Security enforcement aware software development

Dries Vanoverberghe\textsuperscript{a}, Frank Piessens\textsuperscript{a}
\textsuperscript{a}Distrinet, Dept. Computer Science, K.U.Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium

Abstract

In the domain of security policy enforcement, the concerns of application developers are almost completely ignored. As a consequence, it is hard to develop useful and reliable applications that will function properly under a variety of policies. This paper addresses this issue for application security policies specified as security automata, and enforced through run-time monitoring. Our solution consists of three elements: the definition of an abstract interface to the policy that is being enforced, a sound construct to query that policy, and a static verification algorithm that guarantees absence of security policy violations in critical blocks of code.

Key words:
security automata, runtime enforcement, inline reference monitor, static verification

1. Introduction

In today’s networked world, code mobility is ubiquitous. Applications can be downloaded over the internet, or received as an attachment of emails. Even mobile phones and Personal Digital Assistants increasingly support the installation of third party applications from a variety of sources. 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 and so forth.

The research community has developed a variety of countermeasures for addressing the threat of untrusted mobile code. These countermeasures are typically based on run-time monitoring \cite{10,4,11}, static analysis \cite{15}, or a combination of both \cite{21,13,20}.

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 \cite{19}. These systems monitor security-relevant events 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 \cite{10,4,11}, and security automata have been shown to express exactly the policies enforceable by run-time monitoring \cite{19}. The code access security architectures present in Java and .NET are an instance of such systems \cite{10}. In short, run-time monitoring of compliance with a security automaton is a very promising technique for addressing the security problems with untrusted mobile code.

Until now, the most important factors in designing a policy enforcement system are soundness and expressivity of the policy language. The concerns of application writers are almost completely ignored. For instance, a consequence of policy enforcement is that an application can encounter security exceptions or even program aborts almost everywhere in the program. Using as little privileged operations as possible only partly mitigates this problem. It does not allow the code producer to execute alternatives when these operations fail. In short, policy enforcement significantly complicates the task of building useful and reliable applications.

An illustration of the effect that unexpected security policies can have on the reliability of applications is the fact that many applications designed for the (insecure) Windows 9x line of operating system behaved very erratically on the (more secure) Windows NT/2000/XP systems. Stronger security policies at the code consumer side forced code producers to rewrite their applications.

Some effort has been done to address the lack of involvement of the code producer as a best effort approach at the code consumer side (e.g. \cite{14}). Only recently, proactively involving the code producer has gained interest \cite{2,20}. We discuss these systems in more detail in the related work section.
We propose a new language element, the check construction, to make applications more robust in the presence of unknown security policies. The code producer can use this check construction to describe the security-relevant behavior of a block of code. Our solution makes sure that this block will never be interrupted by the security policy by checking this behavior against the policy before executing the block.

In Section 2, we first analyze existing work with a focus on the concerns of the application writer. Before we introduce the check construction in Section 5, we define an abstract interface to the runtime policy (Section 3), and we provide a way to ask the policy at runtime if certain security-relevant behavior will lead to an interruption (Section 4). Next, Section 6 and 7 discuss alternative solutions and describe the implementation status. Finally, we give an overview of related work and we present our conclusions.

2. Background

We start out this section with a motivation of our decision to focus on techniques that are applied by the code producer. Next, we identify three important dimensions that explain how different approaches relate to each other. Then we position a number of existing approaches in these dimensions, and we explain how our work contributes in this area.

2.1. Code producer, code consumer

In the domain of secure execution of untrusted code, two important actors can be identified [15]. The code producer is the actor responsible for creating an application. This application is then distributed to the code consumer. Often, the code consumer does not fully trust the code producer. Moreover, the application can potentially be modified during distribution. Nonetheless, the code consumer wishes to execute the application but he also wants to ensure this does not violate his security requirements specified in a policy.

The most common solution for this problem is to monitor the execution of the application at the site of the code consumer and abort the execution whenever the application tries to violate the policy [10,19,8,11,4]. From a security point of view, this is the best place to enforce the policy because it is as close to the code consumer as possible. On the other hand, because the code producer is absent in this process, it becomes very hard for him to make a robust application that provides its functionality in an optimal way in a wide variety of deployment contexts that may enforce different security policies on the application.

A number of approaches do involve the code producer [16,15,21,13,20]. The code producer is assumed to know the policy, and to develop his application compliant with the policy. When shipping the application, the code producer includes evidence (for instance a proof) of this compliance. The code consumer has to verify the evidence before running the application.

However, the assumption that the policy is fully known by the code producer is not always realistic. There is currently no research on how to produce policy compliant code for possibly unknown policies. To fill the void, we re-analyze existing research with a focus on the concerns of the code producer.

2.2. Dimensions

We classify existing policy enforcement research based on three dimensions: Enforcement technique, knowledge about the policy and locality of the enforcement. In the rest of this section, we discuss these dimensions.

**Technique:** First, the enforcement technique can be static or dynamic. With static techniques, the enforcement of the policy happens before the application is executed. Therefore, there is no overhead during the execution of the application. Unfortunately, complex data structures can often make it hard to gather all the information statically. On the other hand, dynamic techniques enforce the policy by executing extra checks at runtime. Because information is easily available at runtime, these techniques are typically more complex than static techniques. Finally, static and dynamic techniques are often combined to strengthen each other.

**Knowledge:** Most enforcement approaches expect that there is full knowledge of the policy. From the viewpoint of the code producer, this expectation is not always realistic. At worst, there is no information at all about the policy, making the enforcement at development time impossible. Between those extremes, it is possible to expose an interface of the policy. For the code producer, this interface could for instance include a list of events that are considered security-relevant. Because of this information, he knows when he does something which can potentially be dangerous. The actual policy must still be plugged into the interface by the code consumer.

**Locality:** Finally, each enforcement technique has a locality. Locality represents the area of the program analyzed by the technique to make sure that the policy is not violated. For instance when a technique performs checks around one single event at time, its locality is one event. We say it is highly local. On the other hand, when the technique performs checks that span multiple events, its locality is a sequence of events. Being less local makes it possible to optimize the checks for the entire sequence. Moreover, it reduces the amount of places in a program where the application can be aborted. On the other hand, techniques with a high locality are typically much less complex.

2.3. Positioning existing research

With classic inline reference monitoring [9], a program rewriter inserts security checks inside an untrusted applica-
tion. When the application is executed, these checks monitor the behavior of the application and prevent it from violating the policy. The enforcement technique is dynamic. To insert the checks, the program rewriter has full knowledge about the policy. The checks are inserted immediately before and after the execution of a security-relevant event, thus the locality is one event. In Figure 1, classic inline reference monitoring is placed at point 1.

Full static verification is a technique that is static in nature. The policy is translated into method contracts on the security-relevant events, therefore full knowledge about the policy is required. Preconditions describe in which states the policy allows the event and postconditions describe how the state of the policy is modified by the event. A general verification tool is used to prove that the untrusted application does not violate these method contracts. This analysis happens statically. Finally, the locality of this technique is a sequence of events. In Figure 1, full static verification is placed at point 2.

When an application is inlined at the side of the code producer, and it is distributed, the consumer might not trust the producer or the distribution process. Certified inlining [21,13] solves this trust issue by including enough information into the application so that the receiver can check that the application has been correctly inlined with the desired policy. The code producer adds the information by using type annotations, and the compiler verifies that the annotations are correct with respect to the code. When we look at the entire picture, there is a combination of static and dynamic enforcement. The code producer needs full knowledge of the policy. To keep the type checking efficient, the locality of the technique is a single event. In Figure 1, certified inlining is placed at point 3.

In Model-Carrying code [20], the code producer and code consumer also work together to ensure safety. In this case, the code producer distributes his application equipped with a concise high level model of its security-relevant behavior. At the side of the code consumer, an automatic verification procedure decides if the model satisfies their security policies. To make sure that the application is correct with respect to its model, the application is monitored at runtime. In Figure 1, Model-Carrying code is placed at point 4.

2.4. Contribution

When the code producer performs classic inlining, he needs full knowledge about the policy. This is not always realistic, but it is not hard to adapt the inlining to only use partial information about the policy. Instead of embedding the state of the policy, and statements to check and update this state inside the application, we isolate them in a separate policy component. The inliner inserts calls to this component and therefore only depends on its interface. This interface is fully defined by a list of security-relevant events. Just like classic inlining, this technique is dynamic and its locality is a single event. In Figure 1, this contribution corresponds to moving from point 1 to point A.

Unfortunately, the code producer cannot know if his application will be allowed by the policy of the code consumer. The high locality of inlining might even interrupt the execution in the middle of a sequence of events that must either be performed completely, or not at all. Therefore, we provide a way to test if a sequence of events will be allowed by the policy at runtime. In Figure 1, this contribution corresponds to moving from point A to point B.

Finally, we introduce a static verification method to make sure that the code producer queries the correct sequence of events for a given block of code. In Figure 1, this contribution corresponds to moving from point B to point C. Although this is the same position as Model-Carrying Code, we use static verification and dynamic checks with other goals. We discuss the relation between Model-Carrying Code and this paper in more detail in the related work section.

3. Basic security policy enforcement

This section describes how we have adapted classic inlining to make it more usable from the viewpoint of the code producer. First, we describe what it means for an event to be security-relevant in our system. Then we describe how a policy is represented in our system, and we present some example policies. Finally, we explain how we inline an untrusted application.

3.1. Security Relevant Events

When designing a runtime security enforcement mechanism, the fine-grainedness of the events that we consider
security-relevant is important. If an event is a very fine operation, the policy will be queried very often so there will be a large performance overhead. The policy will have a lot of information present to make decisions, therefore it can be very expressive. On the other hand, when the events are restricted to coarse operations, it will be less expressive but more efficient.

The execution of arbitrary byte code instructions is the most fine grained event we can monitor. This approach has been used in the past to enforce memory safety on untrusted code [8]. Very expressive policies can be enforced [19], even in the absence of type safety. On the other hand, strong optimizations are necessary to keep the performance overhead acceptable.

In our work, we intercept the invocation of method calls. This approach is less fine grained than the first one but does not suffer from the same high overhead. It is already adopted by various research projects e.g. PoET/PSLang [9] and Polymer [4].

Given the fact that events are method calls, these methods can still have a different level in the hierarchy of method calls. Low level methods are typically the last method before handing the task over to the operation system. High level method calls are close to the programming interface of the system libraries, and one high level method often uses multiple low level methods. Current research [9,4] usually focuses on low level method calls because this implicitly secures all higher level methods that use these low level calls.

From the viewpoint of the code producer, choosing low level methods is less interesting because these methods are much further away from the high level interface used in the application. To secure the application, the code producer needs some kind of mapping between the high level method calls he uses, and the low level method calls that are described in the security policy.

Because this mapping can be very hard, we use high level methods as events in the policy. More precisely, in our system, a security-relevant event is a method call from inside the untrusted application, into the trusted system libraries.

Traditional enforcement approaches consider the events that are security-relevant as something that is defined by the code consumer only. This is not always realistic: the code producer does not know what events he must be careful with. In our system, the choice of what method calls are security-relevant is attached to the system libraries. The code consumer can only mention methods marked as security-relevant in the policy. In return, the code producer knows what methods might potentially throw security exceptions.

This approach is similar to permissions in .NET Code Access Security [17] and the Java 2 Security Architecture [12]. The base class library defines a set of permissions, and certain methods need a permission. In the security policy, the user only defines the permissions that are granted to a component. He does not define the methods that need permissions.

### 3.2. Security Automata

The policies in our system are a particular instance of Security Automata [19]. A security automaton consists of a set of states and a set of transitions. Each transition is labeled with an event, and connects an input state with an output state. The meaning of such a transition is that the event can be executed when the automaton is in the input state, and that the resulting state is the output state.

Figure 2 contains a simple example of such a security automaton. The goal of this policy is to prevent sensitive information from being sent over the network. This is enforced by only allowing the event `Send()` when the event `Read()` has not been performed yet.

We use a representation of security automata similar to that of PSLang [9]. We represent the policy as an instance of a class `Policy`. This class contains a field for every state variable. For each security-relevant event, it contains a before and an after method. The before method is executed before the event and can be used to stop the application from executing it. It takes the same parameters as the original event. When the event is an instance method, the implicit “this” parameter is added. The after method is executed after the event. It takes the same parameters and implicit “this” parameter if necessary. If the event returns a value, a parameter is added to pass this value to the policy.

We use this presentation style because it eases the presentation of the novel ideas of this paper. In practice, the policy class is automatically generated from a higher level policy language [1] to ease policy development.

In the rest of this section, we discuss some example policies. Because sending text messages on a mobile device can be costly, a user may limit the number of messages that can be sent by an application. The example policy in Figure 3 enforces a maximum of three messages. The event `Send` models the sending of a text message (e.g. through the SMS system).

The 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 afterward. Figure 4 encodes this policy. In the context of this example, the security-relevant events are the calls of the `Open`, `Close`, and `Read` platform APIs. Note the return value of `Open` can be used to update the security policy.
class Policy {
    int count = 0;
    public void BeforeSend()
    {
        if (count >= 3) abort;
    }
    public void AfterSend()
    {
        count++;
    }
}

Fig. 3. Maximum three text messages.

class Policy {
    bool hasRead = false;
    List conn = new List();
    public void BeforeOpen()
    {
        if (hasRead) abort;
    }
    public void AfterOpen(Connection retval)
    {
        conn.Add(retval);
    }
    public void BeforeClose(Connection c)
    {
        conn.Remove(c);
    }
    public void BeforeRead()
    {
        if (!conn.IsEmpty()) abort;
    }
    public void AfterRead()
    {
        hasRead = true;
    }
}

Fig. 4. No communication after read.

3.3. Inlining

With classic inline reference monitoring [9], a program rewriter inserts security checks before and after the execution of a security-relevant event. To generate the security checks, full knowledge about the policy is required.

In the previous section, we represented our policies in an executable way. This representation can be further split into an interface and an implementation. In Section 3.1 we explained why we consider the security-relevant events as a fixed set of events with respect to the system libraries. Because the interface contains a before and after method for each security-relevant event, it only depends on the set of security-relevant events. The actual policy is then the implementation of this interface.

In our approach, the inliner only inserts calls to the before and after methods of the interface. The inlined application only depends on this interface and the actual policy must be plugged in before executing the application.

4. Querying the policy

4.1. Problem statement and general solution

Even when the actual policy is unknown, the code producer now knows what methods potentially result in security exceptions. Unfortunately, the only way to find out if an application is allowed to execute an event, is to ship it to the customer and see if there are complaints.

With inlining, a security check is executed before and after the execution of any security-relevant event, thus the security policy is enforced one single event at a time. This high locality makes it impossible to either execute an entire sequence of security-relevant events, or none at all: When the execution is interrupted, some events might already have been performed. In general, it is impossible to roll back security-relevant operations (e.g. reading a secret, or sending something over the network). Nor is it possible to perform the remaining operations until the next consistent state. This denies the code producer from executing multiple security-relevant events in an atomic way.

We illustrate this by means of an example. Consider an application that sends a message to all contacts having their birthday. Figure 5 shows a typical implementation of this idea. If this application is deployed on a device that enforces the “max three text messages” policy (Figure 3) on all downloaded applications, it will run fine most days, but will be interrupted by the run-time monitor on a day where four or more contacts have their birthday. When the application is interrupted, three messages have already been sent, and it is impossible to un-send these messages. On the other hand, the application cannot complete the functionality that was intended by the code producer.

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

Fig. 5. Example program.

To solve this, we provide the code producer with a way to query the security policy at runtime. This query construct takes a sequence of events as input, representing the security-relevant behavior of a part of the application. It returns true when the sequence of events does not violate the policy, and false when it does.

For instance, to solve the problem with the application in Figure 5, the code producer first calculates the amount of messages that will be sent. Then he asks the policy whether he is allowed to perform the event send for each message 1. If this query succeeds, the code producer sends the messages. Otherwise, he can provide an alternative solution, for instance allow the user to select the most important contacts.

4.2. Testing sequences of events

This query construct takes a sequence of events as input, and returns true when the sequence of events does not vio-

---

1 The syntax for event sequences will be explained in Section 4.6.
BeforeRead() 

Policy

BeforeSend()

AfterSend()

BeforeRead()

AfterRead()

Send()

Read()

Policy

BeforeSend()

AfterSend()

BeforeRead()

AfterRead()

query(Send(),Read(),...)

Fig. 8. Example query

(iii) Having an expressive event sequence language is important. Section 4.5 shows how we extended the simple notion of event sequence to use return values of events inside a sequence and Section 4.6 adds iteration.

4.3. State changes

When we execute a sequence of security methods from the interface of the policy, the state of the policy changes. This means that after a query, we have to execute this sequence of events, and it is no longer possible to execute an alternative after using the query method. This is not acceptable, instead we want the query to execute in an effectless way.

We solve this problem by creating a temporary version of the state of the policy. For this, we execute an extra method in the interface of the policy to clone the state. During the query, all security checks are executed on this temporary version. When the entire query has been executed and the result is available, this version can be safely discarded. The real policy is updated by the cloned security checks before and after each event.

4.4. Return values

An important challenge for the query is to handle return values of security-relevant events. Inside a sequence of events, some of the events may return a value. This value might be used in the policy to update the state and thereby it influences the query for the rest of the sequence. Because the query is not actually executing the security-relevant events, the actual value returned by those events is not known.

For example, consider the “No communication after read” policy in Figure 4. The event Open returns a new connection object. When this event is used in a query, this connection is not available to call the after method on the

List<Contact> targets = new List<Contact>();
foreach (Contact contact in ContactFolder.contacts) {
    if (contact.HasBirthdayToday()) targets.Add(contact);
}

int N = targets.Count;
if (query(for N times Send())) {
    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 = ...
    if (query(Send())) {
        SmsMessage msg = new SmsMessage(....);
        msg.Send();
    } else {
        break;
    }
}

Fig. 6. Example program illustrating our approach using the query construct.

late the policy, and false when it does. The meaning of sequences of events can be explained using two basic building blocks: single events and sequential composition. A single event corresponds to the execution of the events described in Section 3.1. It takes an argument for each parameter of the security-relevant event. These arguments can be fixed values as well as variables that are within scope at the location of the query. Sequential composition takes an existing sequence of events, and adds a single event after it. Figure 7 illustrates the syntax of query and event sequences.

Fig. 7. Syntax of event sequences

In the enforcement mechanism, only the interface of the policy is available. To test this sequence of events, the only primitives that are available are the before and after methods of each event in this interface. In order to check whether a given sequence of events is allowed, the query must call exactly the same sequence of methods on the interface of the policy as the sequence of events would after we inlined calls to the interface. For example, in Figure 8, the left hand side represents the sequence of calls made with inlining, while the right hand side represents the sequence of calls made using the query construct.

While querying the policy is a simple concept, implementing a sound query mechanism is non-trivial. For instance:

(i) While querying, the state of the policy changes. Section 4.3 discusses how our system keeps the query effectless.

(ii) Certain events have a return value. This return value is not present at the time of a query (see Section 4.4).
policy. In Figure 9, we illustrate this with the question marks.

![Fig. 9. Example query with return](image)

The easiest solution is excluding events with return values from sequences of events, but this severely restricts the queries we can do.

The other extreme is to execute the after method for any possible return value of the security-relevant event. If the after method does not fail for any return value, then it will not fail if we execute it with the actual return value, thus this approach is also sound. Besides from having a bad performance or being impossible when there is an infinite set of return values, this also causes the query to be less precise, for instance when the query works for the actual return value, but it fails on another possible value.

In this work, we apply restrictions on the policy and the semantics of events that return a value in such a way that we only have to perform the after method with one well-chosen value, and that the result of this test will be the same as doing the test for all possible values.

First, we change the semantics of events that return a value: Instead of returning any possible value, they can only return a new object reference. We primarily use these methods to model events that create a handle to a resource. Therefore we call them resource handle creator (or shorter, methods). For instance, method calls and field lookups are prohibited. Because all return values are new object references, and an equality test on a new reference always returns false, the actual reference we use does not affect the result of the after method of the policy, as long as the reference is fresh.

During the execution of the query, the temporary reference can be stored in the state of the policy. But because this state is dropped at the end of the query, this does not cause any problems.

4.5. Binder

So far, sequences of events are only described in terms of single events and sequential composition. Although this works fine for small code samples, more complex applications often need the return values of resource creators in the rest of the sequence of events.

For example, if we try to construct a sequence of events for the code in Figure 10, the closest we get is `Open(), Send(?,”Foo”), Close(?). It is impossible to fill in the question marks because we cannot capture the return value of the event `Open`.

\[
\text{Connection } c = \text{Open(url)}; \\
\text{Send}(c, \text{data}); \\
\text{Close}(c);
\]

![Fig. 10. Use of resource handle.](image)

To solve this problem, we introduce a way to bind the return value of a security-relevant event to a variable. A binder is an assignment statement with a type and variable name at the left hand side, and a resource creator at the right hand side. For example, the code fragment in Figure 10 then performs the sequence `Connection x = Open(), Send(x, “Foo”), Close(x)`. In the previous subsection, we described that we use an arbitrary fresh value in the query instead of the actual return value. Using the binder, this value is stored, and used in the rest of the event sequence. In this way, we have the same effect as if we actually substituted the actual value with our chosen value. Because of the restrictions we introduced in the previous subsection, the query stays sound.

4.6. Example language for sequences of events

So far, we have informally explained what a sequence of events is, and how we query a sequence of events against the policy. In this section, we choose one example syntax to describe sequences of events, and we describe the algorithm to query such sequences against the policy in a precise way.

Figure 11 defines event sequences. An event sequence can be empty, it can be a sequential composition of an existing event sequence and basic event, or it can be a finite repetition of an existing sequence. For the repetition, the integer expression `N` denotes how much iterations must be performed. Basic events are either normal events without a return value or resource creators in which the return value is bound to the variable `Var`. Both events can take a number of argument values. These arguments can be fixed values as well as variables that are within scope at the location of the query. The same applies for the integer expression used in the repetition.

For example: in Figure 12, a small code fragment repeatedly opens and closes a connection. This corresponds with executing the sequence “for \(N\) times (empty, `Connection x = Open()`, `Close(x)`).”
Repetition is defined inductively: a repetition of 0 times evaluates to true, and then we evaluate the basic event.

Sequential composition, we first check if the event sequence changing the context or the state of the policy. For a sequence that must be evaluated. The context contains the three components: the context, the policy and the event sequence from the code. Though finding a quick-and-dirty algorithm that works well for a given application is possible, of code is to define an inference algorithm to extract the sequence of events specified by the code. Furthermore, we create a way to check statically that the sequence of events specified by the code is correct with respect to the block of code, it is still possible to be interrupted.

At this moment, the code producer is responsible for making sure that he queries the right sequence of events for a given block of code. If the sequence of events is not correct with respect to the block of code, it is still possible to be interrupted.

In this section, we propose a more disciplined usage of the query method. Furthermore, we create a way to check statically that the sequence of events specified by the code producer is correct with respect to the block of code it applies to.

5.2. Discovering the sequence of events

In this section, we assume that the code producer has a clearly delimited piece of code that must be protected from interruptions by the security policy. First, our system needs a precise way to discover which sequence of events the code performs. Given this sequence, the query method can be used before actually executing the code to guarantee that it will not be interrupted.

The first way to obtain a sequence of events from a piece of code is to define an inference algorithm to extract the sequence from the code. Though finding a quick-and-dirty algorithm that works well for a given application is possible, inference in general is hard.

Because specification and verification is easier than inference, we tackle this problem first and leave inference as future work. Therefore, we allow the code producer to specify a sequence of events for the code, and create a static verification algorithm to test if the specification is correct with respect to the code.

Figure 13 gives a precise definition of the semantics of query method. It contains a number of evaluation rules that execute the correct before and after methods. Each rule has three components: the context, the policy and the event sequence. Because specification and verification is easier than inference, a repetition of 0 times always evaluates to true and for repeating for N greater than or equal to 1, we first evaluate the sequence and then re-evaluate the repetition for N − 1.

For basic events, we first lookup the variables inside the arguments. Then we execute the corresponding before and after method. We write policy \( \xrightarrow{\pm} policy' \) to express that the method \( x \) of the policy executes without throwing an exception, and the state of the policy is updated to \( policy' \).

Resources creators are treated in a special way. Because the return value of the event is not present at the time the query is executed, we use an arbitrary new object reference instead. Furthermore, the context is updated so that the rest of the sequence can use this variable as well.

5. Static Verification

5.1. Problem statement and general solution

In the previous section, we provided a way for the code producer to test if a sequence of events is allowed by the security policy. We focussed on sequences of events so that a sequence of events can be executed either entirely, or not at all. Unfortunately, this query method alone is not enough to guarantee that certain pieces of code will never be interrupted by the security policy.

At this moment, the code producer is responsible for making sure that he queries the right sequence of events for a given block of code. If the sequence of events is not correct with respect to the block of code, it is still possible to be interrupted.

In this section, we propose a more disciplined usage of the query method. Furthermore, we create a way to check statically that the sequence of events specified by the code producer is correct with respect to the block of code it applies to.

\[
(a', p') \rightarrow (c, p', true)
\]

\[
\begin{align*}
(a, p) \rightarrow (c, p', true) \\
(c, p', BasicEvent) \rightarrow (c', p'', true) \\
(c, p, "EventSeq", BasicEvent) \rightarrow (c'', p'', true)
\end{align*}
\]

\[
\begin{align*}
N \geq 1 & \rightarrow (c, p, EventSeq) \rightarrow (c', p', true) \\
(c', p', "for \ N - 1 \ times \ (EventSeq)\) & \rightarrow (c'', p'', true)
\end{align*}
\]

\[
\begin{align*}
(c, p, "for \ 0 \ times \ (EventSeq)\) & \rightarrow (c, p, true) \\
(c, p, "EventSeq") & \rightarrow (c', p', true)
\end{align*}
\]

\[
\begin{align*}
(a') & = lookupVars(c, a) \\
p & \rightarrow p'
\end{align*}
\]

\[
\begin{align*}
(a', p') & \rightarrow (c, p''', true) \\
(c, p', "NormalEvent(a')") & \rightarrow (c', p'', true)
\end{align*}
\]

\[
\begin{align*}
Var & = newObject() \\
(a', p') & \rightarrow (c, p'', true) \\
(c, p', "Type\ Var = RC(a')") & \rightarrow (c', p'', true)
\end{align*}
\]
5.3. Check statement

In this section, we introduce the check statement (Figure 14). A check statement specifies a sequence of events for a block of code (the CheckBlock) and provides an alternative block of code (the ElseBlock).

```java
check(EventSeq)
  CheckBlock
else
  ElseBlock
```

Fig. 14. Syntax of check statement

At runtime, the translation of the check statement uses the query defined in Section 4. When a check statement is encountered, the runtime system uses a query to test if the sequence specified in the check block is allowed by the policy. If this test succeeds, the CheckBlock is executed. Otherwise, the execution continues with the ElseBlock. Figure 15 precisely describes how the check statement is translated.

```java
if(query(EventSeq))
  CheckBlock
else
  ElseBlock
```

Fig. 15. Dynamic semantics of check statement

The check statement can be seen as a more disciplined use of the query method. The code producer can ignore the result of a query method. With the check construction, the code producer commits to actually executing the sequence if the query succeeds.

5.4. Verification technique for example language

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. To do so, we use the sequences of events as defined in Section 4.6.

Because event sequences can include finite repetition of other sequences, our verification technique needs to be able to reason precisely about multiplication and addition over integer expressions. We use a verification condition generation based approach because classic model checking and abstract interpretation are less precise with respect to reasoning over integers.

Verifying an application with respect to a sequence of events (under the form of temporal logic) already exists for many years using model checking. Using verification condition generation however, our paper is the first and only attempt so far to verify sequences of events.

As a specification language, we use SpecLeuven, a slightly modified version of Spec#. Spec# is an language extension for C# with specification constructs. For this work, method contracts under the form of preconditions and postconditions are most important. Preconditions and postconditions are both boolean expressions. The caller of a method must always make sure that the preconditions of this method evaluate to true. On the other hand, the implementation is responsible for ensuring that the postconditions always evaluate to true.

Spec# comes with a tool to verify that implementations satisfy their specification. For this paper, it suffices to look at this tool as a black box that uses verification condition generation and a fully-automatic theorem prover to achieve its goal.

The intuition behind our verification process is to translate the event sequence into a class with methods for all security-relevant events. These methods have method contracts associated with them to make sure that these methods can only be called in a sequence corresponding to the event sequence. In the body of the check statement, the actual events are replaced by calls to the corresponding method in the contract class we generated. Finally, we use the verifier to check that the modified body respects the method contracts. If no violations are found, we can be sure the original body generates the same security-relevant events as the sequence of events that has been specified.

5.4.1. Translating event sequences

The translation algorithm gets a sequence of events as input and delivers a class with method contracts as output.

Each such class has a specification variable counter. This counter determines how many events have already occurred.

Furthermore, the class contains one method for each security-relevant event. With preconditions we specify for which values of the counter an event can be executed. A postcondition indicates that the value of the counter is increased by one.

Next, the class contains an extra method Terminate. After the entire block of code has been executed, this method is called. Using the preconditions, we check that the value of the counter is equal to the length of the sequence of events.

Finally, all variables used in the sequence of events are added to the contract class as local fields. Variables that are defined in the context must be passed through to the contract class using the constructor. To support reasoning about the local fields, we add a postcondition that the value of these local fields do not change to each method.
BuildSpec :: EventSequence => ClassSpec

BuildSpec(Empty) = New()

BuildSpec(EventSequence, NormalEvent(Arg)) =
seq = BuildSpec(EventSequence);
seq.AddAllVars(Arg);
seq.AddEvent(NormalEvent(Arg));
return seq;

BuildSpec(EventSequence, Type Var = RC(Arg)) =
seq = BuildSpec(EventSequence);
seq.AddAllVars(Arg);
seq.AddEvent(RC(Arg));
seq.AddBinderVariable(Var, RC(Arg));
return seq;

BuildSpec(for N times (EventSeq)) =
seq = BuildSpec(EventSequence);
seq.SubstituteCounter(counter % seq.GetLength());
seq.AddIntVar(N);
seq.MultiplyLength(N);
return seq;

Fig. 17. Algorithm to translate event sequences.

Figure 17 shows the algorithm to build the method contracts. Because it takes 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 build a new contract class (Operation New()). The constructor of a new contract class has a postcondition that the counter equals zero. The terminate method gets a precondition that the counter still equals zero. For each event, there is a method with a precondition false, and a postcondition that the value of the counter is increased by one. In Figure 18, we see the result for a system with three security-relevant events: Open, Close and Read.

public class ContractClass {
    public ContractClass() {
        ensures counter == 0;
        public void Terminate() {
            requires counter == 0;
            public int counter;
            public Connection Open() {
                requires false
                ensures counter == old(counter) + 1;
                public void Close(Connection c) {
                    requires false
                    ensures counter == old(counter) + 1;
                    public void Read() {
                        requires false
                        ensures counter == old(counter) + 1;
                    }
                }
            }
        }
    }
}

Fig. 18. Contract class for the empty sequence.

To translate the sequential composition of a sequence of events with a resource creator bound to a variable, we first perform the same operations as if it was a normal event. After that, we add the variable as a field of the contract class, and we introduce a postcondition that the result of calling the method that corresponds to the resource creator equals the field (Operation AddBinderVariable(Var, RC)).

For instance, to generate the contract class for the sequence Close(x) with x a connection which is defined in the context, the result is displayed in Figure 19.

public class ContractClass {
    public ContractClass(Connection x) {
        ensures counter == 0;
        ensures this.x == x;
        public void Terminate() {
            requires counter == 1;
        }
        public void Close(Connection c) {
            requires false
            ensures counter == old(counter) + 1;
            ensures this.x == old(this.x);
        }
        public void Read() {
            requires false
            ensures counter == old(counter) + 1;
            ensures this.x == old(this.x);
        }
    }
}

Fig. 19. Contract class for Close(x).

To translate the sequential composition of a sequence of events with a resource creator bound to a variable, we first perform the same operations as if it was a normal event. After that, we add the variable as a field of the contract class, and we introduce a postcondition that the result of calling the method that corresponds to the resource creator equals the field (Operation AddBinderVariable(Var, RC)).

For instance, to generate the contract class for the sequence Connection x = Open(), Close(x) we first generate the class for sequence Connection x = Open(). After that, we add the event Close(x) in the same way as we did in the previous example. The variable x is no longer defined by the context. It is now defined by binding it to the return value of Open(). Furthermore, the position of the event Close(x) is different. The result is shown in Figure 20.

Finally, to translate the repetition of a sequence, we first generate the contract class for the existing sequence. In the preconditions on the events, we substitute the counter with a counter modulo the length of the original sequence (Operation SubstituteCounter(newcounter)). Next,
public class ContractClass {
  public ContractClass()
      ensures counter == 0;
  public void Terminate()
      requires counter == 1 + 1;
  public int counter;
  public Connection x;
  public Connection Open()
      requires false || counter == 0;
      ensures counter == old(counter) + 1;
      ensures this.x == result;
  public void Close(Connection c)
      requires false || counter == 0;
      ensures counter == old(counter) + 1;
      ensures this.x == old(this.x);
  public void Read()
      requires false;
      ensures counter == old(counter) + 1;
      ensures this.x == old(this.x);
}

Fig. 20. Contract class for $Connection x = Open(), Close(x)$.

We introduce all variables used in the integer expression $N$ as fields of the contract class, and we make sure that they are initialised by the constructor (Operation AddIntVar(expression)). To make sure that the sequence is repeated the correct amount of times, we multiply the length of the contract class by $N$ (Operation MultiplyLength(amount)).

For instance, to generate the contract class for the sequence for $N$ times ($Connection x = Open(), Close(x)$) we can reuse the contract class we generated in the previous example. The result is shown in Figure 21.

public class ContractClass {
  public ContractClass(int N)
      ensures counter == 0;
      ensures this.N == N;
  public void Terminate()
      requires counter == (1 + 1) * N;
  public int counter;
  public int N;
  public Connection x;
  public Connection Open()
      requires false || counter % 2 == 0;
      ensures counter == old(counter) + 1;
      ensures this.x == result;
      ensures this.N == old(this.N);
  public void Close(Connection c)
      requires false || counter % 2 == 1 && c == this.x;
      requires counter == old(counter) + 1;
      requires this.x == old(this.x);
      ensures this.N == old(this.N);
  public void Read()
      requires false;
      ensures counter == old(counter) + 1;
      ensures this.x == old(this.x);
      ensures this.N == old(this.N);
}

Fig. 21. Contract class for for $N$ times ($Connection x = Open(), Close(x)$).

5.4.2. 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 22.

```java
int N = ...;
check(for $N$ times ($Connection x = Open(), Close(x)$)) {
  ContractClass cc = new ContractClass();
  int i = 0;
  while(i < N) {
    Connection c = cc.Open();
    cc.Close(c);
    i++;
  }
  cc.Terminate();
}

Fig. 22. Transformed check block.
```

5.4.3. Practical issues

To verify loops, the code producer 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 23, a small piece of code with a loop is annotated with a loop invariant.

The invariant specifies that the counter will always be equal to the variable $i$ multiplied by two.

```java
int N = ...;
check(for $N$ times(Send(), Read())) {
  int i = 0;
  while(i < N) {
    Send();
    Read();
    invariant counter == 2 * i;
    i++;
  }
}

Fig. 23. 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. Otherwise, the code producer has to write annotations describing the security-relevant behavior of all those methods.
6. Discussion

In this section, we discuss a number of alternative design decisions we could have made.

6.1. Query as part of the interface

In Section 4, we provided the code producer with a way to query a sequence of events against the security policy even if only an interface of the policy is known. We introduced an example language for expressing sequences of events and we gave an algorithm that explains how this sequence of events must be turned into a sequence of calls to the interface of the policy.

Because there is only limited information about the policy, we create a general implementation for the query that works with any policy and event sequence. Therefore we apply certain restrictions on the events and the policy to keep this query sound.

An alternative is to move the implementation of the query method to the side of the code consumer and provide the query as part of the interface, just like the normal events of the policy. When more information about the policy is available, it might be possible to create an optimized implementation for that specific policy.

In this case, it also is possible to execute queries that represent multiple sequences of events without paying a high performance cost. Although this technique is less precise, it would be useful to investigate how it complements the use of resource creators.

6.2. Optimized implementation of the check block

In Section 5, we introduced the check block as a more disciplined use of the query method. In the current system, the check block also uses the query to perform a check before entering the main block. Because the query method alone cannot be used to enforce the security policy, we still rely on inlining to do this. Unfortunately, this causes a reduction in performance because all checks are executed twice.

An optimized implementation of the check block is possible where all checks happen before the main body and all state updates at the end of it. In a way, this is the same as with inlining, only there are more statements in between the security checks now.

Such system relies on the fact that the code producer has specified the correct sequence of events for his block of code. We defined a static verification technique to check this. Because this verification is executed at the side of the code producer, we need some kind of proof-carrying code [15] solution to carry the results over to the code consumer.

There is an interesting relation between inlining and the check construction. Both approaches can be used to enforce the security policy. However, the check construction focuses on an entire block of code, while inlining focuses on a statement that corresponds to the execution of one single event. On the other hand, the query focuses on allowing the code producer to test if the policy allows a sequence of events. Therefore it can be seen as a more liberal counterpart of both inlining and the check construction.

6.3. Events with return values

In Section 4.4, we focus on one specific category of events with a return value, namely the ones that return a new object reference. These events are usually used to model events that create a new resource handle, therefore we called them resource creators.

It is possible to introduce another category of events, where the return value is not passed to the after method of the policy. Although this reduces the expressiveness of the policy with respect to this event, it would be useful to investigate how well this category complements the class of resource creators.

7. Implementation

The research reported on in this paper is done in the context of the project Security of Software and Services for Mobile Systems (S3MS) [18]. This is a research project under the 6th Framework Programme of the European Commission that addresses the shortcomings of the security model of mobile devices, by integrating a variety of newly-developed and existing security technologies into all phases of the mobile applications lifecycle.

As part of this project, the solution presented in this paper has been partially implemented. We have implemented an inliner for the system described in Section 3. This inliner is compatible with applications written for the .NET Compact Framework as well as with desktop or server applications targeting the full .NET Framework. An algorithm for compiling queries has been worked out, but implementing a full compiler is future work. The static verification approach described in Section 5 has been tested on a number of example applications: the translation of the check block as described in Section 5.4 was performed manually for these examples, but the generation and discharge of the verification conditions was done fully automatically by the SpecLeuven tool developed within our research group as an extension to the Spec# program verifier.

The performance overhead of inlining and querying depends strongly on the overhead of the actual policy, and the time it takes to execute an event. Therefore, we have tested the performance overhead on real security-relevant events, namely opening and closing files. The program we monitored repeatedly opens a file, reads its content and closes it. All measurements where executed on a 3.2 Ghz Pentium D with 1 Gb ram.

To overcome large fluctuations on the execution time of the experiment, we repeated it for 10000 times and we report the total time. In total, the application spent 60,78
ms on querying the policy, and 73027.71 ms executing the rest of the inlined application. Thus the query causes a performance overhead of less than 0.1 percent.

8. Related work

In Section 2, we already discussed the most important related work on security policy enforcement. In this section, we give a broader view on related work.

In the context of resource acquisition, the idea of reserving resources before executing a piece of code is a well-established concept. It is used to avoid resource allocation failures at arbitrary points, leaving the application in an inconsistent state. The query method explores the same idea in the context of security policy enforcement. Since enforcing resource bounds can be implemented using security automata, we treat this problem in a more general context.

The work of Chander et al. [5] on enforcing resource bounds also focuses on resource acquisition. The dynamic check is split from the actual allocation of a resource, and a static verification technique is used to make sure that the code producer has performed sufficient checks before allocating the resource. In our work, the check block does something similar in the context of security policies described as security automata. Therefore, we work with sequences of events instead of integer amounts of resources. Model-Carrying code [20] also uses a combination of static verification and dynamic checks to enforce a policy upon an application. In our work, the task of the static and the dynamic parts are reversed. We use static verification to make sure that a piece of code executes a given sequence of events. In MCC, the model is enforced upon the application at runtime. We use a dynamic test to check if a sequence of events is allowed by the policy. MCC uses a model checker to test if the model is allowed by the policy. Hence, the goals of both systems differ.

In our work, we statically check that a piece of code executes a given sequence of events. Cheon et al. [6] dynamically check that a certain sequence of events is performed on a certain library, only their focus is on enforcing correct use of this library. Moreover, our sequences of events are more flexible. In Session types [7], correct sequencing of send and receive operations is achieved using a type system.

Aspinall et al. [2] extend Java MIDP with a mechanism to do bulk querying for resources. Like in our work, the goal is to support the creation of more robust applications, but our work is more general in supporting arbitrary sequences of events.

The work of Koved et al. [14] uses an interprocedural dataflow analysis to compute the permissions that an application needs for correct execution. Unlike our approach, it is a best effort approach on the side of the code consumer, and the code producer can still be forced to rewrite his application.

9. Conclusion

In this paper we argued that run-time policy enforcement complicates the task of building useful and reliable applications, especially in the presence of multiple unknown policies. We analyzed existing research and contributed in three fields: First, we introduced a way for the code producer to reason about policy enforcement, even when only limited information about the policy is available. Second, we proposed the query construct to check at runtime that a sequence of events is allowed by the policy. Finally, we introduce a static verification technique that guarantees the absence of security-related interruptions inside the check construction.

In the future, it would be interesting to experiment with alternative languages to describe sequences of events. Alternatively, we could provide a heuristic inference algorithm that discovers the sequence of events from a piece of code in the easy cases.

References