Unguessable Atoms: A Logical Foundation for Security

Mark Bickford *

June 13, 2006

Abstract

We propose a new foundation for security based on a logical concept of protected information that can be enforced in the mathematical structure of a computation system. We describe a computation system based on event structures and a logic of events, and we show how to model all limitations on the capabilities of adversaries using a theory of atoms.

1 Introduction

Anyone attempting formal specification and verification of security properties immediately confronts two perplexing problems: how are properties such as confidentiality and authentication to be defined in a logical system, and how can we represent the cryptographic assumptions on which implementations of these properties depend? We were faced with these questions when we began our on-going formal analysis of the APSS [**?**](asynchronous, proactive secret sharing) algorithm, which uses sophisticated cryptographic algorithms. To define security properties we need some way to represent information that one set of agents, the *secure agents*, keeps secret from the rest of the world, the *adversaries*. A proof of a security property is, in general, a proof that adversaries cannot learn some protected information, so it depends on the formal definition of what it means for an agent to *learn* protected information. We can only prove impossibility with respect to agents in some *computation system*. All the obstacles that prevent the adversaries from learning protected information, including time, space, or complexity

*This research sponsored by the Information Assurance Institute (IAI) under AFSOR contract F49620-02-1-0170

bounds are included in the computation model. Although the adversaries are understood to be human agents assisted by a computer with some resource bounds, the mathematical analysis usually leaves out adversarial behaviors, like breaching physical security, bribery, threats of violence, etc, that are not easily expressed as algorithms. So, in all of the formal models we consider, the agents are modeled as computational processes.

The foundations for formal analysis of security thus depend on three key ingredients: agents in a computation system, a representation of protected information, and a definition of information learning. In this paper we identify these ingredients in two existing foundational approaches to security that we call the *analytic model* and the *algebraic model*. A crucial difference between the two is how the learning of information is modeled. In the analytic model agents can acquire information by guessing, so there is a distinction between acquiring information and learning information. In the algebraic model, the computation system is restricted so that there is no distinction between learning information and merely acquiring it. We propose a new foundation, based on the use of *unguessable atoms* to represent protected information, which combines the generality of the analytic model with the simplicity of the algebraic model and which can be fully formalized in current formal proof systems.

1.1 The Analytic Model

We call a fully detailed model of all the foundations assumed in theoretical discussions of security, including probability and complexity theory, an *analytic model*. In an analytic model, the computation system is assumed to be "all computer programs". Because of the Church/Turing thesis and the insensitivity of complexity classes, like P, NP, P-SPACE, etc. to the choice of computing model, the details are often omitted, however, a fully formal analytic model must include a general theory of computation, and it should cover programs that are distributed, and can use randomization and real time. Formalization of such theories is non-trivial, but can be done.

With such a computation system, the agents are programs and their semantics includes a notion of program state. The protected information can be any specific part of the states of the secure agents, so it can be some collection of boolean functions of these states, which we may call *bits*. Security has been compromised if the adversaries come to *learn* the values of any of these bits.

Since the computation system is completely general and the concept of protected information is so general, the third ingredient, the definition

of information learning, is the most difficult to agree on, and there is a huge literature on this topic. Intuitively, an agent has learned protected information when it has deduced its value and can act on the result of this deduction in some practical way, but formal definitions of this intuition can be quite complex and valid inference rules about it hard to find. In any such definition, the fact that the agents are resource bounded has to play a part. If protected information is a *bit* then an adversary is obviously able to *guess* its value, so the definition of learning is about how the adversary can *know* that a value has some significance e.g., the adversary has a strategy that improves his chances, over repeated trials, of guessing the value of some particular boolean function of the secret state to a probability > 0.5. An analytic model define concepts of learning and knowledge from first principles and proofs of security reason directly about these concepts. The background theory in which such a definition can be made usually includes a theory of probability and the theory of polynomial-time complexity, all of which must be included in the formal model.

In an analytic model that includes probability and complexity theory, there is no problem stating cryptographic assumptions on which to base implementations. In such a theory, one can express the assumption of a *one-way function*, such as $E(x) = g^x$, that is easily computed, while its inverse, such as $D(g^x) = x$, is difficult to compute.

The merit of the analytic foundation is that it attempts to model the real science and mathematics of secure systems. The disadvantage is that, although current formal systems such as Nuprl, Coq, HOL, etc. have the expressive power to define suitable versions of the three key ingredients, creating a full formalization of the analytic model, including complexity theory, would be a big job, and the resulting theory might be difficult to use.

1.2 The Algebraic (Dolev-Yao) Model

A *security model* like the Dolev-Yao model [3] postulates a computation system and an abstract representation of protected information such that an adversary, working within the computation system, is deemed to know a piece of information merely from his ability to generate its representation. Thus, the computation system must make the guessing of these representations impossible and place constraints on the capability of adversaries to generate them. We will discuss only the most general structure of the Dolev-Yao model as an example of what we are calling an *algebraic model*.

The Dolev-Yao model is a model for public-key cryptography in which

the protected information is a finitely generated algebra of expressions over some possibly infinite set of atomic pieces. The atomic pieces are a set of symbols like $k_1, k_2, \ldots$, $N_1, N_2, \ldots$, and $m_1, m_2, \ldots$ that represent keys, nonces, and atomic messages. The generators of the algebra include operators like $KeyP(k, k')$, representing a public/private key-pair, $\{m\}_k$, representing message m encrypted with key k, and $m_1 m_2$, representing the concatenation of messages m_1 and m_2. The computation system allows an agent that holds a set of messages in this algebra to generate other messages by applying a finite set of *rewrite rules* and the computation system also provides a model of message transmission over the network and allows the adversary to eavesdrop, to acquire any message sent over the network. The rewrite rules allow an agent, for example, to decrypt an encrypted message if he also holds the decryption key, so $\{m\}_k, KeyP(k, k') \rightarrow m, KeyP(k, k')$ is one of the rewrite rules the adversary may use.

In this model, an agent has learned some information if he has generated it from his own initial state and messages acquired by eavesdropping by rewriting using the rewrite rules. There is no need to model when the adversary knows the significance of the acquired information, i.e. there is no distinction between *learning* information and *acquiring* information. Proofs of security protocols in this model can be reduced to case analysis and can sometimes be automated with model checking.

1.3 Unguessable Atoms

We would like a way to specify and prove security properties that has the generality of the analytic model and some of the simplicity of the Dolev-Yao, algebraic model. We would like, as in the analytic model, to model agents as "all distributed programs" and to carry out our proofs in a general purpose logic of distributed systems. By doing this, our security theorems have a greater significance since we will be proving impossibility results for adversaries in the general computation system rather than the limited algebraic computation system. We will also be using the same logical framework for all of our proofs about programs–security proofs will not be done in a special logical system. As we use our formal system and our tools to prove properties of programs we gain confidence in the soundness of the logic and the correctness of the tools, and this confidence includes our security proofs since they are part of the same formal system. If a security property depends on cryptography as well as a non-trivial distributed algorithm, then we can verify both parts of the protocol in one system.

But we also want the simplicity of the algebraic model, in that protected

information can be represented in a way that allows us to model *learning* the information simply as *acquiring* the information. Thus, in particular, the protected information must be *unguessable*, for if an adversary had a procedure that could generate guesses that eventually included all protected information, then we would have to resort to modeling the fact that the adversary does not *know* the significance of his guess.

How can this be possible? A theory of "all programs" must allow a program to apply any computable function and surely, any data-type T that the secure agents can use to store protected information in their state would have some finite representation and hence the set of values in T would be enumerable by a computable function.

Constraints on the Logical Model Our solution to this dilemma is to take T to be the type $Atom$. The members of type $Atom$, called atoms, are terms that evaluate to canonical forms represented by tokens $tok(a)$, $tok(b)$, ..., but the names a,b,... are have no significance in the logic. This is made explicit by the *permutation rule* that says

> From any judgement, $J(a, b, \ldots)$, in the logic, mentioning a finite set $a, b, \ldots$ of these names, we may infer the truth of $J(a', b', \ldots)$ whenever the mapping $a \mapsto a', b \mapsto b', \ldots$ is a permutation.

To make this rule valid, we must constrain the logic in several ways. The first constraint is that the names $a, b, \ldots$ are *unhideable*. This means that a definition like $f(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$ is not allowed because the names a and b occur on the righthand side of the definition but not on the lefthand side. If this were allowed, the permutation rule would be inconsistent: we could prove a judgement that $f(1) = tok(a)$, and use the permutation rule to conclude $f(1) = tok(b)$ and then conclude that $tok(a) = tok(b)$, which will compute to $False$. Every name mentioned in the righthand side of a definition must also be included among the parameters on the lefthand side, so $f\{a, b\}(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$ is an allowed definition.

The second constraint is that the evaluation rules which allow one term to be shown equal to another term by computation (for example $(\lambda x.\ x + x)(2)$ evaluates to 4), must obey the permutation semantics for atoms. This means that the only way that names $a, b, \ldots$ can be used in the computation rules is to branch, uniformly, on whether names are equal. So it is a violation of the constraints to introduce a primitive term operator $xxx()$ and add a rule that $xxx()$ evaluates to $tok(a)$.

The semantics of atoms is presented in [1]. That paper explains how the *unhideable tokens* are added to the terms of the system. In this paper we will call the unhideable tokens *names* and assume only that the set of names *mentioned by* a closed term t is unambiguous and that we can decide whether names are equal.

Thesis Our thesis is that by making the constraints on the logical system necessary to introduce the *Atom* type, we obtain, first, a theory of atoms that corresponds with the intuition that atoms have no structure and can be tested only for equality, and second, we will be able to have the best of both worlds of security models. We will be able to introduce a type of data values (atoms) that are "unguessable," relieving us from the need for a distinction between learning an atom and acquiring an atom. The theory of "all programs" is just as it was before the introduction of atoms, and programs can now compute with atoms as well, but cannot use any hidden structure of atoms, not because of a restriction on the concept of programs, but because of a global constraint on the mathematical framework. Thus, the simplicity of the concepts of protected information and learning of protected information in the algebraic model can be combined with the generality of the computation system in the analytic model.

Note that for any finite set of atoms A there is still a computable function f that enumerates A, but any such function f is essentially a table lookup like $f\{a,b\}(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$. Now, an adversary using the function f to "guess" atoms must have the function f available in its state or in its program, and since the names a and b are unhideable, they are "in" f already so the agent already "has" them. Thus, a table lookup function like f is of no use in guessing atoms since the only atoms it can guess are atoms that it already has. This is the sense in which we say that atoms are *unguessable*.

Dolev-Yao is a model of a whole crypto-system, but our model only provides us with a foundation from which to build such a model. In section 5 we discuss the general form that a model of a crypto-system takes in our computation system, but we will leave to a future report a full description of how a public-key crypto-system is modeled and how a proof of a protocol like Needham-Schoeder-Lowe can be accomplished. Our goal in this paper is to fully explain a foundation for formal security proofs whose representation of protected information is *atoms*, whose computation system is a general model of distributed programs called *message automata*, and defines learning (= acquiring) in terms of a simple proposition called *independence*, which is

the topic of the next section.

2 Independence

Intuitively, an agent will have learned an atom a at some point in its history if it has the atom a in its state at that point. The semantics of the computation system described in section 3 provides an expression (**state after** e) that denotes the state of an agent after an event e has occurred. Events e are essentially points in space/time, so if we can define what it means for an expression like (**state after** e) to contain an atom a, then we can use that to define when an agent has acquired atom a. We have discovered that in a logic based on *constructive type theory* the rules for the primitive proposition $(x : T\|a)$, saying that an expression x of type T *does not* contain atom a, are simpler than the rules for its negation (which says that x *does* contain a). We read $(x : T\|a)$ as "x of type T is *independent of* atom a". We sometimes also say "a is independent of x (in type T)" or that "x and a are independent".

The complete set of rules for independence as implemented in the current Nuprl system are listed in table 1.

Table 1: Rules for Independence

INDEPENDENTEQUALITY

$$\frac{H \vdash T_1 = T_2 \in Type \qquad H \vdash x_1 = x_2 \in T_1 \qquad H \vdash a_1 = a_2 \in Atom}{H \vdash (x_1 : T_1\|a_1) = (x_2 : T_2\|a_2) \in Type}$$

INDEPENDENTTRIVIALITY

$$\frac{H \vdash x \in T \qquad H \vdash a \in Atom \qquad \text{closed x mentions no names}}{H \vdash (x : T\|a)}$$

INDEPENDENTBASE

$$\frac{H \vdash \neg(x = a \in Atom)}{H \vdash (x : Atom\|a)}$$

INDEPENDENTAPPLICATION

$$\frac{H \vdash (f : (v : A \to B)\|a) \qquad H \vdash (x : A\|a)}{H \vdash (f(x) : B[x/v]\|a)}$$

INDEPENDENTABSURDITY

$$\frac{}{H, (a : Atom\|a), J \vdash T}$$

INDEPENDENTSET

$$\frac{H \vdash (x : T\|a) \qquad H \vdash x \in \{v : T|\ P\}}{H \vdash (x : \{v : T|\ P\}\|a)}$$

The triviality rule, "independentTriviality" says that if x is a closed term that mentions no names at all, then, if x has type T and a is any atom, x and a are independent. The base rule "independentBase" says that distinct atoms are independent, and the "independentAbsurdity" rule says that an atom is not independent of itself.

The application rule "independentApplication" is crucial; it says that if function f and argument x are both independent of atom a then the application $f(x)$ is also independent of atom a. This rule formally captures the fact that atoms can not be built from pieces (or from nothing).

The equality rule "independentEquality" says that independence is well-defined from its constituent pieces, so that these pieces can be replaced by equivalent pieces without changing the meaning. Rules of this form are needed because in constructive type theory terms may have many different types and two terms may stand for the same member of one type but different members of another type. Finally, the rule "independentSet" is somewhat technical and concerns the relation between independence and Nuprl's *Set type*. The rule says that if x is independent of a as a member of type T, then it is also independent of a as a member of any subset of T that contains x.

In appendix A we give the formal meta-theoretic definition of the type $(x : T\|a)$ and prove that these rules are true. Using these six primitive inference rules we program *tactics* that automate most simple reasoning about independence and these tactics suffice for all of our security applications.

As a simple example of what follows from these rules, let us prove that integers and atoms are independent. We will use this lemma in appendix B.

Lemma 1

$$\forall a : Atom.\ \forall i : \mathbb{Z}.\ (i : \mathbb{Z}\|a)$$

Proof By induction on i: the case $i = 0$ is $(0 : \mathbb{Z}\|a)$ which follows by using "independentTriviality" since 0 is closed and mentions no names.
Now assume $(i : \mathbb{Z}\|a)$ and show $(i + 1 : \mathbb{Z}\|a)$ and $(i - 1 : \mathbb{Z}\|a)$:
Since $i + 1$ is $(\lambda x.\ x + 1)(i)$, using the "independentApplication" rule it is enough to show $(\lambda x.\ x + 1 : \mathbb{Z} \to \mathbb{Z}\|a)$ and $(i : \mathbb{Z}\|a)$. The former again follows trivially since $\lambda x.\ x + 1$ is closed and mentions no names, and the latter is the induction hypothesis. The $i - 1$ case is the same. □

3 Message Automata and Event Structures

Our computation system will be the class of programs that are expressible as *Message Automata* and the semantics of a message automaton is defined

to be the set of *event structures* that are *consistent* with it. Message automata and event structures provide a perfectly general model of distributed computation and the model is abstract enough to make specifications and proofs of distributed algorithms simple and elegant. At the same time, the model is not too abstract since we can generate runnable code (we currently generate Java) from a message automaton. Since we can *extract* a message automaton from a proof that a specification is implementable, we can develop an algorithm from specification to running code with automated support at every stage of the development process.

In this section we will present just an overview of this theory. We will give examples of message automata and specifications written in the language of event structures and define what it means for a message automaton to *realize* a specification. This will provide enough context for a discussion of how security properties can be specified and realized; we can postpone the full semantics of message automata to appendix B. For further examples and a more informal introduction to event structures, see [**?**].

Message Automata A message automaton is a finite set of clauses; and there are nine sorts of clauses. Of these nine sorts, only the four used in the automaton M_1 shown below are *active clauses*. The other five sorts of clauses are used to constrain the active clauses.

message automaton M_1
at location i **initially** $x = 1$
at location j **initially** $y = 0$
at location i **initially** $done$ = false

precondition internal(i, go) **is** $done$=false
effect of internal(i, go) **is** $done$:=true
internal(i, go) **sends on link** l_1 : $\langle num, x \rangle$

effect of rcv(l_1, num)(v) **is** y:=y+v
rcv(l_1, num) **sends on link** l_2 : $\langle num, y \rangle$

effect of rcv(l_2,num)(v) **is** x:=x+v
rcv(l_2, num) **sends on link** l_1 : $\langle num, x \rangle$

In automaton M_1 the four sorts of active clause are used to initialize state variables, guard internal actions, update state variables, and send messages; each line is a single clause, and the order of clauses is immaterial. Automaton M_1 mentions two *locations* or *agents*, i and j, and mentions two

communication links, l_1 and l_2. It initializes state variables x, y, and *done*, mentions three *event kinds*, internal(i, *go*), rcv(l_1, *num*), and rcv(l_2, *num*). Identifier *num* is being used as a *message tag* and *go* as an internal *action name*. In the abstract syntax, locations, state variables, tags, and action names are simply identifiers. A link is a triple $\langle source, destination, name\rangle$ of identifiers, which allows for multiple links between a source location and destination location (the link l_1 used in automaton M_1 would be $\langle i, j, z\rangle$ for some name z, and likewise l_2 has the form $\langle j, i, z'\rangle$.) An *event kind* is either *internal*, internal(i, a), (where a is an action name and i is a location), or a *receive*, rcv(l,*tag*), where l is a link and the tag *tag* is used to partition the messages received on the link into classes of different types or different meanings. Every action kind k has a unique location: if k is $internal(i, a)$ then its location is i and if k is $rcv(l, tag)$ then its location is the destination of link l.

Message automata A and B are sets of clauses, and we write $A \oplus B$ for the union of the clauses from both A and B, and call it the *join* of A and B. The join is the basic composition operator on automata.

The behavior of automaton M_1 is as follows. Because of the clauses mentioning the state variable *done*, an event of kind internal(i,*go*) will eventually occur and send a message with tag *num* on link l_1 containing the value of state variable x. If the clauses in M_1 are the only clauses affecting x, then this value would be 1. The message will be received at location j by an event of kind $rcv(l_1, num)$ and this will cause the current value of y to be sent back to location i and also add the received value to state variable y. Afterward i and j continue to send messages back and forth, the receipt of each message sending the next message and also adding to x or y. If nothing else affects variables x and y or sends on links l_1 or l_2 with the tag *num*, then the sequence of values received on l_1 with tag *num* will begin 1,1,1,2,4 and the values received on l_2 with tag *num* will begin 0,1,2,3,5.

In order to guarantee this behavior we may add the constraints in automaton M_2 (forming automaton $M_1 \oplus M_2$)

message automaton M_2
only [internal(i, *go*)] **affect** *done* **at location** i
only [rcv(l_2, *num*)] **affect** x **at location** i
only [rcv(l_1, *num*)] **affect** y **at location** j

only [internal(i, *go*), rcv(l_2,*num*)] **send on link** l_1 **with tag** *num*
only [rcv(l_1, *num*)] **send on link** l_2 **with tag** *num*

The first three clauses in M_2 are *frame* clauses, which constrain the kinds of events that may affect a given state variable, and the last two clauses are *send-frame* clauses, which constrain the kinds that may send on a given link with a given tag. The automata M_1 and M_2 are *compatible* because M_1 satisfies the frame and send-frame clauses in M_2.

The clauses in the following automaton M_3 are also compatible with M_1.

message automaton M_3
only [internal(i, *go*), rcv(l_2, *num*)] **read** x **at location** i
internal(i, *go*) **affects only** [*done*] **at location** i
internal(i, *go*) **sends only on link** l_1
rcv(l_2, *num*) **affects only** [x] **at location** i
rcv(l_2, *num*) **sends only on link** l_1

These clauses all constrain *data flow*. Imagine that the value of x is confidential. The first clause in M_3 is a *read-frame* clause, which constrains the kinds of events that may read x. If events of a certain kind can read a confidential state variable x, then we will want to constrain what these events may do. The second clause in M_3 is an *action-frame* clause, which lists all the state variables that events of the given kind may affect, and the third clause is an *action-send-frame* clause, which lists all links on which the event may send. Clauses of these three sorts can constrain the possible behavior of message automata enough that we can prove that confidential information, in the form of unguessable atoms, stays in a given set of state variables.

We have now introduced by example all nine sorts of clauses from which message automata are built. We call any of the five sorts of clauses in automata M_2 and M_3 *frame clauses* and the sorts of clauses in M_1 *active clauses*. For the sake of readability, we have omitted from the syntax of these examples the declarations of the types of the state variables. Any active clause that depends on a state variable includes a type declaration for that variable, and this makes it possible to define a syntactic check for whether an active clause is *compatible* with a frame clause (since we can conservatively assume that a clause declaring x reads x). Two active clauses or two frame clauses are compatible if they are identical when they constrain the same state variable or event kind, and an automaton M is *feasible* if its clauses are pairwise compatible.

Event Structures In a run of a message automaton, internal actions and the receipts of messages are the *events*. When an event e occurs, it occurs at

a particular location $loc(e)$ and it has a particular kind, $kind(e)$. Every event e has a value, $val(e)$: the value of a receive event is the message received, and the value of an internal action can be a random choice [1]. When event e occurs, the value of the state variable x at location $loc(e)$ is (x **when** e) and its value after event e is (x **after** e) (the same identifier x may be used for different state variables at different locations.) Events at a given location are totally ordered and every receive event e was caused by another event $sender(e)$. The transitive closure of the union of the local ordering of events and the relation $sender(e) < e$ defines (following Lamport) the *causal order*, written $e < e'$.

An *event structure* is an abstract mathematical structure that has a type E of events as its domain and the operators *loc*, *kind*, *val*, **when**, **after**, and $<$ in its signature, and which satisfies a few simple axioms. These axioms, and a more complete definition of event structures are given in appendix B. We can define many more concepts in terms of the primitive operators of an event structure. One simple definition is the state when an event occurs:

$$\textbf{state when } e \equiv (\lambda x.\ x \textbf{ when } e)$$

Semantics of message automata For each of the nine sorts of message automaton clauses C we define (in appendix B) a formula ψ_C in the language of event structures and say that event structure es is *consistent with* clause C if $es \models \psi_C$. For example, if C is the *effect clause*

effect of k **is** x:=f(state,val)

then ψ_C is

$$\forall e.\ kind(e) = k \ \Rightarrow\ x \textbf{ after } e = f((\textbf{state when } e), val(e))$$

An event structure es is consistent with a message automaton M if es is consistent with each clause of M. The semantics of a message automaton M is the set of event structures consistent with M.

Feasibility and Realizability We have made a formal definition of a predicate $Feasible(M)$ on message automata, which captures the pairwise compatibility of the clauses of M, and we have a (rather difficult) fully formalized, constructive proof that

$$Feasible(M) \ \Rightarrow\ \exists es.\ Consistent(es, M)$$

[1] The general form of the precondition clause allows every internal action to choose a random value, but since we will not need this feature of message automata in this paper, we have omitted that part of the syntax from the examples.

We say that M *realizes* specification ψ if every event structure consistent with M satisfies ψ and M is feasible (so that it has at least one consistent event structure.)

$$M \textbf{ realizes } \psi \equiv (Feasible(M) \ \wedge \ \forall es.\, Consistent(es, M) \ \Rightarrow \ es \models \psi)$$

If M **realizes** ψ then any feasible extension $M \oplus X$ of M also realizes ψ (because any event structure consistent with $M \oplus X$ is also consistent with M). We use the word *specification* for a formula ψ in the language of event structures, and a *realizer* for ψ is a message automaton M such that M **realizes** ψ.

4 Specification and proof of security properties

To give a formal proof that a protocol, using some cryptographic services, satisfies a security property, we represent the cryptographic service by a specification ψ_{crypto}, the protocol as a message automaton M, and the security property as a specification ψ_{secure}. For example, a security property might be that a given list L of agents should share a common secret. The safety part of this specification could be

$$\psi_1(a) \equiv \ \forall e.\, (loc(e) \in L \ \vee \ ((\textbf{state } \textbf{when } e) \| a))$$

This says that the state of any agent is always independent of the secret a unless the agent in a member of L. The liveness part of the specification could say that every agent in L eventually has the secret in a given state variable x.

$$\psi_2(a) \equiv \ \forall i.\, i \in L \ \Rightarrow \ \exists e.\, (loc(e) = i \ \wedge \ x \ \textbf{when } e = a)$$

The full specification could be

$$\exists a : Atom.\, \psi_1(a) \ \wedge \ \psi_2(a)$$

In section 5 we will further support our claim that the logic of events with atoms and independence is expressive enough to specify the cryptographic assumptions ψ_{crypto} and desired property ψ_{secure} and we will discuss how cryptography can be modeled using message automata exchanging atoms. We claim that a formal proof of the security property is just a proof of M **realizes** ψ where $\psi = (\psi_{crypto} \ \Rightarrow \ \psi_{secure})$, but this claim deserves some further discussion.

The reader may ask where is the model of the adversaries and how are the adversaries able to eavesdrop on the messages sent by the protocol M? The answer lies in the meaning of M **realizes** ψ, which we reiterate, "M is feasible and every event structure consistent with M satisfies ψ". If M **realizes** ψ then any feasible extension $M \oplus X$ of M also realizes ψ, so adversaries expressible as a message automaton, X, *compatible with* M cannot violate ψ. Thus, the model of the adversaries is any message automata X compatible with M; but what sort of assumption is compatibility with M?

For agents X distinct from the agents in M, compatibility with M requires only *interface compatibility*, and this requires only that messages of type T sent on a link between M and X will be received with a value of type T. Thus, for "external" adversaries, the only restriction is that they can not use a type-coercion attack. This means that if M sends protected information (an atom) a that is received by an adversary X, then X can not coerce a to some other type. This restriction reflects our idealization of cryptography to the use of atoms, so such type-coercion attacks are not covered by our formal security proof.

For agents X that are not distinct from the agents in M, compatibility requires much more. It requires that the clauses of X obey the restrictions given by the frame clauses in M, so if M says that only actions of kind k_1 or k_2 read state variable z, and that k_1 and k_2 only send on certain links, then X may not cause an action of some other kind to read z, and it may not cause k_1 or k_2 to send information (which could include z) on any other link (thus leaking z). Our model of adversaries thus includes "Trojan horses" in that we are not assuming that adversaries must be external. For a protocol M to satisfy a security specification it must include enough frame clause restrictions to prevent code added to its own process from leaking information. Many security proofs model the protocol as a given, closed, program and the attackers as external. We are not assuming that the protocol is closed, but rather require it to explicitly restrict the rest of the code at its location. These explicit restrictions could be used by a compiler to add extra protection measures around the access to protected state variables and thus guard against Trojan horses or "accidental" leaks. Since we express all formal properties of distributed systems in the form M **realizes** ψ, we are not treating security properties differently from any other kind of property, and all proofs are done in the same theory of event structures.

If a message automaton M includes an action-send-frame clause like

k **sends only on links** l_1,l_2

then the messages sent by events of kind k will be received only on links l_1 and l_2. This implies that such messages cannot be received by eavesdroppers and hence that links l_1 and l_2 are assumed to be secure. Without such a clause, if M has an action of kind k that can read a secret x and send it on some link l, then some feasible extension M' of M could also send x on any other link, including a link l' to any adversary. To prove that a protocol represented as a message automaton M guarantees a security property ψ even when adversaries may eavesdrop on all messages that M sends, we prove that M **realizes** ψ and check that M does not contain any action-send-frame clauses. Thus eavesdropping is modeled by a lack of clauses restricting the links on which messages may be sent, so proofs have to consider cases where every message sent to any agent could be broadcast to all agents.

Having action-send-frame clauses included in message automata allows us to realize a specification *assuming* that a given link l is secure. As we will discuss below, one way to model cryptographic functions is by sending messages on links to *virtual* agents. In the strongest security proofs we will only assume that links to virtual agents are secure and all other links are subject to eavesdropping.

5 Modeling of Cryptographic Systems

Atoms, independence, and message automata provide the foundation for our formal theory of security, but to reason about real cryptographic protocols we must build on this foundation another "layer" to model the ability of agents to choose nonces, encrypt messages, establish a public/private key pairs, and perform other more complex cryptographic operations such as secret shares. We start by discussing how we might model the simplest of these, the ability of agents to choose nonces.

Suppose that we know how to realize a specification ψ given the ability to choose nonces. Then we would know how to build an automaton M that realizes specification ψ if we are allowed to use effect clauses like

effect of k **is** x:=nonce

that allow an agent to set a state variable to a new nonce whenever a certain kind of event occurs. The proof that M realizes ψ will depend on some properties of these nonces such as the assertion that they are all distinct and cannot be guessed. Since we don't have such effect clauses in message automata, we have to model such behavior using what we do have. The

obvious choice is for the nonces to be atoms, and there are at least two ways we can model the effect x:= nonce. One way is to *assume* that every agent has a supply of atoms unique to that agent by adding a state variables $nonces : List(Atom)$ and $nextn : \mathbb{Z}$ to each agent and letting the assumption $NonceAssumption$ be

$$\forall i, j : Loc.\ i \neq j \ \Rightarrow\ \forall a : Atom.\ a \in (nonces\ \textbf{initially}\ i) \ \Rightarrow\ (j \| a)$$

which says that agent j is initially independent of every atom a[2] in the list of nonces at i. If we add to the $NonceAssumption$ the assertion that the lists of nonces at each agent have no repeats, then we can use

$$\textbf{effect of}\ k\ \textbf{is}\ x, nextn := nonces[nextn], nextn + 1$$

to model **effect of** k **is** x:=nonce. The resulting automaton M' will realize not specification ψ but rather the specification $(NonceAssumption \ \Rightarrow\ \psi)$.

When we generate code from automaton M' we would recognize the special state variables and effect clauses and generate special code for them by replacing the assignment x:=nonces[nextn] with x:=random($parameters$). This changes the automaton so we no longer have a mathematical certainty that the generated code satisfies specification ψ, but we would claim that the transformation provides a good approximation to the $NonceAssumption$. The analysis of the approximation would be in terms of probability or complexity theory and is outside the scope of our idealized logical model.

Another way to model nonces, which generalizes more easily to other cryptographic services, is to provide a *virtual* agent playing the role of a *crypto server*. A *crypto system* is a collection of services, *encrypt*, *decrypt*, etc. The crypto server models these services as messages. For example an agent encrypts x with key a by sending a message containing the pair $\langle x, a \rangle$ to the server and receives the encrypted value in response. We reason about the crypto server like any other agent, as a message automaton. When we generate code from a message automaton that sends and receives from a virtual agent we replace the sends and receives by the code that calls the crypto system the virtual server is modeling.

In the case of a nonce server, we would have a designated agent, i.e. a location, "nsvr" with state variables $nonces : List(Atom)$ and $nextn : \mathbb{Z}$. The $NonceAssumption$ is now

$$\forall j : Loc.\ j \neq nsvr \ \Rightarrow\ \forall a : Atom.\ a \in (nonces\ \textbf{initially}\ nsvr) \ \Rightarrow\ (j \| a)$$

[2]The full definition of $(j \| a)$ is in section B.4.

i.e. that the nonces in the initial state of the nonce server are independent from all other agents initial states and programs. Instead of an effect x:=nonce, an agent will send a request to the nonce server and when the server receives a request it will send back nonces[nextn] and increment nextn. When the original agent receives a reply e from the server it sets x:=$val(e)$. All of this behavior can be realized with standard message automata, and we can prove once and for all that the nonce server satisfies useful properties such as never sending the same atom twice.

We would like the nonce server to have an infinite supply of atoms in the *nonces* state variable, but this assumption would inconsistent since the basic objects in our logical system are finite terms and can mention only a finite set of atoms. Thus, the nonce server can have only a finite list of nonces and hence it will eventually run out of nonces. This is realistic, since if nonces were "really" n-bit random numbers they would run out after at most 2^n steps. Safety specifications and their proofs will not be affected by this limitation of the nonce server, but liveness specifications will have the form "an event e such that $P[e]$ will occur *unless* the nonce server runs out of nonces".

Using virtual agents to model the crypto system also makes generating code from the resulting automata less ad hoc. We replace messages sent to the virtual agents with calls to the actual crypto-system library and make their return values into receives of messages. This could be done by adding wrappers around the crypto-system calls to make them have the same interface as sends and receives on special links.

With this approach, a model of a cryptographic service is a specification ψ_{crypto} of the behavior of a virtual agent which includes assumptions like the $NonceAssumption$ that certain atoms in the state of the virtual agent are independent of all other agents. Then automaton M realizes security specification ψ_{secure} assuming the cryptographic service if M realizes the specification $(\psi_{crypto} \Rightarrow \psi_{secure})$.

5.1 Model of Public Key Cryptography

We conclude this section with a sketch of a model, suggested by Robbert van Rennesse, of public key cryptography as a virtual agent we call a *secret server*. All of its behavioral constraints can be written in event logic to provide a specification ψ_{ss} for the secret server, but here we will describe its behavior informally.

The state variables consist of a (large) table *tab* and an index *next* into the table. The n^{th} row of the table has the form $\langle a_n, k_n, v_n \rangle$ where a_n is an

atom, k_n is a *key* of type $\mathbb{N} + Atom$, i.e. either a number or an atom, and v_n is some data. We assume that all the a_n are distinct and independent from the initial state and program of all other agents (the same assumption as for the nonce server). Initially *next* is zero and the initial values of the k_n's and v_n's are irrelevant.

If the server receives a message of the form $encrypt(k, v)$ when $next = n$, then it updates row n to $\langle a_n, k, v\rangle$, sends back the pair $\langle n, a_n\rangle$, and increments *next* to $n + 1$. If the server receives a message of the form $decrypt(k', a)$ when $next = n$, then it searches the first n rows for one of the form $\langle a, k, v\rangle$ (there is at most one such) and if it finds one and if the keys k and k' match, then it sends back the value v. The keys k and k' match if one is a number $i < n$ and the other is the atom a_i. The server never changes its state or sends anything for any other reason.

To see how this model corresponds to public key cryptography consider the following sequence of events. Agent Bob sends message $encrypt(0, 0)$ and receives the pair $\langle 1, a_1\rangle$. Bob can now use the number 1 as his public key and atom a_1 as his private key (so he must store a_1 in his state and never send it anywhere but to the secret server). Bob sends messages to the world advertising the fact that his public key is the number 1. Agent Alice sends message $encrypt(1, \textit{ms-for-Bob})$ and receives $\langle 17, a_{17}\rangle$ and sends a message to Bob containing a_{17} asking Bob to decrypt it. Bob sends message $decrypt(a_1, a_{17})$ and receives message *ms-for-Bob*. Or, Bob sends $encrypt(a_1, \textit{ms-from-Bob})$ and receives $\langle 29, a_{29}\rangle$ and includes a_{29} in a message to Alice. Alice sends $decrypt(1, a_{29})$ and receives *ms-from-Bob*.

So, in this model, public/private key pairs are pairs $\langle n, a_n\rangle$ from the state of the virtual agent, and *all encrypted values are atoms*. Since the server always sends a new atom for each encryption request, we may also use dummy encrypted messages as nonces, so these will be atoms, too. A typical security specification will say that all agents outside a given *cabal* are independent of certain private keys, nonces, or encrypted messages.

To make a model of a different crypto-system we would vary the state and interface of the server to correspond with the services the system provides. For example, to model a system for secret shares, we would need a more complex table that associates a list of atoms with a piece of data and will decrypt only when given enough of these secret shares. All of these models are just specifications in event logic of the behaviors of suitable servers and we believe that all crypto systems can be modeled at this level of idealization in this way.

6 Related Work

Related work on the use of atoms in logic is by Gabbay and Pitts [4] who use atoms to explain variable binding. They use hereditarily finite sets with urelements as a foundation and some of the general theory they develop is related to our concept of independence.

The related work on formal proofs of security is vast. We mention here only some of the work that contains ideas that translate easily into our model of message automata, event structures, and atoms. We think that this model could form a unifying framework for many of these ideas and methods.

Strand spaces [8] provide extra structure to an algebraic model. The extra structure is a graph of interactions between *strands*, which are local orderings of events (sends or receives of messages) at a single agent. The graph defines a causal ordering which is well-founded, so any subset of its domain has a minimal element. This allows proofs by induction over the causal ordering and makes it possible to prove some general properties for whole classes of protocols and adversaries. One of the useful concepts is that of the *originator* of a message m, the minimal (in a suitably closed subgraph called a *bundle*) event containing m as a subexpression.

Since a local ordering of events and the well-founded causal order are axioms of event structures (see appendix B) the concepts of strand spaces translate directly to the semantics of message automata. We can prove general properties of data flow for atoms, such as lemma 7 in section B.4, and define concepts like the originator of an atom a (the agent that held a initially).

Paulson's inductive approach to proving security [7, 6] and it's elaboration by Millen and Rueß [5] provide organizing principles for carrying out proofs of secrecy. A protocol-independent *secrecy theorem* proved by induction on a well-founded ordering on traces reduces proofs of secrecy to a "local" condition on the honest agents called *discreetness*. We expect to be able to translate all these concepts and theorems into our framework. By doing this we will strengthen the results by proving them for a more general adversary model and we will be able to generate running code from the message automata extracted from applications of these theorems.

7 Conclusions

The concepts of atoms and independence are well-behaved and can be added to a logical system without restricting its expressiveness.

We can combine a general model of computation with a simple model of acquisition of secret information.

Cryptographic protocols can be modeled as automata exchanging atoms, and this model can provide a unifying framework for reasoning about security.

(Acknowledgements and References may be found after the appendices.)

A Definition and Rules for Independence

In constructive type theory, propositions are types, and a type is a *partial equivalence relation* on terms, so to define new propositions we must

1. define the type expressions that represent the new propositions
2. define when two of these type expressions represent the same type
3. define the members of the type and the equivalence relation on the members
4. show that inference rules about the type are valid

These tasks are done in the *metatheory* where the primitive objects are *terms* and there is a primitive *evaluates to* relation on terms. Validity of inference rules is defined in terms of the semantics of *sequents* for which details can be found in [2], but here we simply assert that the validity of the rules for independence reduces to the lemmas below.

Our new type expressions have the form $(x : T\|a)$. Such an expression will be a type if $a \in Atom$, $T \in Type$[3], and $x \in T$. Two expressions $(x : T\|a)$ and $(x' : T'\|a')$ represent the same type if $a = a' \in Atom$, $T = T' \in Type$, $x = x' \in T$. The equivalence relation for the type $(x : T\|a)$ will be trivial: all its members are equivalent (so it is either empty or a single non-empty equivalence class). The members of the type $(x : T\|a)$ will also be trivial: when $(x : T\|a)$ is *true* the members will be just the terms that evaluate to

[3]In Nuprl. there is no type *Type* of all types; instead there is a cumulative hierarchy of *universes*. In this paper, we use the symbol *Type* for an arbitrary universe.

a particular canonical term $Axiom$, and when $(x : T\|a)$ is not true the type has no members. So we only have to define when $(x : T\|a)$ is true and make sure that the definition respects the type equality (lemma 2).

Definition The proposition $(x : T\|a)$ is true if and only if a evaluates to $tok(b)$ for some name b, and there exists a term y in type T such that $x = y \in T$ and y does not mention name b.

Lemma 2 *If* $(x : T\|a) = (x' : T'\|a')$ *then* $(x : T\|a) \Leftrightarrow (x' : T'\|a')$. *So, the previous definition is well-formed.*

Proof Assume that $a = a' \in Atom$, $T = T' \in Type$, $x = x' \in T$, $y \in T$, $x = y \in T$, a evaluates to $tok(b)$, and y does not mention b. We must show that a' evaluates to $tok(b')$ for some name b', and that there exists a $y' \in T'$ for which $x' = y' \in T'$ and y' does not mention b'. Atoms have unique canonical forms, so since $a = a' \in Atom$, a' must evaluate to $tok(b)$, so we can take b'=b. Then we can take $y' = y$, since y does not mention b and $x = y \in T$ and $x = x' \in T$ and $T = T' \in Type$ imply that $x' = y \in T'$. □

Note that this lemma also shows that the rule "independentEquality" is valid.

This completes our definition of the new primitive proposition $(x : T\|a)$. The validity of the other inference rules in table 1 is proved in the following lemmas.

Lemma 3 (independentApplication)

$$((f : (x : A \rightarrow B[x])\|a) \wedge (x : A\|a)) \Rightarrow (f(x) : B[x]\|a)$$

Proof If a evaluates to $tok(b)$ and $f = f' \in (x : A \rightarrow B[x])$ and f' does not mention b, and if $x = x' \in A$ and x' does not mention b, then the term $f'(x')$ does not mention b and by the definition of the dependent function type $x : A \rightarrow B[x]$, we have $f(x) = f'(x') \in B[x]$. □

Lemma 4 (independentAbsurdity)

$$(a : Atom\|a) \;\Rightarrow\; False$$

Proof If a evaluates to $tok(b)$ and $y = a \in Atom$ and y does not mention b, then we have $y = tok(b) \in Atom$, by computation, and $y = tok(c) \in Atom$, by the permutation rule for names. Thus $tok(b) = tok(c) \in Atom$, and this implies $False$. □

Lemma 5 (independentBase)

$$\neg(x = a \in Atom) \;\Rightarrow\; (x : Atom\|a)$$

Proof If a evaluates to $tok(b)$ and $\neg(x = a \in Atom)$ then x must evaluate to $tok(c)$ for some $c \neq b$, so $tok(c) = x \in Atom$ and $tok(c)$ does not mention b. $\square$

Lemma 6 (independentTriviality) *If x is a closed term in type T and x mentions no unhideable token names, then for any $a \in Atom$, $(x : T \| a)$*

Proof This is indeed trivial since we can choose the y in the definition of independence to be x. $\square$

In general, independence is not preserved by subtypes. If $(x : B \| a)$ and x is a member of a subtype A of B, then it may not be true that $(x : A \| a)$. A simple example of this is that for $a \in Atom$ we have $(a : Top \| a)$, $Atom$ a subtype of Top, but $\neg(a : Atom \| a)$. This is because Top is the type which has every closed term as a member but in which any two members are equal, so $a = 17 \in Top$ and 17 mentions no names. If, however, A is a subtype of B in which equality is just the restriction of the equality in B to the members of A, then independence is preserved. This is the case for the *set type* $\{v : B |\ P\}$, and this is the justification for the rule "independentSet" in table 1.

A.1 Inherence

We originally attempted to define the proposition $(x : T \gg a)$ that means that atom a is *inherent* in x (the negation of independence $(x : T \| a)$). To base our theory on this primitive, we need the "dual" of the rule "independentApplication", which is the following *inherence application principle*

$$(f(x) : B[x/v] \gg a) \Rightarrow (f : (v : A \to B) \gg a) \ \vee \ (x : A \gg a)$$

i.e. if an atom is inherent in the result of applying a function f to an argument x then the atom is either inherent in the function f or else inherent in the argument x. This principle captured our intuition that atoms cannot be built from parts so that an atom can't appear in $f(x)$ partly from f and partly from x. However, for this principle to be true constructively would require that there be an effective procedure to determine whether the atom was inherent in f or in x. This might seem trivial: just examine the expressions f and x and see which mentions atom a, but, for inherence to be a proposition in our logic, it must be closed under replacing f by an equivalent f' in its type, and x by an x' that denotes the same value of its type. So, for example, a is not inherent in $(\lambda x, y.\ y)(a)$ since this evaluates to $\lambda y.\ y$. It might be possible to obfuscate a class of functions f

and arguments x by adding irrelevant occurrences of a in such a way that it is an undecidable problem to identify which of a pair f,x from the class for which $(f(x) : T \gg a)$ is the one that contributed the atom.

We were not able to find a definition of inherence for which we could justify the constructive inherence application principle. We considered various additions to the computation system that would allow the tracing of atoms during evaluation and which would allow us to construct the witness to the inherence application principle, but no mechanism we considered was compatible with the permutation semantics on atoms and the other constraints on the logical system. We sketch below a proof that the principle is, in fact, not constructive. Once we realized that independence was a sufficient basis for security properties in our logic we abandoned the notion of inherence and restated all security specifications in terms of independence. In a classical logic, the inherence application principle is equivalent to the independentApplication rule, so users of classical logic could use either inherence or independence as a primitive (the ability to add a type of atoms satisfying the permutation semantics does not depend on the logic being constructive, so our whole approach is compatible with classical logic).

A.1.1 Inherence application principle not constructive

We can show that the inherence application principle is not constructively valid if we admit some functions that are not recursive but can still be considered constructible. These functions are *random choice sequences* analogous to Brouwer's free choice sequences. For the purpose of this section we need only sequences of outcomes from the space $\{0, 1\}$ where 0 and 1 each have probability $1/2$, but the theory of random sequences we use generalizes to sequences of outcomes from any finite probability space.

We can add to the computation system a table $T(a, n)$ whose rows are indexed by atoms a and whose columns are indexed by natural numbers n. At all times, table T is finite, so that it contains only finitely many triples $\langle a, n, T(a, n)\rangle$, but as time progresses more triples are added as follows. Whenever the computation system evaluates a term of the form $rand(a, n)$, it computes the canonical forms of a and n and returns the value of $T(a, n)$ if it is in the table. Otherwise it "flips a fair coin" and fills in the value of $T(a, n)$ according to whether it gets a head or tail. Thus the term $rand(a) = \lambda n.\ rand(a, n)$ is a member of the type $\Omega = \mathbb{N} \to \mathbb{N}_2$. The concept of an open subset of Ω of measure one can be easily defined (with no need to define topology or measure in general). It is consistent to add the following **axiom**:

> If C is an open set of measure one and C is independent of atom a, then there exists n such that the basic open set determined by $rand(a,0),\ldots,rand(a,n)$ is contained in C.

This axiom asserts that the random sequences in the rows of table T are mutually generic random sequences. In fact, for our argument that the inherence application principle is not constructive we need only one generic random sequence $rand(a)$.

For any function f in the type $\mathbb{N} \to \mathbb{N}$, define the sequence r_f by $r_f(n) = rand(a,n)$, if $f(n)$ is different from $f(0),\ldots,f(n-1)$, and $r_f(n) = 0$, otherwise. If we think of f as enumerating a recursively enumerable set A, then if x is enumerated into A at time n, $r_f(n) = rand(a,n)$, and if nothing is added to A at time n, then $r_f(n) = 0$.

Now we claim that if f itself is independent of atom a, then atom a is inherent in the sequence r_f if and only if the set $A = range(f)$ is infinite. This is the same as asserting that r_f is independent of atom a if and only if A is finite. One direction of this assertion is clear. If A is finite then $r_f(n)$ will be 0 for all but finitely many n, so there is a finite term that does not mention atom a that is equal to sequence r_f. If A is infinite, then the set of sequences g for which there is an n where $g(n) = 0$ and $r_f(n) = 1$ is an open set of measure one (because, since f and hence A is independent of a, $rand(a,n) = 1$ for infinitely many of the n for which something is added to A at time n). If r_f is independent of atom a, then this open set is independent of a, and hence the axiom implies that $rand(a)$ meets the set, but this is impossible since $r_f(n) = 1$ implies $rand(a,n) = 1$. Thus r_f is not independent of a if A is infinite.

Now given two functions, f and g, both independent of a, we can form the pointwise maximum, $max(r_f, r_g)$. Atom a is inherent in this sequence if and only if the union of $range(f)$ and $range(g)$ is infinite. If the inherence application principle were constructive, there would be a constructive procedure to decide whether atom a was inherent in r_f or in r_g, but this would provide a constructive procedure to decide, given two r.e. sets whose union is infinite, which of them is infinite, and there is clearly no constructive procedure to decide this.

B Semantics of Message Automata

A message automaton is a representation of a distributed program. The behavior of such a program is a set of, possibly infinite, histories (which we abstract to form *event structures*) but the program itself is a finite object.

The exact details of how this finite object are coded into the computational type theory are not relevant here, so we may define a *message automaton* as a *finite set of clauses* and a *clause* as an instance of one of the following nine schemes, which we partition into four *active clauses* and five *frame clauses*. We have discussed in section 3 the syntax of locations, links, and kinds. The full syntax of the message automaton clauses includes abstract syntax for declaring the types of state variables and tagged messages, but to avoid overloading the reader with details we will omit those parts of the syntax since the essential concepts can be understood without them. In tables 2 and 3 we indicate, for each of the nine clause schemes, its syntactic form, the name we use for it, and its intended meaning. The formal meaning defines which event structures are consistent with the clause; we discuss event structures in the next section.

Table 2: Message Automaton Active Clauses

at location i **initially** x = v (*init*)
In the initial state of agent i, the state variable x has value v.

effect of k **is** x:=f(state,val) (*effect*)
Every action of kind k with a value, v, updates the state variable x, at the location of k, to the value $f(s, v)$ where s is the current state.

k **sends on link** l : f(state,v) (*send*)
Every action of kind k with a value, v, sends on link l a (possibly empty) list of tagged messages, $f(s, v)$ where s is the current state.

precondition internal(i,a) **is** P(state) (*precondition*)
An internal action of kind $internal(i, a)$ may not occur at i unless P is true in the current state, and, infinitely often, the agent either checks that P is false or performs an action of kind $internal(i, a)$.

When we generate Java code from message automata, only the active clauses generate any code; the frame clauses merely restrict the set of active clauses whose inclusion in the message automaton is consistent. A message automaton is *feasible* if there is at least one event structure consistent with it, and in particular, a feasible automaton must obey all of its own frame clauses. So, for example, an automaton that contained both clauses

effect of internal(i,a) **is** x:=f(state,val)

only [rcv(l,tag), internal(i,b)] **affect** x **at location** i

or both clauses

Table 3: Message Automaton Frame Clauses

only k1,k2,... affect x **at location** i (*frame*)
Only actions with kind in the given list may affect the state variable x at location i.

only k1,k2,... send on link l **with tag** tg (*send-frame*)
Only actions with kind in the given list may send messages tagged tg on link l .

only k1,k2,... read x **at location** i (*read-frame*)
Only actions with kind in the given list may read the state variable x at location i.

k **affects only** x,y,... **at location** i (*action-frame*)
An action of kind k affects only state variables in the given list.

k **sends only on links** l1,l2,... (*action-send-frame*)
An action of kind k sends only on links in the given list.

effect of internal(i,a) **is** y:= x + 1

i **only** [rcv(l,tag), internal(i,b)] **read** x **at location** i

would be infeasible (unless a = b), since, in the first case, internal(i,a) affects state variable x but is not listed in the frame clause given for x at location i, and, in the second case, internal(i,a) reads variable x to update variable y, but is not listed in the read-frame clause given for x at location i.

There is an essentially syntactic check for feasibility (modulo type checking, which, in an expressive logic, can require theorem proving), so we could implement a compiler that refuses to generate code for an "illegal" program that fails the feasibility test.

The first two frame conditions, *frame* and *send-frame*, alow us to constrain programs enough to prove the kinds of state invariants needed to prove normal, non-security, specifications of properties like consensus, and mutual exclusion. Thus, the active clauses and the first two frame clauses suffice for a logic of program development for "most" distributed systems. It was only in attempting to specify and prove security properties that we discovered the need for the other frame clauses. These clauses all constrain "data-flow" and are essential in proofs that secret information will not be leaked.

B.1 Event Structures

An *event structure* is a formal mathematical structure in which we can interpret a declarative language that can be understood without direct reference to a computing model or to code. Propositions about event structures, expressed in this language, become specifications about programs once we give a semantics for the programs as sets of event structures. A program then satisfies a specification if all of its event structures satisfy the specified proposition.

In this section we first present enough of the language of event structures (which we call *event logic*) to explain the semantics of all but the read-frame clause. Then we will discuss some extra structure that must be added to allow us to give the semantics for the read-frame clause.

The details of how a mathematical structure is represented (as a dependent product type which includes its axioms via the propositions = types isomorphism) in type theory are not relevant to this paper, so we present event structures by giving the signatures of the operators it provides and describing the axioms. In the following $\mathbb{D}$ denotes a universe of types that have decidable equality tests, and the types **Loc**, **Act**, and **Tag** are all names for the same type **Id** of identifiers. This type is actually implemented, for software engineering reasons related to namespace management and abstract components, as another space of atoms like the ones used here for protected information. But that is a topic for another paper; here **Id** is just a type in $\mathbb{D}$. The type **Lnk** is the product $\mathbf{Loc} \times \mathbf{Loc} \times \mathbf{Id}$, so a link l is a triple $\langle i, j, x \rangle$ with $\mathbf{src}(l) = i$ and $\mathbf{dst}(l) = j$.

B.1.1 Basic Event Structures

The account will be layered, starting with the most basic properties of events and adding layer by layer more expressiveness. To give an idea of how these layers are formally presented, we show in table 4 the signature of some of the layers. In these definitions we use the *disjoint union* of two sets or types, $A + B$ and the computable function space operator $A \to B$. The type $Unit$ has a single distinguished element.

Events with Order Events are the atomic units of the theory. They are the occurrences of atomic actions in space/time. The structure of *event space* is determined by the organization of events into discrete *loci*, each a separate locus of actions through time at which events are sequentially ordered. Loci abstract the notion of an agent or a process. They do not

Table 4: **Signatures in the Logic of Events**

Events with Order
E: $\mathbb{D}$
pred?: $E \rightarrow (E + Loc)$
sender?: $E \rightarrow (E + Unit)$

and with Values
Kind $= Loc \times Act + Lnk \times Tag$
vtyp: $Kind \rightarrow Type$
kind: $e : E \rightarrow Kind$
val: $e : E \rightarrow \mathbf{vtyp}(kind(e))$

and with State
typ: $Id \rightarrow Loc \rightarrow Type$
initially: $x : Id \rightarrow i : Loc \rightarrow typ(x, i)$
when: $x : Id \rightarrow e : E \rightarrow typ(x, loc(e))$
after: $x : Id \rightarrow e : E \rightarrow typ(x, loc(e))$

Definitional extensions
loc: $E \rightarrow Loc$
first: $E \rightarrow Bool$
isrcv: $E \rightarrow Bool$
$x < y \ , \quad x <_{loc} y$

sender: $\{e : E | isrcv(e)\} \rightarrow E$
link: $\{e : E | isrcv(e)\} \rightarrow Link$
tag: $\{e : E | isrcv(e)\} \rightarrow Tag$
Msg $= l : Link \times t : Tag \times \mathbf{vtyp}(rcv(l, t))$

state$(i) = x : Id \rightarrow typ(x, i)$

state-when: $e : E \rightarrow state(loc(e))$
state-after: $e : E \rightarrow state(loc(e))$

share state. All actions take place at these locations.

There are two kinds of events, internal actions and signal detection (message reception). These events are *causally ordered*, e before e', denoted $e < e'$. As Lamport postulated, *causal order* is the structure of time. Causal order is defined in terms of two primitive functions, *pred*? and *sender*? which compute respectively the previous action at its locus (if the event is not the first at that location) and the sender of a received message.

The signature of events with order requires only two discrete types E and Loc, and two partial functions. The function *pred*? finds the predecessor event of e if e is not the first event at a locus or it returns the *location* if e is the first event. The *sender?(e)* is the event that sent e if e is a *receive*, otherwise it is a unit. We can find the location of an event by tracing back the predecessors until the value of *pred* belongs to *Loc*. This is a kind of partial function on E. From *pred? and sender?* we can define Boolean valued functions that identify the first event and receive events. We can define a function *loc* that returns the location of an event. Causal order, $e < e'$, is defined as the transitive closure of the relations $e = pred?(e')$ and $e = sender?(e')$. We can also define the local linear ordering of events at a location, $<_{loc}$, the restriction of causal order, $<$, to a location.

Events with Value We next classify events by their kind, by introducing the type $Kind$ and a function $kind$ from events to kinds. The type $Kind$ is a disjoint union that represents our two basic kinds: an internal action at a location, or the receive of a message on a link with a given tag.

Each kind of action has a value associated with it. The value of a receive event is the message received. The value of an internal action can be chosen randomly or nondeterministically. The value of an event e is $val(e)$ and its type depends only on $kind(e)$.

Events with State We are interested in actions with observable results. Observables are known by *identifiers* and have *types*. At a fixed location or agent, the map of identifiers to values is its *state*. Relations **when**, **after**, and **initially** (which we write with infix notation) connect events to the values of identifiers, e.g. x **when** e is the value of the variable x at the location $loc(e)$ when event e occurs.

Table 5: Axioms of Basic Event Structures

1. On any link, an event sends boundedly many messages; formally:
   $\forall l : Link. \forall e : E. \exists e' : E. \forall e'' : E. R(e'', e) \Rightarrow e'' < e' \wedge loc(e') = dst(l)$
   where $R(e'', e) \equiv isrcv(e'') \wedge sender(e'') = e \wedge link(e'') = l$
2. The predecessor function is one-to-one; formally:
   $\forall e_1, e_2 : E.\ pred?(e_1) = pred?(e_2) \Rightarrow e_1 = e_2$
3. Causal order is (strongly) well-founded; formally:
   $\exists f : E \rightarrow \mathbb{N}. \forall e_1, e_2 : E.\ e_1 < e_2 \Rightarrow f(e_1) < f(e_2)$
4. The location of the sender of an event is the source of the link on which the message was received; formally:
   $\forall e : E.\ isrcv(e) \Rightarrow loc(sender(e)) = src(link(e))$
5. Links deliver messages in FIFO (first in first out) order; formally:
   $\forall e_1, e_2 : E.\ link(e_1) = link(e_2) \Rightarrow sender(e_1) < sender(e_2) \Rightarrow e_1 < e_2$
6. State variables change only at events, so that:
   $\forall e : E. \neg first(e) \Rightarrow (x$ **when** $e) = (x$ **after** $pred(e))$

For the basic event structures, we need only the six simple axioms listed in table 5. In order to constrain data flow and give a semantics to the read-frame clause in message automata we will modify the basic event structures by replacing the **when** and **after** primitives by new primitives functions **Init**, **Trans**, **Choose**, and **Send** that determine the initial state, next state, the value chosen for internal actions, and the messages sent. We replace our

basic axiom 6 by a new axiom about **Trans**, **Choose**, and **Send**. Then we can reintroduce **when** and **after** as definitional extensions and prove the old axiom 6 as a theorem. Before we introduce this modification, we can present a (slightly simplified version of) the semantics of the clauses of message automata, except for the read-frame.

B.2 Semantics of Message Automata

The semantics of a message automaton M is the set of event structures es that are *consistent* with it, so the semantics can be defined by a relation $Consistent(es, M)$. A message automaton is a finite set of clauses, and we define the semantics so that an event structure is consistent with an automaton if and only if it is consistent with each of its clauses. This reduces the definition of the semantics to a base case for each clause scheme, and gives use the rule that

$$Consistent(es, A \oplus B) \Leftrightarrow Consistent(es, A) \wedge Consistent(es, B)$$

The semantics of a clause C is given by a formula, $\Psi(C)$, in event logic, that describes how the clause C constrains the observable history es of the system. The relation $Consistent(es, C)$ is then defined by

$$Consistent(es, C) \Leftrightarrow (es \models \Psi(C))$$

We give a simplified version of the semantics in tables 6 & 7. The simplifications are that, since we have omitted the type declarations from the syntax, we suppress the parts of the constraints relating to the type declarations of the state variables and action values. We also treat all the state variables as *discrete* variables; in the full theory we also allow state variables to be functions of time, so that we can reason about clocks and real-time processes. Also, the full syntax for the precondition clause allows us to specify a finite probability space (like $[0, 1, 2]$ with weights $[1/3, 1/6, 1/2]$) from which the value of a internal action may be *randomly* chosen. The semantics of this is in terms of a theory of *independent random processes*, a topic for another paper, so we omit that part of the semantics of the precondition clause.

B.3 Extended Event Structures (Semantics of Read-Frame)

Secure agents will have to store secrets in their state, and unless there is a way to restrict read access to state variables, security properties cannot be realized. In our computation system, this restriction is provided by the

Table 6: Semantics of Message Automaton Active Clauses

(notation) $\forall e@i.\ P[e] \equiv \forall e : E.\ (loc(e) = i\ \Rightarrow\ P[e])$
state when $e \equiv \lambda x.\ (x$ **when** $e)$
$msgs(e, l) \equiv [\ \langle tag(e'), val(e')\rangle \mid sender(e') = e \wedge link(e') = l\ ]$

$\Psi($**at location** i **initially** x = v$) \equiv$
$\forall e@i.\ first(e)\ \Rightarrow\ x$ **when** $e = v$

$\Psi($**effect of** k **is** x:=f(state,val)$) \equiv$
$\forall e : E.kind(e) = k\ \Rightarrow\ x$ **after** $e = f($**state when** $e, val(e))$

$\Psi($k **sends on link** l : f(state,v)$) \equiv$
$\forall e : E.\ kind(e) = k\ \Rightarrow\ msgs(e, l) = f($**state when** $e, val(e))$

$\Psi($**precondition** internal(i,a) **is** P(state)$) \equiv$
$(\forall e : E.\ kind(e) = internal(i, a)\ \Rightarrow\ P($**state when** $e)) \wedge$
$(\forall e@i.\ \exists e' \geq e.\ kind(e') = internal(i, a)\ \vee\ \neg P($**state after** $e'))$

read-frame clause, **only** ks **read** x **at location** i. The intended meaning of this clause is that for an action whose kind is not in the list ks the value of x has no effect on anything the action does. Actions only update state variables, send messages, and, if internal, choose a value, so we need a way to say that these three activities are unaffected by the value of state variable x. We call two states that agree on all variables but x, equal mod x.

$$s_1 = s_2\ \textbf{mod}\ x \Leftrightarrow \forall z : Id.\ z \neq x\ \Rightarrow\ s_1(z) = s_2(z)$$

If we can define a relation $Same(k, s_1, s_2)$ that means that whatever an action of kind k does in state s_1 it would also do in state s_2, then the semantics $\Psi($**only** ks **read** x **at location** $i)$ of the read frame clause can be

$$\forall k : Kind.\ (k \in ks)\ \vee\ (\forall s_1, s_2.(s_1 = s_2\ \textbf{mod}\ x\ \Rightarrow\ Same(k, s_1, s_2)))$$

In order to define such a relation $Same(k, s_1, s_2)$ we add the signatures in table 8 to make the functions by which the programs compute their next state and messages available in the event structure.

Operator **Init** gives the initial state at each location and **Trans** gives the transition function. We take **when** and **after** out of the set of primitives and instead define them so that for an event e of kind k and value v,

$$\textbf{state after}\ e = \textbf{Trans}(k, v, \textbf{state when}\ e)$$

and

$$\textbf{first}(e)\ \Rightarrow\ \textbf{state when}\ e = \textbf{Init}(loc(e))$$

Table 7: Semantics of Message Automaton Frame Clauses (except read-frame)

$\Psi($**only ks affect** x **at location** i $) \equiv$
$\forall e@i.\ kind(e) \in ks\ \vee\ x$ **after** $e = x$ **when** e

$\Psi($**only ks send on link** l **with tag** tg$) \equiv$
$\forall e : E.\ kind(e) = rcv(l, tg)\ \Rightarrow\ kind(sender(e)) \in ks$

$\Psi(k$ **affects only** xs **at location** $i) \equiv$
$\forall e@i.\ kind(e) = k\ \Rightarrow\ \forall x : Id.\ x \in xs\ \vee\ x$ **after** $e = x$ **when** e

$\Psi(k$ **sends only on links** ls$) \equiv$
$\forall e.\ (isrcv(e)\ \wedge\ kind(sender(e)) = k)\ \Rightarrow\ link(e) \in ls$

Table 8: **Signatures for Extended Events Structures**

Events with Transition Function
Init: $i : Loc \to$ **state**(i)
Trans: $k : Kind \to$ **vtyp**$(k) \to$ **state**$(loc(k)) \to$ **state**$(loc(k))$
Send: $k : Kind \to$ **vtyp**$(k) \to$ **state**$(loc(k)) \to Msg\ List$
Choose: $i : Loc \to a : Act \to \mathbb{N} \to$ **state**$(i) \to$
(**vtyp**$(internal(i, a)) + Unit$)

Two additional axioms relate the values of events to the **Send** and **Choose** operators.

$$\begin{aligned}
&\forall e : E. \quad isrcv(e)\ \Rightarrow \\
&\qquad \langle link(e), tag(e), val(e)\rangle \in \\
&\qquad\quad \mathbf{Send}(kind(sender(e)), val(sender(e)), \mathbf{state\ when}\ sender(e)) \\
&\forall e : E. \quad kind(e) = internal(i, a)\ \Rightarrow \\
&\qquad \exists n : \mathbb{N}.\ val(e) = outl(\mathbf{Choose}(i, a, n, , \mathbf{state\ when}\ sender(e)))
\end{aligned}$$

Thus, the expression **Send**$(k, v,$ **state when** $e)$ gives the messages that an event e of kind k and value v sends, and for an event of kind $internal(i, a)$, the value $val(e)$ is $outl($**Choose**$(i, a, n,$ **state when** $e))$ for some natural number n. This allows non-determinism in the choice of values [4], but

[4]It also allows the values to be chosen *randomly* because the parameter n can be used as an index in a random oracle.

the non-determinism is a function of a hidden number parameter, and (by lemma 1) numbers and atoms are independent, so this kind of nondeterminism does not allow agents to guess atoms.

The resulting event structures have the property that every agent updates its state and sends messages *deterministically* as a function of its current state, and the kind and value of the current event. The only place that non-determinism or randomness can enter into the behavior of an agent is in the choice of values for internal actions, but this still allows enough scope to realize randomized algorithms; it only restricts non-determinism enough to prevent it from being used to guess atoms.

B.3.1 Semantics of Read-Frame

Recall the semantics of the read-frame clause

$$\begin{aligned}&\Psi(\textbf{only } ks \textbf{ read } x \textbf{ at location } i) = \\ &\forall k : Kind.\ (k \in ks)\ \vee \\ &(\forall s_1, s_2 : \textbf{state}(i).(s_1 = s_2\ \textbf{mod}\ x\ \Rightarrow\ Same(k, s_1, s_2)))\end{aligned}$$

We can now define the relation $Same(k, s_1, s_2)$ to be

$$\begin{aligned}\forall v : vtyp(k).\ &\textbf{Trans}(k, v, s_1) = \textbf{Trans}(k, v, s_2) \wedge \\ &\textbf{Send}(k, v, s_1) = \textbf{Send}(k, v, s_2) \wedge \\ k = internal(i, a)\ \Rightarrow\ &\forall n.\ \textbf{Choose}(i, a, n, s_1) = \textbf{Choose}(i, a, n, s_2)\end{aligned}$$

B.4 Basic Lemma on Dataflow of Atoms

Trans, **Init**, **Send**, and **Choose** abstractly define the *program* at each agent i, so we can define when the program at i is independent of atom a, which we write as $(i \| a)$. Suppressing the type parameters in the independence propositions for readability, this is defined by

$$\begin{aligned}(i\|a) = (\textbf{Init}(i)\|a) \wedge (\textbf{Choose}(i)\|a) \wedge \\ \forall k : Kind.\ loc(k) = i\ \Rightarrow\ (\textbf{Trans}(k)\|a) \wedge (\textbf{Send}(k)\|a)\end{aligned}$$

If $(i\|a)$ then the agent at location i does not initially "have" atom a. The fundamental lemma asserts

> If an agent did not have an atom initially and has not received the atom then it does not have the atom.

More formally, for any event e, if the program at the location of e is independent of a, and if the values of all receive events locally before e are independent of a, then the **state when** e is independent of a.

Lemma 7 (Atom Dataflow Lemma) *For all* $e : E$ *and* $a : Atom$
$(loc(e)\|a) \wedge (\forall e' <_{loc} e.\ isrcv(e') \Rightarrow (val(e')\|a)) \Rightarrow (\textbf{state when } e\|a)$

Proof The proof is by induction on the well-founded relation $<_{loc}$. The base case is when $first(e)$, in which case **state when** e is $\textbf{Init}(loc(e))$ which is independent of a by assumption. In the induction step, **state when** $e = \textbf{Trans}(kind(pred(e)), val(pred(e)), \textbf{state when } pred(e))$ so we use the assumptions and the induction hypothesis and the independentApplication rule to reduce this case to proving that $val(pred(e))$ and a are independent. If $pred(e)$ is a receive event, then this follows from the assumptions, but if $pred(e)$ is an internal event, then

$$val(pred(e)) = \textbf{Choose}(loc(e), x, n, \textbf{state when } pred(e))$$

for some number n, and identifier x. We again use the independentApplication rule to show that since all of the pieces are independent of a by induction or by assumption or by lemma 1, so is $\textbf{Choose}(loc(e), x, n, \textbf{state when } pred(e))$. (The complete formal proof has been carried out in Nuprl.) □

C Acknowledgments

We would like to thank: Stuart F. Allen for many hours of discussions about the rules for atoms and independence and pointing out errors in our early versions of the theory; Robbert van Renesse for the virtual server idea; David Guaspari for editing and reorganizing several drafts of this report; Bob Constable for enthusiastic support for this work; and the IAI (Information Assurance Institute) for sponsoring this research.

References

[1] S. F. Allen, *An abstract semantics for atoms in nuprl*, 2006.

[2] R. L. Constable, *The structure of nuprl's type theory*, June 18 1997.

[3] D. Dolev and A. Yao, *On the security of public key protocols*, IEEE Transactions on Information Theory, 29 (1983), pp. 198–208.

[4] M. J. Gabbay and A. M. Pitts, *A new approach to abstract syntax with variable binding*, Formal Aspects of Computing, 13 (2002), pp. 341–363.

[5] J. K. Millen and H. Ruess, *Protocol-independent secrecy*, in IEEE Symposium on Security and Privacy, 2000, pp. 110–209.

[6] Paulson, *Proving security protocols correct*, in LICS: IEEE Symposium on Logic in Computer Science, 1999.

[7] L. C. Paulson, *The inductive approach to verifying cryptographic protocols*, Journal of Computer Security, 6 (1998), pp. 85–128.

[8] F. J. Thayer, J. C. Herzog, and J. D. Guttman, *Strand spaces: Proving security protocols correct*, Journal of Computer Security, 7 (1999).