Security Analysis

Alex Summers Chris Tickner

11th June 2002

Abstract

The project report for our Suprema project on Security Analysis. The aim of this document is to give the reader an overview of what Security Analysis is about, why it is important, and how successful it is as a study.

The first chapter aims to explain the general concept and philosophy of Security Analysis, and demonstrate the broad nature of the area with some examples.

The second chapter is a more-detailed study of a particular security issue; that of Buffer Overflows. It examines what attempts and successes the area of Security Analysis yielded in this particular area.

The third chapter is a look at a particular method of formal reasoning, called Pi Calculus. This is in order to demonstrate how formal reasoning techniques can be developed and made useful in the field.

Contents

1 An Introduction to Security Analysis **2**
- 1.1 What is Security Analysis? . . . 2
- 1.2 The CIA principle . . . 3
- 1.3 Types of Security Attack . . . 4
  - 1.3.1 Software Exploits . . . 4
  - 1.3.2 Timing Attacks . . . 5
  - 1.3.3 Denial Of Service (DOS) attacks . . . 6

2 Buffer Overflow Attacks - A Case Study **7**
- 2.1 Overview . . . 7
- 2.2 Background to Buffer Overflow Attacks . . . 7
  - 2.2.1 Memory Violations . . . 8
  - 2.2.2 Stack Organisation . . . 8
- 2.3 Goals and Methods . . . 9
  - 2.3.1 How can the code be placed in the address space? . . . 9
  - 2.3.2 How can the program be made to jump to the malicious code? . . . 10
- 2.4 Attack Variants . . . 10
- 2.5 Counter-measures . . . 11
  - 2.5.1 Program Analysis . . . 11
  - 2.5.2 Coding Standards . . . 12
  - 2.5.3 Alternative Languages . . . 12
  - 2.5.4 Run-time protection . . . 13
- 2.6 Conclusions . . . 14

3 An example of formal reasoning - Pi Calculus **15**
- 3.1 A Brief overview of Calculi . . . 15
- 3.2 π-Calculus (Re-visited) . . . 17
- 3.3 Examples of communication in π calculus. . . . 18
- 3.4 Conclusions about π-Calculus (And formal reasoning in general) 20

Chapter 1

An Introduction to Security Analysis

1.1 What is Security Analysis?

The study of security in Computer Science is a rapidly growing area, and of increasing importance to the world in general. Computers are now so widely used that it is increasingly important to be able to ensure that guarantees can be made of the functionality and proper security of a system. For example, computers guard most of the top secret information in the world, handle people's day to day banking and offer means of private communication between individuals.

At the same time, computer software and hardware has been increasing rapidly in complexity, and it becomes ever more difficult to say anything with certainty about how a particular system will behave, purely based on the intentions of the programmer or designer. Not only have many exploits been discovered to make use of bugs that may unintentionally exist, but there are many more subtle forms of attacking a 'well-designed' system, and more techniques are developed all the time.

The increase in computer application and correspondingly in computer literacy means that not only is the potential reward for breaking into a 'secure' system increasingly valuable, but more and more people are able to learn the techniques required then ever before.

For these reasons, the formal and scientific study of Security Analysis has become crucial, in order to try and keep up with the explosions both in system complexity and in security attack frequency. Reliable solutions are required to security issues, and methods of formally reasoning about the security of a system are more and more necessary, as abstract discussion of the programming/design techniques becomes less and less adequate.

Security Analysis has given rise to the development of many new methods of formal reasoning; designed specifically for use in the study of computer security. These formal methods are a key feature of the whole study of Security Analysis;

while it is good to be able to apparently solve a security issue, it is much more satisfactory to be able to prove formally that your new system does not suffer from the same kind of weakness. Unfortunately a complete proof is not always possible depending on the nature of the problem (and the formal methods that have been developed for tackling it).

A particular problem for formal Security Analysis is the growth in non-deterministic behaviour of a computer system. Multi-tasking in particular gives rise to a whole new level of complexity in the area, since a system might only be vulnerable if it performs tasks in a particular order, and this order cannot in general be predicted deterministically. For this reason, many systems are considered too complex for any definite statements about their behaviour in a given situation to be possible. Instead, Security Analysis must focus on finding a best-possible solution; perhaps examining the statistical likelihood of a particular event occurring.

1.2 The CIA principle

A simple but widely-applicable principal used in security analysis is the CIA model. The initials stand for Confidentiality, Integrity and Availability; three key principles which should be guaranteed in any kind of secure system. The model is applicable across the whole subject of Security Analysis, from access to a user's internet history to security of encrypted data across the internet. If any one of the three can be breached it can have serious consequences for the parties concerned.

Confidentiality .

Confidentiality is the ability to hide information from those people unauthorised to view it. It is perhaps the most obvious aspect of the CIA triad when it comes to security; but correspondingly, it is also the one which is attacked most often. Cryptography and Encryption methods are an example of an attempt to ensure confidentiality of data transferred from one computer to another.

It is important to understand however that Confidentiality alone is not enough for a secure system. If is is possible that the data to be transferred may not remain intact (for example, if it is corrupted along the way) or even that the system may break down and so prevent access to the data even to the authorised users then this is still not acceptable.

It should also be realised that the concept of Confidentiality in Security Analysis goes far beyond just methods of data encryption. For example, malicious programs such as Trojan horses must be thwarted or else the secret information could be being leaked even as it is being typed. And similarly, the information must be stored securely at its destination or else

the efforts in ensuring its Confidential status in transit will be rendered pointless.

Integrity .

The ability to ensure that data is an accurate and unchanged representation of the original secure information. One type of security attack is to intercept some important data and make changes to it before sending it on to the intended receiver. This is especially applicable in the field of quantum cryptography. While classical cryptography employs various mathematical techniques to restrict eavesdroppers from learning the contents of encrypted messages, in quantum mechanics the information is protected by the laws of physics. In classical cryptography an absolute security of information cannot be guaranteed, but quantum cryptography can detect whether or not a piece of data was intercepted by a third party.

Availability .

It is important to ensure that the information concerned is readily accessible to the authorised viewer at all times. Some types of security attack attempt to deny access to the appropriate user, either for the sake of inconveniencing them, or because there is some secondary effect. For example, by breaking the web site for a particular search engine, a rival search engine may become more popular.

1.3 Types of Security Attack

The variety of security issues that can be considered in the scope of Security Analysis is huge, as there are many different ways in which a system might be compromised. A few of these are described here, but it should be understood that these are only a few common varieties.

Security Analysis is a rapidly evolving field. New security attacks are invented frequently and creatively; perhaps an entirely new breed of exploits might come to light in a year's time.

1.3.1 Software Exploits

Many kinds of Security Attacks involve finding and exploiting undesirable behaviour in existing software; particularly in software which has some security-orientated role. Sometimes the undesirable behaviour results from careless programming, but at others it may be the result of some very obscure effect of the system.

A simple example of a software exploit would be a web browser which stored a user's passwords in plain text form on their computer, or perhaps a website which received the passwords unencrypted. In either case, the nature of the

information means a much greater level of confidentiality would be suitable as the passwords are vulnerable to inspection by some unauthorised third party.

A more complex and subtle example of a software attack is covered in the next chapter; Buffer Overflows are used to exploit specific but relatively common behaviour in C programs, where the lack of memory protection can be exploited to cause malicious code to be executed by a secure (and privileged) application.

1.3.2 Timing Attacks

Sometimes, supposedly secret information can be obtained by examining the timing behaviour of a system. This is an area which most programmers would not consider, but it can yield surprisingly clear results. For example, suppose a password-checking program is written to check the characters of an input password one at a time, and aborts as soon as one does not match. Now, if the user can accurately observe the time it takes for a password to be checked, they can pick apart the target password one character at a time by seeing which one slows the function down a little. Although this may seem implausible in practice, the technique is more readily applicable to much slower and simpler systems, and has been shown to be effective on some kinds of smartcard-security systems.

A similar sort of attack might be possible if a user was able to observe the power consumption of a system while it is carrying out some top-secret task. In fact, research has been made into possible weaknesses in cryptographic algorithms due to the power-consumption behaviour that can be observed while a processor is working on the encryption and decryption algorithms.

Another example of a timing attack; suppose you own a company which markets products on the internet, and would like to know when a potential customer logs on to your site whether they have already visited the sites of your competitors. It is possible to write a short piece of html code which accesses three files sequentially; that is, it won't attempt to access the second until it has successfully accessed the first etc. Now, by making the first and third of these three files be accessible through your web site, and the second through the web-site of your competitor (perhaps an image on their page), you get your site to log the times of the accesses to files 1 and 3, and a simple subtraction gives you an approximation as to how long it took to access file 2 from your competitor.

Why is this information of any use? Well, the scheme is to build this html into your web-site, so that it is executed silently by a user whenever they connect to your home page. Now, if the user has previously visited the web site of your competitor then the address will be cached by the user's browser (and/or web cache etc.) and this will speed up the access to the file on your competitors site. So by comparing your results with a number of calibrating samples you can work out if the competitors website was cached by the user or not.

The use of this technique highlights a security flaw which cannot be solved

perfectly. An extreme solution is to disable all caching, so that no time-difference can be observed between sites you have visited and sites you have not. Alternatively, random delays could be introduced into the behaviour of your browser, in order to disguise what is happening. However both of these solutions involve a trade-off with convenience; it is obviously not possibly to randomly speed-up the behaviour of the browser, only to introduce delays.

This kind of trade-off situation is reflected in many areas of Security Analysis; where no ideal solution exists but approximate solutions or compromises can be reached.

1.3.3 Denial Of Service (DOS) attacks

Some kinds of security attack aim specifically to crash a particular piece of software or entire system. Various methods can be employed; the techniques described in the next chapter will often cause crashes. A fairly common method is to find some service that responds to requests from the outside world by allocating resources to the request in some capacity, and then repeating requests of this kind over and over again until the computer is starved of resources and crashes.

One interesting counter-measure developed against this style of attack is known as Client Puzzles. Here, the worry is that a client may attempt to open hundreds and hundreds of connections to a server at great speed, and the server will not be able to cope. The problem arises from the fact that it is very inexpensive for a client to initiate a connection, while it is relatively expensive for the server to allocate the appropriate resources.

The solution comes by reacting to an incoming request by first giving the client computer a 'puzzle' to solve before it will allocate any resources to that client. This might be along the lines of brute-force breaking a low-security cryptographic function by trial and error. The puzzle is not intended to be very costly; but the aim is to change the balance of expenditure so that now it is more costly to the client to complete the initiation of a connection than it is to the server. Since in most applications there is no real reason why many many connections should be required all at once and at great speed this is a very elegant solution; the delays encountered are not significant enough to be noticeable to a user, but are long enough to prevent a DOS attack of this kind.

Chapter 2

Buffer Overflow Attacks - A Case Study

2.1 Overview

There exist several kinds of security flaw which result simply from 'bad' programming. By this it is meant that they could be avoided if more-careful analysis of programming style and structure were employed. Perhaps the most significant of these is the Buffer Overflow attack, which is discussed in this chapter.

The aim of this chapter is to provide an example of a real-life security issue, and to demonstrate how different Security Analysis approaches can be applied to attempt to solve the problem. Buffer Overflows are a good example to study as the concept is relatively straightforward, but they are a highly-significant issue in real software development.

Buffer Overflows are also an area in which Security Analysis is particularly fruitful. While some security problems have no perfect solution and always involve a difficult trade-off, the issues raised by Buffer Overflows can in principle be overcome entirely by various approaches to analysis of the problem.

All example code given in this section is in the C language, as are most genuine examples of Buffer Overflow and similar attacks.

2.2 Background to Buffer Overflow Attacks

> "Buffer overflows appear to be the most common problems reported in May, with degradation-of-service problems a distant second. Many of the buffer overflow problems are probably the result of careless programming, and could have been found and corrected by the vendors, before releasing the software, if the vendors had performed elementary testing or code reviews along the way." – Michele Crabb, The SANS Network Security Digest. SANS, 1997.

```
void CopyPrint(char * source)
{
  char buffer[100];

  strcpy(buffer,source);
  printf(buffer);
}
```

Figure 2.1: CopyPrint example function

2.2.1 Memory Violations

The basic concept of a buffer overflow attacks is to exploit the lack of bounds-checking on memory access in languages such as C. An array in C is just a contiguous (each element is consecutive to the next) block of allocated memory. While it is guaranteed that attempts to modify memory within the bounds of a declared array are permissible, the converse is not necessarily true; there is no guarantee that it is forbidden to modify the contents of memory outside the bounds of the array.

Segmentation violations and the like protect against some illegal memory accesses (depending on the Operating System etc.), and in particular against illegal accesses to faraway memory locations (e.g. an application trying to write to Operating-System memory). However, A program in C is given near-complete freedom within its own memory space, and in particular it is unlikely that memory accesses in the vicinity of an array are going to be noticed. It is this weakness that buffer overflows exploit.

2.2.2 Stack Organisation

Consider the simple CopyPrint function given in Figure 2.1

The array 'buffer' is local to this function (an automatic variable) and will be allocated on the stack, in the context of this function call. In particular, when the function is called, the return address and other return information (e.g. previous stack pointer) will be stored on the stack, and all local variables are kept on the stack somewhere below them.

Now, what happens if the string that source points to is longer than 100 characters? Well, as it turns out, strcpy() will just continue copying from the source to the destination until it reads a null character, so in fact it will overflow the bounds of our buffer. And herein lies the weakness, because if the overflow is large enough then this string-copy can overwrite the return address on the stack, so that when the function completes it now cannot return as it was meant to. The goal then, of this type of attack, is to replace the original return address

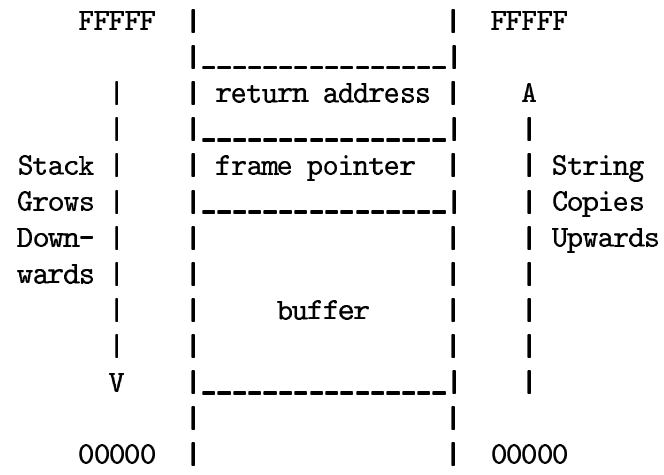

Figure 2.2: Stack organisation during a function call

with a new value; causing the program to 'jump' to a location not intended by the programmer.

2.3 Goals and Methods

In general, the aim of a software exploit like this is to gain privileged access to a computer system; perhaps to gain control of it entirely. For this reason, attacks are often made on software which already has privileged rights (e.g. programs run with root permissions) with the aim of spawning a new shell window with root permissions that the attacker can then make use of. The code for obtaining such a window is often very simple, e.g. exec(sh) , and so is easy to write a machine-code representation for. The task of completing a Buffer Overflow attack then breaks down into two subgoals:

1. Getting the required malicious code into the address space of the program.
2. Getting the program to jump to the malicious code for execution.

These can be achieved by various means as described below:

2.3.1 How can the code be placed in the address space?

It is possible to inject the malicious code as part of the attack; either writing it into the buffer, or writing it at some point beyond the return address in memory. Then, the return address need only be replaced with this appropriate nearby address and the code will get executed.

Alternatively, it may be that the program code that the attacked wants to execute already resides somewhere in memory. Or perhaps there is a system function exec() which is accessible, and then the attacker need only work out how to specify an argument (sh) to this function and then overwrite a return address with the location of this system call.

2.3.2 How can the program be made to jump to the malicious code?

We've already discussed the most common method, which is to overwrite the return details of a function (sometimes referred to as the function's activation record). However, other more-subtle techniques can be employed..

One of the more esoteric features of the C language is the area of function pointers. These essentially provide a type of variable which 'points' to the address of a function and can in turn be modified to point to others, and 'called' by providing suitable arguments for the function it points to. Without wanting to delve to deeply into the syntax, the material point here is that if a function pointer can be overwritten by a buffer overflow technique then this provides an alternative means of attack. When the function pointer is 'called' later on it now causes the program to jump to a new location, specified by the attacker.

There exist other even-more obscure exploits but it is enough to see that the general aims remain the same; target code must be somehow introduced and the program must somehow be made to jump to it.

2.4 Attack Variants

Unfortunately, it is not just the simple strcpy() example given above that can give rise to a buffer overflow vulnerability. For example, consider the code for NewCopyPrint given in Figure 2.3

The size-specifier appears to ensure that the string contained in temp must be 'safe' as if we pass in a very long source string then it will just be cut-off after the 80th character. But what if the string that source points to is in fact of the form

```
"%100d <evil code>"
```

Now what happens? Well in fact, usually the spurious %d conversion causes no error (it is not seen at compile time since it doesn't appear explicitly in the function call to sprintf), but rather takes the unspecified parameter off the stack just as it normally would. This behaviour can incidentally be used to investigate the contents of the stack in order to help work out where the return address is

```
void NewCopyPrint(char * source)
{
  char temp[100];
  char buffer[100];

  sprintf(temp, "%80s", source);
  sprintf(buffer, temp);
  printf(buffer);
}
```

Figure 2.3: NewCopyPrint example function

stored. Imagine the effect of "%x %x %x %x %x %x %x %x %x %x" as an input string to the above function; now it prints the previous 10 integer values off the stack (in hexadecimal) so long as this doesn't cause a crash. However, the real aim in our example was to cause an overflow, and this works because of the specified 100 padding in the

The point of this second example is to show that it is not always trivial to identify a buffer overflow weakness; and while it is easily to list the C-functions which can be considered 'unsafe', it is impractical to ban use of these entirely, and often difficult to discern between a sensible and a dangerous usage.

2.5 Counter-measures

So what counter-measures can Security Analysis provide? As mentioned above, this is in principle a soluble problem, but the ways of going about tackling it vary. We will discuss a number of approaches below.

2.5.1 Program Analysis

One obvious approach is a static analysis of pre-written code, in order to verify its security. This has the advantage of applying human reasoning to the problem, which can often pick up unusually obscure bugs. Human reasoning is of course slow, and a highly expensive commodity, but many consultancy companies exist who will provide such a service.

There also exist a number of software packages for automated analysis of program code (e.g. Berkeley, Stanford, Stake.com have produced tools which perform various kinds of automatic analysis). An automated approach has the advantage of being much faster and cheaper, but is of course limited by the analysis algorithms employed. As well as static analysis, some software tools have taken a fault-injection approach; they automatically try to overflow any

buffers they can find access to and observe the behaviour of the software.

Unfortunately, although many bugs may be detected by Program Analysis, neither manual or automated approaches are perfect, and some will slip through. In addition, analysis will only solve individual problems and must be applied to every piece of software developed. In general, it is better to find some more widely applicable practice.

2.5.2 Coding Standards

Now that the knowledge of Buffer Overflows as a security weakness is widespread it is hoped that more robust code will be produced as standard. However, the introduction of enforced coding standards to prevent these sorts of flaws is generally impractical; at best it can be hoped that individual programmers pay attention to the issues involved, and perhaps some companies make it a target for their employees. Programmers are in general very defensive of their own methods, but the C community in particular has a culture which values performance over other considerations.

Even if common coding flaws are well-known; we have seen that they can creep in in subtle ways, and a general awareness of the issues is still no guarantee that errors do not creep in accidentally.

2.5.3 Alternative Languages

Many alternative languages exist which are 'type-safe'; that is, the sorts of memory exploitations described above are not possible because access to variables is by a much stricter mechanism which ensures that only the allocated memory is available for reading and writing.

A good example of a 'type-safe' language is Java, which contains no pointers explicitly and implements a memory-handling system which is largely transparent to the programmer. Although it is still possible to create complex dynamic data structures in Java, the C-style pointers are replaced by 'references' in this language, which are restricted only to point to the locations of declared variables and certainly cannot be manipulated to point to particular addresses explicitly.

The lack of direct memory manipulation makes it impractical to write very low-level applications in the Java language (e.g. Device Drivers), and in the particular example of Java there is a performance issue as it is an interpreted language (rather than compiled), but for many applications it may be a suitable solution, and certainly provides one approach to avoid buffer overflow vulnerabilities in future code.

A complicating factor is that although Java itself is type-safe, the Java Virtual Machine may well be written in C, and it is sometimes possible to apply Buffer Overflow Exploits to the machine itself! Although this is harder to engineer, in this way it may be possible to produce similar results to those described above, even to supposedly safe Java applications.

2.5.4 Run-time protection

A different policy towards tackling buffer overflows is to introduce some kind of run-time mechanism for intercepting the attacks as they are attempted, rather than trying to achieve perfectly secure code in the first place. The two main approaches in this area are array-bounds checking and code pointer integrity checking.

Array-bounds checking

Array-bounds checking involves ensuring that any array-manipulation functions access memory only within the prescribed bounds of the arrays they are meant to be using. In particular, a rogue strcpy function would be prevented from writing beyond the end of the destination array, and so a buffer overflow could be prevented.

Various research projects have yielded results in this area; perhaps the most complete solution being the gcc patch developed by Paul Kelly and Richard Jones at Imperial College. This performs checks on all pointer and array accesses at runtime, in order to ensure that security flaws (and other pointer bugs) are identified and prevented from executing. Special care has been made to ensure that this patch is backward compatible with other C code, unlike many other attempts. The main disadvantage of the system is the marked performance penalty caused by the overheads of performing the relevant pointer calculations. The most extreme example (simple matrix multiply) shows an execution 30 times slower than normal.

A link to this project can be found at:

`http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html`

Code pointer integrity

Code pointer integrity is an alternative idea, where methods are put in place to ensure that any pointer which is 'jumped' to by the program (for example a return address in a function call) has not been modified before it is followed. One such technique is employed by the StackGuard software (originally developed as a patch to gcc); which uses 'canaries' to protect the stack of a function! These are extra words of data inserted between the return address (the whole activation record, in fact) and the area of the stack used for local variables. The name comes from the analogy with miners canaries which were used to indicate when some subtle danger was afoot! The idea is that before the function returns it checks that the 'canary' is still intact, and if not it aborts the program as the stack has been tampered with. Although it is still possible in principle to overwrite the return address on the stack, it is now impossible to do it by a buffer overflow technique without overwriting the canary.

It is important that the contents of a canary cannot be duplicated easily by an attacker, otherwise they might just overwrite the canary words with their original contents and thus go undetected. In order to avoid this, either randomised canaries are used, or canaries consisting of a mixture of terminating characters (null, newline, carriage return and end of file). Where terminator canaries are used (what a great name!) the would-be attacker is thwarted as whichever C function is used to overflow the buffer it will not be able to write these characters without terminating; for example strcpy() will terminate as soon as it finds a null character in its input.

The StackGuard system has gone on to be marketed commercially and its success may be attributed to its relatively high-performance (it introduces an overhead per function call rather than per pointer reference as with bounds-checking) and reasonable compatibility. It is however impossible to use StackGuard with software which accesses an activation record directly. For example the gnu debugger will not work properly in tracing the stack of a StackGuard-compiled program, and the linux kernel cannot be compiled using StackGuard as it makes use of activation records as defined in the standard way.

Extensions of the StackGuard method exist to cover function pointers and the like (PointGuard) and format string exploits like the %100d example above (FormatGuard) and all appear to be reasonably successful in their attempts.

Further information on these particular projects can be found at the following sites..

`http://www.immunix.org/stackguard.html`
`http://www.wirex.com/Immunix/Guardian/FormatGuard.html`

2.6 Conclusions

From this chapter we have seen a real security threat, and a number of methods which are employed to try and tackle it. Security Analysis is necessary to develop consistent and reliable solutions to security issues; not only in solving any immediately apparent issues but also in preventing them happening again.

While any programmer with some knowledge of the concept and technique of Buffer Overflow attacks could change their own code to remove an apparent weakness, it is much more practical to develop widely-applicable and standard solution techniques, such as run-time protection and fault-injection software. In this way Security Analysis is an exceptionally valuable study which can produce very tangible long-term benefits.

Chapter 3

An example of formal reasoning - Pi Calculus

3.1 A Brief overview of Calculi

As we have seen so far, there are many possible problems with systems and communication. The question which we shall now turn our attention to is: Is there any way to formally specify, and reason about, our systems? More importantly, is there any way to reason about communications between processes in our systems?

The answer to both of these questions is, of course, yes. There are many calculi around that can be used to define systems and reduce the large number of potential actions down to a set of minimal solutions. Here we shall just outline a few of the calculi to give you some idea as to their uses and applications.

λ-Calculus. (A. Church)

> The λ-calculus is a formal mathematical system devised by Alonzo Church in 1934. Its aim was to investigate functions, function application and recursion. There are only two legal operations in λ-calculus, construction of a function with one argument and a specified body or invocation of one of the defined functions.
>
> The body of a function can be any legal expression, that is to say it could be variables, function constructions or invocation of other functions. However, the only possible argument for a function is another function. From this small basis, we can construct a programming language that can express any computation that any other language can express.
>
> It has influenced many programming languages but none more so than the functional programming languages, such as Lisp, Haskell, Miranda and ML. As well as being a firm basis for many functional languages, Lambda calculus provides the metalanguage for formal definitions in denotational

semantics and, although not itself concerned with security analysis in the sense of communication, it is used as a starting point for many other calculi.

Calculus of Communicating Systems. (R. Milner)

The Calculus of Communicating Systems (CCS) is an algebra for specifying and reasoning about concurrent systems. It was created by R. Milner in 1980.

As an algebra, CCS provides a set of terms, operators and axioms that can be used to write and manipulate algebraic expressions. The expressions define the elements of a concurrent system and the manipulations of these expressions reveal how the system behaves.

π-Calculus. (R. Milner, J. Parrow and D. Walker)

Presented in 1989 by Robin Milner, Joachim Parrow and David Walker. Its aim was to present a calculus for communicating systems where the processes and be naturally defined to have a changing structure.

The π-Calculus is a process algebra. That is, it is an algebraic approach to the study of concurrent processes. It allows communication between processes via channels, and channel names are also allowed to be send along a channel. It was designed to extend CCS and to incorporate the ideas of lambda calculus by admitting processes, or links to processes, to be passed as values between concurrent systems. As in lambda calculus, where everything was a function, in pi Calculus, everything is a process.

Note: We shall go into more detail about π -calculus in a moment.

Spi Calculus. (M. Abadi and A. D. Gordon)

Introduced by Martin Abadi and Andrew D Gordon as an extension of π-Calculus designed for the evaluation of cryptographic protocols. In an excerpt from their paper "A Calculus for Cryptographic Protocols - The Spi Calculus." they describe the calculus as

> "The Spi calculus is an extension of the π-calculus with cryptographic primitives. It is designed for the description and analysis of security protocols, such as those for authentication and for electronic commerce."

Ambient Calculus. (L.Cardelli and A. D. Gordon)

The un-typed calculus was introduced in 1998 by Lucia Cardelli and Andrew D. Gordon in their paper "Mobile Ambients"

The calculus allows active processes to move between different sites. Rather than focusing on the communication of values, channels or processes (as in π-calculus) they focus on the movement of the process itself.

This is an extension of the ideas of mobility in Java, where only passive code could move from one site to the next.

3.2 π-Calculus (Re-visited)

To try to give people an idea as to the nature of formal specification in security analysis, we shall now attempt to give an introduction to some of the ideas, syntax and semantics of the π calculus.

In π calculus, we have only two classes of entities, *names* (these can be anything from link names to variables or ordinary data values) and *agents* (any process expression). The set of *names* is denoted as $\mathcal{N}$ and the set of *agent identifiers* is $\mathcal{K}$. Now we can define the following types of operations on the agents as follows:

1. A *summation* $\sum_{i \in I} P_i$, where the index set, *I*, is finite.

   This means that the whole process will act as one or another of the P_i's. An empty summation is denoted as **0** and is known as *inaction*. **0** is a process that can do nothing.

2. A *prefix* form $\bar{y}x.P$, $y(x).P$ or $\tau.P$.

   '$\bar{y}\langle x \rangle.P$' is called the *negative prefix* and can be thought of as a message *x* that is being sent along a link *y* to another process and then behaving like *P*.

   '$y(x).P$' means that the arbitrary variable *z* is received along the link *y* and then behaves as the process $P\{z/x\}$. (NOTE: $P\{z/x\}$ as a substitution, and is defined later.)

   '$\tau.P$' is known as a *silent prefix*. It performs the silent action τ and then behaves like P.

3. A *composition* $P_1|P_2$.

   This is the same as agents P_1 and P_2 acting in parallel. The components may act independently, but also if an output from $\bar{y}x.P_1$ synchronises with an input action $y(x).P_2$ then a silent action is created with respect to the composite agent $P_1|P_2$.

4. A *restriction* $(x)P$.

   This agent will behave like P, although no communication is allowed along port *x* or $\bar{x}$. However, *P* will allow communication between its various components along the link *x* since the communication described would be a silent action τ (via the reasoning above P could be considered as two processes $P_1|P_2$ with a communication across the private link *x*.).

5. A *match* $[x = y]P$.

   This will behave like P if x and y are identical. If not, then P will behave like $\mathbf{0}$.

6. A *defined agent* $A(y_1, \ldots, y_n)$.

   For the agent identifier A, there is a unique *defining equation* $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$ where the names $x_1, \ldots, x_n$ are both distinct and the only names which may occur free in P.

3.3 Examples of communication in π calculus.

Now that we have become acquainted with all the notation in πcalculus, what can we actually *do* with it? We hope to show you a series of examples so that you can get a brief idea of how to use the syntax of π calculus and how you can identify the problems with a system that you have specified.

Over the next three examples, we shall

1. Define a system using agents and names.
2. Look at how this system communicates between the different agents.
3. Look for problems within the system.
4. Try to find a way to deal with those problems.

Example 1. Here we shall look at a simple communication in a system specified using π calculus. the system is modeling a telephone call, with how a message gets from a user to the switchboard via a phone.

```
Speaker = air<M>                 // The speaker will send out a
                                 message through the air
Phone = air(x).wire<x>           // The message is taken from
                                 the air by the Phone and is
                                 transmitted along a wire
BT = wire(x).switchboard<x>      // The message is taken from the
                                 wire by BT and is sent out along
                                 the switchboard link
System = Speaker | Phone | BT
```

Now, we would like to know what is going on in the system, so if we model the communication by reducing System into its least solution, we get:

$$\texttt{Seaker | Phone} \longrightarrow \texttt{wire<M>}.$$

Phone takes an input from *air*, and transfers the data to the *wire* port. Speaker provides us with an input from *air*, so the phone can give us message M through the port *wire*.

$$\texttt{wire<M> | BT} \longrightarrow \texttt{switchboard<M>.}$$

BT takes an input from the port *wire* and transfers it to the port *switchboard*, so if we have `wire<M>|wire(x).switchboard<x>` we will gain the message `M` from the command `wire(x)` and we will then need to replace all occurrences of x with M, so BT will send out `switchboard<M>`

So, now if we apply both of these rules together, then we will have:

$$\texttt{Speaker | Phone | BT} \longrightarrow \texttt{switchboard<M>.}$$

So we have now successfully communicated with the switchboard!

Example 2 Although we have managed to communicate with the switchboard, we have a problem. Suppose we have a person trying to listen in on our conversation. It is obvious that the communication was intercepted if this occurred at either *air* or *switchboard*, but if they were sneaky, they could tap our phoneline!

```
WireTap = wire(x).wire<x>.listener<x>  // This copies the
                                       message from the wire
                                       to the listener!
```

This is possible because of the fact that the port *wire* is globally visible. So now our communication with the switchboard can be monitored by any external party!

```
WireTap | System                  →
WireTap | Speaker | Phone|BT      →
WireTap | wire<M> | BT            →
wire<M>.listener<M> | BT          →
listener<M>|switchboard<M>
```

This is Obviously not a good state of affairs if, for example, we were trying to send sensitive information across our phoneline to the switchboard. However we do have the ability to restrict the communication along a certain port using the ν operation.

Example 3. Since we know that our communication can be monitored by anyone else, we wish to put a stop to people listening in to our conversation. To do this, we shall specify *wire* to be a restricted port for the compositional agent `System`.

System′ ≡ (ν wire)System	//System′ is defined to be equal to our first implementation of System, but with the port wire to be restricted.

Now if anyone tries to listen into the message transmitted on our new system, System′ then we have:

WireTap | System′	$\longrightarrow$
WireTap | (ν wire)(Speaker | Phone | BT)	$\longrightarrow$
wire(x).wire<x>.listener<x> | (ν wire)(wire<M> | BT)	$\longrightarrow$
wire(x).wire<x>.listener<x> | switchboard<M>	

The message *M* has been communicated from the speaker to the switchboard, and can no longer be intercepted. *M* can be transferred within System′ along the port *wire* as it would be the equivalent of a silent action, However as noone outside System′ can listen in to the communication across the port wire. In other words, WireTap can no longer listen in to any information transferred from the phone to the switchboard.

Thus, to conclude these examples, we have shown that not only can we communicate a message from a phone to a switchboard, but the communication itself is protected from external agents listening in to what information is being transferred.

3.4 Conclusions about π-Calculus (And formal reasoning in general)

Obviously in this paper, only a small subset of the π calculus has been discussed, but we hope to have given readers some idea as to its capabilities. It is able to help you identify which communications are *silent* and which are not. By simply looking at this, it becomes quite obvious about whether or not another agent may be able to take advantage of a security flaw which is based upon a flaw in the transfer of the data itself.

A good example of a flaw in the transfer of data was identified in the Needham- -Schroder protocol, identified in the mid nineties. The protocol had been around in common use for 18 years before the flaw was actually identified. If we suppose Alice is sending Bob her credit-card details. To do this, Alice would request a public key, so that she could encrypt her details and send them, securely, to Bob. In the protocol there was a way for Charles to impersonate Bob, and thus Charles knows about Alice's credit-card.

This was done by Charles intercepting Bob's public key while it was on route to Alice. Charles would then impersonate Bob and send Alice Charles' public key. Alice would be unaware of this problem, and so she would encrypt her card details and send them back to Charles, thinking that she was in fact sending

them to Bob. Charles could then decrypt Alice's details and make fraudulent purchases with the information.

This scenario is obviously a problem of the communication between the agents, and not a fault of the cryptographic methods themselves and illustrates why it is so important to have detailed method for formal review of protocols and systems before they are deployed in critical applications.