Abstract

Mobile is an extension of the .NET Common Intermediate Language that supports certified In-Lined Reference Monitoring. Mobile programs have the useful property that if they are well-typed with respect to a declared security policy, then they are guaranteed not to violate that security policy when executed. Thus, when an In-Lined Reference Monitor (IRM) is expressed in Mobile, it can be certified by a simple type-checker to eliminate the need to trust the producer of the IRM.

Security policies in Mobile are declarative, can involve unbounded collections of objects allocated at runtime, and can regard infinite-length histories of security events exhibited by those objects. The prototype Mobile implementation enforces properties expressed by finite-state security automata—one automaton for each security-relevant object—and can type-check Mobile programs in the presence of exceptions, finalizers, concurrency, and non-termination. Executing Mobile programs requires no change to existing .NET virtual machine implementations, since Mobile programs consist of normal managed CIL code with extra typing annotations stored in .NET attributes.


General Terms Security

Keywords program rewriting, reference monitors, execution monitoring, in-lined reference monitoring, security automata

1. Introduction

Language-based approaches to computer security have employed two major strategies for enforcing security policies over untrusted programs.

- Low-level type systems, such as those used in Java bytecode [21], .NET CIL [9], and TAL for x86 [23], can enforce important program invariants such as memory safety and control safety, which dictate that programs must access and transfer control only to certain suitable memory addresses throughout their executions. Proof-Carrying Code (PCC) [24] generalizes the type-safety approach by providing an explicit proof of safety in first-order logic.

- Execution Monitoring technologies such as Java and .NET stack inspection [14, 21, H.22.11], SASI [11], Java-MAC [20], Java-MOP [4], Polymere [11], and Naccio [12], use runtime checks to enforce temporal properties that can depend on the history of the program’s execution. For example, SASI Java was used to enforce the policy that no program may access the network after it reads from a file [10]. For efficiency, execution monitors are often implemented as In-lined Reference Monitors (IRM’s) [22], wherein the runtime checks are in-lined into the untrusted program itself to produce a self-monitoring program.

The IRM approach is capable of enforcing a large class of powerful security policies, including ones that cannot be enforced with purely static type-checking [17]. In addition, IRM’s can enforce a flexible range of policies, often allowing the code recipient to choose the security policy after the code is received, whereas static type systems and PCC usually enforce fixed security policies that are encoded into the type system or proof logic itself, and that therefore cannot be changed without changing the type system or certifying compiler.

But despite their power and flexibility, the rewriters that automatically embed IRM’s into untrusted programs are typically trusted components of the system. Since rewriters tend to be large and complex when efficient rewriting is required or complex security policies are to be enforced, the rewriter becomes a significant addition to the system’s trusted computing base.

In this paper, we present Mobile, an extension to the .NET CIL that makes it possible to automatically verify IRM’s using a static type-checker. Mobile (MOnitorable BIL with Effects) is an extension of BIL (Baby Intermediate Language) [15], a substantial fragment of managed .NET CIL that was used to develop generics for .NET [19]. Mobile programs are CIL programs with additional typing annotations that track an abstract representation of program execution history. These typing annotations allow a type-checker to verify statically that the runtime checks in-lined into the untrusted program suffice to enforce a specified security policy. Once type-checked, the typing annotations can be erased, and the self-monitoring program can be safely executed as normal CIL code. This verification process allows a rewriter to be removed from the
<table>
<thead>
<tr>
<th>1. REPORT DATE</th>
<th>2. REPORT TYPE</th>
<th>3. DATES COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUN 2006</td>
<td></td>
<td>00-00-2006 to 00-00-2006</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE AND SUBTITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certified In-lined Reference Monitoring on .NET</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6. AUTHOR(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cornell University, Department of Computer Science, Ithaca, NY, 14853</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>14. ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>see report</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. SECURITY CLASSIFICATION OF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. REPORT</td>
</tr>
<tr>
<td>unclassified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>17. LIMITATION OF ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same as Report (SAR)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18. NUMBER OF PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19a. NAME OF RESPONSIBLE PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>unclassified</td>
</tr>
</tbody>
</table>

Approved for public release; distribution unlimited
trusted computing base and replaced with a (simpler) type-checker. Even when the rewriter is small and therefore comparable in size to the type-checker, type-checking constitutes a useful level of redundancy that provides greater assurance than trusting the rewriter alone. Mobile thus leverages the power of IRM’s while using the type-safety approach to keep the trusted computing base small.

Figure 1 summarizes a typical load path on a system that executes IRM’s written in Mobile. Untrusted, managed CIL code is first automatically rewritten according to a security policy, yielding an annotated, self-monitoring program written in Mobile. The rewriting can be performed by either a code producer or by a client machine receiving the untrusted code. Since the rewriter, and therefore the self-monitoring program, remains untrusted, the self-monitoring program is then passed to a trusted type-checker that certifies the code with respect to the original security policy. Code that satisfies the security policy will be approved by the type-checker, and is therefore safe to execute; code that is not well-typed will be rejected and would indicate a failure of the rewriter.

In this paper we focus on robust certification of Mobile code. Techniques for efficient rewriting are left to future work, but we describe a na"ive rewriter and suggest some strategies for optimizing it in 3. Our prototype implementation of Mobile consists of a type-checker that verifies sound rewriting with respect to security policies expressed as $\omega$-regular expressions. The implementation can verify both single-threaded and multi-threaded managed CIL applications, and it supports language features beyond those modeled by BIL, such as exceptions and finalizers.

2. Related Work

Type-systems $\lambda_A$ [11] and $\lambda_{hist}$ [28] enforce history-based security policies over languages based on the $\lambda$-calculus. In both, program histories are tracked at the type-level using effect types that represent an abstraction of those global histories that might have been exhibited by the program prior to control reaching any given program point. Mobile differs from $\lambda_A$ and $\lambda_{hist}$ by tracking history on a per-object basis. That is, both $\lambda_A$ and $\lambda_{hist}$ represent a program’s history as a finite or infinite sequence of global program events, where the set of all possible global program events is always finite. Policies that are only expressible using an infinite set of global program events (e.g., events parameterized by object instances) are therefore not enforceable by $\lambda_A$ or $\lambda_{hist}$. For example, the policy that every opened file must be closed by the time the program terminates is not enforceable by either $\lambda_A$ or $\lambda_{hist}$ when the number of file objects that could be allocated during the program’s execution is unbounded. In object-oriented languages such as the .NET CIL, policies concerning unbounded collections of objects arise naturally, so it is not clear how $\lambda_A$ or $\lambda_{hist}$ can be extended to such settings. Mobile enforces policies that are universally quantified over objects of any given class, and therefore allows objects to be treated as first-class in policy specifications.

PCC has been proposed as a framework for supporting certifying rewriting using temporal logic [3]. The approach is potentially powerful, but does not presently support languages that include exceptions, concurrency, and other features found in real programming languages [2, p. 173]. It is therefore unclear whether proof size and verification speed would scale well in practical settings.

CQual [13] and Vault [7] are C-like languages that enforce history-based properties of objects by employing a flow-sensitive type system based on alias types [29]. Security-relevant objects in CQual or Vault programs have their base types augmented with type qualifiers, which statically track the security-relevant state of the object. A type-checker then determines if any object might enter a state at runtime that violates the security policy. Vault’s type system additionally includes variant types that allow a runtime value to reflect an object’s current state. The Vault type-checker identifies instructions that test these state values to ensure that those tests will prevent security violations when the program is executed. Fugue [8] is a static verifier based on Vault that uses programmer-supplied specifications to find bugs in .NET source code. It verifies policies that constrain the use of system resources or that prescribe protocols that constrain the order in which methods may be called on objects. Fugue supports any source language that compiles to managed .NET CIL code, but it does not support exceptions, finalizers, or concurrency. It additionally lacks a formal proof of soundness for its aliasing analysis and type system.

Inspired by CQual, Vault, and Fugue, our work scales these ideas up to a large existing programming language—the full managed .NET CIL (minus reflection)—while providing a formal proof of soundness. In scaling up to a larger-scale language, we adopt a somewhat different approach to tracking object security states at the type level. CQual, Vault, and Fugue assign linear types to security-relevant objects (and, in the case of Vault, to runtime state values), and use aliasing analyses to track changes to items with linear types. However, it is not clear how such analyses can be extended to support concurrency or to support an important technique commonly used by IRM’s to track object security states, wherein security-relevant objects are paired with runtime values that record their states, and then such pairs are permitted to leak to the heap. Existing alias analyses cannot easily track items that are permitted to leak to the heap arbitrarily, or that are shared between threads. We therefore take the approach of $L^3$ [22], wherein linearly-typed items are permitted to leak to the heap by packing them into shared data structures with limited interfaces. These shared object-state pairs, called packages, can be aliased arbitrarily and are not tracked by the type system. Mobile provides trusted operations for packing and unpacking linear-typed items to and from shared package objects. To perform any (security-relevant) operation that might change a value with linear type, it must first be unpacked from any package that contains it. As with ownership types [6], packing and unpacking operations are implemented as destructive reads, so that only one thread can perform security-relevant operations on a given security-relevant object at a time. Mobile’s type system and the CLI permissions system are both leveraged to maintain invariants linking an object to an accurate runtime representation of its state.

---

**Figure 1. A Mobile load path**

trusted computing base and replaced with a (simpler) type-checker. Even when the rewriter is small and therefore comparable in size to the type-checker, type-checking constitutes a useful level of redundancy that provides greater assurance than trusting the rewriter alone. Mobile thus leverages the power of IRM’s while using the type-safety approach to keep the trusted computing base small.

Figure 1 summarizes a typical load path on a system that executes IRM’s written in Mobile. Untrusted, managed CIL code is first automatically rewritten according to a security policy, yielding an annotated, self-monitoring program written in Mobile. The rewriting can be performed by either a code producer or by a client machine receiving the untrusted code. Since the rewriter, and therefore the self-monitoring program, remains untrusted, the self-monitoring program is then passed to a trusted type-checker that certifies the code with respect to the original security policy. Code that satisfies the security policy will be approved by the type-checker, and is therefore safe to execute; code that is not well-typed will be rejected and would indicate a failure of the rewriter.

In this paper we focus on robust certification of Mobile code. Techniques for efficient rewriting are left to future work, but we describe a na"ive rewriter and suggest some strategies for optimizing it in 3. Our prototype implementation of Mobile consists of a type-checker that verifies sound rewriting with respect to security policies expressed as $\omega$-regular expressions. The implementation can verify both single-threaded and multi-threaded managed CIL applications, and it supports language features beyond those modeled by BIL, such as exceptions and finalizers.

2. Related Work

Type-systems $\lambda_A$ [11] and $\lambda_{hist}$ [28] enforce history-based security policies over languages based on the $\lambda$-calculus. In both, program histories are tracked at the type-level using effect types that represent an abstraction of those global histories that might have been exhibited by the program prior to control reaching any given program point. Mobile differs from $\lambda_A$ and $\lambda_{hist}$ by tracking history on a per-object basis. That is, both $\lambda_A$ and $\lambda_{hist}$ represent a program’s history as a finite or infinite sequence of global program events, where the set of all possible global program events is always finite. Policies that are only expressible using an infinite set of global program events (e.g., events parameterized by object instances) are therefore not enforceable by $\lambda_A$ or $\lambda_{hist}$. For example, the policy that every opened file must be closed by the time the program terminates is not enforceable by either $\lambda_A$ or $\lambda_{hist}$ when the number of file objects that could be allocated during the program’s execution is unbounded. In object-oriented languages such as the .NET CIL, policies concerning unbounded collections of objects arise naturally, so it is not clear how $\lambda_A$ or $\lambda_{hist}$ can be extended to such settings. Mobile enforces policies that are universally quantified over objects of any given class, and therefore allows objects to be treated as first-class in policy specifications.

PCC has been proposed as a framework for supporting certifying rewriting using temporal logic [3]. The approach is potentially powerful, but does not presently support languages that include exceptions, concurrency, and other features found in real programming languages [2, p. 173]. It is therefore unclear whether proof size and verification speed would scale well in practical settings.

CQual [13] and Vault [7] are C-like languages that enforce history-based properties of objects by employing a flow-sensitive type system based on alias types [29]. Security-relevant objects in CQual or Vault programs have their base types augmented with type qualifiers, which statically track the security-relevant state of the object. A type-checker then determines if any object might enter a state at runtime that violates the security policy. Vault’s type system additionally includes variant types that allow a runtime value to reflect an object’s current state. The Vault type-checker identifies instructions that test these state values to ensure that those tests will prevent security violations when the program is executed. Fugue [8] is a static verifier based on Vault that uses programmer-supplied specifications to find bugs in .NET source code. It verifies policies that constrain the use of system resources or that prescribe protocols that constrain the order in which methods may be called on objects. Fugue supports any source language that compiles to managed .NET CIL code, but it does not support exceptions, finalizers, or concurrency. It additionally lacks a formal proof of soundness for its aliasing analysis and type system.

Inspired by CQual, Vault, and Fugue, our work scales these ideas up to a large existing programming language—the full managed .NET CIL (minus reflection)—while providing a formal proof of soundness. In scaling up to a larger-scale language, we adopt a somewhat different approach to tracking object security states at the type level. CQual, Vault, and Fugue assign linear types to security-relevant objects (and, in the case of Vault, to runtime state values), and use aliasing analyses to track changes to items with linear types. However, it is not clear how such analyses can be extended to support concurrency or to support an important technique commonly used by IRM’s to track object security states, wherein security-relevant objects are paired with runtime values that record their states, and then such pairs are permitted to leak to the heap. Existing alias analyses cannot easily track items that are permitted to leak to the heap arbitrarily, or that are shared between threads.

We therefore take the approach of $L^3$ [22], wherein linearly-typed items are permitted to leak to the heap by packing them into shared data structures with limited interfaces. These shared object-state pairs, called packages, can be aliased arbitrarily and are not tracked by the type system. Mobile provides trusted operations for packing and unpacking linear-typed items to and from shared package objects. To perform any (security-relevant) operation that might change a value with linear type, it must first be unpacked from any package that contains it. As with ownership types [6], packing and unpacking operations are implemented as destructive reads, so that only one thread can perform security-relevant operations on a given security-relevant object at a time. Mobile’s type system and the CLI permissions system are both leveraged to maintain invariants linking an object to an accurate runtime representation of its state.
3. Overview

A Mobile security policy identifies a set of security-relevant object classes and assigns a set of acceptable traces to each such class. A trace is a finite or infinite sequence of security-relevant events—program operations that take a security-relevant object as an argument. Our implementation expresses security policies as \( \omega \)-regular expressions over the alphabet of events, but the formalisms presented in this article can be leveraged to support alternative policy languages as well. A Mobile program satisfies the security policy if for every complete run of the program, (i) if the run is finite (i.e., the program terminates), the sequence of security-relevant events performed on every object allocated during that run is a member of the set of traces that the security policy has assigned to that object's class; and (ii) if the run is infinite (i.e., the program does not terminate), at each step of the run the sequence of security-relevant events performed so far on each security-relevant object is a prefix of a member of the set of traces assigned to that object's class.

For example, [8, p. 5] proposes a security policy involving a WebPageFetcher class for which proper usage is to call the Open method to acquire the resource, the GetPage method to use the resource, and the Close method to release the resource. A Mobile policy that requires programs to open web pages before reading them, allows at most three reads per opened page, and requires programs to close web pages before the program terminates (but allows them to remain open on runs that never terminate), might assign \((O \cup G \cup C)^*\) as the set of acceptable traces for class WebPageFetcher (where \(O\), \(G\), and \(C\) denote Open, GetPage, and Close events, respectively, and \(\omega\) denotes finite or infinite repetition). Note that this policy can only be enforced by a mechanism that tracks events at a per-object level.

Although Mobile security policies model events as operations performed on objects, global events that do not concern any particular object can be encoded as operations on a global object that is allocated at program start and destroyed at program termination. Thus, Mobile policies can regard global events, per-object events, and combinations of the two.

For example, one might modify the example policy above by additionally requiring that at most ten network sends may occur during the lifetime of the program. In that case, the global object would additionally be identified as a security-relevant object, a Send method call performed on any System.Net.Sockets.Socket object would be identified as a security-relevant event for the global object, and the global object would be assigned the set of traces denoted by \(\epsilon \cup S \cup S^2 \cup \cdots \cup S^{10}\) (where \(S\) denotes a Send event).

A rewriter that produces self-monitoring programs from untrusted CIL code is expected to produce well-typed Mobile code, so that the policy-adherence theorem can be used to guarantee that it is safe to execute. For this rewriting task to be feasible, Mobile’s type system must be flexible enough to permit writers to insert runtime security checks—well-typed code that tracks the state of security-relevant objects at runtime, testing aspects of the state that cannot be verified statically. To that end, Mobile supports a pack operation that pairs a security-relevant object with a runtime value (e.g., an integer) representing an abstraction of the object’s current state, and that encapsulates them into a two-field package object. Mobile’s unpack operation can be used to unpack a package, yielding the original object that was packed along with the runtime value that represents its state. Mobile programs can then test this runtime value to infer information about the associated object’s state. Both pack and unpack are implemented as CIL method calls to a small trusted library (about ten lines of C# code).

To keep type-checking tractable, Mobile does not allow security-relevant operations on objects that are packed. A package class’s two fields are declared to be private so that, to access a security-relevant object directly and perform operations on it, it must first be unpacked. While unpacked, Mobile allows only limited aliasing of security-relevant objects—none of their aliases can escape to the heap. To enforce this restriction, the unpack operation is implemented as a destructive read, preventing the package from being unpacked again before it is re-packed. Packages, however, are permitted to escape to the heap and to undergo unlimited aliasing. These restrictions allow the type-checker to statically track histories of unpacked objects and to ensure that packed objects are always paired with a value that accurately reflects their state. When an object is packed, it is safe for the type-checker to forget whatever information might be statically known about the object, keeping the type-checking algorithm tractable and affording the rewriter a dynamic fallback mechanism when static analysis cannot verify all security-relevant operations.

When pack and unpack are implemented as atomic operations, Mobile can also enforce security policies in concurrent settings. In such a setting, Mobile’s type system maintains the invariant that each security-relevant object is either packed or held by a thread that itself will not escape to the heap. To enforce this restriction, the type system ensures that packed objects are always paired with a value that accurately reflects their state. When an object is packed, it is safe for the type-checker to forget whatever information might be statically known about the object, keeping the type-checking algorithm tractable and affording the rewriter a dynamic fallback mechanism when static analysis cannot verify all security-relevant operations.

For example, [8, p. 5] proposes a security policy involving a WebPageFetcher class for which proper usage is to call the Open method to acquire the resource, the GetPage method to use the resource, and the Close method to release the resource. A Mobile policy that requires programs to open web pages before reading them, allows at most three reads per opened page, and requires programs to close web pages before the program terminates (but allows them to remain open on runs that never terminate), might assign \((O \cup G \cup C)^*\) as the set of acceptable traces for class WebPageFetcher (where \(O\), \(G\), and \(C\) denote Open, GetPage, and Close events, respectively, and \(\omega\) denotes finite or infinite repetition). Note that this policy can only be enforced by a mechanism that tracks events at a per-object level.

Although Mobile security policies model events as operations performed on objects, global events that do not concern any particular object can be encoded as operations on a global object that is allocated at program start and destroyed at program termination. Thus, Mobile policies can regard global events, per-object events, and combinations of the two.

For example, one might modify the example policy above by additionally requiring that at most ten network sends may occur during the lifetime of the program. In that case, the global object would additionally be identified as a security-relevant object, a Send method call performed on any System.Net.Sockets.Socket object would be identified as a security-relevant event for the global object, and the global object would be assigned the set of traces denoted by \(\epsilon \cup S \cup S^2 \cup \cdots \cup S^{10}\) (where \(S\) denotes a Send event).

A rewriter that produces self-monitoring programs from untrusted CIL code is expected to produce well-typed Mobile code, so that the policy-adherence theorem can be used to guarantee that it is safe to execute. For this rewriting task to be feasible, Mobile’s type system must be flexible enough to permit writers to insert runtime security checks—well-typed code that tracks the state of security-relevant objects at runtime, testing aspects of the state that cannot be verified statically. To that end, Mobile supports a pack operation that pairs a security-relevant object with a runtime value (e.g., an integer) representing an abstraction of the object’s current state, and that encapsulates them into a two-field package object. Mobile’s unpack operation can be used to unpack a package, yielding the original object that was packed along with the runtime value that represents its state. Mobile programs can then test this runtime value to infer information about the associated object’s state. Both pack and unpack are implemented as CIL method calls to a small trusted library (about ten lines of C# code).

To keep type-checking tractable, Mobile does not allow security-relevant operations on objects that are packed. A package class’s two fields are declared to be private so that, to access a security-relevant object directly and perform operations on it, it must first be unpacked. While unpacked, Mobile allows only limited aliasing of security-relevant objects—none of their aliases can escape to the heap. To enforce this restriction, the unpack operation is implemented as a destructive read, preventing the package from being unpacked again before it is re-packed. Packages, however, are permitted to escape to the heap and to undergo unlimited aliasing. These restrictions allow the type-checker to statically track histories of unpacked objects and to ensure that packed objects are always paired with a value that accurately reflects their state. When an object is packed, it is safe for the type-checker to forget whatever information might be statically known about the object, keeping the type-checking algorithm tractable and affording the rewriter a dynamic fallback mechanism when static analysis cannot verify all security-relevant operations.

When pack and unpack are implemented as atomic operations, Mobile can also enforce security policies in concurrent settings. In such a setting, Mobile’s type system maintains the invariant that each security-relevant object is either packed or held by a thread that itself will not escape to the heap. To enforce this restriction, the type system ensures that packed objects are always paired with a value that accurately reflects their state. When an object is packed, it is safe for the type-checker to forget whatever information might be statically known about the object, keeping the type-checking algorithm tractable and affording the rewriter a dynamic fallback mechanism when static analysis cannot verify all security-relevant operations.
A Formal Analysis of Mobile

The Abstract Machine

Figure 2 gives the Mobile instruction set. Like Bil, Mobile’s syntax is written in postfix notation. In addition to Bil instructions, Mobile includes:

- instructions `evt` and `e`, which perform security-relevant operations `e` on an object (where `e` is some unique identifier, such as “open”), that we associate with each security-relevant operation,
- instructions `newpackage` and `newwhist` for creating packages and runtime state values,
- instructions `pack` and `unpack` for packing/unpacking objects and runtime state values to/from packages,
- instruction `condst`, which dynamically tests a runtime state value, and
- the pseudo-instructions `if` and `ret`, which do not appear in source code but are introduced in the intermediate stages of the small-step semantics presented in Figure 4. Instruction `if` is a term that has been reduced to value `v`, and instruction `ret` pops the current stack frame at the end of a method call.

These abstract instructions model real CIL instructions. For example, if calls to method `m` are security-relevant operations, the CIL instruction that invokes `m` on object `o` is modeled by the Mobile instruction sequence: `o evt em; o calivirt C::m.Sig m`. A description of how our implementation models other CIL instructions is given in Figure 4.

Figure 3 provides Mobile’s type system. Mobile types consist of untracked, integer, and history-abstraction types (the types of runtime state values). The type of each unpacked, security-relevant object `C(l)` is parameterized by an object identity variable `l` that uniquely identifies the object. All aliases of the object have types with the same object identity variable, but other unpacked objects of the same class have types with different object identity variables. The types `C(l)` of packed classes and security-irrelevant classes do not include object identity variables, and their instances are therefore not distinguishable by the type system. We consider Mobile terms to be equivalent up to alpha conversion of bound variables.

The types `RepC(H)` of runtime state values are parameterized both by the class type `C` of the object to which they refer and by a history abstraction `H`—an `ω`-regular expression (plus variables and intersection) that denotes a set of traces. In such an expression, `ω` denotes finite or infinite repetition.

Closed (i.e., variable-less) history abstractions conform to a subset relation; we write `H1 ⊆ H2` if the set of traces denoted by `H1` is a subset of the set of traces denoted by `H2`. This subset relation induces a natural subtyping relation `⊆` given in Figure 4. Observe that the subtyping relation in Figure 4 does not recognize class subtyping of security-relevant classes. We leave support for subtyping of security-relevant classes to future work.

Type variables in Mobile types are bound by typing contexts, that is, associating a history abstraction `Ψ` with each object identity variable that corresponds to an unpacked, security-relevant object. Since object identity variables uniquely identify each object instance, object history maps can be seen as a spatial conjunction `Ψ` of assertions about the histories of the various unpacked objects in the heap.

A complete Mobile program consists of:

<table>
<thead>
<tr>
<th>Class names</th>
<th>Field types</th>
<th>Class methods</th>
<th>Class policies</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>C</code></td>
<td><code>field : (C × f) → μ</code></td>
<td><code>methodbody : (C::m.Sig) → I</code></td>
<td><code>policy : C → H</code></td>
</tr>
</tbody>
</table>

Figure 2. The Mobile instruction set

Figure 3. The Mobile type system

4. Formal Analysis of Mobile

4.1 The Abstract Machine

Figure 2 gives the Mobile instruction set. Like Bil, Mobile’s syntax is written in postfix notation. In addition to Bil instructions, Mobile includes:

- instruction `evt e`, which performs security-relevant operation `e` on an object (where `e` is some unique identifier, such as “open”), that we associate with each security-relevant operation,
- instructions `newpackage` and `newwhist` for creating packages and runtime state values,
- instructions `pack` and `unpack` for packing/unpacking objects and runtime state values to/from packages,
- instruction `condst`, which dynamically tests a runtime state value, and
- the pseudo-instructions `if` and `ret`, which do not appear in source code but are introduced in the intermediate stages of the small-step semantics presented in Figure 4. Instruction `if` is a term that has been reduced to value `v`, and instruction `ret` pops the current stack frame at the end of a method call.

These abstract instructions model real CIL instructions. For example, if calls to method `m` are security-relevant operations, the CIL instruction that invokes `m` on object `o` is modeled by the Mobile instruction sequence: `o evt em; o calivirt C::m.Sig m`. A description of how our implementation models other CIL instructions is given in Figure 4.

Figure 3 provides Mobile’s type system. Mobile types consist of untracked, integer, and history-abstraction types (the types of runtime state values). The type of each unpacked, security-relevant object `C(l)` is parameterized by an object identity variable `l` that uniquely identifies the object. All aliases of the object have types with the same object identity variable, but other unpacked objects of the same class have types with different object identity variables. The types `C(l)` of packed classes and security-irrelevant classes do not include object identity variables, and their instances are therefore not distinguishable by the type system. We consider Mobile terms to be equivalent up to alpha conversion of bound variables.

The types `RepC(H)` of runtime state values are parameterized both by the class type `C` of the object to which they refer and by a history abstraction `H`—an `ω`-regular expression (plus variables and intersection) that denotes a set of traces. In such an expression, `ω` denotes finite or infinite repetition.

Closed (i.e., variable-less) history abstractions conform to a subset relation; we write `H1 ⊆ H2` if the set of traces denoted by `H1` is a subset of the set of traces denoted by `H2`. This subset relation induces a natural subtyping relation `⊆` given in Figure 4. Observe that the subtyping relation in Figure 4 does not recognize class subtyping of security-relevant classes. We leave support for subtyping of security-relevant classes to future work.

Type variables in Mobile types are bound by typing contexts `Γ`, which assign class or package types to object identity variables `l` and declare any history abstraction variables `θ`. Object identity variables can additionally appear in object history maps `Ψ`, which associate a history abstraction `H` with each object identity variable that corresponds to an unpacked, security-relevant object. Since object identity variables uniquely identify each object instance, object history maps can be seen as a spatial conjunction `Ψ` of assertions about the histories of the various unpacked objects in the heap.

A complete Mobile program consists of:

<table>
<thead>
<tr>
<th>Class names</th>
<th>Field types</th>
<th>Class methods</th>
<th>Class policies</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>C</code></td>
<td><code>field : (C × f) → μ</code></td>
<td><code>methodbody : (C::m.Sig) → I</code></td>
<td><code>policy : C → H</code></td>
</tr>
</tbody>
</table>
We also use the notation \( \text{fields}(C) \) to refer to the number of fields in class \( C \). Method signatures \( \text{Sig} \) will be described in \( \ref{subsec:method-signatures} \).

### 4.2 Operational Semantics

Unlike \( \ref{subsec:operational-semantics} \), we provide a small-step operational semantics for Mobile rather than a large-step semantics, so as to apply the policy adherence theorems presented in \( \ref{subsec:policy-adherence} \) to programs that do not terminate or that enter a bad state.

In Mobile’s small-step memory model, presented in Figure \( \ref{fig:memory-model} \), objects consist not only of an assignment of values to fields but also a trace \( \vec{e} \) that records a history of the security-relevant operations performed on the object. Although our model attaches a history trace to each object, we prove in \( \ref{subsec:history-traces} \) that it is unnecessary for the virtual machine to track and store object traces because well-typed Mobile code never exhibits a trace that violates the security policy.

The small-step operational semantics of Mobile, given in Figures \( \ref{fig:semantics} \) and \( \ref{fig:semantics} \), define how a given store \( \psi \) and instruction \( I \) step to a new store \( \psi' \) and instruction \( I' \), written \( \psi, I \rightarrow \psi', I' \). Rules \( \ref{rule:execute-event} \) and \( \ref{rule:execute-event} \) model the behavior of the new instructions introduced by Mobile. Rule \( \ref{rule:execute-event} \) appends event \( e_1 \) to the sequence of events exhibited on object \( \ell \). Rule \( \ref{rule:execute-event} \) introduces a new package object to the local context. Rule \( \ref{rule:execute-event} \) assigns an object \( \ell' \) and runtime state value \( \text{rep}_C(H) \) to the fields of package \( \ell \). Rule \( \ref{rule:execute-event} \) yields the object and runtime state value stored in package \( \ell \) and erases \( \ell' \)’s fields.

Rules \( \ref{rule:execute-event} \) and \( \ref{rule:execute-event} \) use notation not previously defined and therefore deserve special note. Runtime operations \( \text{test}_{C,k} \) and \( \text{hc}_{C,k} \) test runtime state values and construct new runtime state values, respectively. Rather than fixing these two operations, we allow Mobile to be extended with unspecified implementations of them. Different implementations of \( \text{test}_{C,k} \) and \( \text{hc}_{C,k} \) can therefore be used to allow Mobile to support different collections of security policies. For example, a Mobile system that supports security policies expressed as DFA’s might implement runtime state values as 32-bit integers and might support tests that compare runtime state values to integer constants (to determine which state the DFA is in). In that case, one could define for each \( k \in \{0, 2^{32}\} \), \( \text{hc}_{C,k}() \) = \( k \) and \( \text{test}_{C,k}(i) = \{1 \) if \( i = k \), else 0\). A more powerful (but more computationally expensive) Mobile system might implement runtime state values as dynamic data structures that record an object’s entire trace and might provide tests to examine such structures. In this paper, we assume only that a countable collection of state value constructors and tests exists and that this collection adheres to typing constraints \( \ref{constraint:typing-1} \), \( \ref{constraint:typing-2} \), \( \ref{constraint:typing-3} \) and \( \ref{constraint:typing-4} \) presented in \( \ref{subsec:typing} \).

![Figure 5. The Mobile memory model](image)

![Figure 6. Mobile Evaluation Contexts](image)

![Figure 7. Small-step Operational Semantics for Mobile](image)
1 (newobj C()) starg 1;
2 (ldarg 1) evt e1;
3 (ldarg 1) evt e2;
4 (newpackage C) starg 2;
5 (ldarg 2) (ldarg 1) (newhist C, 0) pack;
6 (...) (ldarg 2) stfd ...;
7 (ldarg 2) unpack 4) starg 3;
8 (ldarg 3) ((ldarg 4) evt e1) (...) condst C, 0

Figure 8. Sample Mobile program

The operational semantics given in Figure 7 are for a single-threaded virtual machine without support for finalizers. To model concurrency, one could extend our stacks to consist of multiple threads and add a small-step rule that non-deterministically chooses which thread to execute next. Finalizers could be modeled by adding another small-step rule that non-deterministically forks a finalizer thread whenever an object is unreachable. Our implementation supports concurrency and finalizers, but to simplify the presentation, we leave the analysis of these language features to future work.

4.3 Type System

Mobile’s type system considers each Mobile term to be a linear operator from a history map and frame list (describing the initial heap and stack, respectively) to a new history map and frame list (describing the heap and stack yielded by the operation) along with a return type. That is, we write \( \Gamma \vdash I : (\Psi; FF) \rightarrow \exists \gamma'. (\Psi'; FF'; \tau') \) if term \( I \), when evaluated in typing context \( \Gamma \), takes history map \( \Psi \) and frame list \( FF \) (in which any typing variables are bound in context \( \Gamma \)) to new history map \( \Psi' \) and new frame list \( FF' \), and yields a value of type \( \tau' \) (if it terminates). Any new typing variables appearing in \( FF' \) and \( \tau' \) are bound in context \( \Gamma' \). A method signature (see Figure 4) is the type assigned to the term comprising its body.

Below, we provide an informal description of Mobile’s typing rules by walking the type-checking algorithm through the sample Mobile program given in Figure 8. A complete list of typing rules is stated formally in the appendix.

Line 1 of the sample program creates a new object of class \( C \) and stores it in local register 1. When a new security-relevant object is created, Mobile’s type system assigns it a fresh object identity variable \( \ell \). The return type of the newly created object is thus \( C(\ell) \) and the new history map yielded by the operation satisfies \( \Psi(\ell) = e \); that is, new objects are initially assigned the empty trace.

As security-relevant events are performed on the object, the type system tracks these changes by statically updating its history map to append these new events to the sequence it recorded in its history map. So for example, after processing lines 2–3 of the sample program, which perform events \( e_1 \) and \( e_2 \) on the object in local register 1, the type-checker’s new history map would satisfy \( \Psi(\ell) = e_1e_2 \). At each point that a security-relevant event is performed, the type system ensures that the new trace satisfies a prefix of the security policy. For example, when type-checking line 3, the type-checker would verify that \( e_1e_2 \subseteq \text{pre}(\text{policy}(C)) \), where \( \text{policy}(C) \) denotes the set of acceptable traces assigned by the security policy to class \( C \), and \( \text{pre}(\text{policy}(C)) \) denotes the set of prefixes of members of set \( \text{policy}(C) \).

Security-relevant objects of type \( C(\ell) \) are like typical objects except that they are not permitted to escape to the heap. That is, they cannot be assigned to object fields. In order to leak a security-relevant object to the heap, a Mobile program must first store it in a package using a \texttt{pack} instruction. This requires three steps: (1) A package must be created via a \texttt{newpackage} instruction. (2) A runtime state value must be created that accurately reflects the state of the object to be packed. This is accomplished via the \texttt{newhist} instruction, which is described in more detail below. (3) Finally, the \texttt{pack} operation is used to store the object and the runtime state value into the package. Lines 4 and 5 of the sample program illustrate these three steps. Line 4 creates a new package and stores it in local register 2. Line 5 then fills the package using the object in local register 1 along with a newly created runtime state value.

In order for Mobile’s type system to accept a \texttt{pack} operation, it must be able to statically verify that the runtime state value is an accurate abstraction of the object being packed. That is, if the runtime state value has type \( \text{Rep}_C'H \), then the type system requires that \( \Psi(\ell) \subseteq H \) where \( \ell \) is the object identity variable of the object being packed. Additionally, since packed objects are untracked and therefore might continue to exist until the program terminates, packed objects must satisfy the security policy. That is, we require that \( \Psi(\ell) \subseteq \text{policy}(C) \).

Packages that contain security-relevant objects can leak to the heap, as illustrated by line 6 of the sample program, which stores the package to a field of some other object. Since only packed objects can leak to the heap, the restriction that packed objects must be in a policy-adherent state is a potential limitation of the type system. That is, it might often be desirable to leak an object that is not yet in a policy-adherent state to the heap, but later retrieve it and restore it to a policy-adherent state before the program terminates. In [5] we show how Mobile implementations can use finalizer code to avoid this restriction and leak objects to the heap even when they are not yet in a policy-adherent state.

After a \texttt{pack} operation, the type system removes object identity variable \( \ell \) from the history map. Hence, after line 5 of the sample program, \( \Psi(\ell) \) is undefined and the object that was packed becomes inaccessible. If the program were to subsequently attempt to load from local register 1 (before replacing its contents with something else), the type-checker would reject the code because that register now contains a value with an invalid type. Object identity variable \( \ell \) can therefore be thought of as a capability that has been revoked from the local scope and given to the package.

In order to perform more security-relevant events on an object, a Mobile program must first reacquire a capability for the object by unpacking the object from its package via an \texttt{unpack} instruction. Line 7 of the sample programunpacks the package in local register 2, storing the extracted object in local register 3 and storing the runtime state value that was packed with it in local register 4. Since packages and the objects they contain are not tracked by the type system, the type system cannot statically determine the history of a freshly unpacked object. All that is statically known is that the runtime state value that will be yielded at runtime by the \texttt{unpack} instruction will be an accurate representation of the unpacked object’s history. To reflect this information statically, the type system assigns a fresh object identity variable \( \ell' \) to the unpacked object and a fresh history variable \( \theta \) to the unknown history. The unpacked object and runtime state value then have types \( C(\ell') \) and \( \text{Rep}_C'\theta \), respectively, and the new history map satisfies \( \Psi(\ell') = \theta \). The type \( C(\ell') \) of a package can hence be thought of as an existential type binding type variables \( \ell' \) and \( \theta \).

If the sample program were at this point to perform security-relevant event \( e \) on the newly unpacked object, Mobile’s type system would reject because it would be unable to statically verify that \( \theta e \subseteq \text{policy}(C) \) (since nothing is statically known about history \( \theta \)). However, a Mobile program can perform additional \texttt{evt} operations on the object by first dynamically testing the runtime state value yielded by the \texttt{unpack} operation. If a Mobile program dynamically tests a value of type \( \text{Rep}_C'\theta \), Mobile’s type system can statically infer information about history \( \theta \) within the branches of the conditional. For example, if a \texttt{condst} instruction is used to test a value with type \( \text{Rep}_C'\theta \) for equality with a value of type
Mobile supports many possible schemes for representing histories at runtime and for testing them, so rather than fixing particular operations for constructing runtime state values and particular operations for testing them, we instead assume only that there exists a countable collection of constructors `newhist C, k` and conditionals `condst C, k` for all integers `k`, that construct runtime state values and test runtime state values (respectively) for objects of class `C`. We then abstractly define `HC.(),...` to be the type `RepC(·)` of a history value constructed using constructor `k` for security-relevant class `C`, and we define `ctxC,k(·, ·)` and `ctxC,k(·, ·)` to be the object history maps that refine `Ψ` in the positive and negative branches (respectively) of a conditional that performs test `k` on a history value of type `RepC(·)`. Mobile supports any such refinement that is in the sense that

\[
\text{test}_{C,k}(H) = \begin{cases} 0 & \Rightarrow \Psi \leq \text{ctx}_{C,k}(H, \Psi)(f) \\
\neq 0 & \Rightarrow \Psi \leq \text{ctx}_{C,k}(H, \Psi)(f) \end{cases}
\]

We further assume that each history type constructor `HC.()` accurately reflects its runtime implementation, in the sense that for all history value types `RepC1,1,...,RepCn,n(H)` such that `n = \text{arity}(HC.C, k)`, there exists some `H` such that

\[
HC.C, k(RepC1,1,...,RepCn,n(H)) = RepC(·)
\]

and

\[
\text{hc}_{C,k}(RepC1,1,...,RepCm,m(H)) = RepC(·)
\]

In the sample program, suppose that history value constructor `newhist C, 0` takes no arguments and yields a runtime value that represents history `e1,e2`; and suppose that conditional test `condst C, 0` compares a runtime state value to the value that represents history `e1,e2`. Formally, suppose that `HC.C, 0()` = `RepC1,1,...,RepCm,m` and `ctxC,0(0, Ψ) = Ψ[θ → e1]`. Thus, in the positive branch of such a test, the type-checker’s object history map can be refined by substituting `e1,e2` for any instances of the history variable being tested. Then if `policy(C) = (e1,e2)`, a Mobile type-checker would accept the sample program. In the negative branch of the conditional in line 8, the type-checker would infer that the object in local register 4 has history `e1,e2`, and therefore it is safe to perform event `e1` on it. However, if `policy(C) = e1,e2`, then the type-checker would reject, because `e1,e2,e1` is not a prefix of `e1,e2,e2`.

In the negative branch of this conditional the type-checker can infer that the object in local register 4 has a history represented by a state value other than the one that it was tested against. The history map could therefore be refined by substituting history variable `θ` with the union of all of the history abstractions associated with all of the other possible runtime state values defined for that object’s class.

Our implementation of Mobile implements history abstraction values as integers. Thus, it provides `2^{32} newhist` operations for each security-relevant class `C`, defining `hc.C, k()` = `k` for all `k` in `0..2^{32} − 1`. Tests `condst` of runtime state values are implemented as equality comparisons between the integer runtime state value to be tested and an integer constant. Thus, we define

\[
\text{test}_{C,k}(\text{rep}_C(θ)) = \begin{cases} 1 & \text{if } \text{rep}_C(θ) = k \\
0 & \text{otherwise} \end{cases}
\]

\[
\text{ctx}_{C,k}(θ, Ψ) = Ψ[θ ↦ θ \cap (H_k)]
\]

\[
\text{ctx}_{C,k}(θ, Ψ) = Ψ[θ ↦ θ \cap (H_k)]
\]

for each integer `k` in `0..2^{32} − 1`, where `H_k` is a closed history abstraction statically assigned to integer constant `k`. The assignments of closed history abstractions `H_k` to integers `k` are not trusted, so this mapping can be defined by the Mobile program itself (e.g., in settings where self-monitoring programs are produced by a common rewriter or where separately produced programs do not exchange objects) or by the policy-writer (in settings where the mapping must be defined at a system global level for consistency).

The above scheme allows a Mobile program to represent object security states at runtime with a security automaton of `2^{32}` states or less. Each state of the automaton is assigned an integer constant `k`, and history abstraction `H_k` would denote the set of traces that cause the automaton to arrive in state `k`.

### 4.4 Policy Adherence of Mobile Programs

The operational semantics of Mobile presented in \[\[\]\] permit untyped Mobile programs to enter bad terminal states—states in which the Mobile program has not been reduced to a value but no progress can be made. For example, an untyped Mobile program might attempt to load from a non-existent field or attempt to unpack an empty package (in which case no small-step rule can be applied). Mobile’s type system presented in \[\[\]\] prevents both policy violations and bad terminal states, except that it does not prevent unpack operations from being performed on empty packages. This reflects the reality that in practical settings there will always be bad terminal states that are not statically preventable. We prove below that Mobile programs well-typed with respect to a security policy will not violate the security policy when executed even if they enter a bad state.

Formally, we define well-typed by

**Definition 1.** A method `C::m.Sig` with `Sig = ∀\text{in}.(Ψ_{\text{in}}, Fr_{\text{in}}) → ∀\text{out}.(Ψ_{\text{out}}, Fr_{\text{out}}, τ)` is well-typed if and only if there exists a derivation for the typing judgment `Γ_i → m : (Ψ_{\text{in}}, Fr_{\text{in}}) → (Ψ_{\text{out}}, Fr_{\text{out}}, \tau)` where `I = \text{methodbody}(C::m.Sig).

**Definition 2.** A Mobile program is well-typed if and only if (1) for all `C::m.Sig ∈ \text{Dom(methodbody)}`, method `C::m.Sig` is well-typed, and (2) there exists a method `\text{main}::\text{main}.Sig_{\text{main}} ∈ \text{Dom(methodbody)}` with `Sig_{\text{main}} = ∀\text{in}.(Ψ_{\text{in}}, (τ_1, ..., τ_n)) → ∀\text{out}.(Ψ_{\text{out}}, Fr_{\text{out}}, τ_{\text{out}})` such that for all substitutions `σ : θ → τ` and all object identity variables `ε C ∈ (Γ_{\text{in}}, Γ_{\text{out}})`, if `Ψ_{\text{out}}(f) = H` then `σ(H) ⊆ \text{policy}(C).

Part 2 of definition 2 captures the requirement that a Mobile program’s entry method must have a signature that complies with the security policy on exit.

Policy violations are defined differently depending on whether the program terminates normally. If the program terminates normally, Mobile’s type system guarantees that the resulting heap will be policy-adherent; whereas if the program does not terminate or enters a bad state, Mobile guarantees only that the heap at each evaluation step will be prefix-adherent, where policy- and prefix-adherence are defined as follows:

**Definition 3 (Policy Adherent).** A heap `h` is policy-adherent if, for all class objects `obj_{C} ∈ \text{Ring}(h)`, `ε C ⊆ \text{policy}(C).

**Definition 4 (Prefix Adherent).** A heap `h` is prefix-adherent if, for all class objects `obj_{C} ∈ \text{Ring}(h)`, `ε C ⊆ \text{pre(policy}(C))`. 
To formalize the theorem, we first define a notion of consistency between a static typing context and a runtime memory state. We say that a memory store $\psi$ respects an object identity context $\Psi$ and a list of frames $Fr$, written $\Gamma \vdash \psi : (\Psi; Fr)$ if all object fields and stack slots in $\psi$ have values of appropriate types, and the heap in $\psi$ is prefix-adherent. (See Figure 3 for a formal definition.) The following two theorems then establish that well-typed Mobile programs do not violate the security policy.

**Theorem 1 (Terminating Policy Adherence).** Assume that a Mobile program is well-typed, and that, as per Definition 2, its main method has signature $\Sigma_{main} = \forall \leq n. (\Psi_{in}, (\tau_1, \ldots, \tau_n)) \Rightarrow \exists \leq n. (\Psi_{out}, \tau_1, \ldots, \tau_n)$. If $\Gamma \vdash \psi : (\Psi_{in}; Fr)$ holds and if $\psi, \text{methodbody}(C_{main}::\text{main}::\Sigma) \Rightarrow^* (h', s')$ holds, then $h'$ is policy-adherent.

**Proof.** Omitted for brevity. See [16].

**Theorem 2 (Non-terminating Prefix Adherence).** Assume that a Mobile program is well-typed, and assume that $\Gamma \vdash \Gamma' : (\Psi; Fr) \Rightarrow \exists \tau'. (\Psi'; Fr'; \tau)$ and $\Gamma \vdash (h, s) : (\Psi; Fr)$ hold. If $h$ is prefix-adherent and $(h, s), I \Rightarrow^* (h', s')$, $\Gamma'$ holds, then $h'$ is prefix-adherent.

**Proof.** Omitted for brevity. See [16].

An important consequence of both of these theorems is that Mobile can be implemented on existing .NET systems without modifying the memory model to store object traces at runtime. Since a static type-checker can verify that Mobile code is well-typed, and since well-typed code never exhibits a trace that violates the security policy, the runtime system need not store or monitor object traces to prevent security violations.

5. Implementation

Our prototype implementation of Mobile consists of a type-checker for Mobile’s type system extended to the full managed subset of Microsoft’s .NET CIL (minus reflection). The type-checker was written in Ocaml (about one thousand lines of code) and uses Microsoft’s .NET ILX SDK [30] to read and manipulate .NET bytecode binaries. Mobile programs are .NET CIL programs with typing annotations encoded as .NET method attributes. The Mobile type-checker reads these (untrusted) annotations and verifies them in the course of type-checking.

Our implementation allows security policies to identify method calls as security-relevant events. Thus, security policies can constrain the usage of resources provided by the CLR by monitoring CLR method calls and the objects they return. Our type-checker can, in principle, regard any CIL instruction as a security-relevant event, but we leave practical investigation of this feature to future work.

Operations `pack` and `unpack` are implemented as method calls to the (very small) trusted C# library given in Figure 5. Observe that C#’s `lock` construct is used to make both operations atomic. History abstraction values are implemented as integers. Thus, our `newlist` operation is simply a `lock` instruction that loads an integer constant onto the evaluation stack. Policies can statically declare for each integer constant a closed history abstraction that integer represents when used as a runtime state value. Tests of runtime state values consist of equality comparisons with integer constants in the manner described in [14].

As described in [14], this implementation suffices to support IRM’s that model security policies as finite-state security automata.

The type-checker must verify subset relations over the language of history abstractions given in Figure 3. Although deciding subset for $\omega$-regular expressions with variables and intersection is not tractable in general, the task is simplified by observing that real Mobile code only introduces history variables at the beginnings of expressions (when an object is unpacked) and only introduces intersections that involve a variable and a closed history abstraction (when a runtime state value is tested). The resulting sub-language can be decided using a simple regular expression subset algorithm (proof omitted for brevity).

Our type-checker also recognizes method annotations attached to finalizers of security-relevant classes. A finalizer’s precondition must be satisfied whenever an object of its class escapes to the heap (i.e., when it is packed), since at any point after that, its package object could become orphaned and then garbage-collected. By the time a program terminates, all of its objects are guaranteed to satisfy their finalizers’ postconditions, since at that point any remaining objects will be garbage-collected. This allows an IRM to leak security-relevant objects to the heap (in packages) even when they are not yet in a policy-adherent state, as long as the object’s finalizer suffices to restore it to a policy-adherent state once garbage-collection occurs.

To test our implementation, we wrote a simple rewriter like that described in [3] and used it to enforce a security policy that allows each .NET network socket object to accept at most $n$ connections during the program’s lifetime (where $n$ is a parameter specified by the policy-writer). Such a policy might be used, for example, to force applications to relinquish control of network ports after a certain amount of activity. We applied this policy to a small multithreaded webserver written in C#. The original application binary was 20K in size and rewriting did not alter its size. (Padding introduced by the CLI binary format masked the small overhead introduced by additional instructions and annotations.)

Hand-counting the material inserted by the rewriter revealed approximately 83 bytes in additional instructions and 117 bytes in annotations. Rewriting took 0.12 seconds and type-checking took 0.09 seconds on a 1.8GHz Pentium. We benchmarked both the original and rewritten webservers by using WebStone to simulate two clients retrieving live webpages ranging in size from 500 bytes to 5 megabytes. WebStone reported that the rewritten webservice exhibited an average throughput rate that was 99.97% of the original webservice’s.

The aforementioned test is obviously not a definitive evaluation of the feasibility of automated rewriting; much work remains to be done in terms of evaluating the approach on richer policies and applications. However, we consider it to be preliminary evidence that rewriting can be automated in a way that produces acceptable annotations.
6. Conclusions and Future Work
Mobile’s type system and the theorems presented in §6.4 show that a common style of IRM, in which extra state variables and guards that model a security automaton have been in-lined into the untrusted code, can be independently verified by a type-checker, eliminating the need to trust the rewriter that produced the IRM. We verify policies that are universally quantified over unbounded collections of objects—that is, policies that require each object to exhibit a history of security-relevant events that conforms to some stated property. The language of security policies is left abstract and could consist of DFA’s, LTL expressions, or any computable language of finite and infinite event sequences.

Our implementation of Mobile for managed Microsoft .NET CIL expresses security policies as ω-regular expressions. We verify such policies in the presence of exceptions, concurrency, finalizers, and non-termination, demonstrating that Mobile can be scaled to real type-safe, low-level languages.

Our presentation of Mobile has not addressed issues of object inheritance of security-relevant classes. Future work should examine how to safely express and implement policies that require objects related by inheritance to conform to different properties. A type-checker for such a system would need to identify when a type-cast at runtime could potentially lead to a violation of the policy and provide a means for policy-adherent programs to perform necessary type-casts.

Another open problem is how to support a wider range of IRM implementations. Mobile supports only a specific (but typical) treatment of runtime state, wherein each security-relevant object is paired with a dynamic representation of its state every time it is leaked to the heap. In some settings, it may be desirable to implement IRM’s that store an object’s dynamic state differently, such as in a separate array rather than packaged together with the object it models. Type systems for coordinated data structures could potentially be leveraged to enforce invariants over these decoupled objects and states.

We chose a type system for Mobile that statically tracks control flow in a data-insensitive manner, with ω-regular expressions denoting sets of event sequences. This approach is appealing because there is a natural rewriting strategy (outlined in §6.5) whereby well-typed Mobile code can be automatically generated from untrusted CIL code. A more powerful type system could employ a richer language like Hoare Logic [18] to track data-sensitive control flow. This could allow clever rewriters to eliminate additional run-time checks by statically proving that they are unnecessary. However, formulating a sound and complete Hoare Logic for .NET that includes objects and concurrency is challenging; furthermore, the burden of producing useful proofs in this logic would be pushed to the rewriter. Future work should investigate rewriting strategies that could make such an approach worthwhile.

Finally, not every enforceable security policy can be couched as a computable property that is universally quantified over object instances. For example, one potentially useful policy is one that requires that for every file object opened for writing, there exists an encryptor object to which its output stream has been linked. Such a policy is not supported by Mobile because it regards both universal and existentially quantified properties that relate multiple object instances. Future work should consider how to implement IRM’s that enforce such policies, and how these implementations could be type-checked so as to statically verify that the IRM satisfies the security policy.

Acknowledgments
The authors are indebted to Matthew Fluet, Michael Hicks, and Amal Ahmed for their helpful critiques of this paper.

References
The following is a formal statement of Mobile’s typing rules.

The following is a formal statement of Mobile’s typing rules. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 1–12, Snowbird, Utah, June 2001.


Appendix

The following is a formal statement of Mobile’s typing rules.

The following is a formal statement of Mobile’s typing rules. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 1–12, Snowbird, Utah, June 2001.


The following is a formal statement of Mobile’s typing rules.

The following is a formal statement of Mobile’s typing rules. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 1–12, Snowbird, Utah, June 2001.


