

Supporting security monitor-aware development

Dries Vanoverberghe, Frank Piessens
{Dries.Vanoverberghe, Frank.Piessens}@cs.kuleuven.be

Abstract

With the emergence of support for third-party applications on mobile devices such as cell phones and PDA's, support for setting application security policies is also built into these devices. While this can significantly increase security for end-users, it also significantly complicates the task of building useful and reliable applications for these devices. Different devices will set different policies, and violations of the policy will lead to security exceptions or even immediate abortion of the application potentially leaving it in an inconsistent state.

This paper addresses this issue in the context of application security policies specified by means of security automata, and enforced by means of run-time monitoring. We propose a language element, the check block, that developers can use to make their applications more security monitor-aware. At run-time, a check block will query the security policy enforced by the monitor to make sure that the body of the block will not lead to policy-violations. At compile time, a static check ensures that the generated run-time check is adequate. We present a formalization of the static and dynamic semantics of the check block, and we outline how it can be implemented on top of C# or Java.

1. Introduction

Mobile phones and PDA's have evolved over the past years to general purpose computation platforms. Many of these devices support downloading of third-party applications. However, this support for applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications can lead to denial of service, financial damage, leaking of confidential information on the device and so forth.

Current devices already provide some countermeasures against these threats but these are inadequate. The research community has developed a variety of more flexible countermeasures for addressing the threat of untrusted mobile code. These countermeasures are typically based on run-time monitoring [6, 2, 7], static analysis [9], or a combina-

tion of both [13, 8, 11].

In this paper, we limit our attention to systems based on run-time monitoring, where the policy is specified by means of a security automaton [10]. These systems monitor security-relevant events (e.g. system calls or platform API method calls) that an application generates, and abort the application as soon as the sequence of security-relevant events is not accepted by the security automaton. Several such systems have been designed and prototyped [6, 2, 7], and security automata have been shown to express exactly the policies enforceable by run-time monitoring [10]. Run-time monitoring of compliance with a security automaton is a very promising technique for addressing the security needs of mobile devices.

However, given the flexibility and expressive power of security automata, it is to be expected that different users will set and enforce different policies. Some users will be more concerned about privacy, and set a policy that limits communication possibilities after an application has read personal information. Other users will be more concerned about cost, and limit for example the number of text messages that an application can send.

The key question we address in this paper is: how should developers of bona fide applications deal with this situation? How should one design and build an application so that it can provide its functionality in an optimal way in a wide variety of deployment contexts that may enforce different security policies on the application? A generic answer to this question is that the application should adhere to the principle of least privilege, and hence be compliant to as many policies as possible. However, this is not sufficient, for two reasons:

First, even least-privilege implementations will sometimes lead to policy violations, and hence to abortion of the program. This leads to situations where the program has partly fulfilled its functionality, and this can be undesirable, for instance because the program's persistent state is left in an inconsistent state.

Second, depending on the policy set by the user of the device, the application might want to realize its functionality in different ways. For instance, one user might forbid WiFi internet access in his policy (because it is relatively easy to

eavesdrop on), while another user might forbid sending of text messages (because this costs money). An application should be able to adapt the communication means used to the policy active on the device.

Our solution is based on a new language element, the check block, that developers can use to interact with the security policy active on the device. In particular, this primitive enables a developer to test dynamically whether a given block of code (potentially generating a long sequence of security-relevant events) would pass the security policy. This enables the developer to deal with the two issues enumerated above.

The rest of this paper is structured as follows: in Section 2 we elaborate on the problem statement, and give a high-level overview of our solution. Section 3 defines the semantics of our new language primitive more formally, and states a soundness theorem. Finally, Section 4 discusses related work and Section 5 concludes the paper.

2. Overview

2.1. Background

2.1.1. Policies assumptions

The policies in our approach are a particular instance of security automata [10] but we make a number of simplifying assumptions. We present an automaton as an instance of a class *Policy* (for instance Figure 1). The state of the automaton is represented by fields. For each security-relevant event, the class *Policy* contains a check method to check if a transition is allowed, and an update method to update the state of the automaton.

First of all, our approach assumes that a fixed set of security-relevant events is defined.

To simplify the presentation of the security policy, we assume that events take at most one argument, and that the argument is an object reference. Moreover, object references may only be used in equality tests in the policy.

Furthermore, we split events in two groups: normal events and resource handle creators (or shorter: resource creators). Normal events must not have a return value. Resource creators model the method calls that return a new resource handle (for instance a handle to a file).

Finally, we assume that one security-relevant event does not trigger another security-relevant event. Therefore, we can safely perform the state update of the automaton after the execution of the event. For resource creators, this is convenient because the return value of the event can then be used to update the state (for instance to track a set of open handles). Correspondingly, we add a parameter for the return value to the update method.

2.1.2. Example policies

Here, we introduce two example policies that will be used in the paper.

Because sending text messages can be costly, a user may limit the number of messages that can be sent by an application. The example policy in Figure 1 enforces a maximum of three messages. The event *Send* models the sending of a text message (e.g. through the SMS system).

```
class Policy {
    int count = 0;

    public void CheckSend(){
        if(count >= 3) abort;
    }
    public void UpdateSend(){
        count++;
    }
}
```

Figure 1. Maximum three text messages.

Our second example policy implements a privacy measure by requiring that all network connections are closed at the moment of reading confidential information and that no connections may be opened afterwards. Figure 2 encodes this policy. In the context of this example, the security-related events are the calls of the *Open*, *Close*, and *Read* platform APIs. Note the return value of the resource creator *Open* can be used to update the security policy.

```
class Policy {
    bool hasRead = false;
    List conn = new List();

    public void CheckOpen(){
        if(hasRead) abort;
    }
    public void UpdateOpen(Connection retval){
        conn.Add(retval);
    }
    public void CheckClose(Connection c){}
    public void UpdateClose(Connection c){
        conn.Remove(c);
    }
    public void CheckRead(){
        if(!conn.IsEmpty()) abort;
    }
    public void UpdateRead(){
        hasRead = true;
    }
}
```

Figure 2. No communication after read.

2.2. Problem statement

We illustrate the problem addressed by our approach by means of an example. Consider an application that sends a message to all contacts having their birthday. Figure 3 shows a typical implementation of this idea.

```
foreach(Contact contact in ContactFolder.contacts){
  if(contact.HasBirthdayToday()){
    SmsMessage msg = new SmsMessage(...);
    msg.Send();
  }
}
```

Figure 3. Example program.

If this application is deployed on a device that enforces the "max three text messages" policy on all downloaded applications, it will run fine most days, but will be aborted by the run-time monitor on a day where four or more contacts have their birthday.

When the application aborts, three messages have already been sent, so the user cannot simply re-run the application while relaxing the security policy. More generally speaking, the security violation might abort the program while the state of the program is inconsistent. Because security-relevant events cannot always be rolled back, it is impossible to restore the state.

On the other hand, the developer had no intention to harm the end user. Without a means to query the security policy in advance, the developer cannot really fix this problem or offer alternatives. Even if policy violations are signaled using exceptions, it is often too late to offer an alternative when handling them.

2.3. Solution overview

The core problem is that the code can be interrupted by the security policy at any security-related event. To solve this, developers need to be able to query the security policy in advance. We augment the programming language with *check blocks*. A check block has a header and a body. The header specifies a sequence of events. At run time, our approach checks if the sequence of events is allowed by the policy. A static check ensures that the body performs exactly the specified sequence of events. If the run-time test succeeds, the execution of the body will not be interrupted by the policy.

On the other hand, now that the developer can query the policy in advance, he can provide alternative scenarios. Therefore, our construction also takes an else block.

When we return to the example program, the developer can restructure his code as follows. First he calculates how many contacts are having their birthday. Then he specifies

how many messages he will send. Finally, he sends the messages inside the body of a check block. The developer can now test beforehand if the policy would be violated, so he can write an alternative. He can, for instance, inform the user of the situation and ask him to select the most important candidates one by one.

The result for this application is shown in Figure 4. A check block takes a sequence of events as an argument. The sequence $Send() \wedge N$ (read as *send power n*) means that the event send is repeated N times. If the policy allows this sequence, all messages are sent without user interaction and the program will not be interrupted by the policy. Otherwise, the execution continues with the else block.

```
List<Contact> targets = newList<Contact>();
foreach(Contact contact in ContactFolder.contacts){
  if(contact.HasBirthdayToday()) targets.Add(contact);
}
int N = targets.Count;
check(Send() ^ N){
  for(int i = 0; i < N; i++){
    SmsMessage msg = new SmsMessage(...);
    msg.Send();
  }
} else {
  Console.WriteLine("I cannot send "+N+" messages.");
  while(true){
    Console.WriteLine("Select one person or abort.");
    int index = ...;
    check(Send()){
      SmsMessage msg = new SmsMessage(...);
      msg.Send();
    } else { break; }
  }
}
```

Figure 4. Example program illustrating our approach using check block.

In the rest of our paper, we deal with two important challenges. If we want to ensure that the security policy will never interrupt the body of our check construction, we must verify that it actually performs the specified sequence.

Another challenge for our approach is to handle return values of security-relevant events. For instance, consider again the policy in Figure 2. The return value of the event *Open* is used to update the state of the policy. When we check a sequence of events before the event *Open* has actually been performed, we do not have the actual value that is returned by the event.

3. Semantics

In this section, we formalize the dynamic and static semantics of the check block. First, we define a syntax extension relative to a Java-like programming language. Next, we

```

Stmt ::= ...
      | check (EventSeq) CheckBlock else ElseBlock
EventSeq ::=  $\epsilon$ 
          | MethodName ( Var? )
          | EventSeq, EventSeq
          | EventSeq  $\wedge$  IntegerExpr
          | Type Var = ResourceCreator in EventSeq

```

Figure 5. Syntax of event sequences and check block

explain the run-time semantics of the check block and how it can be implemented. Then, we describe how we verify that the code block complies with the specified sequence of events. Finally, we state a soundness theorem and we sketch a proof.

3.1. Syntax

Figure 5 shows the syntax of our extension. It introduces one extra statement called the check block. A check block has a sequence of events and two blocks of code. Both *CheckBlock* and *ElseBlock* denote normal Java blocks.

Figure 5 also defines event sequences. The first case is the empty sequence. Next, we have an individual event. It has a method name and possibly one variable. The next case is sequential composition. Semantically, this means that the first sequence will be executed before the second sequence. The fourth case is finite repetition. The number of repetitions is an integer expression whose value is possibly only known at runtime. Finally, we have a binder. This construct introduces a variable that can be used in the sequence followed by it. The variable must be initialized by calling a particular event that must be a resource creator.

3.2. Dynamic check

At the entrance of the check block, our approach dynamically checks that a sequence of events is allowed by the policy. In this section, we explain how the check is performed and we give an algorithm to perform it.

The general idea of the dynamic check is to generate a block of code that can be inlined at the entrance of the check block. Before the check is reached, all parameters used in the event sequences must have been initialised. When the inlined block is executed, it generates a sequence of dynamic checks against a clone of the policy. When one of the events fails, the entire check fails and the execution continues with the body of the else block. Otherwise, the body of the block executes. In both cases the clone of the policy is discarded. The state of the real policy is updated by the actual execution of the events.

Figure 6 gives a more formal definition. **GenerateCheck** clones the policy and uses it for checking the sequence. **CheckSeq** takes the cloned policy as input and gen-

```

GenerateCheck :: Policy  $\rightarrow$  EventSeq  $\rightarrow$  Block
GenerateCheck(policy, sequence) =
{
  Policy clone = policy.Clone();
  CheckSeq(clone, sequence)
}
CheckSeq :: Policy  $\rightarrow$  EventSeq  $\rightarrow$  StatementList
CheckSeq(policy,  $\epsilon$ ) =
  nop;
CheckSeq(policy, MethodName(Var?)) =
  policy.CheckMethodName(Var?);
  policy.UpdateMethodName(Var?);
CheckSeq(policy, EventSeq1, EventSeq2) =
  CheckSeq(policy, EventSeq1)
  CheckSeq(policy, EventSeq2)
CheckSeq(policy, EventSeq  $\wedge$  N) =
  for(int i = 0; i < N; i++){
    CheckSeq(policy, EventSeq)
  }
CheckSeq(policy, Type Var = RC in EventSeq) =
  policy.CheckRC(Rc);
  Object Var = new Object();
  policy.UpdateRC(Rc, Var);
  CheckSeq(policy, EventSeq)

```

Figure 6. Algorithm for dynamic check

erates a list of statements to check the sequence of events in the check block's header.

For empty sequences, no checks need to be generated. A single event is translated to the corresponding check and update call on the *Policy* class. For sequential composition, we first generate the code for the left sequence and then for the right one. For repetition, we simply place the code to check the sequence in a loop. Finally, for a binder, we generate the corresponding check call to the policy. For the update call, the return value of the resource creator is unknown, so we use a new object reference instead. Then we translate the rest of the event sequence.

The return value of a resource might be used by the security policy in equality tests. Because this value is a new object reference, none of the equality tests will succeed. If we want our approach to be sound, we must ensure that the actual return value has the same behaviour. In our approach, resource creators are required to return new object references.

3.3. Static verification

In this section, we explain how we use static verification to ensure that the specified sequence is equal to the sequence executed by the actual code. Because our process will need to take data flow and control flow into account, we have chosen a verification condition based approach. Typically, applications are annotated with a specification and the implementation is verified against the specification. For this purpose, we use the existing tool Spec# [1].

```

BuildSpec :: EventSequence- > ClassSpec
BuildSpec( $\epsilon$ ) = New
BuildSpec(MethodName(Var?)) =
  let em = New
  in AddEv(SetLength(em, 1), MethodName(Var?))
BuildSpec(EventSeq1, EventSeq2) =
  let em1 = BuildSpec(EventSeq1)
  em2 = BuildSpec(EventSeq2)
  in let em2s = Shift(em2, Length(em1))
  len = Length(em1) + Length(em2)
  in SetLength(Combine(em1, em2s), len)
BuildSpec(EventSeq  $\wedge$  N) =
  let em = BuildSpec(EventSeq)
  in let em2 = SetLength(em, N * Length(em))
  in AddIntVar(Mod(em2, Length(em)), N)
BuildSpec(Type Var = RC in EventSeq) =
  let em = BuildSpec(RC, EventSeq)
  in AddResourceVar(em, Var)

```

Figure 7. Algorithm to translate event sequences.

The intuition behind our verification process is to translate the event sequence into a class specification. This class specification has methods for all security-relevant events, and the pre and post conditions ensure that these methods can only be called in a sequence corresponding to the event sequence. In the body of the check block, the calls of an event are replaced by calls to the specification class. If this block of code verifies against the class specification we generated, we can be sure it generates security-relevant events conforming to the event sequence used to build the generated class specification.

3.3.1. Translating event sequences

The translation algorithm gets a sequence of events as input and delivers a class specification as output. Each such specification has a specification variable `eventcounter`. This counter determines how many events have already occurred. When an object of this class is created, the counter is initialized to zero. For each security-relevant event, a method is introduced. Each such method gets a postcondition saying that the event counter increases by one. Finally, we add an extra method `Terminate` with a precondition stating that the event counter must equal the length of the sequence. The length is in general an expression because some information might not be statically known.

Figure 7 shows the algorithm to build the specification. Because it would take too much space to formally specify the entire algorithm, we describe it in function of a number of basic operations. Then we informally explain the basic operations.

To translate the empty sequence, we simply build a new class specification (Operation **New**). For such specification,

each security-relevant event gets a precondition false making it impossible to call the method. The length of this sequence is zero.

To translate a single event, we also build a new class specification (Operation **New**). To accept one event, we set the length expression to one (Operation **SetLength**). Next, we make sure that it is possible to call the event when the counter is zero. We do this by relaxing the precondition by performing a logical or between the original value (false) and an equality test between the eventcounter and zero (Operation **AddEv**).

For example, consider a security policy with two security-relevant events: `Send` and `Read`. If we would have to translate a sequence that performs `Send` for one single time, we can see the result in Figure 8.

```

public class EventMachine {
  public EventMachine()
    ensures eventcounter == 0;
  public void Terminate()
    requires eventcounter == 1;
  public int eventcounter;
  public void Send()
    requires false | eventcounter == 0;
    ensures eventcounter == old(eventcounter) + 1;
  public void Read()
    requires false;
    ensures eventcounter == old(eventcounter) + 1;
}

```

Figure 8. Example class specification for `Send()`.

Next, we discuss how the sequential composition can be translated. What we actually want is that the first sequence is accepted when the event counter is between zero and the length of the first event counter and that the second sequence starts at the length of the first sequence and ends at the sum of both lengths. We can do this by replacing the event counter of the second specification by the event counter minus the length of the first specification (Operation **Shift**). Then we can recombine the specification by performing a logical or between the requires clauses of the events of both (Operation **Combine**). Finally, we have to correct the length expression (Operation **SetLength**).

Consider again the policy with two security-relevant events: `Send` and `Read`. Figure 9 shows the resulting preconditions for the sequence `Send()`, `Read()`.

```

public void Send()
  requires false | eventcounter == 0;
public void Read()
  requires false | eventcounter - 1 == 0;

```

Figure 9. Example class specification for `Send()`, `Read()`.

The next case is the translation of the repetition of a sequence. First we generate the specifications for the sequence. Then we substitute the counter by a counter modulo the length of the sequence (Operation **Mod**) and multiply the length expression with the variable that specifies how many times we have to execute the sequence. (Operation **SetLength**). To have access to the variable, we have to add it as a local variable. The variable must then be initialized using the constructor (Operation **AddIntVar**). Figure 10 shows an example.

```

public class EventMachine {
  public EventMachine(int N)
    ensures eventcounter == 0;
    ensures this.N == N;
  public void Terminate()
    requires eventcounter == (1 + 1) * N;
  public int eventcounter;
  public int N;
  public void Send()
    requires false | eventcounter % 2 == 0;
    ensures eventcounter == old(eventcounter) + 1;
  public void Read()
    requires false | eventcounter % 2 - 1 == 0;
    ensures eventcounter == old(eventcounter) + 1;
}

```

Figure 10. Example class specification for $(Send(), Read())^N$.

Finally, we translate the binder. To do so, we first build the specification of the sequence without considering the extra variable. Then we add the variable as a local variable. At the resource creator, we add a postcondition that the local variable is equal to the result of the event. For all events that use the variable as argument, we add a precondition that the argument must equal the local variable and a postcondition stating that the local variable is unchanged (Operation **AddResourceVar**).

Again, we illustrate this with a small example. This time we consider the events $Connection\ x = Open()$ and $Close(Connection\ x)$ as security-relevant. Figure 11 shows the impact of the binder on the specification of the methods $Open$ and $Close$.

```

public Connection x;
public Connection Open()
  ensures x == result;
...
public void Close(Connection x)
  requires this.x == x;
  ensures this.x == old(this.x);
...

```

Figure 11. Example class specification for $Connection\ x = Open()\ in\ Close(x)$.

3.3.2. Transforming the body

Now we know how to construct the class specification, we can describe its use. Basically, we make a new instance of the class at the beginning of the body of the check block. Inside the body, we replace each call of an event with a call of the corresponding method on that instance. Finally, at the exit of the check block, we add a call to the method $Terminate$. The result on a simple program that repeatedly opens and closes a connection can be seen in Figure 12.

```

int N = ...;
check((Connection x = Open() in Close(x)) ^ N){
  EventMachine em = new EventMachine();
  int i = 0;
  while(i < N) {
    Connection c = em.Open();
    em.Close(c);
    i++;
  }
  em.Terminate();
}

```

Figure 12. Transformed check block.

3.3.3. Practical issues

To verify loops, the programmer will in general need to annotate his code fragments with loop invariants. For instance, consider again the example with two security-relevant events: $Send()$ and $Read()$. In Figure 13, a small piece of code with a loop is annotated with a loop invariant. The invariant specifies that the eventcounter will always be equal to the variable i multiplied by two.

```

int N = ...;
check((Send(), Read()) ^ N){
  int i = 0;
  while(i < N) invariant eventcounter == 2 * i; {
    Send();
    Read();
    i++;
  }
}

```

Figure 13. Example code fragment with loop invariant.

In general, methods can be called inside a check block. Those methods could potentially execute security-relevant events. For simplification purposes, we only allow calls to methods that do not perform security-relevant events. To check this in a modular way, it is again the task of the programmer to specify the methods that do not perform security-relevant events (by placing an attribute on them).

Finally, we must also verify that the arguments for the events have not changed since the entrance of the block.

3.4. Soundness

When using a check block, the dynamic checks against the policy are placed at the entrance of the block. To show that our approach is sound, we need to show that the same dynamical checks are executed as when the checks would be executed together with the actual events.

More formally, at the entrance of a check block, the algorithm *CheckSeq* generates a block of code. When this code is executed, a sequence of interactions with the security policy takes place. We call this the expanded trace. On the other hand, if we just execute the code inside the check block and perform the checking together with the events, we get another sequence. This trace is called the actual trace. In order to prove that our approach is sound, we need to prove that the expanded trace is equal to the actual trace up to renaming of object identifiers.

We have not developed a full proof yet, but here is an outline of the soundness argument. First, if the code of the check block verifies, the actual trace satisfies the specifications we generated out of the event sequence. Then, we prove that the expanded trace satisfies the same specifications. Finally, we prove that the class specification only accepts one trace (up to renaming of object identifiers). Because the specification accepts both sequences, and it can only accept one sequence, the two sequences must be equal.

4. Related work

As far as we know, research on enforcement of security policies always focused on the expressiveness of the policy and the correctness of the enforcement mechanism. We are not aware of existing research to enable a developer to make reliable applications in the presence of arbitrary application security policies.

Our work inserts dynamic checks in an application and statically verifies that those checks are sufficient to enforce a security automaton. A similar combination can be found in [13, 8, 12, 3]. Our approach is more general because it is based on sequences of events.

The idea of specifying a sequence of events has already been explored in [4, 5]. In Model Carrying Code [11], a model can represent multiple sequences of events. Therefore it is more general, but also less precise.

5. Conclusion and future work

In this paper we argued that application sandboxing using run-time enforcement complicates the development of reliable applications, especially in the presence of multiple unknown policies. To alleviate this problem, we introduced a check block that specifies a sequence of events for a piece

of code, defined an algorithm to perform runtime checks against the policy for a sequence of events and we use static verification to prove that the code performs the specified sequence. In combination, this guarantees the absence of security-related interruptions inside the check block.

In the future, this work could be extended by inferring the sequence of events from a piece of code. Alternatively, less precise sequences could be used to make the approach more convenient for developers.

6. Acknowledgements

We thank Jan Smans and Bart Jacobs for their comments and feedback on the paper. Thanks to Dilian Gurov and Irem Aktug for discussing the ideas with us. Finally, we thank Folker den Braber for the motivating example.

References

- [1] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview.
- [2] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *PLDI '05*, pages 305–314, New York, NY, USA, 2005. ACM Press.
- [3] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP*, pages 311–325, 2005.
- [4] Y. Cheon and A. Perumandla. Specifying and checking method call sequences in JML. In *SERP '05*, 2005.
- [5] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
- [6] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.
- [7] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.
- [8] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .net. In *PLAS '06*, pages 7–16, New York, NY, USA, 2006. ACM Press.
- [9] G. C. Necula. Proof-carrying code. In *POPL '97*, pages 106–119, Paris, Jan. 1997.
- [10] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
- [11] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *SOSP '03*, pages 15–28, New York, NY, USA, 2003. ACM Press.
- [12] J. Smans, B. Jacobs, and F. Piessens. Static verification of code access security policy compliance of .net applications. *Journal of Object Technology*, 5(3):35–58, 2006.
- [13] D. Walker. A type system for expressive security policies. In *POPL '00*, pages 254–267, New York, NY, USA, 2000. ACM Press.