By transforming any piece of TXML that executes and returns a result into an equivalent piece of TXML that will terminate, and showing that this transformation is sound, trusted execution modules written in TXML can be analysed for correctness.

Secondly, it provides a useful means to automate analysis of trusted execution. TXML can be automatically translated into StatVerif, integrated with a StatVerif model of the TPM and constrained into a form that is likely to terminate. By automating this process, the translation is made consistent; compared to earlier, manual work, a number of small errors were discovered. A proof of the correctness of the transformation is a major undertaking that would be a topic for future work. The StatVerif that is generated is similar to that which would be written by hand were the compilers not to exist; generating StatVerif automatically and then checking it by hand is faster and less error-prone than writing the same StatVerif from scratch for each verification.

In this section, I set out the structure of the toolchain used to convert TXML descriptions of units of trusted execution into models that StatVerif can analyse.

The toolchain consists of a sequence of steps:

- TXML is used as the input to a compiler to generate StatVerif, using the StatVerif extensions.
- The StatVerif translation of the TXML is combined with a StatVerif model of a TPM and a set of StatVerif processes in order to generate a complete model.
- The StatVerif is translated in Horn clauses, using the StatTrans translator.
- The Horn clauses are modified to ensure termination.
- The Horn clauses are finally processed with StatVerif in order to test reach-ability of the queries in the model.

5.5.1 Generating StatVerif from TXML

TXML was designed to be easy to tokenise and parse. Although its origin is simply a list of operations, I had automated processing in mind from the initial stages. As is almost inevitable

TXML Program *t*	txml2statverif(*t*)
`x := f(a, b, c); t1`	`let x = f(a, b, c) in txml2statverif(t1)`

`x := seal(pcr, y); t1`	`out(loc2priv, tuple3 (seal, pcr, y));`
	`in(loc2priv, x);`
	`txml2statverif(t1)`

`x := unseal(y); t1`	`out(loc2priv, tuple2 (unseal, y));`
	`in(loc2priv, x);`
	`txml2statverif(t1)`

`extend(x); t1`	`read state as xState';`
	`let xBoot' = fst3(xState') in`
	`  let xPcr' = snd3(xState') in`
	`    let xFlag' = trd3(xState') in`
	`      state := tuple3(xBoot', h(xPcr', x), xFlag');`
	`txml2statverif(t1)`

`reset; t1`	`read state as xState';`
	`let xBoot' = fst3(xState') in`
	`  let xFlag' = trd3(xState') in`
	`    state := tuple3(xBoot', u0, xFlag');`
	`txml2statverif(t1)`

`check x = y; t1`	`if x = y then txml2statverif(t1)`

`result := SKINIT {`
`     t2;`
`}; t1`
	`free Skinit000_measurement.`
	`let Skinit000 =`
	` lock;`
	` read state as xState;`
	` let xBoot = fst3(xState) in`
	`   let xPcr = snd3(xState) in`
	`     let xFlag = trd3(xState) in`
	`       if xFlag = false then`
	`         if xBoot = b0 then`
	`           state := tuple3(bt(b0), h(u0, Skinit000_measurement), true);`
	`           (* PAL operation starts *)`
	`           in(c, inputsof (t2));`
	`           let xMessage = txml2statverif(t2) in`
	`             out(c, xMessage);`
	`             (* PAL operation ends *)`
	`             read state as xState';`
	`             let xBoot' = fst3(xState') in`
	`               let xPcr' = snd3(xState') in`
	`                 state := tuple3(xBoot', xPcr', false);`
	`                 unlock.`
	
	`let AuxProc001 =`
	` (* PAL operation starts *)`
	` in(c, inputsof (t2));`
	` let xMessage = txml2statverif(t2) in`
	`   out(c, xMessage).`
	`   (* PAL operation ends *)`

Figure 5.6: Mapping of TXML into StatVerif

in such languages, changes made in the early stages to enhance functionality complicated the language slightly, but it can be parsed with an LR(1) parser.

The venerable *lex* [160] and *yacc* [161] were used to generate a lexical analyser and a parser, respectively. As is usual with compilers built using this tool chain, a concrete syntax tree was built in memory, with a symbol table used to track all the names. After some consistency checks the equivalent StatVerif is generated by walking the concrete syntax tree.

The design work lay in mapping the TXML into equivalent StatVerif (the StatVerif that would have been written by hand had TXML not been involved) and in providing a sufficient range of options such that various alternative scenarios and techniques could be explored. The basic transformations are shown in Figure 5.6.

5.5.2 StatVerif options

As well as mapping TXML operations into StatVerif, the tooling provides a range of extra choices for use when exploring models. Some of these are implemented as TXML operations, because they need to happen at a particular point during the execution of the model.

Others of them are implemented as “pragmas” (the name taken from `#pragma` in more recent C compilers) which affect the whole translation.

The least interesting of the pragmas control various aspects of the names that are given to objects in the emitted StatVerif.

skinit_name:

subr_name:

process_name: These set the name of the emitted functions, so that multiple TXML fragments can be combined in on model. Without these pragmas default names are used in the emitted StatVerif, which means that only one TXML file can be incorporated into the model.

skinit_measurement: This sets the name that is to be used as the measurement of the SKINIT. Because StatVerif does not have an capability for introspection, the measurement of the PAL itself (which in reality is an SHA1 hash of the code itself) is represented as StatVerif name. Again, the name can be changed into order to permit multiple PALs to be present in a single model.

In order to permit future extension of the TPM model, the pragma use_tpi_api takes a set of values which alter whether certain TPM operations are performed within the StatVerif generated in response to the TXML, or performed by sending messages to the TPM model.

use_tpm_api u: is set by default. Unseal, which is the core security method which we are analysing, is always performed by the TPM model, with appropriate checks on PCR values and the TPM Proof.

If this option is unset, any unseal requested by the PAL is immediately performed within the PAL itself. This feature was added to speed up debugging of models; it removes any fidelity to reality, but speeds up termination (which can be up to ten times faster). With this option unset, the results of the termination do not say anything useful about correctness with respect to trusted execution, but might speed up the process of debugging a model which has problems elsewhere.

use_tpm_api e:

use_tpm_api r:

use_tpm_api s: are unset by default. The current TPM model does not support extension, reset or sealing, because in our StatVerif the PCR registers are directly accessible. These flags are provided to permit later development. I did experiment with a version of the TPM model which supports extension of PCRs and reading of PCRs via messages, rather than directly: although it appeared correct, it was not possible to get even simple models to terminate in reasonable time.

The “composable” and “alpha” pragmas are present to help debugging: they force the renaming of symbols that might be re-used between multiple SKINITs in the same model, simply to make reading StatVerif output easier.

5.5.3 Optimising StatVerif

In order to reduce the time taken for models to terminate, some optimisation is done to merge together related operations. The main such optimisation relates to the common case of the last action of an SKINIT being extension of a PCR. Rather than rebuild the state tuple twice, which generates a lot of Horn clauses, we instead combine the extension and the exit from the SKINIT into one rebuilding of the state.

Each time the state tuple is manipulated, it is broken apart with a destructuring “let”, the individual elements are modified and it is then reassembled into a tuple. Doing this repeatedly slows down execution of the model markedly, to the point that termination on non-trivial models is extended by a factor of two or more.

Typically, the state tuple is modified twice in rapid succession. The first modification is usually triggered by the last step in the TXML, which is the extension of the PCR with the fixed public constant. The second modification is the book-keeping done by the model, involving re-setting flags that indicate that a PAL is being executed.

With a naive translation of TXML into StatVerif, these two modification are done separately, each with its own destructuring and re-building of the state tuple. The adds a substantial number of Horn clauses to the model and slows down execution. We optimise the StatVerif by merging the book-keeping modification of the state tuple into any modification that takes place as the last action of a PAL. A command-line option is provided to turn off this optimisation so as to generate the most obvious, direct StatVerif.

5.5.4 Modifying the generated Horn clauses

The output from StatTrans would normally be fed directly into ProVerif. Modification of the Horn clauses is only required if they would not terminate in their initial form, and is only legitimate if there is an associated proof of the validity of the transformation.

In this case, our problem is that the StatVerif model is highly unlikely to terminate when given the clauses generated from the original TXML. Our model of the TPM permits extension at any time, and ProVerif is unable to see that this repeated extension cannot yield a useful result. Rather than modify StatVerif, we instead modify the Horn clauses, consistent with our proof of the validity of the transformation.

The state of the system is represented with a triple whose second element is the value of the PCR. Each clause whose left hand side (which is matched against the current knowledge) references the PCR as a single variable (which will match any PCR state) is replaced by multiple clauses, each matching a possible value of the PCR. This allows us to restrict the set of left hand sides that can be matched to those reachable within a specific number of extensions from the starting state. Once the PCR has reached the state furthest from the start state for which a clause matches, no further matches will take place and therefore no further extensions will be made.

This is the concrete implementation of constraining the search space: rather than having a set of clauses which match any state, instead they only match up to the limit of the search. In the tooling as it currently stands, the default limit to the matching is two extensions beyond the starting state, so that the search terminates at three extensions. In practice, no example that has been analysed requires more than this, and for simple cases with a smaller limit the

Initialisation data steps and starting processes:

```
new sk_Srk; new sk_Slb; new pwd; new salt; new r;
out(c, pk(sk_Srk)); out(c, pk(sk_Slb));
out(c, salt);
out(c, aenc(pk(sk_Srk), r,
   tuple3(tpmPf, h(u_0, slbA), sk_Slb)));
(Tpm | !Skinit | !Client | !Server | !Auxproc)
```

Additional process for the client:

```
let Client =
   new r; in(c, x_Nonce);
   let x_Cipher = aenc(pk(sk_Slb), r,
                            tuple2(pwd, x_Nonce)) in
   out(c, x_Cipher).
```

Additional process for the server:

```
let Server =
   in(c, xHash);
   if xHash = md5(salt, pwd) then
   out(c, allow_login).
```

Figure 5.7: Inserted lines for password authentication

saving in execution time is not worth the effort involved in selecting the correct limit.

The clauses output by StatTrans are parsed with a recursive descent parser; the syntax is sufficiently simple that the tokeniser is a single rule, and the grammar itself only consists of nine rules. The parser recognises Horn clauses in the format used as an intermediary between StatTrans and ProVerif; it makes some assumptions about the naming scheme in use in order to recognise the Horn clauses that relate to the manipulation of the state tuple. Each clause which operates on the PCR to perform an extension will match any value, as in “reality” the PCR can be extended starting from any state. We replace that one clause with a sequence of clauses which match pre-set patterns for the PCR: initial states, and then initial states extended by one or more values, up to the limit of the number of extensions we intend to search.

5.6 Results

5.7 Password authentication

In this section, we take the SSH password authentication application from [44] as a case study.

5.7.1 Description

An additional authentication mechanism for OpenSSH is proposed by McCune *et al* [44]. The defender has an authentication SLB (slbA) which is used to provide the hash of a password the client supplies, without the server having access to the plain-text. The goal is to prevent any malicious code on the server from learning the user's password, even if the server is compromised. Users often use the same passwords for multiple services; with this extension to OpenSSH, a user who logs onto a compromised server will at least not disclose a password which may be used by the attacker to compromise further accounts.

A keypair is shared between the authentication SLB and the client. The private part of the keypair ($\mathsf{sk}_{\mathsf{Slb}}$) is sealed against PCR17 with the value u_0 extended with the measurement slbA, and is conveyed to the user in a way which allows him to confirm the key generation was done correctly (see [44] §6.3.1 for details).

When the client wishes to authenticate, a nonce is generated by the ssh server and sent to the client. The client sends this nonce and the client's password (pwd), encrypted using the public part of the keypair it has previously obtained, to the server. The server sends the nonce and the cipher text to the SLB, together with the sealed key and the salt (salt) extracted from the password file.

The SLB unseals the private part of the keypair, and uses that to decrypt the message from the client. The nonce contained in that message is compared with the nonce supplied by the server in order to confirm freshness. The password extracted from the message is hashed together with the salt provided by the server, in order to form a value that the server can compare with the copy from the password file. The plain-text of the password is not available outside the SLB; the hash is available more widely.

As with the previous example, we can see that no sealed data is output. Although the output is not plain-text (it is the hash of a password together with some salt) it will does not contain the TPM proof and is not encrypted.

5.7.2 Modelling

We model the SSH password authentication application in TXML. As well as the salt and the public part of all keys, we assume that the attacker has the private part of $\mathsf{sk}_{\mathsf{Slb}}$ sealed against u_0 extended with the measurement of slbA.

$$\begin{array}{l}
\mathsf{K}_{\mathit{init\_SSH}} = \{ \\
\qquad x_{\mathit{salt}} = \mathsf{salt}, \\
\qquad x_{\mathit{pksrk}} = \mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \\
\qquad x_{\mathit{pkslb}} = \mathsf{pk}(\mathsf{sk}_{\mathsf{Slb}}), \\
\qquad x_{\mathit{sdata}} = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, (\mathsf{tpmPf}, \\
\quad \mathsf{h}(0, \mathsf{measure}(\mathsf{slbA})), \mathsf{sk}_{\mathsf{Slb}})) \; \}
\end{array}$$

where slbA is the program:

```
result := SKINIT {
  xsk_Slb := unseal(xSdata);
  xTemp := adec(xsk_Slb, xCipher);
  xPwd := fst2(xTemp);
  xNonce' := snd2(xTemp);
  check xNonce = xNonce';
  hash := md5(xSalt,xPwd);
  extend(fpc);
  rtn hash;
}.
```

5.7.3 Analysis in StatVerif

The compiler generates StatVerif representing components of the system. The inserted parts for the protocol are shown in Figure 5.7.

- In lines 3–6, the public parts of SRK and the communication keypair, the salt, and the pre-sealed private part of the keypair are output to the attacker via public channel c. This models the knowledge the attacker is assumed to have.
- Additional processes including Client, Server and Auxproc are started.
- Process Client models the client receiving x_{Nonce} and generating an encrypted blob containing its own password pwd and the nonce under the public part of $\mathsf{sk}_{\mathsf{Slb}}$.
- Process Server acts only as a sanity check for the protocol, because the server has already been assumed to be compromised.

The security property we wish to check is the secrecy of the client password pwd. We also confirm that a login token is generated in order to provide some assurance that the model is correct. The queries are written in the StatVerif calculus as follows:

$$\begin{array}{ll} \mathsf{query\ att}(u, \mathsf{pwd}) & (F_3) \\ \mathsf{query\ att}(u, \mathsf{allow\_login}) & (F_4) \end{array}$$

5.7.4 Result of our analysis

After using StatTrans to generate Horn clauses from our StatVerif input, we apply the modification described in §5.5.4. Using the syntactic criteria from [94], we can bound the number of PCR extensions. ProVerif then terminates with F_4 reachable and F_3 unreachable. This shows that login tokens are being generated, and that there is no short attack on the secrecy of the user's password pwd. Based on $\mathsf{K}_{\mathit{init\_SSH}}$ and slbA, the model conforms to the conditions of Theorem 1. Therefore there is no attack on the secrecy of pwd.

5.8 A Certification Authority

5.8.1 Description

This certification authority example is also taken from [44]. It consists of two SLBs, one to perform key generation, the other to perform key signing.

The key generation SLB constructs a keypair ($\mathsf{sk}_{\mathsf{SignKey}}$) suitable for use in signing other keys, and the private part of $\mathsf{sk}_{\mathsf{SignKey}}$ is sealed against u_0 extended with the measurement of the second SLB.

For the signing SLB (slbC), the client forms a certificate signing request (CSR) containing a public key along with details of the client's identity. The client submits this to the key signing SLB, which has access to the sealed form of its own private key. The SLB checks the signing policy, then unseals its private part of the keypair in order to sign the CSR. The result is returned to the client.

5.8.2 Modelling

We make some abstractions in our model. Firstly, we check that the signing SLB maintains the secrecy of any signing key with which it is used. This allows us to leave the key-generation SLB unmodelled and use a simple process that produces sealed keys instead. Secondly, as the required security property is the secrecy of the CA's signing key rather than the authenticity of the CSRs, the signing policy is not modelled.

We assume that the attacker has the public parts of the storage root key and the signing key $\mathsf{sk}_{\mathsf{SignKey}}$. We also assume that the attacker has the private part of $\mathsf{sk}_{\mathsf{SignKey}}$ sealed against u_0 extended with the measurement of slbC.

As in the previous example, we are able to determine that $m = 1$, as the application does not output any sealed objects.

$$
\begin{aligned}
&\mathsf{K}_{\mathit{init\_CA}} = \{ \\
&\qquad x_{\mathit{pksrk}} = \mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \\
&\qquad x_{\mathit{pkSignKey}} = \mathsf{pk}(\mathsf{sk}_{\mathsf{SignKey}}), \\
&\qquad x_{\mathit{sdata}} = \mathsf{aenc}(\mathsf{pk}(\mathsf{sk}_{\mathsf{Srk}}), \mathsf{r}, \\
&\qquad\quad (\mathsf{tpmPf}, \mathsf{h}(0, \mathsf{measure}(\mathsf{slbC})), \mathsf{sk}_{\mathsf{SignKey}})) \\
&\}
\end{aligned}
$$

where slbC is the program:

```
result := SKINIT {
  xskSignKey := unseal(xKeyBlob);
  xCert := sign(xskSignKey,xCSR);
  extend(fpc);
  rtn xCert;
}.
```

5.8.3 Analysis in StatVerif

We again obtain the StatVerif processes for the certification authority protocol by filling the holes in the pattern and using the compiler to generate the Skinit and Auxproc processes. The inserted lines are shown in Figure 5.8.

- The public parts of SRK and the CA's signing key and the pre-sealed private part of the signing key are output to the attacker via public channel c.

Initialisation data steps and starting processes:

```
$\mathsf{new\ sk_{Srk};\ \ new\ sk_{SignKey};\ \ new}\ r;$
$\mathsf{out}(c, \mathsf{pk}(\mathsf{sk_{Srk}}))$
$\mathsf{out}(c, \mathsf{pk}(\mathsf{sk_{SignKey}}))$
$\mathsf{out}(c, \mathsf{aenc}(\mathsf{pk}(sk_{\mathsf{Srk}}), r,$
    $\mathsf{tuple3}(\mathsf{tpmPf}, \mathsf{h}(\mathsf{u_0}, \mathsf{slbC}), \mathsf{sk_{SignKey}})));$
$(\mathsf{Tpm} \mid !\mathsf{Skinit} \mid !\mathsf{Auxproc})$
```

Figure 5.8: Inserted lines for the CA example

- Additional processes including Auxproc are started.

The security property we are checking is the secrecy of the CA signing key $\mathsf{sk_{SignKey}}$. As a partial check that the model is correct we also check for the existence of certificates signed by $\mathsf{sk_{SignKey}}$. The queries are written in the StatVerif calculus as follows:

$$\begin{array}{ll} \mathsf{query\ att}(u, \mathsf{sk_{SignKey}}) & (F_5) \\ \mathsf{query\ att}(u, \mathsf{sign}(\mathsf{sk_{SignKey}}, x_{CSR})) & (F_6) \end{array}$$

5.8.4 Result of our analysis

We bound the number of PCR extensions as in §5.7.4. ProVerif then terminates with F_6 reachable, which shows that the model does in fact produced signed certificates, and F_5 unreachable, which shows that there are no short attacks on the secrecy of the CA signing key $\mathsf{sk_{SignKey}}$. Based on $\mathsf{K}_{\mathit{init\_CA}}$ and slbC, the model conforms to the conditions of Theorem 1. Therefore there is no attack on the secrecy of $\mathsf{sk_{SignKey}}$.

5.9 Counter-results

As well as proving that the method verifies correct pieces of application logic as correct, it is valuable to run the same process on components that are known to be incorrect, to confirm that the weaknesses are found. It is also useful to be able to isolate the sensitive data in a PAL, confirming that each item that is being sealed (which is an expensive process) actually needs to be sealed.

With this in mind, TXML is extended with the `reveal` function, which takes one expression as an argument. This optionally inserts into the generated ProVerif an `out` on a public channel, making the expression public and therefore available to the attacker. The functionality is switched on and off with a command line flag to the compiler. This permits the same file of TXML to be built in two forms, one which should pass verification and one which should exhibit a security flaw, without an error-prone editing of the file itself.

The expression that is `reveal`ed is typically the result of an unsealing, but can be an arbitrary expression. It allows experimentation with “what happens if the attacker discovers this?” or “what happens if the attack is able to guess this?”. The disclosure may not be of a secret *per se*; for example, if a seal contains the hash of a secret, revealing it may or may not be useful to an attacker depending on the details of the protocol in use. The `reveal` function covers all these situations. The compiler as described in this thesis does not have the functionality to permit the turning on or off of individual reveal operations: either everything is revealed or nothing is revealed. A development version of the compiler had the ability to add a name to each `reveal` and on the command-line specify which of the `reveals` is to be actioned.

Using this functionality shows that each of our examples is insecure (fails verification, with ProVerif halting and showing a plausible attack) if any of the sealed elements are revealed. This is unsurprising, but provides two benefits:

- There has been extensive research into the trusted execution mechanisms, and they have withstood detailed examination. If this current work had found problems, they would probably have been subtle, multi-step attacks. A verification which indicates there are no weaknesses in a system which has been extensively examined by less formal methods has a problem: it may simply be that the tooling and model are insufficiently sensitive to detect weaknesses, but as the result aligns with expectations no further attention is paid. But introducing bugs into code which is believed to be secure guards against complacency: the bugs (if appropriately chosen) break security. If the model and verification techniques do not detect this security failure, then they are immediately highly suspect. That the method detects security problems provides some informal confidence that when it does not detect a problem, there is no problem.
- When executing code that uses a TPM, sealing and unsealing confidential data is expensive: it involves the execution of complex, or at least lengthy, cryptographic code on a processor several orders of magnitude slower than the main CPU. Avoiding those operations is valuable, and if it turns out that some of the seal data in fact need not be sealed, that will allow the sealing and unsealing to be bypassed safely. The reveal operation allows this to be tested.

5.10 Weaknesses in the verifications

The verification technique described has some limitations. Intutively there is no reason to believe that they invalidate the verifications, but it is important to recognise the limits on what has been formally proved.

Firstly, the method used to bound the search space for ProVerif has the effect that attestations cannot be used by PALs. An attestation is an important part of the process of using a PAL, as it provides evidence to an external party that the result was calculated using the particular PAL. There may be other reasons why this is true; for example, if the calculation relies on the unsealing of a piece of data, the result may implicitly assure the user that the unsealing took place, and therefore that the PCRs were in the correct state. Attestations involve extending the PCRs with the inputs and output to the PAL, so that after the PAL has executed a subsequent TPM Quote operation provides evidence that a PAL with a particular measurement was executed with a particular set of inputs to produce a particular output. My proof of correctness relies on being able to simulate the actions of a PAL (SKINIT) with a SUBR, which does not modify the PCRs. A clear requirement for future work would be to overcome this restriction, but it is not likely to be achieved by modification of the current method.

Secondly, there are limitations on the ability of PALs to produce sealed data as their output. Repeated recursive sealing is prohibited. In practice this restriction is less serious than the restriction on attestation, because extensive consideration of plausible PALs has not revealed any that need to do more than unseal a piece of data, update it, and then reseal that result. However, a general verification technique would not have this restriction.

5.11 Summary of chapter

We have seen a technique which permits the automated verification of pieces of application logic expressed in a high-level language, running within the context of a model of trusted execution. The verification technique places restrictions on the classes of PALs that can be verified, which do not preclude the verification of a wide range of realistic cases, but also unfortunately excludes the verification of PALs which rely on the production of attestations, which is a significant limitation. Further work would be required to overcome this problem.

Chapter 6

Summary, conclusions and future directions

6.1 Achievements

This work set out to verify that the trusted execution technologies SVM and TXT had a sound foundation, to provide a means to extend symbolic verification of security protocols to include those which make use of trusted execution, and to provide some concrete examples of applications which could benefit from these techniques. In this it largely succeeded, but it is important to note the limitations of the methods used, which we set out below in Section 6.2

The work done provides a modelling language capable of expressing the functionality of plausible PALs, including examples taken from the literature, and allows their verification as part of a system using trusted execution. This modelling language can be automatically transformed into the input to the StatVerif system, which, with the use of another tool to impose some constraints — for which we have a proof of validity — will evaluate the security properties of the model.

6.2 Limitations

The main issue that we have is that the simplifications made to the model remove the ability to reason about systems which form attestations. This is not fatal to the utility of the work, but it is worth discussing the implications.

The method used for verification imposes one restriction on PALs: that each time they are executed, they must produce the same PCR state upon exit. This does not materially weaken the attacker, as they are given the power to run the PAL an arbitrary number of times and obtain the output each time, but does remove our ability to model faithfully the large class of PALs that need to prevent replay attacks by incorporating a proof of freshness provided by the invoker. The additional extensions would not assist the attacker in breaching properties of confidentiality, but it is frustrating that we cannot at the same time as proving confidentiality also prove resistance to replay attacks.

Unfortunately, the method is not amenable to modification to solve this problem. By bounding the number of executions of SKINIT, we bound the attacker's ability to execute the PAL.

Introducing the SUBR mechanism resolves this, allowing the attacker to obtain an arbitrary number of results from the PAL without increasing the number of times SKINIT is executed. Solving the problem of modelling a system in which the attacker can execute SKINIT, complete with modification of the PCR state, to different values each time, would require a substantially different approach, using a tool which was by some means able to avoid the problem an searching unbounded space.

6.3 Lessons learned

There are obvious temptations to choose both a problem area and a tool set that are already being studied within one's research group. There is expertise, wisdom and past experience to draw on. However, it is not a given that an adjacent problem to those already being addressed is tractable using the same approaches.

That was the case here: in the end, an extensive range of tooling, including complex transformations with associated proofs of correctness, was required to support the use of our initially selected tool. It is possible that other tools would have worked more naturally, although the general problem (that of searching a potentially unbounded space of possible state changes) would have remained the same. We looked in depth at the work of Datta *et al* [92] but decided to use StatVerif instead. StatVerif's developers were local, but by the time TXML was developed there would potentially have been profit in re-examining Datta's work [92]. On the other hand, ProVerif is a well-established tool with a wide range of literature, and its attacker model is an extremely good fit to our assumptions. Short of repeating the work using an alternative tool set, it is hard to judge which approach would be easier.

The toolchain that was eventually produced is simple to use. A problem is modelled in an imperative language plus a small amount of ProVerif (the additional functionality of StatVerif is required by the compiler, but does not normally need to be written by the user) and the tool chain then operates without further intervention. The end result is a set of Horn clauses that ProVerif can work on with a good chance of termination. The manually written ProVerif is mostly concerned with the relationship between processes; I sketched a language to automate the production of this in the majority of cases, but the language was of similar complexity to the subset of ProVerif normally used, and lacked the expressiveness to capture more complex cases. Given the familiarity with ProVerif within the community this did not seem like progress.

With hindsight, however, the toolchain has several problems.

The first is that the TXML to StatVerif compiler is very traditional in nature, using the standard Unix lexical analyser *lex* [160] and its usual partner the parser generator *yacc* [161] ("yet another compiler compiler"). The concrete parse tree generated by this is then processed by a program written in C. I also tested the compiler using the GNU re-implementations *flex* [162] and *bison* [163], but did not use any of their enhanced functionality: the reason for ensuring the GNU equivalents worked was in ensure portability to Linux platforms which do not have the original tools.

When the compiler was first developed, the transformations were relatively straightforward, and that the StatVerif patterns were embedded in the C source code was not a particular problem. The initial version of the compiler was written over a short period, following past experience in writing similar tools. Over the course of the research, the complexity of the transformations grew and the compiler became more complex to provide better error checking and code-generation: it acquired a symbol table, more complex semantic analysis (almost to the point of generating an abstract syntax tree) and a range of options affecting the code that is generated. When more recently I taught a course on compiler construction I was able to use the TXML compiler as an exemplar, as it contained each of the phases I was discussing. With hindsight, I would have realised in advance that this tool was going to become complex, and (a) written it using more modern tools rather than those that came most easily to hand and (b) made it table- or template-driven.

Similar comments apply to the second stage in the tooling, the modification of the Horn clauses produced by the StatVerif compiler. The basic transformation is simple: a clause which matches any state and performs a reset or extension is replaced by n clauses, each only matching the nth step in a sequence of resets or extensions. This started out as a shell script using the timeworn *sed* [164] and other standard Unix command-line tools; as more functionality was required it became a traditional lexical analyser, parser and code-generator compiler (this time using a recursive descent parser). However, it still acts more lexically than semantically — the compiler is coded with detailed knowledge of how the StatVerif compiler's code generator works and makes assumptions based upon, for example, the lexical form of names. Again with hindsight, it would have been better to modify the StatVerif compiler to annotate its output with the information that the transformation stage intuited, to avoid the rewriting process having to make assumptions based on the format of names.

The choice of symbolic verification tool meant that there is no way to use it to prove properties about cryptographic primitives: they are assumed to work by construction. When the variation on Diffie-Hellman arose as an idea, the toolchain was able to verify that the sealing and unsealing of the initial secret works, but cannot prove anything useful about the cryptographic security of the construction. Alternative tools (see above) would have been no better at this, but it is nonetheless a cause for regret.

6.4 Research directions

The work of this thesis shows that the trusted execution technology has a sound design. We have seen that, using well-proven verification tools, the confidentiality of data sealed to code can be assured, and we have presented a toolchain which allows this verification to be extended to applications which use trusted execution. The benefits are well summarised — albeit from the perspective of a manufacturer keen to sell product — in an Intel position paper [165].

We are left, however, with the inevitable conclusion that none of the TPM-based technologies touched on in this thesis have obtained significant market penetration. Over the period of time during which this thesis was being written, there was no apparent improvement in the situation. TPMs are widely deployed in “enterprise” class hardware, but are rarely enabled, let alone used; it is a matter of some conjecture as to why, and looking to the future it remains to be seen if newer entrants into the market achieve any more success.

Technologies which do not achieve substantial adoption within their first few years of availability rarely make a sudden recovery. Sadly, the TPM and its derivatives appear to have fallen into the category of promising technologies which offer much to developers, but which have failed to achieve any sort of critical mass in the marketplace. Their use as key stores has had some success on mobile platforms, but their use in enterprise environments has been overtaken by widespread use of virtualisation. The hopes I had at the outset of this work of verifying a technology which could be used to secure access to cloud-based services have not been fulfilled: it is difficult, at the time of writing, to envisage cloud architectures in which hardware roots of trust that are not amenable to use in virtualised environments can be successful.

There are some areas of the work described in this thesis which would benefit from further work. First and foremost, the extension of the verification of TXT and Flicker to include the production of attestations would be useful. Unfortunately, this is not likely to be a simple task, and will require a completely new approach to the proofs: without the ability to assume that PCRs are in the same state when the same PAL is executed with distinct inputs the line the method described in this thesis cannot work.

Secondly, the variation of the Diffie-Hellman cipher key exchange protocol that is described would benefit from further analysis. This seems a useful variation, even without the ability to embed it in a PAL.

Thirdly, in the event of one of the technologies described in Section 6.5 gaining substantial market presence, it would be interesting to examine the use of the technology to protect secrets, and repeat the verification that has been done for TXT.

6.5 Alternative architectures and developments

TXT and SVM, which use the current TPM as the root of trust, are not the only mechanisms for improving assurance of execution. There are technologies at various stages of development from Intel, ARM and Microsoft which address related issues, and it is worth looking briefly at each of them.

6.5.1 Intel Server Guard Extensions (SGX)

Intel has published a set of papers [166, 167] which describe a technology called SGX (Software Guard eXtensions). SGX allow developers to create what Intel refer to as Enclaves, whose function is similar to PALs. They have fewer restrictions on their function, which both extends the capability of the techology and complicates verification.

An enclave is a unit of code and data that executes in a trusted environment and is encrypted whenever it is exposed to untrusted code. A physically and logically protected Enclave Page Cache (EPC) is present in the processor (it is not clear to me if there is one per core or one per die). The EPC contains code and data, unencrypted, during the execution of an enclave. Attempts to access or modify the EPC from outside the enclave are prevented by processor access control. In order to permit larger, or more numerous, enclaves, data from the EPC can be paged to platform memory. Encryption is performed during paging, so that the paged-out data is not accessible to untrusted code. There is appropriate protection against rollback and other interference which pages that are being faulted back in.

Unlike other solutions based on TXT, interrupts and other events are enabled while the enclave is executing. Prior to an event handler being called, the contents of registers and caches are pushed onto the stack, encrypted, before the caches and registers are scrubbed (the EPC is not part of this process, as it is only accessible to the enclave). The event handler can then execute. If the event handler decides to continue the enclave, the process of saving the cache and register state is reversed and control is returned to the enclave.

The enclave cannot initiate I/O directly, but this does not prevent I/O continuing on behalf of other processes running on the machine. The enclave can also be time-sliced with other processes, so long-running enclaves do not stall the machine. This is a major benefit over Flicker type solutions, which have to run to completion. The enclave can also return intermediate results, so it possible to generate a stream of (for example) network packets which an external process then passes to real hardware.

The enclave can only be executed from defined entry points; an attacker cannot jump into the enclave and bypass input checks. This removes the opportunity for anything akin to return-orientated programming [168]: the hardware mechanism enforces the use of complete execution units, preventing an attacker from executing sections of code while bypassed bounds checking or other security measures.

An enclave is built by adding pages to a structure. As they are added, these pages are measured and a hash is computed. The page is then moved to the EPC. Once completed, the enclave can be executed. When execution finally completes, the enclave is destroyed and its space in EPC is available for other uses. As mentioned, if the total size of enclaves in use is greater than the size of the EPC the pages can be faulted to and from main memory, with encryption protecting their privacy and integrity.

Because the enclave is destroyed on exit, any persistent data must be stored externally. This is done with a sealing mechanism similar to a TPM. However, there are some significant differences. The sealing mechanism does not define the encryption technology, rather it performs key management; this means that encryption performance is gated by the processor, not a TPM, and the application can choose an appropriate encryption mechanism for the sensitivity of the data it is processing.

The data can be sealed in one of two ways: to the measurement of the enclave itself (the Enclave Identity) or to an identity used to sign the enclave (the Sealing Identity). The former means that data is sealed to a precise piece of code; the latter means that data is sealed to anything signed as being equivalent. There is support for versioning, to prevent older, insecure versions of an enclave being used to unseal data sealed by a newer, more secure version.

Data is inherently mobile from platform to platform. If a user is running a particular enclave on a platform that supports SGX, they can unseal data which was sealed to that enclave on another platform. If they need to seal data to a particular hardware instance, the facility exists, but it is not done by default (contrast with the TPM_Proof in a TPM seal, which is mandatory). Aside from its practical benefits in a cloud environment, this is a consequence of not using a single TPM-type element; if the enclave data were sealed to a particular processor instance, then it would be sealed to either one individual core or one individual die, which would make multiprocessor architectures very challenging.

There is an attestation mechanism, EREPORT, which permits an attestation to be generated of enclave state, together with up to 256 bits of ”user data” (this could be used to attest to the hash of a computed result, for example). There is a mechanism for exchanging attestations between enclaves and checking that they are correctly signed.

Intel has also published a paper on applications [167]. It contains one small application which could be implemented in Flicker: a one-time password generator and checker similar in outline to the ssh example described previously. But there is also a substantial application, the use of SGX to protect a video conferencing system. It relies on trusted audio and video, rather than arbitrary I/O, which is a complex layer of interfacing code between the enclave (which generates SRTP [169] packets) and the networking hardware which is not considered within the paper. Intel claims to have this running with good performance, which means that moving in and out of the enclave on each packet is an acceptable overhead.

If this technology is widely deployed, it will provide interesting avenues for developers. It is not, however, available on the market at the time of writing.

All the Flicker example applications described above would run faster and would not impact other threads running on the processor any more than some ordinary thread would. Intel claims convincingly that you could use SGX to implement any Flicker PAL after making fairly minimal changes.

That said, if there actually is a requirement to tie an enclave to a particular piece of hardware, for example to replicate an HSM, it will be somewhat more complex than it would be with a TPM. For example, with a TPM, the TPM_Proof ties decryption to a unique TPM, while in the SGX world the seal is either to the measurement of the enclave or the signer of the enclave, not a hardware instance. If an application in fact does want to tie it to a single instance then the application has to organise this itself. A TPM has a secure serial number and is intended to be securely bonded to a motherboard; an ID in a socketed processor or a reflashable BIOS chip is not as secure. For example, if a CA were being secured using SGX, the ability to clone the application and run multiple copies in parallel would be useful to an attacker; this is prevented by the semantics of unseal on a TPM system, but has to be implemented by the application on SGX, with no clear hardware root of trust available.

The same paper also suggests that with appropriate library support the same source code could run both as an SGX enclave or, if SGX technology were not available, a Flicker PAL. There is probably a useful subset of applications for which this would work.

In summary SGX is

- faster than Flicker-type solutions;
- has a smaller TCB than Trustvisor and other hypervisor solutions;
- will run most of the things that those will; and
- does not require a TPM.

Of course, the reality is that Flicker and other trusted execution technologies have not achieved any market penetration, so discussing how to improve the performance of them is slightly besides the point. If people were saying ”we prototyped an application using it, but the performance was poor so we are waiting for it to get faster” then things would be different, but that is not the case. What is needed is some truly compelling example applications, and they do not as yet appear to be present. Consider the applications in the cited papers.

The first is a rather complex one time password implementation. At its heart, the problem is getting a shared key agreed between two parties, such that the key material is sealed into an enclave at each end. As an example of agreeing a persistent key between two communicating parties, it is very plausible, and it demonstrates a range of the capabilities of an SGX enclave.

But as a genuinely useful example to convince a purchaser that SGX offers something for real systems architects, it is unconvincing. It relies on the new technology, SGX, being on the **client** system. No-one could adopt this mechanism until there was a significant penetration of the SGX into client devices, with OS APIs available to allow a browser plugin to access the functionality. If someone were reading the paper looking for reasons to experiment with trusted execution, they would not be convinced: a solution which only works if client devices have the latest hardware and software from one particular manufacturer is not likely to achieve sufficient volume to justify development work.

The other example is a malware-resistant secure video conferencing system.

Again, as an example of end to end encryption (”how can I use trusted execution to protect the private key associated with a certificate and to protect the session key while the connection is active?”) it is convincing.

But as an example of a real problem with a real threat model, it seems quite forced. Current secure video conferencing systems work by putting an external encryption box between the conferencing system and the untrusted network, which is easier to accredit, gives a wider choice of video conferencing hardware and protects against a similar range of threats. The set of people who on the one hand need video conferencing secure against some fairly exotic attacks, but on the other hand cannot use a commodity firewall and VPN to protect the connection, must surely be small.

6.5.2 Trustzone

ARM approach the problem of hardware-enforced security from a different perspective. Conceptually, each processor that implements the “TrustZone” technology is in fact two processors: one secure, able to access the entirety of memory and peripheral space, and the other insecure, with limitations on what it can access. The intent is that trusted code, subject to additional scrutiny and control, runs in the trusted context, and the large majority of application code runs in the untrusted context. Over time, the architecture has grown such that instead of there being physical assets associated with the trusted zone, rather there is a mode bit specifying whether the processor is in trusted mode, and the ability to time-slice trusted and untrusted execution.

The technology answers a need in ARM’s early market of mobile telephony, where there are regulatory requirements surrounding the software control of radio transmitters. Manufacturers of mobile phones must be able to show that they have taken sufficient measures to prevent the device being modified to transmit (and in some markets receive) outside the licensed frequencies, power outputs and modes. The definition of “sufficient” varies, but certainly a phone which could be modified to operate on unauthorised frequencies simply by modifying the software would not be acceptable.

Earlier phones used sealed modules to implement the radio frequency element of the phone, so that even if the rest of the phone were modified to request transmission outside the authorised envelope the RF module would not permit it. This is expensive and bulky; manufacturers would rather control the operation of the whole phone from one processor. So ARM’s TrustZone model suits this space perfectly: the RF can be controlled from the secure context, with the user interface and other element subject to upgrade and change controlled from the insecure context.

TrustZone does not, however, incorporate a full range of confidentiality services, nor does it have any direct equivalent of configuration registers, sealing and so on. It is not intended to; the architecture offers to those implementing operating systems a set of components to develop their own mechanisms to suit their own requirements. For the case of linking data to programs, or any other particular use-case, this flexibility comes at the expense of shifting the assurance burden from the processor manufacturer[1] to the operating system vendor. Because of this, and the fragmentation in the ARM market between a wide range of cores, systems on a chip and standalone processors, TrustZone has had the same issues of adoption as TPM-based solutions have had: the technology is attractive, but the challenge of delivering solutions that will run on a wide range of platforms and be maintainable is substantial.

6.5.3 Cloud TPM

Microsoft Research have proposed a set of extensions to the TPM 2.0 specification which they have named cTPM, the Cloud TPM [170]. There is no hardware implementation of this as yet; the work is only being carried out in simulation.

The researchers propose extending the TPM 2.0 specification with an additional key that is shared with the cloud. This key can be used to leverage sharing of other keys between multiple TPMs owned by the same user. This does not extend trusted execution to the cloud, but would permit sealed data to be stored in the cloud by one device and then operated on by another. Similar functionality is available from the Server Guard Extensions, and it will be interesting to see if either can attain production and then some market traction.

Interestingly, the authors use ProVerif to verify their extensions to the TPM 2.0. They do not have a model for the whole TPM, and would almost certainly need to use techniques similar to those in this thesis to deal with stateful operations.

6.5.4 Research possibilities

There are several pieces of work that are suggested by this thesis.

- The mechanism for using trusted execution to leverage Diffie-Hellman key exchange without relying on continuous access to a trusted random number generate has been prototyped, but a full implementation with a full mathematical evaluation would be a substantial project. The mechanism is proposed as an exemplar of a task where being able to securely link code and a piece of data is important; a concrete implementation would be worthwhile both as a demonstration of this and as a security protocol in its own right.
- The verification described in this work makes assumptions about constant outputs and PCR states from pieces of trusted code; it explicitly cannot prove results about code which forms attestations. The technique used in this work is almost certainly not capable of extension to deal with this: the assumptions are too deep-rooted, and the proof

[1] Which is particularly complicated in the case of ARM as they are a vendor of designs rather than of implemented silicon.

of the soundness of limiting the search relies on them. A verification of a fuller system which includes attestations would be valuable, but would require a root-and-branch consideration of tooling.

- TXML arose originally to support verification, and was extended to provide a means to model applications and generate StatVerif code. With additional annotation, it might be possible to use the same TXML source to generate both StatVerif and executable code. With the availability of verification tools for subsets of C [50] it might be possible to bring together a verification of the mapping from the TXML to C (by use of a model checker on the generated code) and also the program's use of trusted execution facilities (by the toolchain already presented).

Although probably the most challenging, the last of these directions is perhaps the most interesting. To be able to write and verify programs which execute on a platform with a hardware root of trust would be valuable for a range of security applications, and would bring together several disparate strands of research into one tool. To be able to run, for example, a certification authority whose code was amenable to verification and which also had a hardware root of trust would be a welcome development.

Bibliography

[1] Intel Corporation, Intel Trusted Execution Technology (Intel TXT) Software Development Guide Measured Launched Environment Developer's Guide. Intel, 2015.

[2] G. Strongin, "Trusted computing using AMD "Pacifica" and "Presidio" secure virtual machine technology," Information Security Technical Report, vol. 10, no. 2, pp. 120–132, 2005.

[3] ISO/IEC, ISO/IEC 11889-1:2009 Information technology – Trusted Platform Module – Part 1: Overview. ISO/IEC, 2009.

[4] M. Arapinis, E. Ritter, and M. D. Ryan, "StatVerif: Verification of Stateful Processes," in Proc. of the 24th IEEE Computer Security Foundations Symposium. IEEE Computer Society Press, 2011, pp. 33–47.

[5] B. Blanchet, V. Cheval, X. Allamigeon, and B. Smyth, "Proverif: Cryptographic protocol verifier in the formal model," http://prosecco.gforge.inria.fr/personal/bblanche/proverif, 2010.

[6] S. Xu, I. Batten, and M. Ryan, "Dynamic Measurement and Protected Execution: Model and Analysis," in Trustworthy Global Computing, M. Abadi and A. L. Lluch, Eds. Springer International Publishing, 2014, pp. 45–63.

[7] O. Ritchie and K. Thompson, "The UNIX time-sharing system," Bell System Technical Journal, The Bell System Technical Journal, The, vol. Bell System Technical Journal, The, no. 6, pp. 1905–1929, 1978.

[8] S. Smalley and P. Loscocco, "Integrating flexible support for security policies into the Linux operating system," https://www.nsa.gov/research/_files/selinux/papers/freenix01-abs.shtml, 2001.

[9] D. Nowitz, "Uucp implementation description," in UNIX Programmer's Manual. Bell Laboratories, NJ, 1978.

[10] K. Thompson and D. M. Ritchie, UNIX Programmer's Manual. Bell Telephone Laboratories, 1975.

[11] H. Chen, D. Wagner, and D. Dean, "Setuid Demystified." in USENIX Security Symposium, 2002, pp. 171–190.

[12] D. Tsafrir, D. Da, Silva, and D. Wagner, "The murky issue of changing process identity: revising "setuid demystified"," USENIX Login, vol. 33, no. 3, pp. 55–66, 2008.

[13] H. Orman, “The Morris worm: a fifteen-year perspective,” IEEE Security & Privacy, vol. 1, no. 5, pp. 35–43, 2003.

[14] W. Enck, M. Ongtang, and P. McDaniel, “Understanding android security,” IEEE security & privacy, vol. 7, no. 1, pp. 50–57, 2009.

[15] M. Ryan, “Personal Communication.”

[16] D. C. Andrews and J. M. Newman, “Personal Jurisdiction and Choice of Law in the Cloud,” Md. L. Rev., vol. 73, p. 313, 2013.

[17] D. Svantesson and R. Clarke, “Privacy and consumer risks in cloud computing,” Computer Law & Security Review, vol. 26, no. 4, pp. 391–397, 2010.

[18] P. T. Jaeger, J. Lin, J. M. Grimes, and S. N. Simmons, “Where is the cloud? Geography, economics, environment, and jurisdiction in cloud computing,” First Monday, vol. 14, no. 5, 2009.

[19] B. T. Ward and J. C. Sipior, “The Internet jurisdiction risk of cloud computing,” Information systems management, vol. 27, no. 4, pp. 334–339, 2010.

[20] M. Campbell-Kelly, ICL: a business and technical history. Cambridge University Press, 1990.

[21] H. Devonald and D. Eldridge, “How the ICT 1900 series Evolved,” Computer Resurrection, vol. 16, 1996.

[22] S. H. Lavington, “Manchester computer architectures, 1948-75,” Annals of the History of Computing, IEEE, vol. 15, no. 3, pp. 44–54, 1993.

[23] J. Buckle, “The origins of the 2900 series,” ICL Tech. J, vol. 1, no. 1, pp. 5–22, 1978.

[24] R. N. Ibbett, “The University of Manchester MU5 Project,” Annals of the History of Computing, IEEE, vol. 21, no. 1, pp. 24–33, 1999.

[25] P. E. Mounier-Kuhn, “Bull: A world-wide company born in europe,” Annals of the History of Computing, vol. 11, no. 4, pp. 279–297, 1989.

[26] CIA, “ICL COMPUTERS FOR THE USSR,” http://www.foia.cia.gov/sites/default/files/document_conversions/89801/DOC_0000969851.pdf, 1971.

[27] E. MacAskill, J. Borger, N. Hopkins, N. Davies, and J. Ball, “GCHQ taps fibre-optic cables for secret access to world,Äôs communications,” The Guardian, vol. 21, p. 2013, 2013.

[28] K. Martersteck and A. Spencer, “The 5ESS Switching System: Introduction,” AT&T Technical Journal, vol. 64, no. 6, pp. 1305–1314, 1985.

[29] P. Young, “The history of STC,” IEE Proceedings A (Physical Science, Measurement and Instrumentation, Management and Education, Reviews), vol. 133, no. 5, pp. 319–327, 1986.

[30] D. Moralee, “The System X project,” Electronics & Power, vol. 25, no. 8, pp. 544–551, 1979.

[31] A. Suciu and T. Carean, “Benchmarking the True Random Number Generator of TPM Chips,” http://arxiv.org/pdf/1008.2223v1.pdf, 2010.

[32] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978.

[33] R. Rivest, A. Shamir, and L. Adleman, “Cryptographic communications system and method,” vol. United States, 1983.

[34] British Standards Institute, BS ISO/IEC 27001:2005 BS 7799-2:2005 Information technology — Security techniques — Information security management systems — Requirements. London: BSI, 2005.

[35] ——, BS ISO/IEC 27002:2005 BS 7799-1:2005 Incorporating corrigendum no. 1 Information technology — Security techniques — Code of practice for information security management. London: BSI, 2005.

[36] M. Auty, S. Creese, M. Goldsmith, and P. Hopkins, “Inadequacies of Current Risk Controls for the Cloud,” in Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, 2010, pp. 659–666.

[37] Communication Electronics Security Group, CESG Tailored Assurance Methodology. Cheltenham: GCHQ, 2007.

[38] ISO/IEC, ISO/IEC 15408-1:2009 Information technology – Security techniques – Evaluation criteria for IT security – Part 1: Introduction and general model. ISO, 2009.

[39] D. Kleidermacher, “Towards a High Assurance Multi-level Secure PC for Intelligence Communities,” in Technologies for Homeland Security, 2008 IEEE Conference on Technologies for Homeland Security, 2008 IEEE Conference on, 2008, pp. 609–614.

[40] J. Jones, “The Importance of the “Evaluated Configuration” in Common Criteria Evaluations,” http://blogs.microsoft.com/cybertrust/2006/05/24/, 2006.

[41] R. Glott, E. Husmann, A. Sadeghi, and M. Schunter, “Trustworthy Clouds Underpinning the Future Internet,” The Future Internet, pp. 209–221, 2011.

[42] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds,” in Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009, pp. 199–212.

[43] Department of Defense, DoD 5200.28-STD Department of Defense Trusted Computer System Evaluation Criteria. US Department of Defence DoD, 1985.

[44] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, “Flicker: An execution infrastructure for TCB minimization,” in ACM SIGOPS Operating Systems Review, vol. 42(4). ACM, 2008, pp. 315–328.

[45] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in Proc. of the 2010 IEEE Symposium on Security and Privacy. IEEE Computer Society Press, 2010, pp. 143–158.

[46] L. Duflot, O. Grumelard, O. Levillain, and B. Morin, "ACPI and SMI handlers: some limits to trusted computing," Journal in computer virology, vol. 6, no. 4, pp. 353–374, 2010.

[47] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in Proceedings of the 3rd ACM workshop on Cloud computing security workshop. ACM, 2011, pp. 29–40.

[48] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud." in USENIX Security symposium, 2012, pp. 159–173.

[49] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013, pp. 430–444.

[50] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in 2988, K. Jensen and A. Podelski, Eds. Springer Berlin Heidelberg, 2004, pp. 168–176.

[51] M. Christodorescu, R. Sailer, D. Schales, D. Sgandurra, and D. Zamboni, "Cloud security is not (just) virtualization security: a short paper," in Proceedings of the 2009 ACM workshop on Cloud computing security. ACM, 2009, pp. 97–102.

[52] A. Sabelfeld and A. Myers, "Language-based information-flow security," IEEE Journal on Selected Areas in Communications, vol. 21, no. 1, pp. 5–19, 2003.

[53] P. Ryan and S. Schneider, "Process algebra and non-interference," in Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE. IEEE, 1999, pp. 214–227.

[54] S. K. Nair, "Remote Policy Enforcement Using Java Virtual Machine," Ph.D. dissertation, Amsterdam University, Amsterdam, 2010.

[55] V. Haldar and M. Franz, "Mandatory access control at the object level in the java virtual machine," http://www.vivekhaldar.com/pubs/macvm.pdf, 2008.

[56] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for Java," in Proceedings of the 21st Annual Computer Security Applications Conference. Citeseer, 2005, pp. 303–311.

[57] ——, "Semantic remote attestation: a virtual machine directed approach to trusted computing," in Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium. USENIX Association, 2004.

[58] A. Farzan, F. Chen, J. Meseguer, and G. Roşu, "Formal analysis of Java programs in JavaFAN," in Computer Aided Verification. Springer, 2004, pp. 242–244.

[59] P. Giambiagi, "Secrecy for mobile implementations of security protocols," Licentiate Thesis, Royal Institute of Technology, Stockholm, 2001.

[60] M. Dam and P. Giambiagi, “Confidentiality for mobile code: The case of a simple payment protocol,” in csfw. Published by the IEEE Computer Society, 2000, p. 233.

[61] Microsoft, “A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003,” https://support.microsoft.com/en-us/kb/875352, 2006.

[62] Hewlett Packard, “Data Execution Prevention,” http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf, 2004.

[63] Aleph One, “Smashing the stack for fun and profit,” Phrack magazine, vol. 7, no. 49, 1996.

[64] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” Proceedings of the 14th ACM conference on Computer and communications security, pp. 552–561, 2007.

[65] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing return-oriented programming to RISC,” Proceedings of the 15th ACM conference on Computer and communications security, pp. 27–38, 2008.

[66] L. Davi, A. Sadeghi, and M. Winandy, “Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks,” in Proceedings of the 2009 ACM workshop on Scalable trusted computing. ACM, 2009, pp. 49–54.

[67] B. Cantrill, M. Shapiro, and A. Leventhal, “Dynamic instrumentation of production systems,” in Proceedings of the annual conference on USENIX Annual Technical Conference. USENIX Association, 2004, pp. 15–28.

[68] L. Davi, A. Sadeghi, and M. Winandy, “ROPdefender: A detection tool to defend against return-oriented programming attacks,” in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011, pp. 40–51.

[69] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A. Sadeghi, and C. Stüble, “A protocol for property-based attestation,” in Proceedings of the first ACM workshop on Scalable trusted computing. ACM, 2006, pp. 7–16.

[70] A. Sadeghi and C. Stüble, “Property-based attestation for computing platforms: caring about properties, not mechanisms,” in Proceedings of the 2004 workshop on New security paradigms. ACM, 2004, pp. 67–77.

[71] U. Kühn, M. Selhorst, and C. Stüble, “Realizing property-based attestation and sealing with commonly available hard-and software,” in Proceedings of the 2007 ACM workshop on Scalable trusted computing. ACM, 2007, pp. 50–57.

[72] P. Sangster, H. Khosravi, M. Mani, K. Narayan, and J. Tardo, “RFC 5209: Network Endpoint Assessment (NEA): Overview and Requirements,” https://tools.ietf.org/html/rfc5209, 2008.

[73] T. C. Group, "TCG Trusted Network Communications TNC Architecture for Interoperability," http://www.trustedcomputinggroup.org/files/resource_files/DDBB9EF7-1A4B-B294-D03A12D6C4A8B356/TNC_Architecture_v1_5_r4.pdf, 2012.

[74] I. Bente and J. von, Helden, "Towards Trusted Network Access Control," in Proceedings of the First International Conference Future of Trust in Computing, D. Gawrock, H. Reimer, A.-R. Sadeghi, and C. Vishik, Eds. Vieweg+Teubner, 2009, pp. 157–167.

[75] Bell Telephone Laboratories Inc, "Unix Seventh Edition Manual," http://plan9.bell-labs.com/7thEdMan/index.html, 1979.

[76] A. Singh, Mac OS X internals: a systems approach. Addison-Wesley Professional, 2006.

[77] D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman, "Microkernel operating system architecture and Mach," in In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, 1992, pp. 11–30.

[78] D. Solomon, "The Windows NT kernel architecture," Computer, vol. 31, no. 10, pp. 40–47, 1998.

[79] S. Delaune, S. Kremer, M. Ryan, and G. Steel, "Formal analysis of protocols based on TPM state registers," Proc. of the 24th IEEE Computer Security Foundations Symposium (CSF 2011). IEEE Computer Society Press, 2011.

[80] D. Solomon, M. Russinovich, and A. Ionescu, Windows internals. Microsoft Press, 2009.

[81] Communication Electronics Security Group, "BitLocker Drive Encryption Suitable for the protection of RESTRICTED data on Microsoft Windows Vista SP1 and SP2 and Microsoft Windows 7," http://www.cesg.gov.uk/news/docs_pdfs/BitLocker-drive-encryption-jan-2011.pdf, 2011.

[82] A. Lin, "Automated analysis of security APIs," Ph.D. dissertation, MIT, 2005.

[83] W. McCune and L. Wos, "Otter-the cade-13 competition incarnations," Journal of Automated Reasoning, vol. 18, no. 2, pp. 211–220, 1997.

[84] D. Jackson, "Alloy: a lightweight object modelling notation," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 11, no. 2, pp. 256–290, 2002.

[85] Mitre, "CVE-2008-0166," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166, 2008.

[86] Debian, "Debian Security Advisory DSA-1571-1 openssl – predictable random number generator," http://www.debian.org/security/2008/dsa-1571, 2008.

[87] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga, "Security Evaluation of Scenarios Based on the TCG's TPM Specification," in ESORICS 2007, J. Biskup and J. Lopez, Eds. Berlin / Heidelberg: Springer, 2007, pp. 438–453.

[88] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," International Journal of Information Security, vol. 10, no. 2, pp. 63–81, 2011.

[89] L. de Moura, S. Owre, H. Rueü, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in 3114, R. Alur and D. Peled, Eds. Springer Berlin Heidelberg, 2004, pp. 496–500.

[90] J. Millen, J. Guttman, J. Ramsdell, J. Sheehy, and B. Sniffen, "Analysis of a Measured Launch," http://www.mitre.org/work/tech_papers/tech_papers_07/07_0843/07_0843.pdf, 2007.

[91] K. L. McMillan, "The SMV system," in Symbolic Model Checking. Springer, 1993, pp. 61–85.

[92] A. Datta, J. Franklin, D. Garg, and D. Kaynar, "A logic of secure systems and its application to trusted computing," in Proc. of the 30th IEEE Symposium on Security and Privacy. IEEE Computer Society Press, 2009, pp. 221–236.

[93] C. Fournet and J. Planul, "Compiling information-flow security to minimal trusted computing bases," Programming Languages and Systems, pp. 216–235, 2011.

[94] S. Delaune, S. Kremer, M. Ryan, and G. Steel, "Formal analysis of protocols based on TPM state registers," in Proc. of the 24th IEEE Computer Security Foundations Symposium. IEEE Computer Society Press, 2011.

[95] S. Berger, R. Caceres, K. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: virtualizing the trusted platform module," in Security '06: 15th USENIX Security Symposium. USENIX Association, 2006.

[96] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 164–177, 2003.

[97] M. Rosenblum, "VMware "virtual platform"," in Proceedings of hot chips, vol. 1999, 1999, pp. 185–196.

[98] N. Kumar and V. Kumar, "Bitlocker and Windows Vista," http://www.nvlabs.in/uploads/projects/nvbit/nvbit_bitlocker_white_paper.pdf, 2008.

[99] S. Bajaj and R. Sion, "TrustedDB: A trusted hardware-based database with privacy and data confidentiality," Knowledge and Data Engineering, IEEE Transactions on, vol. 26, no. 3, pp. 752–765, 2014.

[100] M. Lindemann, R. Perez, R. Sailer, L. Van, Doorn, and S. W. Smith, "Building the IBM 4758 secure coprocessor," Computer, vol. 34, no. 10, pp. 57–66, 2001.

[101] T. Chumash and D. Yao, "Detection and prevention of insider threats in database driven web services," Trust Management III, pp. 117–132, 2009.

[102] M. Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in Proceedings of the 11th Working Conference on Reverse Engineering, 2004, pp. 27–36.

[103] Google Inc, "Certificate Transparency," http://www.certificate-transparency.org, 2013.

[104] M. V. Wilkes, Time sharing computer systems. Elsevier Science Inc., 1975.

[105] R. Morris and K. Thompson, "Password security: A case history," Communications of the ACM, vol. 22, no. 11, pp. 594–597, 1979.

[106] D. Wagner and I. Goldberg, "Proofs of security for the Unix password hashing algorithm," in Advances in Cryptology— ASIACRYPT 2000. Springer, 2000, pp. 560–572.

[107] Data Encryption Standard, "FIPS pub 46," Appendix A, Federal Information Processing Standards Publication, 1977.

[108] J. Klensin, R. Catoe, and P. Krumviede, "RFC2095: IMAP/POP AUTHorize extension for simple challenge/response," https://tools.ietf.org/html/rfc2095, 1997.

[109] P. Leach, C. Newman, and A. Melnikov, "RFC2831: Using Digest Authentication as a SASL mechanism," https://tools.ietf.org/html/rfc2831, 2000.

[110] J. Viega, "Practical random number generation in software," in Computer Security Applications Conference, 2003. Proceedings. 19th Annual. IEEE, 2003, pp. 129–140.

[111] P. Gutmann, "Software Generation of Practically Strong Random Numbers." in Usenix Security, 1998.

[112] B. Barak, R. Shaltiel, and E. Tromer, "True Random Number Generators Secure in a Changing Environment," in 2779, C. D. Walter, C. Koc, and C. Paar, Eds. Springer Berlin Heidelberg, 2003, pp. 166–180.

[113] B. Dole, S. Lodi, and E. Spafford, "Misplaced trust: Kerberos 4 session keys," in Network and Distributed System Security, 1997. Proceedings., 1997 Symposium on. IEEE, 1997, pp. 60–70.

[114] I. Goldberg and D. Wagner, "Randomness and the Netscape browser," Dr Dobb's Journal-Software Tools for the Professional Programmer, vol. 21, no. 1, pp. 66–71, 1996.

[115] P. Hellekalek, "Good random number generators are (not so) easy to find," Mathematics and Computers in Simulation, vol. 46, no. 5, pp. 485–505, 1998.

[116] M. D. MacLaren and G. Marsaglia, "Uniform random number generators," Journal of the ACM (JACM), vol. 12, no. 1, pp. 83–89, 1965.

[117] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," International Journal of Information Security, vol. 1, no. 1, pp. 36–63, 2001.

[118] D. Bernstein, T. Lange, and P. Schwabe, "The Security Impact of a New Cryptographic Library," in Progress in Cryptology, LATINCRYPT 2012 7533, A. Hevia and G. Neven, Eds. Springer Berlin Heidelberg, 2012, pp. 159–176.

[119] L. Blum, M. Blum, and M. Shub, "A Simple Unpredictable Pseudo-Random Number Generator," SIAM Journal on Computing SIAM J. Comput., vol. SIAM Journal on Computing 15, no. 2, pp. 364–383, 1986.

[120] N. Ferguson and B. Schneier, Practical Cryptography (Computer Science). John Wiley & Sons, 2003.

[121] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 197, Specification for the ADVANCED ENCRYPTION STANDARD (AES)," http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[122] S. Checkoway, M. Fredrikson, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham, "On the practical exploitability of Dual EC in TLS implementations," in USENIX Security, vol. 1, 2014.

[123] A. Young and M. Yung, "The prevalence of kleptographic attacks on discrete-log based cryptosystems," in 1294, J. Kaliski, BurtonS., Ed. Springer Berlin Heidelberg, 1997, pp. 264–276.

[124] D. Shumow and N. Ferguson, "On the possibility of a back door in the NIST SP800-90 Dual Ec Prng," Proc. Crypto '07, 2007.

[125] N. Perlroth, "Government Announces Steps to Restore Confidence on Encryption Standards," New York Times, 2013.

[126] G. Taylor and G. Cox, "Digital randomness," Spectrum, IEEE, vol. 48, no. 9, pp. 32–58, 2011.

[127] Oracle, "Developer's Guide to Oracle Solaris 11 Security. Introduction to the Oracle Solaris Cryptographic Framework," https://docs.oracle.com/cd/E23824_01/html/819-2145/chapter1-1.html, 2011.

[128] Cryptotronix, "Using the TPM's random number generator," http://cryptotronix.com/2014/08/28/tpm-rng/, 2014.

[129] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the linux random number generator," in 2006 IEEE Symposium on Security and Privacy. IEEE, 2006, pp. 15–25.

[130] F. Goichon, C. Lauradoux, G. Salagnac, and T. Vuillemin, "Entropy transfers in the Linux random number generator," , 2012.

[131] D. P. Bovet and M. Cesati, Understanding the Linux kernel. O'Reilly Media Inc, 2005.

[132] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the random number generator of the windows operating system," ACM Transactions on Information and System Security (TISSEC), vol. 13, no. 1, p. 10, 2009.

[133] D. Davis, R. Ihaka, and P. Fenstermacher, "Cryptographic randomness from air turbulence in disk drives," in Advances in Cryptology, CRYPTO '94. Springer, 1994, pp. 114–120.

[134] J. Postel, "RFC 791: Internet protocol," https://tools.ietf.org/html/rfc791, 1981.

[135] ——, "RFC 793: Transmission control protocol," https://tools.ietf.org/html/rfc793, 1981.

[136] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, The design and implementation of the 4.4 BSD operating system. Pearson Education, 1996.

[137] T. A. Beardsley and J. Qian, “The TCP Split Handshake: Practical Effects on Modern Network Equipment,” Network Protocols and Algorithms, vol. 2, no. 1, pp. 197–217, 2010.

[138] D. Lyons, “IEN176: The DECSYSTEM-20 TCP/IP user interface,” https://www.rfc-editor.org/ien/ien176.txt, 1981.

[139] S. M. Bellovin, “Security problems in the TCP/IP protocol suite,” SIGCOMM Comput. Commun. Rev., vol. 19, no. 2, pp. 32–48, 1989.

[140] R. T. Morris, “A Weakness in the 4.2BSD Unix TCP/IP Software,” Computer Science Technical Report Number 117, 1985.

[141] S. J. Templeton and K. E. Levitt, “Detecting spoofed packets,” in DARPA Information Survivability Conference and Exposition, 2003. Proceedings, vol. 1. IEEE, 2003, pp. 164–175.

[142] K. Scarfone and P. Hoffman, “Guidelines on firewalls and firewall policy,” NIST Special Publication, vol. 800, p. 41, 2009.

[143] M. Zalewski, “Strange Attractors and TCP/IP Sequence Number Analysis,” http://lcamtuf.coredump.cx/oldtcp/tcpseq.html, 2001.

[144] ——, “Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later,” http://lcamtuf.coredump.cx/newtcp/, 2002.

[145] D. Kaminsky, “Black ops 2008: It’s the end of the cache as we know it,” Black Hat USA, 2008.

[146] D. Boneh, “Twenty years of attacks on the RSA cryptosystem,” Notices of the AMS, vol. 46, no. 2, pp. 203–213, 1999.

[147] W. Diffie and M. E. Hellman, “New directions in cryptography,” Information Theory, IEEE Transactions on, vol. 22, no. 6, pp. 644–654, 1976.

[148] L. Chen, “Recommendation for key derivation through extraction-then-expansion,” NIST Special Publication, vol. 800, p. 56C, 2011.

[149] Q. H. Dang, “SP 800-135 Rev. 1. Recommendation for Existing Application-Specific Key Derivation Functions,” http://csrc.nist.gov/publications/nistpubs/800-135-rev1/sp800-135-rev1.pdf, 2011.

[150] V. S. Miller, “Use of elliptic curves in cryptography,” in Advances in Cryptology, CRYPTO 85 Proceedings. Springer, 1986, pp. 417–426.

[151] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, and A. Heckert, “NIST special publication 800-22,” A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2001.

[152] M. Bellare and P. Rogaway, “Random oracles are practical: A paradigm for designing efficient protocols,” in Proceedings of the 1st ACM conference on Computer and communications security. ACM, 1993, pp. 62–73.

[153] D. Grawrock, Dynamics of a Trusted Platform: A Building Block Approach. Intel Press, 2009.

[154] Advanced Micro Devices, Secure Virtual Machine Architecture Reference Manual. Advanced Micro Devices, 2005.

[155] B. Blanchet, “An efficient cryptographic protocol verifier based on prolog rules,” in Proc. of the 14th IEEE Computer Security Foundations Workshop. IEEE Computer Society Press, 2001, pp. 82–96.

[156] ——, “Automatic verification of correspondences for security protocols,” Journal of Computer Security, vol. 17, no. 4, pp. 363–434, 2009.

[157] M. Abadi and C. Fournet, “Mobile values, new names, and secure communication,” ACM SIGPLAN Notices, vol. 36, no. 3, pp. 104–115, 2001.

[158] A. Horn, “On sentences which are true of direct unions of algebras,” The Journal of Symbolic Logic, vol. 16, no. 01, pp. 14–21, 1951.

[159] M. H. Van, Emden and R. A. Kowalski, “The semantics of predicate logic as a programming language,” Journal of the ACM (JACM), vol. 23, no. 4, pp. 733–742, 1976.

[160] M. E. Lesk and E. Schmidt, lex: A lexical analyzer generator. Bell Laboratories, 1987.

[161] S. C. Johnson, Yacc: Yet another compiler-compiler. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[162] V. Paxson, W. Estes, and J. Millaway, “Lexical Analysis with Flex,” University of California, p. 28, 2007.

[163] C. Donnelly and R. Stallman, “Bison. The YACC-compatible Parser Generator,” http://www.gnu.org/software/bison/manual/bison.pdf, 2004.

[164] B. W. Kernighan, “The Unix system and software reusability,” IEEE Transactions on Software Engineering, vol. 10, no. 5, pp. 513–518, 1984.

[165] W. Futral and J. Greene, “The Future of Trusted Computing,” in Intel Trusted Execution Technology for Server Platforms, A Guide to More Secure Datacenters. Springer, 2013, pp. 119–128.

[166] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, “Innovative technology for cpu based attestation and sealing,” in Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP, vol. 13, 2013.

[167] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del, Cuvillo, “Using innovative instructions to create trustworthy software solutions,” in Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. ACM, 2013, p. 11.

[168] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” ACM Transactions on Information and System Security (TISSEC), vol. 15, no. 1, p. 2, 2012.

[169] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, “RFC 3711: The Secure Real-time Transport Protocol (SRTP),” https://tools.ietf.org/html/rfc3711, 2004.

[170] C. Chen, H. Raj, S. Saroiu, and A. Wolman, “cTPM: a cloud TPM for cross-device trusted applications,” in Proceedings of the 11th USENIX conference on networked systems design and implementation, 2014, pp. 187–201.

All URLs were re-checked during May 2015, confirming that they were at that point valid and matched the referenced work.