Static Program Analysis for Security

K. Gopinath

Department of Computer Science and Automation

Indian Institute of Science

Bangalore 560 012, India

Contents

Abstract

In this chapter, we discuss static analysis of the security of a system. First, we discuss the background on what types of static analysis is feasible in principle and then move on to what is practical. We next discuss static analysis of buffer overflow and mobile code followed by access control. Finally, we discuss static analysis of information flow expressed in a language that has been annotated with flow policies.

1 Introduction

Analysing a program for security holes is an important part of current computing landscape as security has not been an essential ingredient in a program's design for quite some time. With the critical importance of a secure program becoming clearer in the recent past, designs based on an explicit security policy are likely to gain prominence.

Static analysis of a program is one technique to detect security holes. Compared to monitoring executions runtime (which may not have the required coverage), a static analysis - even if incomplete due to loss of precision - gives potentially an analysis on all runs possible instead of just the ones seen so far.

However, security analysis of an arbitrary program is extremely hard. First, what security means is often unspecified or underspecified. Either the definition is too overly strict and cannot cope with the "common sense" requirement, or too broad and not useful. For example, one definition of security is through the notion of "non-interference"[26]. If it is very strict, even cryptoanalytically strong encryption and decryption does not qualify as secure as there is information flow from encrypted text to plain text[27]. If it is not very strict, by definition, some flows are not captured that are important in some context for achieving security and hence again not secure. For example, if electromagnetic emissions are not taken into account, the key may be easily compromised[37]. A model of what security means is needed and this is by no means an easy task[38]. Schneider[25] has a very general definition: a security policy defines a binary partition of all (computable) sets of executions: those that satisfy and those that do not. This is general enough to cover access control policies (a program's behavior on an arbitrary individual execution for an arbitrary finite period), availability policies (behavior on an arbitrary individual execution over an infinite period) and information flow (behavior in terms of the set of all executions).

Secondly, the diagonalization trick is possible and many analyses are undecidable. For example, there are undecidable results with respect to viruses and malicious logic[34]: it is undecidable whether an arbitrary program contains a computer virus. Similarly, there exist viruses for which no error-free detection algorithm exists.

Recently, there has been some interesting results on computability classes for enforcement mechanisms[25] with respect to execution monitors, program rewriting and static analysis. Execution monitors intervene whenever execution of an untrusted program is about to violate the security policy being enforced. They are typically used in operating systems using structures such as access control lists or used when executing interpreted languages by runtime type checking. It is possible to rewrite a binary so that every access to memory goes through a monitor. While this can introduce overhead, optimization techniques can be used to reduce the overhead. In many cases, the test can be determined to be not necessary and removed by static analysis. For example, if a reference to a memory address m has already been checked earlier, it may not be necessary for a later occurrence.

Program rewriting modifies the untrusted program before execution to make it incapable of violating the security policy. An execution monitor can be viewed as a special case of program rewriting but Hamlen et al[25] point out certain subtle cases. Consider a security policy that makes halting a program illegal; an execution monitor cannot enforce this policy by halting as this would be illegal! There are also classes of policies in certain models of execution monitors that cannot be enforced by any program rewriter.

As is to be expected, the class of statically enforceable policies is the class of recursively-decidable properties of programs (class Π_0 of the arithmetic hierarchy): a static analysis has to be necessarily total (i.e. terminate) and return safe or unsafe. If precise analysis is not possible, we can relax the requirement by being conservative in what it returns (i.e. tolerate false positives). Execution monitors are the class of co-recursively enumerable languages (class Π_1 of the arithmetic hierarchy)[1].

A system's security is specified by its security policy (such as access control or information flow model) and implemented by mechanisms such as physical separation or cryptography[2]. Consider access control. An access control (AC) system guards access to resources whereas an information flow model classifies information to prevent disclosure. Access control is a component of security policy, while cryptography is one technical mechanism to effect the security policy. Systems have employed basic access control since timesharing systems began ('60) (eg. Multics, Unix). Simple access control models for such "standalone" machines assumed the universe of users known resulting in the scale of model being "small". A set of access control moves can unintentionally leak a right (i.e. give access when it should not be). If it is possible to analyse the system and ensure that such a result is not possible, the system can be said to be secure. But theoretical models inspired by these systems (the most well known being the Harrison, Ruzzo, Ullman (HRU) model) showed "surprising" undecidability results[23] chiefly due to the unlimited number of subjects and objects possible. Technically, in this model, it is undecidable whether a given state of a given protection system is safe for a given generic right. However, the need for automated analysis of policies was small in the past as the scale of the systems was small.

Information flow analysis, in contrast to access control, concerns itself with the downstream use of information once obtained after proper access control. Carelessness in not ensuring proper flow models has resulted recently in the encryption system in HD-DVD and BluRay disks to be compromised (the key protecting movie content is available in memory)[3].

Overt models of information flow specify the policy how data should be used explicitly whereas covert models use "signalling" of information through "covert" channels such as timing, electromagnetic emissions, etc. Note that the security weakness in the recent HD-DVD case is due to

[1]A security policy P is co-recursively enumerable if there exists a Turing machine M that takes an arbitrary execution monitor EM as an input and rejects it in finite time if $\sim P(EM)$ otherwise $M(EM)$ loops forever.

[2]While cryptography has rightfully been a significant component in the design of large scale systems, its relation to security policy, esp its complementarity, has not often been brought out in full. "If you think cryptography is the solution to your problem, you don't know what your problem is," Roger Needham.

[3]The compromise of the security system DeCSS in DVDs was due to cryptanalysis but in the case of HD-DVD it was simply improper information flow.

improper overt information flow. Research on some overt models of information flow such as Bell and LaPadula[4] was initiated around the 60's, inspired by the classification of secrets in the military. Since operating systems are the inspiration for work in this area, models of secure operating systems were developed such as C2, B2, etc.[61]. Work on covert information flow progressed in the 70's. However, the work on covert models of information flow in proprietary operating systems (eg. DG-UNIX) was expensive and too late that it could not be used on the by then obsolescent hardware. Showing that Trojan horses did not use covert channels to compromise information was found to be "ultimately unachievable" [39].

Currently, there is a considerable interest in studying overt models of information flow as the extensive use of distributed systems and recent work with widely available operating systems such as SELinux and OpenSolaris has expanded the scope of research. The scale of model has become "large" with the need for automated analysis of policies being high.

Access control and information flow policies can both be statically analysed, with varying effectiveness depending on the problem. Abstract interpretation, slicing and many other compiler techniques can also be used in conjunction. Most of the static analyses induce a constraint system that needs to be solved to see if security is violated. For example, in one buffer overflow analysis[33], if there is a solution to a constraint system, then there is an attack. In another analysis in language based security, non-existence of a solution to the constraints is an indication of a possible leak of information.

However, static analysis is not possible in many cases[19] and has not yet been used on large pieces of software. Hence, exhaustive checking using model checking[7] is increasingly being used when one wants to gain confidence about a piece of code. In this chapter, we consider model checking as a form of static analysis. We will discuss access control on Internet that uses model checking (Section 4.4). We will also discuss this approach in the context of SELinux where we check if a large application has been given only sufficient rights to get its job done[49].

In spite of the many difficulties for analysing the security of a system, policy frameworks such as access control and information flow analysis, and mechanisms such as cryptography have been used to make systems "secure". However, for any such composite solution, we need to trust certain entities in the system such as the compiler, the BIOS, the (Java) runtime system, the hardware, digital certificates, etc.: essentially the "chain of trust" problem. That this is a tricky problem has been shown in an interesting way by Ken Thompson[24]; we will discuss it below. Hence the need for a "small" trusted computing base (TCB): all protection mechanisms within a system (hardware, software, firmware) for enforcing security policies.

In the past (early '60's), operating systems were small and compilers larger in comparison. The TCB could justifiably be the OS, even if uncomfortably larger than one wished. In today's context, the compilers need not be as large as current operating systems (for eg. Linux kernel or Windows is many millions lines of code) and a TCB could profitably be the compiler. Hence, a compiler analysis of security is meaningful nowadays and likely to be the only way large systems (often a distributed system) can be crafted in the future. Using a top level security policy, it may be possible to automatically partition the program so that the resulting distributed system is secure by design[28].

To illustrate effectiveness of static security analysis, we first discuss a case where static analysis fails completely (Ken Thomson's Trojan Horse). We then outline some results on the problem of detecting viruses, and then a case study where static analysis can be in principle be very hard (equivalent to cryptanalysis in general) but is actually much simpler due to a critical implementation error. We will also briefly touch upon obfuscation that exploits difficulty of analysis.

We then discuss static analysis of buffer overflows, loading of mobile code and access control and information flow, illustrating the latter using Jif[53] language on a realistic problem. We conclude with likely future directions in this area of research.

1.1 A "Dramatic Failure" of Static Analysis: Ken Thompson's "Trojan Horse"

Some techniques to defeat static analysis can be deeply effective; we summarize Ken Thompson's ingenious "Trojan Horse" trick[24] in a compiler that uses self-reproduction, self-learning and self-application. We follow his exposition closely.

Firstly, self-reproduction is possible; for example, one can construct a self-reproducing program such as `((lambda x.  (list x x)) (lambda x.  (list x x)))`. Second, it is possible to teach a compiler (written in its own language) to compile new features (the art of bootstrapping a compiler): this makes "self-learning" possible. We again give Ken Thompson's example to illustrate it. Let us say that a lexer knows about `'\n'` (newline) but not `'\v'` (vertical tab). How does one teach it to compile `'\v'` also? Let the initial source be:

```
c=next();
if (c !='\\') return(c);
c=next();
if (c !='\\') return('\\')
if (c ='n') return('\n')
```

Adding to the compiler source

```
if (c ='v') return('\v')            //error
```

and using previous compiler binary does not work as that binary does not know about `'\v'`. However, the following works:

```
 if (c ='v') return(11)
```

Now a new binary (from the new source using the old compiler binary) knows about `'\v'` in a portable way! Now we can use it to compile the previously uncompilable statement:

```
if (c ='v') return('\v')
```

The compiler has "learned". Now, we can introduce a Trojan Horse into login program that has a backdoor to allow a special access to log in as any user:

```
compile(char *s) { (* compiler main that accepts s as input (a program) *)
    if match(s, "pattern_login") {
       (* ``Trojan'' compile for a login program in the system*)
       compile("bug_login"); return
    }
    ... (* normal compile *)
}
```

But this is easily detectable (by examining the source). To make this not possible, we can add a second Trojan Horse aimed at the compiler:

```
    if match(s, "pattern_compiler") {
       (* ``Trojan'' compile for a compiler program in the system*)
       compile("bug_compiler"); return
    }
```

Now we can code a self-reproducing program that reintroduces both Trojan Horses into the compiler with a learning phase where the buggy compiler binary with 2 Trojan Horses now reinserts it into any new compiler binary compiled from clean source! The detailed scheme is as follows: first a clean compiler binary (A) is built from a clean compiler source (S). Next, as part of the learning phase, a modified compiler source (S') can be built that incorporates the bugs. The logic in S' looks at the source code of any program submitted for compilation and, say by pattern matching, decides whether a program submitted is a login or a compiler program. If it decides that it is one of these special programs, it reproduces[4] one or more Trojan horses (as necessary) when presented with a clean source. Let the program S' be compiled with A. We have a new binary A' that reinserts the two Trojans on any clean source!

The virus exists in the binary but not in the source. This is not possible to discern unless the history of the system (the sequence of compilations and alterations) is kept in mind. Static analysis fails spectacularly!

1.2 Detecting Viruses

As discussed earlier, the virus detection problem is undecidable. A successful virus encapsulates itself so that it cannot be detected; the opposite of "self-identifying data". For eg, a very clever virus would put logic in I/O routines so that any read of suspected portions of disk returns the original "correct" information! A polymorphic virus inserts "random" data to vary signature.

[4]It is easiest to do so in the binary being produced as the compiled output.

More effectively, it can create a random encryption key to encrypt rest of virus and store the key with the virus.

1.2.1 Cohen's results

Cohen's impossibility result states that it is impossible for a program to perfectly demarcate a line, enclosing all and only those programs that are infected with some virus: There is no algorithm that can properly detect all possible viruses[34]: $\forall Alg.\exists Virus.Alg$ does not detect $Virus$. For any candidate computer virus detection algorithm A, there is a program `p(pgm): if A(pgm) then exit; else spread`. Here, "spread" means behave like a virus. We can diagonalize this by setting `pgm=p` and a contradiction follows immediately as `p` spreads iff `not A(p)`.

Similarly, Chess and White[35] show that there exists a virus which no algorithm perfectly detects i.e. with no false positives: $\exists Virus.\forall Alg.Alg$ does not detect $Virus$. Also, there exists a virus which no algorithm loosely-detects, i.e. claims to find the virus but infected with some other virus: $\exists Virus.\forall Alg.Alg$ does not loosely-detect $Virus$. In other words, there exist viruses for which, even with virus analyzed completely, no program that detects just that virus with no false positives. Further, another Chess & White's impossibility result states that there is no program to classify programs only those with a virus `V` and not include any program without any virus.

Furthermore, there are interesting results concerning *polymorphic viruses*: these viruses generate a set of other viruses that are not identical to itself but related to it in some way (for example, be able to reproduce the next one in sequence). If the size of this set is greater than 1, call the set of viruses generated the viral set S.

An algorithm A detects a set of viruses S iff for every program p, $A(p)$ terminates and returns TRUE iff p is infected with some virus V in S. If W is a polyvirus, for any candidate W-detection algorithm C, there is a program $s(pgm)$ that is an instance of W:

```
  if subroutine_one(pgm) then exit, else {
    replace text of subroutine_one with a random program;
    spread; exit; }
subroutine_one: return C(argument)
```

This can be diagonalized by `pgm=s` resulting in a contradiction: `if C(s) true exit; otherwise polyvirus`.

Hence, it is clear that there are serious limits to static analysis.

1.3 A Case Study for Static Analysis: GSM Security Hole

We will now attempt to situate static analysis with other analyses, using the partitioning attack[37] on GSM hashing on a particular implementation as an example. GSM uses COMP128 algorithm for encryption. A 16B (128b) key of subscriber (available with the base station) along with a 16B challenge from the base station) is used to construct a 12B (96b) hash. The first 32b is sent as response to the challenge and the rest 64b used as the session key. The critical issue is that there should be no leakage of the key in any of the outputs; *this includes any electromagnetic leakages!* during the computation of the hash. The formula for computation of hash (it has a butterfly structure) is as follows:

```
//X is 32B (16B key || 16B challenge)
//Tj[r] is a lookuptable of 8-j bits value with r rows:
//alg uses T0[512], T1[256], T2[128], T3[64], T4[32]
for j=0..4
  for k=0..2^j -1
    for l=0..2^(4-j) -1
      m=l+k*2^(5-j)
      n=m+2^(4-j)
      y=(X[m]+2*X[n]) mod 2^(9-j)
      z=(2*X[m]+X[n]) mod 2^(9-j)
      X[m]=Tj[y]
      X[n]=Tj[z]
```

If we expand y in the expression for X[m], we have

```
X[m]=Tj[(X[m]+2*X[n]) mod 2^(9-j)]
```

A simple-minded flow analysis will show that there is direct dependence of the EM leakage on key; hence, this is not information flow secure. However, cryptographers, using the right "confusion" and "diffusion" operators such as the above code, have shown that the inverse can be very difficult to compute. Hence, even if very simple static analysis clearly points out the flow dependence of the EM leakage on the key, it is not good enough to crack the key. However, even if the mapping is cryptanalytically strong, there are often some "implementation" bugs that can give away the key. One attack is possible if one does not follow the following principle[37] of statistical independence (or more accurately *non-interference* that will be discussed later):

Relevant bits of all intermediate cycles and their values should be statistically independent of the inputs, outputs and sensitive information.

Normally, those bits that emit high EM emissions are good candidates for analysis. One set of candidates are the array and index values as they need to be amplified electrically for addressing memory. They are therefore EM sensitive whereas other internal values may not be so.

Due to the violation of this principle, a cryptographically strong algorithm may have an implementation that leaks secrets. For example, many implementations use 8b microprocessors as COMP128 is optimized for them, so the actual implementation for T0 is 2 tables T00, T01 (each 256 entries):

```
//X is 32B (16B key || 16B challenge)
//Tj[r] is a lookuptable of 8-j bits value with r rows:
//alg uses T0[512], T1[256], T2[128], T3[64], T4[32]
for j=0..4
  for k=0..2^j -1
    for l=0..2^(4-j) -1
      m=l+k*2^(5-j)
      n=m+2^(4-j)
      y=(X[m]+2*X[n]) mod 2^(9-j)
      z=(2*X[m]+X[n]) mod 2^(9-j)
      if (y<256) X[m]=Tj0[y] else Tj1[y-256] // old: X[m]=Tj[y]
    if (z<256) X[m]=Tj0[z] else Tj1[z-256] // old: X[n]=Tj[z]
```

If the number of intermediate lookups from tables `T00` or `T01` have statistical significance, then due to the linearity of the index `y` with respect to `R` for the first round, some information can be gleaned about the key. The technique of differential cryptanalysis is based on such observations. In addition, if it is possible to know when access changes from one table (say, `T00`) to another (`T01`) by changing `R`, then the `R` value where it changes is given by `K` + 2*`R`=256, from which `X`, the first byte of the GSM key, can be determined.

In general, we basically have a set of constraints, such as

$0 \leq x + 2y_1 \leq 127$

$128 \leq x + 2y_2 \leq 256$

where y_1 and y_2 are two close values that map the index into different arrays (T0 or T1)[5].

If there is a solution to these Diophantine equations, then we have an attack. Otherwise, no. Since the cryptographic confusion and diffusion operations determine the difficulty (esp, with devices such as S-boxes in DES), in general the problem is equivalent to the cryptanalysis problem. But if we assume that the confusion and diffusion operations are linear in subkey and other parameters (as in COMP128), we just need to solve a set of linear diophantine equations[29].

What we can learn from the above is the following: We need to identify EM sensitive variables. Other values can be "declassified"; even if we do not take any precautions, we can assume they cannot be observed from an EM perspective. We need to check the flow dependence of the EM sensitive variables (i.e. externally visible) on secrets that need to be safeguarded.

Recently, there has been work that implies that success or failure of branch prediction presents observable events that can be used to crack encryption keys.

The above suggests the following problems for study:

- Automatic downgrading of "insensitive" variables.

- What is the minimal declassification to achieve desired flow properties?

[5]in general, $l \leq f(a, b) \leq u$,... with f being an affine function for tractability.

1.4 Obfuscation

Given that static analyses are often hard, there are some applications that use it to good effect. An example is the new area of “obfuscation” of code so that it cannot be reverse-engineered easily. Obfuscation is the attempt to make code “unreadable” or “unintelligible” in the hope that it cannot be used by competitors. This is effected by performing semantic preserving transformations so that automatic static analysis can reveal nothing useful. In one instance, control flow is intentionally altered so that it is difficult to understand or use it by making sure that any analysis that can help in unravelling code is computationally intractable (for eg, PSPACE-hard or NP-hard). Another example is introducing aliases intentionally as certain alias analysis problems are known to be hard (if not undecidable), esp in the interprocedural context. Since it has been shown that obfuscation is, in general, impossible[30], static analysis in principle could be adopted to undo obfuscation unless it is computationally hard.

2 Static Analysis of Buffer Overflows

Since the advent of the Morris worm in 1988, buffer overflow techniques to compromise systems has been widely used. Most recently, the SQL slammer worm in ’03, using a small UDP packet (376B), compromised 90% of all target machines worldwide in less than 10 minutes.

Since buffer overflow on a stack can be avoided, for example, by preventing the return address from being overwritten by the “malicious” input string, array bounds checking of the input parameters by the callee is one technique. Due to the cost of this check, it is useful to explore compile time approaches that eliminate the check through program analysis. Wagner[31] uses static analysis (integer range analysis) but it has false positives due to imprecision in pointer analysis, interprocedural analysis, etc. and lack of information on dynamically allocated sizes.

CCured[36] uses static analysis to insert runtime checks to create a type-safe version of C program. CCured classifies C pointers into SAFE, SEQ or WILD pointers. SAFE pointers require only a null check. SEQ pointers require a bounds check as these are involved in pointer arithmetic but the pointed object is known statically while WILD ones require a bounds check as well as a runtime check as it is not known what type of object it points to at runtime. For such dynamically-typed pointers, we cannot rely on the static type; instead, we need, for example, run-time tags to differentiate pointers from non-pointers.

Ganapathy et al. [33] solve linear programming problems arising out of modelling C string programs as linear programs to identify buffer overruns. Constraints result from buffer declarations, assignments and function call/returns. C source is first analysed by a tool that builds a program dependence graph for each procedure, an interprocedural CFG, ASTs for expressions, along with points-to and side-effect information. A constraint generator then generates 4 constraints for each pointer to a buffer (between max/min buffer allocation and max/min buffer index used), 4 constraints on each index assignment (between previous and new values as well as for the highest and lowest values), 2 for each buffer declaration, etc. A taint analysis next attempts to identify and remove any uninitialized constraint variables to make it easy for the constraint solvers.

Using LP solvers, the best possible estimate of the number of bytes used and allocated for each buffer in any execution is computed. Based on these values, buffer overruns are inferred. Some false positives are possible due to the flow-insensitive approach followed; these have to be manually resolved. Since infeasible linear programs are possible, they use an algorithm to identify irreducibly inconsistent sets. After such sets of constraints are removed, further processing is done before solvers are employed. Their approach also employs techniques to make program analysis context sensitive.

Ashcraft and Engler[48] use a "metacompilation" (MC) approach to catch potential security holes. For example, any use of "untrusted input" [6] could be a potential security hole. Since a compiler potentially has information about such input variables, a compiler can statically infer some of the problematic uses and flag them. To avoid hardwiring some of these inferences, the MC approach allows implementers to add rules to the compiler in the form of high-level system-specific checkers. Jaeger et al.[49] use a similar approach to make SELinux aware of 2 trust levels to make information flow analysis possible; as of now, it is not possible. We will discuss this further in

[6]Examples in the Linux kernel code are system call parameters, routines that copy data from user space, and network data.

Section 4.3.

3 Static Analysis of Safety of Mobile Code

The importance of safe executable code embedded in webpages (such as Javascript), applications (as macros in spreadsheets), OS kernel (as drivers, packet filters[50], profilers such as DTrace[51]), cellphones or smartcards is becoming more and more important everyday, With unsafe code (especially one that is a Trojan), it is possible to get elevated privileges that can ultimately compromise the system. Recently, Google Desktop search[32] could be used to compromise a machine (to make all of its local files available outside, for example) in the presence of a malicious Java applet as Java allows an applet to connect to its originating host,

The most simple model is the "naive" sandbox model where there are restrictions such as limited access to local filesystem for the code but this is often too restrictive. A better sandbox model is that of executing the code in a virtual machine implemented either as an OS abstraction or as a software isolation layer and using emulation. We will discuss the latter here. In the latter solution, the safety property of the programming language and the access checks in the software isolation layer are used to guarantee security.

Since OO languages such as Java and C# have been designed for making possible "secure" applets, we will consider OO languages here. Checking whether a method has access permissions may not be local. Once we use a programming language with function calls, the call stack has information on the current calling sequence. Depending on this path, a method may or may not have the permissions. Stack inspection can be carried out to protect the callee from the caller by ensuring that the untrusted caller has the right credentials to call a higher privileged or trusted callee. However, it does not protect the caller from the callee in case of callback or event-based system. We need to compute the intersection of permissions of all methods invoked per thread and base access based on this intersection. This protects in both the directions.

Static analysis can be carried out to check security loop holes introduced by extensibility in OO languages. Such holes can be introduced through subclassing that override methods that check for corner case important for security. We can detect potential security holes by using a combination of model checking and abstract interpretation: first, compute all the possible execution histories; pushdown systems can be used for representation. Next, use temporal logic to express properties of interest (such as: a method from an applet cannot call a method from another applet). If necessary, use abstract interpretation and model checking to check properties of interest.

Another approach is that of the proof carrying code (PCC). Here, mobile code is accompanied by a proof that the code follows the security policy. As a detailed description of the above approaches for the safety of mobile code is given in the 1st edition of this handbook[18], we will not discuss it here further.

4 Static Analysis of Access Control Policies

Butler Lampson[6] introduced access control as a mapping from {entity, resource, op} to {permit, deny} (as commonly used in operating systems). Later models have introduced structure for entities such as roles ("role based access control") and introduced a noop to handle the ability to model access control modularly by allowing multiple rules to fire: {role, resource, op} to {permit, deny, noop}. Another significant advance is access control with anonymous entities: the subject of trust management, which we discuss in Section 4.4.

Starting from the early simple notion, theoretical analysis in the HRU system[23] of access control has the following primitives:

- create subject s: creates new row, column in access control matrix (ACM)
- create object o: creates new column in ACM
- destroy subject s: deletes row, column from ACM
- destroy object o: deletes column from ACM
- enter r into A[s, o]: adds r rights for subject s over object o
- delete r from A[s, o]: removes r rights from subject s over object o

Adding a generic right r where there was not one is "leaking". If a system S, beginning in initial state $s0$, cannot leak right r, it is safe with respect to the right r. It turns out with the above primitives, there does not exist an algorithm for determining whether a protection system S with initial state $s0$ is safe with respect to a generic right r. A Turing machine can be simulated by the access control system by the use of the infinite two-dimensional access control matrix to simulate the infinite Turing tape, using the conditions to check the presence of a right to simulate whether a symbol on the Turing tape exists, adding certain rights to keep track of where the end of the corresponding tape is, etc.

Take grant models[1], in contradistinction to HRU models, are decidable in linear time[1]. Instead of generic analysis, specific graph models of granting and deleting privileges, etc. are used. Koch et al.[44] have proposed an approach in which safety is decidable in their graphical model if each graph rule either deletes or adds graph structure but not both. However, the configuration graph is fixed.

Recently, there has been work on understanding and comparing the complexity of DAC (Graham-Denning) and HRU models in terms of a state transition systems[46]. HRU systems have been shown to be not as expressive as DAC. In the Graham-Denning model, if a subject is deleted, the objects owned are atomically transferred to its parent. In a highly available access control system, however, there is usually more than one parent (a DAG structure rather than a tree) and we need to decide how the "orphaned" objects are to be shared. We need to specify further models (for eg, dynamic separation of duty). If subject is the active entity or leader, further modelling is necessary. The simplest model usually assumes a fixed static alternate leader but this is inappropriate in many critical designs. The difficulty in handling a more general model is that leader election also requires resources that are subject to access control. But for any access control reconfiguration to take place, authentications and authorizations have to be frozen for a short duration till the reconfiguration is complete. Since leader election itself requires access control decisions as it requires network, storage and other resources, we need a special mechanism to keep these outside the purview of the freeze of access control system. The modelling thus becomes extremely complex. This is an area for investigation.

4.1 Case Studies

4.1.1 Firewalls

Firewalls are one widely known access control mechanism. A firewall examines each packet that passes through the entry point of a network and decides whether to accept the packet and allow it to proceed or to discard the packet. A firewall is usually designed as a sequence of rules; each rule is of the form <pred> → <decision> where <pred> is a boolean expression over the different fields of a packet, and the <decision> is either accept or discard. Designing the sequence of rules for a firewall is not any easy task as it needs to be consistent, complete, and compact. Consistency means that the rules are ordered correctly, completeness means that every packet satisfies at least one rule in the firewall, and compactness means that the firewall has no redundant rules. Gouda and Liu[47] have examined the use of "firewall decision diagrams" for an automated analysis and present polynomial algorithms for achieving the above desirable goals.

4.1.2 Setuid Analysis

The access control mechanism in Unix-based systems is based critically on the setuid mechanism. This is also known to be a source of many privilege escalation attacks if this feature is not used correctly. Since there are many variations of setuid in different Unix versions, the correctness of a particular application using this mechanism is difficult to establish across multiple Unix versions. Static analysis of an application along with model of the setuid mechanism is one attempt at checking the correctness of an application.

Chen, Wagner and Dean[20] develop a formal model of transitions of the user IDs involved in the setuid mechanism as a finite state automaton (FSA) and develop techniques for automatic construction of such models. The resulting FSA are used to uncover problematic uses of the Unix API for uid-setting system calls, to identify differences in the semantics of these calls among various Unix systems, to detect inconsistency in the handling of user IDs within an OS kernel, and to check the proper usage of these calls in programs automatically. As an Unix-based system maintains per-process state (e.g., the real, effective, and saved uids) to track privilege levels, a suitably abstracted FSA (by mapping all user ids into a single "non-root" composite id) can be devised to maintain all such relevant information per state. Each uid-setting system call then leads to a number of possible transitions; FSA transitions are labelled with system calls. Let this FSA be called the setuid-FSA. The application program can also be suitably abstracted and modelled as an FSA (the program FSA) that represents each program point as a state and each statement as a transition. By composing the program FSA with the setuid-FSA, we get a composite FSA. Each state in the composite FSA is a pair of one state from the setuid-FSA (representing a unique combination of the values in the real uid, effective uid, and saved uid) and one state from the program FSA (representing a program point). Using this composite FSA, questions such as the following can be answered:

- Can the setuid system call fail? This is possible if an error state in the setuid-FSA part in a composite state can be reached.
- Can the program fail to drop the privilege? This is possible if a composite state can be reached that has a privileged setuid-FSA state but the program state should be unprivileged at that program point.
- Which parts of an application run at elevated privileges? By examining all the reachable composite states, this question can be answered easily.

4.2 "Dynamic" Access Control

Recent models of access control are declarative using rules that encode the traditional matrix model. An access request is evaluated using the rules to decide whether access it to be provided or not. It also helps to separate access control policies from business logic.

Dougherty et.al.[42] use Datalog to specify access control policies. At any point in the evolution of the system, there are facts ("ground terms") that interact with the policies ("datalog rules"); the resulting set of deductions is a fixpoint that can be used to answer queries whether an access is to be allowed or not. In many systems, there is also a temporal component to access control decisions. Once an event happens (eg. a paper is assigned to reviewer), certain accesses get revoked (eg. the reviewer cannot see the reviews of other reviewers of the same paper until he has submitted his own) or allowed. We can therefore construct a transition system with edges being events that have a bearing on the access control decisions. The goal of analysis is now either safety or availability (a form of liveness): namely, is there some accessible state in the dynamic access model which satisfies some boolean expression over policy facts? These questions can be answered efficiently as any fixed Datalog query can be computed in polynomial time in the size of the database, and the result of any fixed conjunctive query over a database Q can be computed in space $O(\log|\ Q\ |)$[42, 22].

Analysis of access control by abstract interpretation is another approach. Given a language for access control, we can model leakage of a right as an abstract interpretation problem. Consider a simple language with assignments, conditionals and sequence (";"). If A is an user, let $[A,\ stmt]s$ represent whether A can execute *stmt* in state s. Also, let $r(q,A)s$ mean that A can read q in state s and $w(p,A)s$ means A can write p in state s. Then we have the following interpretation:

$[A,\ p{=}q]s = r(q,A)s$ and $w(p,A)s$

$[A,$ if c then p else $q]s = r(c,A)s$ and $((c$ and $[A,p]s)$ or (not c and $[A,q]s))$

$[A,\ (a;\ b)]s = [A,\ a]s$ and $[A,\ b]s'$
where s' is the new state after executing a.

Here, A can be a set of users also. Next, if $[A,\ prog](startstate) = 1$, then A can execute *prog*. The access control problem now becomes: Does there exist a program P that A can execute and the program also computes the value of some forbidden value and writes it to a location that A can access? With HRU type of models, the set of programs to be examined is essentially unbounded and we have undecidability. However, if we restrict the programs to be finite, decidability is possible.

It is also possible to model "dynamic access control" by other methods such as using pushdown systems[40], graph grammars[5], or frameworks such as "formal concept analysis and concept lattices"[21] but we will only discuss the access control problem on the Internet that can be modelled using pushdown systems.

4.3 Retrofitting Simple MAC Models for SELinux

Since SELinux is easily available, we will use SELinux as an example for discussing access control. In the past, the lack of such operating systems made research difficult; they were either classified or very expensive.

Any machine hosting some services on the net should not get totally compromised if there is a break-in. Can we isolate the breach to those services and not to affect rest of the system? It is possible to do so if we can use “mandatory access policies” (MAC) rather than the standard “discretionary access policies” (DAC). In a MAC, the system decides how you share your objects whereas in a DAC you can decide how you share your objects. A break-in into a DAC system has the potential to usurp the entire machine whereas in a MAC system the kernel or the system still validates each access according to a policy loaded beforehand.

For example, in some recent Linux systems (eg. Fedora Core 5/6 that is based on Security Enhanced Linux or SELinux) that employ MAC, there is a “targeted” policy where every access to a resource is allowed implicitly but deny rules can be used to prevent accesses. By default, most processes run in an “unconfined” domain but certain daemons or processes[7] (“targeted ones”) run in “locked down” domains after starting out as “unconfined”. If cracker breaks into apache and gets a shell account, it can run only with the privileges of the “locked down” daemon and thus rest of the system is usually safe. The rest of the system is not safe only if there is a way to effect a transition into the “unconfined” domain. With the more stringent “strict policy”, also available in Fedora Core 5/6 that implicitly denies everything and “allow” rules are used to enable accesses, it is even more difficult.

Every subject (process) and object (e.g. file, socket, IPC object, etc) has a security context that is interpreted only by security server. Policy enforcement code typically handles security identifiers (SIDs); SIDs are nonpersistent and local identifiers. SELinux implements a combination of:

- Type Enforcement and (optional) MultiLevel Security: Typed models have been shown to be more tractable for analysis. Type enforcement requires that the type of domains and objects be respected when making transitions to other domains or when acting on objects of a certain

[7] httpd, dhcpd, mailman, mysqld, named, nscd, ntpd, portmap, postgresql, squid, syslogd, winbind, snmpd

type. It also some preliminary support for models that have information at different levels of security. The bulk of the rules in most policies in SELinux is for type enforcement.

- Role Based Access Control (RBAC): Roles for processes. Specifies domains that can be entered by each role and specifies roles that are authorized for each user with an initial domain associated with each user role. It has the ease of management of RBAC with fine granularity of type enforcement.

The security policy is specified through a set of configuration files.

Overt flows transfer data directly; these are often high-bandwidth and easily controllable by a policy. Covert flows are indirect; e.g., file existence or CPU usage; these are often low-bandwidth and difficult to control in SELinux. Examples of overt flows:

- Direct Information Flow: eg: `allow subject_t object_t:file write`. Here, the domain `subject_t` is being given the permission to write to a file of type `object_t`.

- Transitive Information Flow: eg: `allow subject_a_t object_t : file write; allow subject_b_t object_t : file read`. Here, one domain is writing to an object of type file that is being read by another domain.

However, one downside is that very fine level control is needed. Every major component such as NFS or X needs extensive work on what permissions needs to be given before it can do its job. As the default assumption is “deny”, there could be as many as 30,000 allow rules! There is a critical need for automated analysis. If the rules are too lax, we can potentially have a security problem. If we have too few (too strict), there can be a failure of program at runtime as the program does not have enough permissions to carry out its job. Just as in software testing, we need to do code coverage analysis. In the simplest case, without alias analysis or interprocedural analysis, it is possible to look at the static code and decide what objects are needed to be accessed. Assuming that all the paths are possible, one can use “abstract interpretation” or “program slicing” to determine the needed rules. However, these rules will necessarily be conservative. Without proper alias analysis in the presence of aliasing, we will have to be even more imprecise; similar is the case in the context of interprocedural analysis.

Ganapathy[8] discuss automated authorization policy enforcement for user-space servers and the Linux kernel. Here, legacy code is retrofitted with calls to a reference monitor that checks permissions before granting access (MAC). For example, information "cut" from a sensitive window in an X server should not be allowed to be "pasted" into an ordinary one. Since manual placing of these calls is error-prone, an automated analysis based on program analysis is useful. First security sensitive operations to be checked ("MAC'ed") are identified. Next, for each such operation, the code-level constructs that must be executed are identified by a static analysis as a conjunction of several code-level patterns in terms of their ASTs. Next, locations where these constructs are potentially performed have to be located and, where possible, the "subject" and "object". Next the server or kernel is instrumented with calls to a reference monitor with subject, object and op triple as the argument, with a jump to the normal code on success or with call to a code that handles the failure case.

We now discuss the static analysis for automatic placement of authorization hooks, given, say, the kernel code and the reference monitor code[9]. Assuming no recursion, the call graph of the reference monitor code is constructed. For each node in the call graph, a summary is produced. A summary of a function is the set of $(pred, op)$ pairs that denotes the condition under which op can be authorized by the function. For computing the summary, a flow and context-sensitive analysis is used that propagates a predicate through the statements of the function. For example, at a conditional statement with q as the condition, the "if" part is analysed with $pred \wedge q$, and the "then" part by $pred \wedge \neg q$. At a call site, each pair in the summary of the function is substituted with the actuals of the call and the propagation of the predicate continues. When it terminates, we have a set of pairs as summary. Another static analysis on the kernel source recovers the set of conceptual operations that may be performed by each kernel function. This is done by searching for combinations of code patterns in each kernel function. For each kernel function, it then searches through a set of idioms for these code patterns to determine if the function performs a conceptual operation; an idiom is a rule that relates a combination of code patterns to conceptual operations.

Once the summary of each function h in the reference monitor code and the set of conceptual operations (S) for each kernel function k is available, finding the set of functions h_i in the monitor code that guards k reduces to finding a cover for the set S using the summary of functions h_i.

Another tractable approach is for a less granular model but finer than non-MAC systems. For example, it is typically the case in a large system that there are definitely forbidden accesses and allowable accesses but also many "gray" areas. "Conflicting access control subspaces" [10] result if assignments of permissions and constraints that prohibit access to a subject or a role conflict. Analysing these conflicts and resolving them, an iterative procedure, will result in a workable model.

4.4 Model Checking Access Control on Internet: Trust Management

Access control is based on identity. However, on the Internet, there is usually no relationship between requestor and provider prior to request (though cookies are one mechanism used). When users are unknown, we need 3rd party input so that trust, delegation and public keys can be negotiated. With public-key cryptography, it becomes possible to deal with anonymous users as long as they have a public key: authentication/ authorization is now possible with models such as SPKI/SDSI[41] (Simple public key infrastructure/Simple Distributed Security Infrastructure) or trust management. An issuer authorizes specific permissions to specific principals; these credentials can be signed by the issuer to avoid tampering. We can now have credentials (optionally with delegation) with the assumption that locally generated public keys do not collide with other locally generated public keys elsewhere(!). This allows us to exploit "local namespaces": any local resource controlled by a principal can be given access permissions to others by signing this grant of permission using the public key.

We can now combine access control and cryptography into a larger framework with logics for authentication/authorization and access control. For example, an authorization certificate (K, S, D, T, V) in SPKI/SDSI can be viewed as an ACL entry, where keys or principals represented by the subject S are given permission, by a principal with public key K, to access a "local" resource T in the domain of the principal with public key K. Here, T is the set of authorizations (operations permitted on T), D is the delegation control: whether S can in turn give permissions to others and V is the duration during which the certificate is valid.

Name certificates define the names available in an issuer's local name space whereas authorization certificates grant authorizations, or delegate the ability to grant authorizations. A certificate chain provides proof that a client's public key is one of the keys that has been authorized to access a given resource either directly or transitively, via one or more name-definition or authorization-delegation steps. A set of SPKI/SDSI name and authorization certificates defines a "pushdown system"[40] and one can "model check" many of the properties in polynomial time. Queries in SPKI/SDSI[41] can be as follows:

- Authorized access: Given resource R and principal K, is K authorized to access R? Given resource R and name N (not necessarily a principal), is N authorized to access R? Given resource R, what names (not necessarily principals) are authorized to access R?
- Shared access: For two given resources R1 and R2, what principals can access both R1 and R2? For two given principals K1 and K2, what resources can be accessed by both K1 and K2?
- Compromisation assessment: Due (solely) to the presence of maliciously or accidentally issued certificate set $C0 \subset C$, what resources could principal K have gained access to? What principals could have gained access to resource R?
- Expiration vulnerability: If certificate set $C0 \subset C$ expires, what resources will principal K be prevented from accessing? What principals will be excluded from accessing resource R?
- Universally guarded access: Is it the case that all authorizations that can be issued for a given resource R must involve a cert signed by principal K? Is it the case that all authorizations that grant a given principal K0 access to some resource must involve a cert signed by K?

Other models of trust management such as RT ("RBAC based Trust management") [43] are also possible. The following rules are available in the base model RT[]:

- Simple Member: $A.r \rightarrow D$. A asserts that D is a member of A's r role.
- Simple Inclusion: $A.r \rightarrow B.r1$. This is delegation from A to B.

The model RT[$\cap$] adds to RT[] the following "Intersection Inclusion" rule: $A.r \rightarrow B1.r1 \textit{ and } B2.r2$. This adds partial delegations from A to B1 and to B2. The model RT[$\Leftarrow$] adds to RT[] the following

"Linking Inclusion" rule: $A.r \rightarrow A.r1.r2$. This adds delegation from A to all the members of the role A.$r1$. RT[$\cap, \Leftarrow$] is all of the above 4 rules. The kinds of questions that we would like to ask are:

- Simple safety (Existential): does a principal have access to some resource in some reachable state?
- Simple availability: in every state, does some principal have access to some resource?
- Bounded safety: in every state, is the number of principals that have access to some resource bounded?
- Liveness (Existential): Does there exist a reachable state in which no principal has access to a given resource?
- Mutual Exclusion: In every reachable state, are 2 given properties (or two given resources) mutually exclusive, i.e., no principal has both properties (or access to both resources) at the same time?
- Containment: In every reachable state, does every principal that has one property (e.g., has access to a resource) also have another property (e.g., is an employee)? Containment can express safety or availability (e.g., by interchanging the two example properties in the previous sentence).

The complexity of queries such as simple safety, simple availability, bounded safety, liveness, and mutual exclusion analysis for RT[$\cap, \Leftarrow$] is decidable in poly time in size of state. For containment analysis[43], it is P for RT[], coNP-complete for RT[[$\cap$], PSPACE-complete for RT[$\Leftarrow$], and decidable in coNEXP for RT[$\cap, \Leftarrow$].

However, permission-based trust management cannot authorize principals with a certain property easily. For eg.[45], to give a 20% discount to students of a particular institute, the book store can delegate discount permission to the institute key. The institute has to delegate its key to each student with respect to "bookstore" context; this can be too much burden on the institute. Or, the institute creates a new group key for students and delegates it to each student key but this requires that the institute create a key for each meaningful group; this is also too much burden again! One answer to this problem is attribute-based approach: it combines RBAC and trust management.

The requirements in an attribute-based system[45] are decentralization, provision of delegation of attribute authority, inference, attribute-based delegation of attribute authority, conjunction of attributes, attributes with fields (expiry, age, ...) with the desirable features of expressive power, declarative semantics, and tractable compliance checking. Logic programming languages such as Prolog or, better, Datalog can be used for a delegation logic for ABAC: this combines logic programming with delegation and possibly with monotonic or non-monotonic reasoning. With delegation depth and complex principals such as k out of n (static/dynamic) thresholds, many more realistic situations can be addressed.

Related to the idea of attribute based access control and to allow for better interoperability across administrative boundaries systems, an interesting approach is the use of proof carrying authentication[11]. An access is allowed if a proof can be constructed for an arbitrary access predicate by locating and using pieces of the security policy that have been distributed across arbitrary hosts. It has been implemented as modules that extend a standard web server and web browser to use proof-carrying authorization to control access to web pages. The web browser generates proofs mechanically by iteratively fetching proof components until a proof can be constructed. They provide for iterative authorization, by which a server can require a browser to prove a series of challenges.

5 Language Based Security

As discussed earlier, current operating systems are much bigger than current compilers and therefore it is worthwhile to make the compiler part of the TCB than an OS. If it is possible to express security policies using a programming language that can be statically analysed, a compiler as part of a TCB makes eminent sense.

The main goal of language based security is to check the ***non-interference*** property, i.e., to detect all possible leakages of some sensitive information through computation, timing channels, termination channels, I/O channels, etc. But non-interference property is too restrictive to express security policies, since many programs do *leak* some information. For example, sensitive data after encryption can be "leaked" to outside world, which is agreeable with respect to security as long as the encryption is effective. Hence, non-interference property has to be *relaxed* by some mechanisms like *declassification*.

Note that static approaches cannot quantify the leakage of information as the focus is whether a program violated some desired property with respect to information flow. It is possible to use a dynamic approach that quantifies the amount of information leaked by a program as the entropy of the program's outputs as a distribution over the possible values of the secret inputs, with the public inputs held constant[19]. Noninterference has a entropy of 0. Such a quantitative approach will often be more useful and flexible than a strict static analysis approach, except that analysis has to be repeated multiple times for coverage.

One approach to static analysis for language based security has been to use type inference techniques, which we discuss next.

5.1 The type-based approach

Type systems establish safety properties (invariants) that hold throughout the program whereas noninterference requires that two programs give the same output inspite of different input values for its "low" values. Hence, a noninterference proof can be viewed as a bisimulation. For simpler languages (discussed below), a direct proof is possible but for languages with advanced features such as concurrency and dynamic memory allocation, noninterference proofs are more complex. Before we proceed to discuss the type-based approach, we will briefly describe the lattice model of information flow.

The lattice model of information flow started with the work of Bell and LaPadula[4] and Denning[52]. Every program variable has a static security class (or label); the security label of each variable can be global (as in early work) or local for each owner as in the decentralized label model (DLM) developed for Java in Jif[59].

If x and y are variables, and there is (direct) information flow from x to y, it is permissible iff the label of x is less than that of y. Indirect flows arise from control flow such as `if (y=1) then x=1 else x=2`. If label of $x \leq$ label of y, some information of y flows into x (based on whether x

is 1 or 2) and should be disallowed. Similarly, `if (y=z) then x=1 else w=2`, the *lub* of the levels of y and z should be $\leq$ *glb* of the levels of x and w. To handle this situation, we can assign a label to the program counter (*pc*). In the above example, we can assign the label of the *lub* to *pc* just after evaluating the condition; the condition now that needs to be satisfied is that both the arms of the `if` should have atleast the same level as the *pc*.

Dynamic labels are also possible. A method may take parameters and the label of the parameter itself could be an another formal. In addition, array elements could have different labels based on index and hence an expression could have a dynamic label based on the runtime value of its index.

Checking that the static label of an expression is atleast as restrictive as the dynamic label of any value it might produce is now one goal of analysis (preferably static). Similarly, in the absence of declassification, we need to check that the static label of a value is atleast as restrictive as the dynamic label of any value that might affect it. Due to the limitations of analysis, static checking may need to use conservative approximations for tractability.

Denning proposed program certification as a lattice-based static analysis method[52] to verify secure information flow. However, soundness of the analysis was not addressed. Later work such as Volpano and his colleagues[55] showed that a program is secure if it is "typable", with the "types" being labels from a security lattice. Upward flows are handled through subtyping. In addition to checking correctness of flows, it is possible to use "type inference" to reduce the need to annotate the security levels by the programmer. Type inference computes the type of any expression or program. By introducing type variables, a program can be checked if it can be typed by solving the constraint equations (inequalities) induced by the program. In general, simple type inference is equivalent to first-order unification whereas in the context of dependent types it is equivalent to higher-order unification.

For example, consider a simple imperative language with the following syntax[55]:

```
(phrases) p::= e | c

(expr)    e::= x | l | n | e arith e' | e relop e'

(cmds)    c::= e:=e' | c; c' | if e then c else c' |

                while e do c | let var x:=e in c
```

Here, `l` denotes locations (i.e., program counter values), `n` integers, `x` variables and `c` constants. The types in this system are types of variables, locations, expressions and commands; these are given by one of the partially ordered security labels of the security system. A `cmd` has a type t_{cmd} only if it is guaranteed that every assignment in `cmd` is made to a variable whose security class is `t` or higher. The type system for security analysis is as follows (L and T are location and type environments respectively):

$L; T \vdash n : t$ An integer constant can be typed

$L; T \vdash x : t$ if $T(x) = t$

$L; T \vdash l : t$ if $L(l) = t$

$$\frac{\begin{array}{l} L; T \vdash e : t \\ L; T \vdash e' : t \end{array}}{L; T \vdash e \ relop \ e' : t}$$

$$\frac{\begin{array}{l} L; T \vdash e : t \\ L; T \vdash e' : t \end{array}}{L; T \vdash e \ arithop \ e' : t}$$

$$\frac{\begin{array}{l} L; T \vdash x : t \\ L; T \vdash e' : t \end{array}}{L; T \vdash x := e : t_{cmd}}$$

$$\frac{\begin{array}{l} L; T \vdash c : t_{cmd} \\ L; T \vdash c' : t_{cmd} \end{array}}{L; T \vdash c; c' : t_{cmd}}$$

$$\frac{\begin{array}{l} L; T \vdash e : t \\ L; T \vdash c : t \\ L; T \vdash c' : t \end{array}}{L; T \vdash if \ e \ then \ c \ else \ c' : t}$$

$L; T \vdash e : t$

$L; T \vdash c : t$

$L; T \vdash \mathit{while}\ e\ \mathit{do}\ c : t$

Consider the rules for assignment above. In order for information to flow from e' to e, both have to be at the same security level. However, upward flow is allowed: for secrecy, for example, if e is at a higher level and e' is at a lower level. This is handled by extending the partial order by subtyping and coercion: the lowlevel ("derived type") is smaller (for secrecy) in this extended order than the highlevel ("base type"). Note that the extended relation has to be contravariant in the types of commands t_{cmd}.

It can be proved[55] that if an expression e can be given a type t in the above type system, then, for secrecy, only variables at level t or lower in e will have their contents read when e is evaluated ("no read up"). For integrity, every variable in e stores information at integrity level t. If a command has the property that every assignment within c is made to a variable whose security class is atleast t, then the confinement property for secrecy says that no variable below level t is updated in c ("no write down"). For integrity, every variable assigned in c can be updated by a type t variable.

Soundness of the typesystem induces the non-interference property, i.e. a high value cannot influence any lower value (or, information does not leak from high values to low values).

Information flow in multithreaded programs has been studied by Volpano[56]. The above type system does not guarantee noninterference; however, by restricting the label of all the while loops and its guards to low, the property is restored. Abadi has modelled encryption as declassification[57] and presented the resulting type system.

Myers and his colleagues have developed static checking for DLM-based Jif language[58, 59] while Pottier and his colleagues[54] have developed OCaml-based FlowCAML. We discuss the Jif approach in some detail.

5.2 Java Information Flow (Jif) Language

Jif is Java based information flow programming language that adds *static analysis* of information flow for improved security assurance. Jif is mainly based on *static type checking.* Jif also performs some *run-time* information-flow checks.

Jif is based on decentralized labels. A "*Label*" in Jif defines the security level, represented by a set of *policy expressions* separated by semicolon. A policy expression $\{owner : reader_1, reader_2, ...\}$ means that principal *owner* wants to allow labeled information to flow to at most the principals $reader_i$. Unlike MAC model, these labels contains fine-grained policies, which have an advantage of being able to represent *decentralized access control.* These labels are called decentralized labels because they enforce security on behalf of the owning principals, not on behalf of an implicitly centralized policy specifier. The policy $\{o_1 : r_1, r_2; o_2 : r_2, r_3\}$ specifies that both o_1, o_2 own the information with each allowing either r_1, r_2 or r_2, r_3 respectively. For integrity, another notation is adopted.

Information can flow from label L_1 to label L_2 only if $L_1 \sqsubseteq L_2$ (i.e. L_1 is less restrictive than L_2), where $\sqsubseteq$ defines a preorder on labels in which the equivalence classes form a join semilattice. To label an expression (such as $w+x$), a join operator is defined as the *lub* of the labels of the operands, as it has to have a secrecy as strong as any of them. In the context of control flow, such as `if cond then x=... else x=...`, we also need the *join* operator. To handle implicit flows through control flow, each program visible location is given an implicit label.

A *principal hierarchy* allows one principal to ***actfor*** another. This helps in simplifying the policy statements in terms of representation of groups or roles. For example, suppose principal *Alice* ***actsfor*** *Adm* and principal *Bob* ***actsfor*** *Adm*, then in the following code whatever value that *Adm* has is also readable by *Alice* and *Bob*.

```
void examplePrincipalHierarchy(){
    int{Alice:} a;
    int{Bob:} b;
    int{Adm:} c = 10;
    a = c; /* is valid stmt */
    b = c; /* is valid stmt */
}
```

The *declassification* mechanism gives the programmer an explicit escape hatch for releasing information whenever necessary. The *declassification* is basically carried out by relaxing the policies of some labels by principals having sufficient *authority.* For example:

```
void exampleDeclassification() where authority (Alice) {
    int{Alice:} x;
    int{Alice:Bob} y;
    int{Bob:} a;
    int{Bob:Alice} b;
      /*Here PC has label {}*/
    if (x > 10) {
       /*Here PC has label {Alice:}*/
      declassify(y = 25, {Alice:Bob});
       /*stmt ''y = 25" is declassified from {Alice:} to {Alice:Bob}*/
       /*valid because has Alice's authority*/
    }
       /*Here PC has again the label {}*/
      b = declassify(a, {Bob: Alice});
      /*invalid because doesn't have Bob's authority*/
}
```

JiF has label polymorphism. This allows the expression of code that is generic with respect to the security class of the data it manipulates. For example,

```
void exampleLP(int{Alice:;Bob:}i){
    ...
    /*Assures a security level upto {Alice:;Bob:}*/
  }
```

To the above function (which assures a security level upto {*Alice:;Bob:*}) one can pass any integer variable having one of the following labels:

- {}
- {*Alice:Bob*}
- {*Alice:*}
- {*Bob:*}
- {*Alice:;Bob:*}

Jif has *automatic label inference*: this makes it unnecessary to write many *type-annotations*. For example, suppose the following function is called (which can only be called from a program point with label atmost {*Alice:;Bob:*}) from a valid program point, "*a*" will get a default label of {*Alice:;Bob:*}.

```
void exampleALI{Alice:;Bob:}(int i){
    int a;
    /* Default label of ‘‘a"’ is {Alice:;Bob:}*/
   }
```

Run-time label checking and first-class label values in Jif make it possible to discover and to define new policies at run time. Run-time checks are statically checked to ensure that information is not leaked by the success or failure of the run-time check itself. Jif provides a mechanism for comparing runtime labels, and also a mechanism for comparing runtime principals. For example,

```
void m(int{Alice:pr} i, principal{} pr){
 int{Alice:Bob} x;
 if (Bob actsfor pr) {
   x = i;
    /* OK, since {Alice:pr} <= {Alice:Bob}*/
 }
 else {
   x = 0;
 }
```

```
  }
 void n(int{*lbl} i, label{} lbl){
   int{Alice:} x;
   if (lbl <= new label {Alice:}) {
     x = i;
      /* OK, since {*lbl} <= {Alice:Bob}*/
   }
   else {
     x = 0;
   }
}
```

Note that in the above function ***n(...)***, **lbl* represents an actual label held by the variable *lbl*. Whereas just *lbl* inside a label represents label of the variable *lbl* (i.e. {} here).

Labels and principals can be used as first-class values represented at runtime. These **dynamic** labels and principals can be used in the specification of other labels, and used as the parameters of parameterized classes. Thus, Jif's type system has **dependent types**. For example,

```
   class C[label L] { ... }
  ...
  void m() {
     final label lb = new label {Alice:  Bob};
     int{*lb; Bob:} x = 4;
     C[lb] foo = null;
       /*Here ''lb" acts as first-class label*/
     C[{*lb}] bar = foo;
  }
```

Note that, unlike in Java, method arguments in Jif are always implicitly **final**. Some of the limitations of Jif are that there is no support for **Java Threads**, *nested classes*, *initializer blocks* or '**native**' methods.

Interaction of Jif with existing Java classes is possible by generating *Jif **signatures*** for the interface corresponding to these java classes.

5.3 A case study: VSR

The context of this work[8] is the security of archival storage. The central objective usually is to guarantee the availability, integrity, and secrecy of a piece of data *at the same time.* Availability is usually achieved through redundancy, which reduces secrecy (it is sufficient for the adversary to get one copy of the secret). Although the requirements of availability and secrecy seem to be in conflict, an information-theoretic secret sharing protocol has been proposed by A. Shamir in 1979 [12]. But this algorithm does not provide data integrity. Loss of shares can be tolerated upto a threshold but not to arbitrary modifications of shares.

A series of improvements have therefore been proposed over time to build secret sharing protocols resistant to many kinds of attacks. The first one takes into account the data-integrity requirement, and leads to *Verifiable* Secret Sharing (VSS) algorithms [13, 14]. The next step is to take into account mobile adversaries that can corrupt any number of parties given sufficiently ample time. It is difficult to limit the number of corrupted parties on the large time scales over which archival systems are expected to operate. An adversary can corrupt a party but redistribution can make that party whole again (in practice, this happens, for example, following a system reinstallation). Mobile adversaries can be tackled by means of proactive secret sharing (PSS) wherein redistributions are performed periodically. In one approach, the secret is reconstructed and then redistributed. However, this causes extra vulnerability at the node of reconstruction. Therefore, another approach, viz., redistribution without reconstruction is used [15]. A combination of VSS and PSS is Verifiable Secret Redistribution (VSR); one such protocol is proposed in [16]. In [17], we proposed an improvement of this protocol relaxing some of the requirements.

Modelling the above protocol using Jif can help us to understand the potential and the diffi-

[8]This is joint work with a former student S. Roopesh.

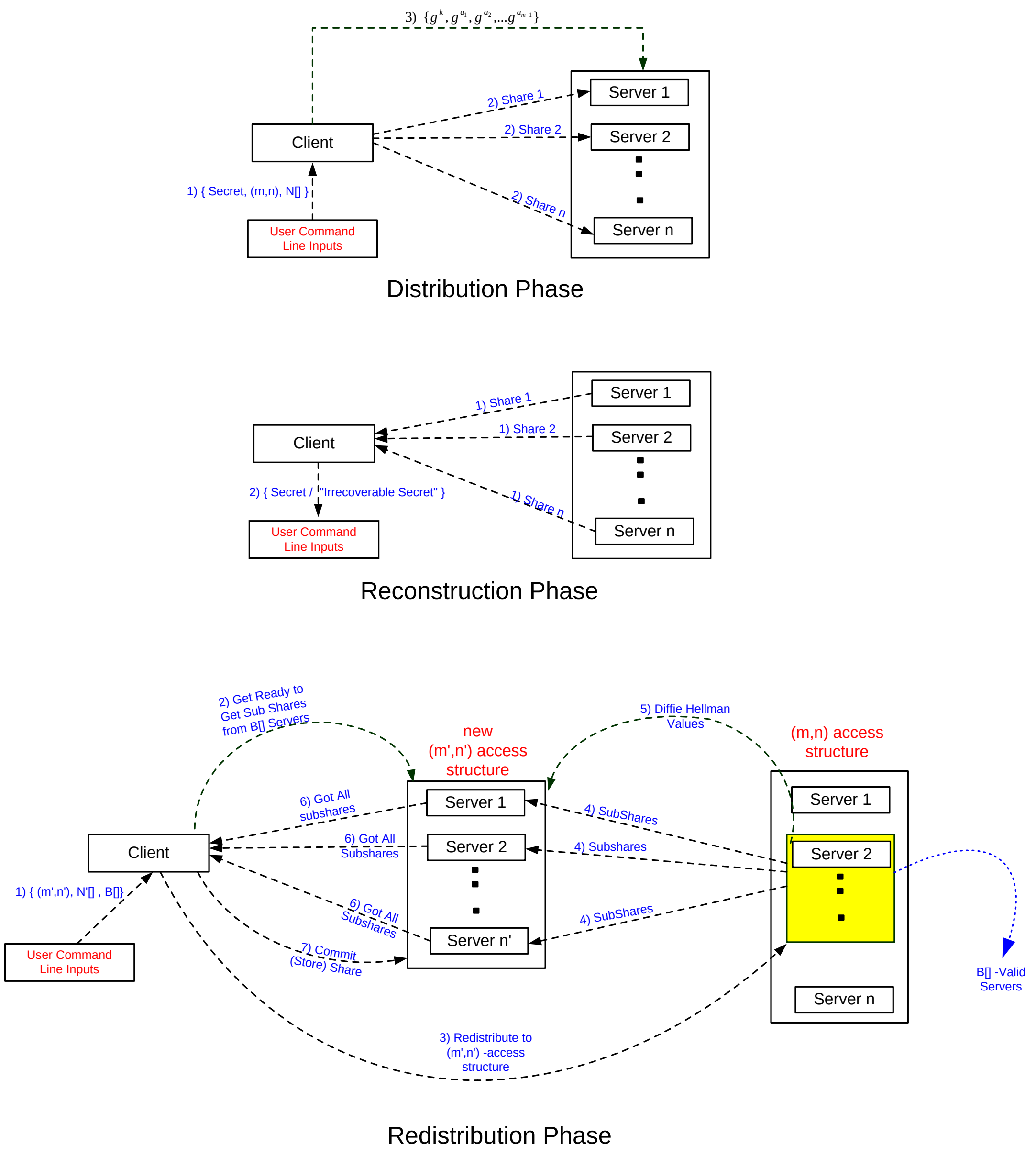

Figure 1: Simplified VSR Protocol

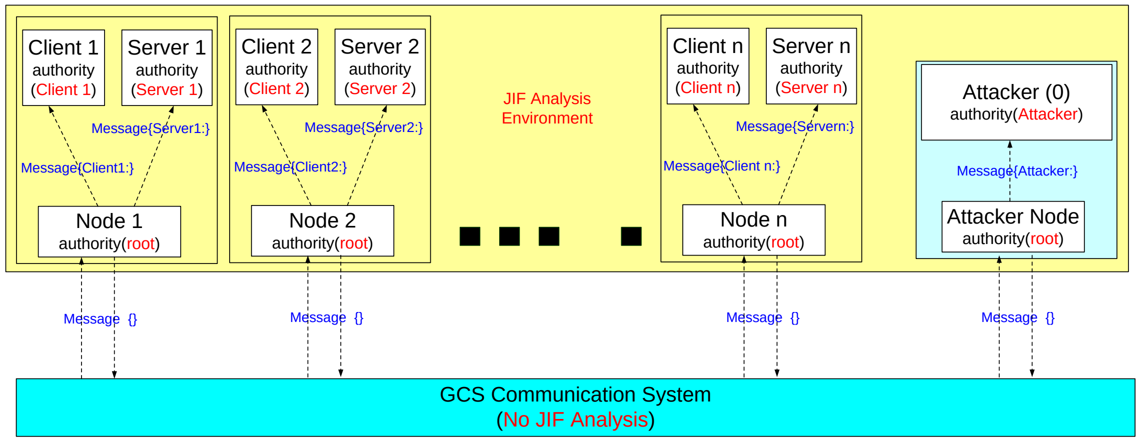

Figure 2: Jif Analysis on VSR

culties of Jif static analysis. We now discuss the design of simplified VSR[16] protocol:

- **Global Values**: This class contain the following variables that are used for generation and verification of shares and subshares during reconstruction and redistribution phases.
  - ***(m,n)***: m - threshold number of servers required for reconstruction of secret, and n - total number of servers to which shares are distributed.
  - ***p***: prime used for Z_p, ***r***: prime used for Z_r
  - ***g***: Diffie-Hellman exponentiator
  - ***KeyID***: Unique for each secret across all clients.
  - ***ClientID***: ID of the owner of the secret.
- **Secret**: This class contain ***secret***'s value (i.e. secret itself), ***polynomial*** used for distribution of shares, **N**[] - array of server IDs to which the shares are distributed.
- **Points2D**: This class contain two values **x** and **f(x)**, where f is the polynomial used for generating the shares (or sub-shares). Used by "*Lagrangian Interpolator*" to reconstruct secret (or shares from sub-shares).
- **Share**: This class contains an *immutable* ***original share*** value (used to check whether shares are uncorrupted or not), and ***redistributing polynomial*** (used for redistribution of this share to new access structure).

- **SubShare**: Same as "Share" class, except the share value actually contain ***sub-share*** value and no redistributing polynomial.
- **SubShareBox**: This class keeps track of the sub-shares from a set of valid servers (i.e. "**B**[]" servers) in redistribution phase. This is maintained by all servers belonging to the new *(m',n')-access structure* to which new shares are redistributed.
- **Client**: This class contains zero or more secrets and is responsible for initial distribution of shares and reconstruction of secret from valid set of servers.
- **Server**: This class maintains zero or more shares from different clients and is responsible for redistribution of shares, after client (who is the owner of the secret corresponding to this share) has approved for the redistribution.
- **Node**: Each node contains two units, one ***Client*** and the other ***Server*** (having the same ids as this Node). On message reception from the reliable communication (GCS: Group Communication System) interface *GCSInterface*, this node extracts the information inside the message and gives it to either the client or the server, based on the message type.
- **GCSInterface**: This communication interface class is responsible for acting as an interface between underlying reliable messaging system (say, the Ensemble GCS[60] system), user-requests, and *Client* and *Server*.
- **UserCommand** (Thread): Handles three types of commands from the user:
  - Distribution of user's secret *S* to *(m,n) - access structure*,
  - Redistribution from *(m,n) - access structure* to *(m',n') - access structure*, and
  - Reconstruction of secret *S*.
- **SendMessage** (Thread): This class packetizes the *messages* (from "Node") (depends on the communication interface), then either sends or multicast these packets to the destination node(s).
- **Attacker**: This class is responsible for attacking the servers, getting their shares and corrupting all their valid shares (i.e. changing the "***y***" values in *Points2D* class to arbitrary values). This class also keeps all valid shares it got by attacking the servers. It also reconstructs all

possible secrets from the shares collected. Without loss of generality, we assume that atmost one attacker can be running in the whole system.

- **AttackerNode**: Similar to *Node*, but instead of *Client* and *Server* instances, it contains only one instance of "***Attacker***" class.
- **AttackerGCSInterface**: Similar to *GCSInterface* class.
- **AttackerUserCommand**: Handles two types of commands from the attacker:
  - Attack server S_i, and
  - Construct all possible secrets from collected valid shares.

We do not dwell on some internal bookkeeping details during redistribution and reconstruction phases. Figure 1 gives the three phases of simplified VSR protocol. The "distribution" and "reconstruction" phases almost remain the same. Only "redistribution phase" is slightly modified, where client acts as manager of the redistribution process. The following additional assumptions are also being made:

- No "**Abort**" or "**Commit**" messages from servers
- In "*redistribution phase*", *client*, who is the owner of the *secret* corresponding to this *redistribution process*, will send the "**commit**" messages, instead from the redistributing servers.
- Attacker is restricted to only attacking the servers and thereby getting all the original shares and corrupting them which are held by the servers.
- No "**reply** and **DoS** attacks'

5.3.1 Jif analysis on simplified VSR

In this section, we discuss an attempt to do Jif analysis on simplified VSR implementation and its difficulties. As shown in the figure 2, every "*Node*" (including the "*AttackerNode*") runs with "**root**" *authority* (who is above all and can "***actfor***" all principals). Every message from and out of the network will have an ***empty label*** (as we rely on underlying Java Ensemble for *cryptographically* perfect end-to-end and multicast communication). The "**root**" (i.e. *Node*) receives the message from the network. Based on the message type, "**root**" will appropriately (*classify* or) *declassify* the contents of the message, and handles them either by giving to *client* or *server*. Similarly for all outgoing messages from *Client/Server* to communication network would be properly *declassified*.

First, the communication interface used (Java Ensemble) uses some "*native*" methods to contact ***ensemble-server***[9]. And in order to do *asynchronous* communication we use *threads* in our VSR implementation. Since Jif does not support **Java Threads** and **native** methods, Jif analysis cannot be done at this level. Hence we have to restrict the Jif analysis above the communication layer[10].

Next, let us proceed to do static analysis on the remaining part using the Jif compiler. Consider the *Attacker Node* (see figure 2). The attacker node (running with **root** authority) classifies all content going to the principal "Attacker" with the label - $\{Attacker\ :\}$. If the attacker has compromised some server, he would get all the valid shares belonging to that server. Hence, all shares of a compromised "Server" output from the communication interface (GCS) goes to the attacker node first. Since it is running with "**root**" authority, and sees that shares are semi-sensitive, hence, (de)classifies these shares to {Server:}. In order that these shares by read by the attacker, the following property should hold: {**Server:**} $\sqsubseteq$ {**Attacker:**}

But since the "Attacker" principal does not *actsfor* this "Server" (principal), the above relation does not hold, hence the attacker cannot read the shares. The Jif compiler detected this as an error. However, if we add *actsfor* relation from "Attacker"(principal) to this "Server" (principal) (i.e. *Attacker* ***actsfor*** *Server*), he could read the *shares*. In this case, the Jif compiler does not report any error, implying that the information flow from {Server:} to {Attacker:} is valid (since {Server:} $\sqsubseteq$ {Attacker:} condition holds).

VSR has a threshold property: if the number of shares (or subshares) is more than the threshold, the secret can be reconstituted. What this implies is that any computation knowing less than the threshold number of shares cannot interfere with any computation that has access to more than this

[9] *ensemble-server* is a daemon serving group communication.

[10] Due to this separation, we encountered a problem with the Jif analysis even if we want to abstract out the lower GCS layer. Some of the classes (like "*Message*" class) are common to both upper and lower layers but they are compiled by two different compilers (Jif and Java) that turn out to have incompatible *.class* files for the common classes.

threshold. We need a **threshold set non-interference** property. This implies that we can atmost declassify less than any set of threshold number of shares. Since such notions are not expressible in Jif, even if mathematically proved as in [16], more powerful dependent type systems have to be used in the analysis. Since modern type inference is based on explicit constraint generation and solving, it is possible to model the threshold property through constraints but solving such constraints will have a high complexity. Fundamentally, we have to express the property that any combination of subshares less than the threshold number cannot interfere (given that the number of subshares available is more than this number); either, every case has to be discharged separately or symmetry arguments have to be used. Approaches such as model checking (possibly augmented with symmetry-based techniques) are indicated.

There is a also a difficulty in Jif to label each array element with different values. In “Client” class, there is a function that calculates and returns the shares of a secret. Its skeleton code is as follows:

```
1 Share[] getShares(int[] ServerID){
2       Share{Client:}[]{Client:} shares = new Share{Client:}[]{Client:};
3       int n = ServerIDs.length;
4       for (int i = 0 ; i < n ; i++){
5         shares[i] = {Client:Server_ServerID[i]} polynomial(ServerID[i]);
6       }
7       return shares;
8 }
```

In line 2, “*shares*” is an array of type “*Share*” having a label of {*Client:*} for both array variable and individual elements of this array. Line 5 calculates different shares for different servers based on ServerIDs, and assigning a different label of {*Client:* $Server_{ServerID[i]}$ } for each of these shares. But, Jif does not allow for different labels for different elements of an array; it only allows a common label for all elements of an array. If this is possible, we should ensure that accessing some array element does not itself leak some information about the index value.

There are many other difficulties for using Jif to do analysis currently such as the need to recode a program in Jif. The basic problem is the (post-hoc) analysis after an algorithm has already been designed. What is likely to be more useful[11] is an explicit security policy before the code is developed[28] and then use Jif or similar language to express and check these properties where possible. McCamant and Ernst[19] argue that Jif-type static analysis is still relatively rare and no "large" programs have yet been ported to Jif or FlowCaml. They instead use a fine-grained dynamic bit-tracking analysis to measure the information revealed during a particular execution.

6 Future Work

Current frameworks such as Jif have not been sufficiently developed. We foresee the following evolution:

- The analysis in Section 5.1 so far has assumed that typing can reveal interesting properties. This needs to be extended to a more general static analysis that deals with values in addition to types as well as when the language is extended to include arrays, etc. Essentially, the analysis should be able to handle array types with affine index functions. This goes beyond the analysis possible with type analysis.
- Incorporate pointers and heaps in the analysis. Use dataflow analysis on lattices but with complex lattice values (such as regular expressions, etc.). Or, use shape analysis techniques.
- Integrate static analysis (such as abstract interpretation, compiler dataflow analysis) with model checking to answer questions such as "what is the least/most privilege a variable should have to satisfy some constraint". This may be coupled with techniques such as CEGAR ("counterexample guided abstraction refinement")[2]. This however requires considerable machinery. Consider for example, a fully developed system such as the SLAM software model checking[3]. It uses counterexample-driven abstraction of software through "boolean" programs (or abstracted programs) using model creation (c2bp), model checking (bebop) and

[11]In the verification area, it is increasingly becoming clear that checking the correctness of finished code is an order of magnitude more difficult than intervening in the early phases of the design.

model refinement (newton). SLAM builds on the OCaml programming language, and uses dataflow and pointer analysis, predicate abstraction and symbolic model checking, and tools such as CUDD, a SAT solver and a SMT theorem prover[3]. Many such analyses have to be developed for static analysis of security properties.

References

[1] Matt Bishop, Computer Security: Art and Science, Addison-Wesley, 2002.

[2] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith: Counterexample-Guided Abstraction Refinement. CAV 2000: 154-169.

[3] Thomas Ball, Sriram K. Rajamani: The SLAM project: debugging system software via static analysis. POPL 2002: 1-3

[4] Bell, D. Elliott and LaPadula, Leonard J. (1973). "Secure Computer Systems: Mathematical Foundations". MITRE Corporation.

[5] Jrg Bauer, Ina Schaefer, Tobe Toben, Bernd Westphal: Specification and Verification of Dynamic Communication Systems. ACSD 2006: 189-200.

[6] Butler W. Lampson: A Note on the Confinement Problem. Commun. ACM, 16(10): 613-615 (1973).

[7] Edmund M. Clarke, Orna Grumberg, Doron A. Peled, "Model Checking," The MIT Press, 2000.

[8] Vinod Ganapathy, Trent Jaeger, Somesh Jha: Retrofitting Legacy Code for Authorization Policy Enforcement. S&P 2006: 214-229

[9] Vinod Ganapathy, Trent Jaeger, Somesh Jha: Automatic placement of authorization hooks in the linux security modules framework. ACM Conference on Computer and Communications Security 2005: 330-339

[10] Trent Jaeger, Xiaolan Zhang, Fidel Cacheda: Policy management using access control spaces. ACM Trans. Inf. Syst. Secur. 6(3): 327-364 (2003)

[11] Andrew W. Appel and Edward W. Felten. Proof-Carrying Authentication. 6th ACM Conference on Computer and Communications Security, November 1999

[12] A. Shamir. How to share a secret. *Comm. ACM*, 22(11):612–613, November 1979.

[13] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. *In Proc. of the 28th IEEE Ann. Symp. on Foundations of Computer Science*, pages 427–437, October 1987.

[14] T. P. Pedersen. Non-interactive and information theoretic secure verifiable secret sharing. *In Proc. of CRYPTO 1991, the 11th Ann. Intl. Cryptology Conf.*, pages 129–140, August 1991.

[15] Sushil Jajodia and Yvo Desmedt. Redistributing secret shares to new access structures and its applications. Technical Report ISSE TR-97-01, George Mason University, July 1997.

[16] Theodore M. Wong, Chenxi Wang, and Jeannette M. Wing. Verifiable secret redistribution for archival systems. *Proceedings of The First IEEE Security in Storage Workshop*, December 2002.

[17] V. H. Gupta and K. Gopinath. An extended verifiable secret redistribution protocol for archival systems. In *International Conference on Availability, Reliability and Security*, pages 100–107, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[18] R.B.Keskar, R.Venugopal, "Compiling Safe Mobile Code," in The Compiler Design Handbook, CRC Press 2002.

[19] Stephen McCamant and Michael D. Ernst Quantitative Information-Flow Tracking for C and Related Languages, MIT-CSAIL-TR-2006-076, 2006.

[20] Hao Chen, David Wagner, Drew Dean, "Setuid Demystified," Proceedings of the 11th USENIX Security Symposium, 2002.

[21] Arul Ganesh, K. Gopinath, A model for Context-Dependent Access Control using Concept Lattice, *in preparation.*

[22] M. Y. Vardi. The complexity of relational query languages (extended abstract). In Symposium on the Theory of Computing, ACM, 1982.

[23] Protection in Operating Systems, Harrison, Ruzzo, Ullman, CACM Aug, 1976.

[24] Ken Thompson. Reflections on Trusting Trust, Communication of the ACM, Vol. 27, No. 8, August 1984

[25] Computability Classes for Enforcement Mechanisms, K. W. Hamlen, Greg Morrisett, F. B. Schneider, ACM TOPLAS 2005.

[26] Joseph A. Goguen, Jos Meseguer: Security Policies and Security Models. IEEE Symposium on Security and Privacy 1982

[27] Steve Zdancewic Andrew C. Myers, Robust Declassification, Proceedings of the 2001 IEEE Computer Security Foundations Workshop, 2001

[28] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, Andrew C. Myers Untrusted Hosts and Confidentiality: Secure Program Partitioning. 2001 Symposium on Operating Systems Principles.

[29] Felix Lazebnik, On systems of linear Diophantine equations, The Mathematics Magazine, vol. 69, no. 4, October 1996.

[30] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In Advances in Cryptology (CRYPTO'01), volume 2139 of Lecture Notes in Computer Science, pp 1-18, August 2001.

[31] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In Symposium on Network and Distributed Systems Security (NDSS00), February 2000. San Diego CA.

[32] Attacks on Local Searching Tools Seth Nielson Seth J. Fogarty Dan S. Wallach, Dec 2004

[33] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), October 2003.

[34] Fred Cohen, "Computer Viruses: Theory and Experiments", Computers and Security 6 (1987).

[35] David M. Chess and Steve R. White. An Undetectable Computer Virus. Virus Bulletin Conference, Sept. 2000.

[36] George C. Necula, Scott McPeak, Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code, POPL 2002.

[37] J. Rao, P. Rohatgi, H. Scherzer, and S. Tinguely. Partitioning attacks: Or how to rapidly clone some GSM cards. Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P02).

[38] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. 42nd FOCS, 2001. Revised version (2005).

[39] Joshua D. Guttman, Amy L. Herzog, and John D. Ramsdel. Information Flow in Operating Systems: Eager Formal Methods, WITS2003.

[40] Analysis of SPKI/SDSI Certificates using Model Checking, Jha and Reps, CFSW'02.

[41] Certificate chain discovery, Clarke et al, JCS 2001.

[42] Specifying and Reasoning about Dynamic Access-Control Policies D. Dougherty, K. Fisler and S. Krishnamurthi, 2006

[43] Beyond Proof-of-compliance: Security Analysis in Trust Management. Ninghui Li, John C. Mitchell, and William H. Winsborough. JACM, May 2005.

[44] Decidability of safety in graph-based models for access control, Koch, Mancini, Parisi-Presicce, ESORICS Oct 2002.

[45] Delegation Logic, A logic-based approach to distributed authorization, Li, Grosof and Feigenbaum, ACM Trans. on Info. and System Security, Feb 2003.

[46] Safety in Discretionary Access Control. Ninghui Li and Mahesh V. Tripunitara, IEEE Symposium on Security and Privacy, May 2005.

[47] Firewall Design: Consistency, Completeness, and Compactness, Mohamed G. Gouda and Alex X. Liu

[48] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions: Proceedings of the 4th Symposium on Operating System Design and Implementation.

[49] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In Proceedings of the 11th USENIX Security Symposium, USENIX, August 2003

[50] Steven McCanne, Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," Winter Usenix Conf., San Diego, 1993.

[51] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal (June 2004). "Dynamic Instrumentation of Production Systems". Proceedings of the 2004 USENIX Annual Technical Conference.

[52] Dorothy E. Denning , Peter J. Denning, Certification of programs for secure information flow, Communications of the ACM, v.20 n.7, p.504-513, July 1977.

[53] Jif: Java + information flow: www.cs.cornell.edu/jif/.

[54] Franois Pottier and Vincent Simonet. Information flow inference for ML. ACM Transactions on Programming Languages and Systems, 25(1):117–158, January 2003.

[55] Dennis M. Volpano, Cynthia E. Irvine, Geoffrey Smith: A Sound Type System for Secure Flow Analysis. Journal of Computer Security 4(2/3): 167-188 (1996)

[56] Geoffrey Smith, Dennis M. Volpano: Secure Information Flow in a Multi-Threaded Imperative Language. POPL 1998: 355-364

[57] Martn Abadi: Secrecy by Typing in Security Protocols. TACS 1997

[58] http://www.cs.cornell.edu/jif/doc/jif-3.0.0/manual.html

[59] Andrew C. Meyers. Mostly-static decentralized information flow control. January 1999.

[60] Mark Hayden and Ohad Rodeh. Ensemble reference manual. February 2004.

[61] Trusted Computer System Evaluation Criteria, http://en.wikipedia.org/wiki/Trusted_Computer_System_Evaluation_C