FLEX: A Flexible Code Authentication Framework for Delegating Mobile App Customization

Gabriele Costa[†], Paolo Gasti[‡], Alessio Merlo[†], and Shung-Hsi Yu[‡]
[†] DIBRIS – University of Genoa (Italy) [‡] New York Institute of Technology (USA)
gabriele.costa@unige.it, pgasti@nyit.edu, alessio.merlo@unige.it, syu07@nyit.edu

ABSTRACT

Mobile code distribution relies on digital signatures to guarantee code authenticity. Unfortunately, standard signature schemes are not well suited for use in conjunction with program transformation techniques, such as *aspect-oriented programming*. With these techniques, code development is performed *in sequence* by multiple teams of programmers. This is fundamentally different from traditional single-developer/ single-user models, where users can verify end-to-end (i.e., developer-to-user) authenticity of the code using digital signatures. To address this limitation, we introduce FLEX, a flexible code authentication framework for mobile applications. FLEX allows semi-trusted intermediaries to modify mobile code without invalidating the developer's signature, as long as the modification complies with a "contract" issued by the developer. We introduce formal definitions for secure code modification, and show that our instantiation of FLEX is secure under these definitions. Although FLEX can be instantiated using any language, we design AMJ—a novel programming language that supports code annotations—and implement a FLEX prototype based on our new language.

1. INTRODUCTION

In recent years, software development has evolved from a centralized to a distributed activity. Modern development techniques and paradigms emphasize multiple code contributors, often working "in series" by adding further functionalities, components, and refinements to an application. Prominent examples of distributed development paradigms include *Aspect Oriented Programming* [15] (AOP), *Reflection* [24], and *Contract-driven development* [18]. Because of the flexibility of these approaches, and because they fit well within the BYOD paradigm, the research community has started to apply distributed development approaches to smartphone software [2].

Smartphones have traditionally relied on closed marketplaces for code distribution. This model involves three parties: (*i*) one or more *developers*, who builds smartphone apps in its entirety; (*ii*) the *app marketplace* (e.g., the Google Play Store [12], the Apple App Store [1], and the Firefox Marketplace [9]), which distributes smartphone apps; and (*iii*) the *user* who runs apps downloaded from the app marketplace. As a prominent example of this model, the Google Play Store guarantees app authenticity by requiring that all apps are cryptographically signed by their respective developers [23]. This prevents code modifications, because it allows users to verify end-to-end (i.e., developer-to-user) app authenticity.

Unfortunately, this approach to code authentication is not well suited for distributed development processes. By having multiple independent developers who contribute code at different points in time, each (legal) code modification invalidates all previously issued signatures.

Further, the current code authentication approach prevents app marketplaces from implementing benign code modifications. For instance, Armando et al. [2] introduced the notion of *meta-market*—an entity that redistributes mobile apps to a group of federated mobile devices. The meta-market performs security analysis of apps and, if needed, refines the apps' code to neutralize possible vulnerabilities, and to add code instrumentation. However, by modifying the application's code, the meta-market invalidates the developer's signature.

Any modification implemented by the meta-market requires a new signature, which can be issued by either the developer or the meta-market itself. We argue that neither option is satisfactory. Clearly, requiring the developer to review and sign potentially hundreds of different modifications is not sustainable. On the other hand, replacing the developer's signature with a new one from the meta-market prevents the user from performing end-to-end app authentication, and gives the developer no control on which modifications are performed by the meta-market. In addition to security issues, this approach can potentially raise both legal (e.g., *does any modifications violate the developer's license agreement?*) as well as technical concerns (*how can modifications be implemented reliably when the app's source code is not available?*).

To address these problems, in this paper we introduce FLEX, a flexible code authentication framework. FLEX allows: (*i*) the developer to define constraints on modifications of his mobile apps, (*ii*) a third party (e.g. the meta-market) to perform targeted modifications and, (*iii*) the user to verify end-to-end app authenticity. In addition, it lets the user check which modifications have been applied to an app.

As a proof of concept, and to provide formal proofs on the framework's properties, we developed a simple programming

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
 ISBN 978-1-4503-4233-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897887

language called *Annotated Middleweight Java* (AMJ), which extends Middleweight Java [4]. At the core of AMJ there are *rewriting rules*: the developer annotates the app's source code using these rules, which specify legal modifications that can be implemented by the meta-market. Annotations have no effect on the semantics of the app at runtime, and are ignored by the execution environment. We emphasize that FLEX can be instantiated with languages other than AMJ. In fact, rewriting rules equivalent to those presented in this work can be developed for any languages that supports late binding, including Java and C#.

In the rest of this paper, we denote the list of all annotations in an app as contract. The developer signs the app and the contract, and then sends the resulting package to the app marketplace for distribution. The meta-market retrieves the app from the marketplace, and uses the contract to determine which modifications can be implemented without invalidating the developer's signature. Once the meta-market has implemented its modifications, it sends the app, the contract, and the modifications to the user. Upon receipt, the user is able to verify the authenticity of the original app, and that the modifications implemented by the meta-market comply with the developer's contract.

The proposed approach has the following benefits: (*i*) developers can easily enforce restrictions on meta-market modifications. The impact of those modification on the development of the app is limited; (*ii*) the meta-market can safely implement modifications according to the specifications provided by the contract; (*iii*) users can verify the integrity of the developer's code, as well as the compliance of the modifications carried out by the meta-market with the restriction imposed by the developer. Moreover, because the original code and the contract are signed by the developer, the user and the meta-market can keep the developer accountable if the application does not work properly; and (*iv*) FLEX introduces no additional overhead during app execution: all checks are performed by the user *before* installing the app.

Organization. The rest of the paper is organized as follows. Section 2 presents a case study. Section 3 reviews related work. In Section 4 we introduce our system and adversary model. Section 5 presents our programming model and defines AMJ. We show how code and annotations are signed and how AMJ is used to guarantee the validity of applications in Section 6. Section 7 presents our prototype. We conclude in Section 8.

2. UNIVERSAL REMOTE: A CASE STUDY

To highlight the benefits of FLEX, we consider a *universal remote* as a case study for our approach. A universal remote is a smartphone app that can control a wide variety of devices, including smart lights, HVAC, garage doors, smart deadbolts, electric shades, and kitchen appliances. Example of universal remotes include Google's OpenHAB [21] and the Wink app [27]. An important challenge when developing a universal remote is to provide support for a large number of protocols, required by different classes of smart devices.[1]

These protocols vary, among other things, in their security requirements. For example, a universal remote connected to the same WiFi as a smart lightbulb should be allowed to turn the light on or off. However, the same universal remote might not be authorized to unlock arbitrary smart deadbolts on the same network.

Ideally, the developer of a universal remote app should not be concerned with implementing each individual protocol. Instead, vendors should be responsible for adding appliance-specific code to the universal remote, without invalidating the app signature.

In this section, we argue that FLEX is well suited to securely enable this model. To do so we discuss how a simplified universal remote, which supports only "on" and "off" commands, can be implemented using our framework. The following AMJ code represents a "toy" universal remote class:[2]

```
1  class URemote {
2
3    Device d;
4
5    // ...
6
7    void on() {
8      Message m, r in {
9        m = new Message(true);
10       this.d.send(m);
11       r = this.d.receive();
12       if(!r.isACK()) { this.prompt(...); }
13     }
14   }
15 }
```

Class `URemote` has a field `d` that represents a device connector, i.e., an object that is used to exchange messages with the device. Among the methods included in `URemote`, we highlight one that is used to turn on the device (`void on()`), and another that is (possibly) invoked to request the user PIN (`Message askPIN()`).

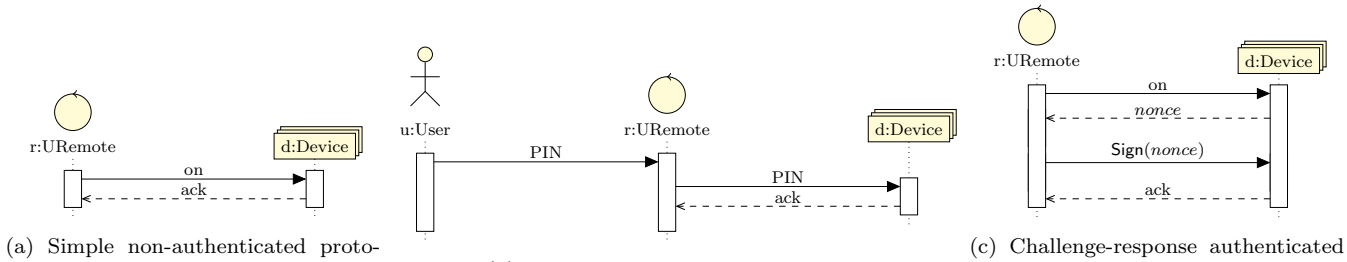
A generic interaction between the universal remote and a device, implementing the protocol in Figure 1a, includes the following steps. The remote creates a new `Message` object that indicates that device `d` should be turned on (line 9). Then, the message is sent to the device (line 10), which returns a message (`r`, line 11) that indicates whether the command was executed successfully. Otherwise, the universal remote handles negative responses at line 12.

Because of the lack of authentication, this code is only suitable for controlling non-security-critical devices. Figures 1b and 1c exemplify two of the many protocols suitable for devices that require authentication. The former is based on a user-provided PIN, while the latter uses a cryptographic challenge-response mechanism. Each smart device will implement one of many variants of these or possibly other protocols.

Allowing the manufacturer to provide a "plugin" or a "device driver", which implements a device-specific protocol, addresses this issue only from a functionality standpoint. We believe that this plug-in- or driver-based approach is far from ideal in terms of security and vendor/developer accountability. In fact, the driver provided by the device manufacturer must be allowed to run together with (or in place of) the app code, and therefore requires the user to trust *both* the app developer and the manufacturer. Additionally, the developer might not be able to specify meaningful restrictions on the driver's behavior.

[1]For example, OpenHAB includes support for tens of protocols.http://www.openhab.org/features-tech.html

[2]The syntax of AMJ is formally introduced in Section 4. However, for this example, the reader may assume a Java-like syntax.



(a) Simple non-authenticated protocol.

(b) PIN-based authenticated protocol.

(c) Challenge-response authenticated protocol.

Figure 1: Three alternative protocols for the universal remote.

We believe that FLEX represents a satisfactory approach for addressing both the functionality and security aspect of this and other related use cases. With FLEX, the developer annotates the code, marking not only which section can be modified by the vendor (who in this context acts as meta-market), but also which restrictions must be enforced on the vendor's code. Upon receiving the universal remote app from the vendor, the user verifies that the original app code has not been illegally modified, and that the legal modifications are valid according to the developer's specifications. We detail this approach in the rest of the paper, and complete this use case in Section 6.

3. RELATED WORK

Proof-carrying Code. Proof-carrying code [20] is a method for augmenting an application with a formal proof (either manually or automatically generated) that guarantees that the app adheres to a set of rules. The proof can be verified automatically, and therefore does not impact usability. This makes proof-carrying code very useful, especially when the security properties can be completely specified by the user within the language supported by the proof framework, and the application can function correctly within these restrictions. Nevertheless, this approach has major limitations in the scenario highlighted in this paper. It is in fact very unlikely [22] that code producers and consumers will agree on a specific set of properties. As a consequence, a variant called *model-carrying code* [22] has been more successful. Model-carrying code consists in instrumenting a model of the application behavior instead of providing a proof of compliance. Although this approach is more flexible than the one based on proof-carrying code, the parties must still agree on the elements appearing in the model.

Secure Meta-Market. Secure meta-market is an application distribution paradigm proposed by Armando et al. [2] to enforce "Bring Your Own Device" (BYOD) security policies on personal mobile devices. A meta-market stands between the app marketplace, the user's organization and the device owner, allowing the organization to automatically enforce BYOD security policies on mobile devices. The enforcement of a policy usually requires modifications to the app (e.g., to *instrument* it). The modifications are performed by the meta-market, and have the side effect of invalidating the developer's signature. This requires the meta-market to re-sign the app before it can be installed on mobile devices.

The meta-market model has been recently adopted outside the BYOD context. For instance, Cassandra [17] is a meta-market architecture that verifies whether Android applications comply with the user's privacy policy. Cassandra allows users to restrict installed applications to those that comply with a particular security policy. Users do not need to trust the meta-market because applications carry their own proof of compliance.

Code Rewriting. The current literature includes several paradigms for program transformation. Among them, Aspect-Oriented Programming [14] (AOP) and reflection [26] are probably the most commonly used. Informally, an aspect consists of a fragment of code and a rewriting rule. When aspects are defined, a program can be modified by inserting invocations to the aspects' code. Also, using Aspect Weaving [5] the fragments carried by the aspects are directly injected in the application code. Instead, reflection allows programs to manipulate their own elements (e.g., procedures and classes) through specific APIs and data structures. Both AOP and reflection are compatible with our approach, and can be used for implementing a program transformation framework similar to FLEX. However, we believe that code annotations contained within comments are easier to understand and use under our assumptions of programming under a contract (see below).

The idea of using comments for annotating programs has been proposed before. For instance, the Java modeling language [16] (JML) allows a developer to attach specifications to her code as comments. A specification can serve for many purposes, such as automatic verification, and contract-based software design. Extending JML with the syntax of our annotation language is feasible and allows the integration of FLEX with a state-of-the-art specification language. However, since we target mobile code, BML [8] is a better choice, as it implements JML specifications at the byte-code level.

Redactable and Sanitizable Signatures. Redactable signatures [13] allow an authorized semi-trusted party to obtain a valid signature from a redacted document without any interaction with the original signer. Unfortunately, redactable signatures are not a viable tool for *adding* code to signed applications. In fact, redactable signatures only support removal of document components.

A more promising approach consists in using sanitizable signatures [3, 7]. Sanitizable signatures allow authorized semi-trusted parties to modify parts of a signed message in a limited fashion. In our scenario, this includes adding and removing code from the mobile app without invalidating the original signature. Although this approach might sound appealing, it has two critical drawbacks: (*i*) by using sanitizable signatures, the developer is able to indicate what parts of the code can be modified, but cannot impose any restriction on the modifications. Because the language

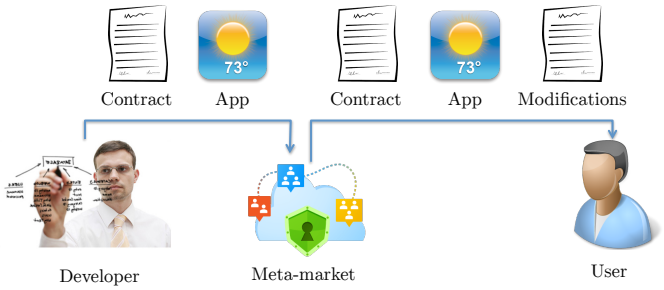


Figure 2: Parties and their interaction.

used by the developer (and therefore by the meta-market) is Turing-complete, arbitrary code injection performed by the meta-market implies virtually no restrictions on the resulting modified code; and (*ii*) the user cannot reliably and securely determine the exact modifications implemented by the meta-market. Hence, there can be no end-to-end trust between the developer and the user.

4. SYSTEM AND ADVERSARY MODEL

In this section, we present the components of our system, and discuss their interaction. Our system includes a developer, a meta-market, and a user (see Figure 2). The developer creates an annotated source code in AMJ, and compiles it using the FLEX compiler. The result of this process is a standard (signed) smartphone application, and a contract. The contract is automatically generated from the code annotations, and specifies which parts of the application can be modified by the meta-market, and how. (The language of our code annotations is discussed in Section 5.4).

The meta-market is in charge of implementing modifications to the developer's app, and to distribute the resulting code to the users. App modifications, also written using AMJ, are distributed in source format, and are signed by the meta-market. After receiving the original app, the contract and the list of modifications from the meta-market, the user verifies all signatures, and checks if the meta-market's modifications comply with the contract. If they do, the user runs a small tool that compiles the modifications and merges them with the original app.

Integrity and authenticity of the contract, the app, and the modifications are guaranteed using a standard signature scheme. The developer and the meta market have access to their respective private (signing) keys sk_{dev} and sk_{mm}, while all parties have access to all public keys.

More formally, let app be an application, contract a contract that lists valid modifications to app, and mods a set of modifications that can be applied to app. The developer issues tuple $D = (\mathsf{app}, \mathsf{contract}, \gamma_D)$, where γ_D is auxiliary information on D (e.g., a signature on app and contract), while the meta-market generates tuple $M = (\mathsf{app}, \mathsf{contract}, \mathsf{mods}, \gamma_M)$ where γ_M is auxiliary information on M. The meta-market runs algorithm $\mathsf{check}_{mm}(D, pk_{dev})$, which returns 1 if the application and the contract are valid and have been constructed by a honest developer (i.e., all signatures verify), and 0 otherwise. The user has access to algorithm $\mathsf{check}_{user}(M, pk_{dev}, pk_{mm})$, which returns 1 if $\mathsf{check}_{mm}(D, pk_{dev}) = 1$ and if the modifications in M are valid with respect to app and contract, and have been produced by the meta-market—and 0 otherwise.

We assume that the adversary can be internal (i.e., one of the protocol participants) or external. In the former case, we consider a malicious meta-market. This meta-market is willing to *covertly* perform modifications to the mobile application in violation of the developer's contract. In the latter, we consider an adversary who relays messages between parties, and can therefore modify them in transit. Although this can be addressed, in some cases, by using tools such as TLS/SSL, there are scenarios in which this is not possible. For example, the meta-market might store applications on a cloud server that is used for distributing applications. If the adversary is able to subvert the cloud server, it can perform arbitrary modifications to the apps or to the contract before they are retrieved by the user.

Internal Adversaries. We allow a malicious meta-market to arbitrarily deviate from the intended behavior by implementing any feasible (i.e., polynomial-time) strategy. The goal of the meta-market is to construct a tuple $M_i = (\mathsf{app}_i, \mathsf{contract}_i, \mathsf{mods}_i, \gamma_{M_i})$ such that $\mathsf{check}_{user}(M_i, pk_{dev}, pk_{mm})$ returns 1, and $D_i = (\mathsf{app}_i, \mathsf{contract}_i, \gamma_{D_i})$ was never issued by the developer. In other words, the malicious meta-market wants to surreptitiously construct a valid application, contract, and set of modifications such that either the applications or the contract (or both) have not been generated by the developer, or the modifications do not match with the contract and the application.

To formally define internal adversaries, we introduce the Forging Application Attack (FAA):

Experiment $\mathsf{FAA}_{\mathcal{A}}(\kappa)$

1. $\mathcal{A}$ receives pk_{dev} and (sk_{mm}, pk_{mm}), and adaptively requests arbitrary tuples $D_i = (\mathsf{app}_i, \mathsf{contract}_i, \gamma_{D_i})$ to the honest developer.

2. Eventually, $\mathcal{A}$ outputs $M^* = (\mathsf{app}^*, \mathsf{contract}^*, \mathsf{mods}^*, \gamma_{M^*})$. The experiment outputs 1 if and only if $\mathsf{check}_{user}(M^*, pk_{dev}, pk_{mm}) = 1$ and $D^* = (\mathsf{app}^*, \mathsf{contract}^*, \gamma_{D^*})$ was never issued by the developer. Otherwise, the experiment outputs 0.

Definition 1 (FAA-security). *A FLEX instantiation is secure under Forging Application Attack if there exists a negligible function* negl *such that for any PPT* $\mathcal{A}$, $\Pr[\mathsf{FAA}_{\mathcal{A}}(\kappa) = 1] \leq \mathsf{negl}(\kappa)$.

External Adversaries. In this scenario, an external malicious party is allowed to perform arbitrary modifications to the messages exchanged by the protocol participants. Informally, the goal of the malicious party is to provide a tuple $M_i = (\mathsf{app}_i, \mathsf{contract}_i, \mathsf{mods}_i, \gamma_{M_i})$ to the user such that either $D_i = (\mathsf{app}_i, \mathsf{contract}_i, \gamma_{D_i})$ was not issued by the developer, or mods_i was not issued by the meta-market, or both, and such that $\mathsf{check}_{user}(M_i, pk_{dev}, pk_{mm}) = 1$. To formalize external adversaries, we introduce the Application Poisoning Attack (APA):

Experiment $\mathsf{APA}_{\mathcal{A}}(\kappa)$

1. $\mathcal{A}$ receives pk_{dev} and pk_{mm}. It then adaptively requests arbitrary tuples $D_i = (\mathsf{app}_i, \mathsf{contract}_i, \gamma_{D_i})$ to the developer, and sends tuples $D_j = (\mathsf{app}_j, \mathsf{contract}_j, \gamma_{D_j})$ and modifications mods_j to the meta-market, which returns $M_j = (\mathsf{app}_j, \mathsf{contract}_j, \mathsf{mods}_j, \gamma_{M_j})$ if $\mathsf{check}_{mm}(D_j, pk_{dev}) = 1$, and $\perp$ otherwise.

Table 1: Syntax of MJ

```
L ::= class C extends D { C̄ f̄; K M̄ }
K ::= C(C̄ x̄) { s }
M ::= C m(C̄ x̄) { s }
e ::= null | x | e.f | e.m(ē) | new C(ē) |
      (C)e | /*@ER@*/ e
s ::= skip; | if (e1 == e2) {s1} else {s2} |
      e.f = e′; | x = e; | C x in {s} |
      s1 s2 | return e; | /*@SR@*/ s |
      /*@CR@*/ s
```

Abbreviations:

$\texttt{true} \triangleq \texttt{null}$

$\texttt{false} \triangleq \texttt{new Object()}$

$\texttt{if (e) \{s1\} else \{s2\}} \triangleq \texttt{if (e == null) \{s1\} else \{s2\}}$

$\texttt{if (!e) \{s1\} else \{s2\}} \triangleq \texttt{if (e) \{s2\} else \{s1\}}$

$\texttt{if (...) \{s\}} \triangleq \texttt{if (...) \{s\} else \{skip;\}}$

$\texttt{s}_1 \ldots \texttt{s}_k \triangleq \texttt{s}_1\ (\texttt{s}_2\ (\cdots\ \texttt{s}_k) \cdots)$

$\texttt{\{s\}} \triangleq \texttt{C x in \{s\}}$ (where $\texttt{x} \notin fv(\texttt{s})$)

$\texttt{C x; C y in \{s\}} \triangleq \texttt{C x in \{ C y in \{s\}\}}$ (with $\texttt{x} \neq \texttt{y}$)

2. Eventually, $\mathcal{A}$ outputs $M^* = (\mathsf{app}^*, \mathsf{contract}^*, \mathsf{mods}^*, \gamma_{M^*})$. The experiment outputs 1 iff $\mathsf{check}_{user}(M^*, pk_{dev}, pk_{mm}) = 1$ and $D^* = (\mathsf{app}^*, \mathsf{contract}^*, \gamma_{D^*})$ was never issued by the developer, or M^* was never issued by the meta-market, and 0 otherwise.

Definition 2 (APA-security). *A FLEX instantiation is secure under Application Poisoning Attack if there exists a negligible function* negl *such that for any PPT* $\mathcal{A}$, $\Pr[\mathsf{APA}_{\mathcal{A}}(\kappa) = 1] \leq \mathsf{negl}(\kappa)$.

In the rest of the paper we define how contract, mods, γ_M, and γ_D are constructed, and how check_{user} and check_{mm} are computed in order to guarantee that the aforementioned security properties hold.

5. PROGRAMMING MODEL

In this section we present our programming framework for developing annotated applications.

5.1 Annotated Middleweight Java

Middleweight Java (MJ) is an object-oriented imperative programming language proposed by Bierman et al. [4]. The main goal of MJ is to provide a compact—yet expressive—subset of the features of Java. With FLEX we introduce an extension of the syntax of MJ, called AMJ, in which some elements of the language can be also *annotated* with patterns for term rewriting. The syntax of MJ is given in Table 1.

With AMJ, a new class `C` is defined by specializing an existing class `D` (`Object` is the predefined, empty class). Each class definition consists of three elements: (*i*) a list of typed fields $\overline{\texttt{C}}\ \overline{\texttt{f}}$; (*ii*) a constructor `K`; and (*iii*) a list of methods $\overline{\texttt{M}}$. The constructor of a class `C` consists of a list of typed parameters $\overline{\texttt{C}}\ \overline{\texttt{x}}$ and a statement `s`. Each method has a name `m`, a return type `C`, a list of parameters (e.g., constructors) and a statement `s`. An expression can be the constant `null`, a variable `x`, a field `e.f`, a method invocation `e.m(ē)`, an object constructor `new C(ē)`, a class cast `(C)e` or an annotated expression `/*@ER@*/e`. A statement is either: an effect-free command `skip;`, a conditional branch `if (e1 == e2) {s1} else {s2}`, a field assignment `e.f = e′;`, a variable assignment `x = e;`, a block `C x {s}`, a sequence `s1 s2`, a return instruction `return e;`, or an annotated statement `/*@SR@*/s` and `/*@CR@*/s`.

Annotations for expressions and statements are discussed in Section 5.4. In the rest of this section, the reader can consider annotations as code comments, which have no effects on the execution of programs.

To improve the readability, we introduce some syntactic sugar. We write `void` in method signature when its return type is irrelevant. Also we use expression `e.m(ē)` as a statement in place of `C x in {x = e.m(ē);}`.

Example 1. *A subset of the class* `Message` *can be defined as follows.*

```
class Message extends Object {
  Object v;
  Message(Object u) {this.v = u;}
  Object isACK() {
    if(this.v == null)
      {return null;}
    else
      {return new Object();}
  }
}
```

A `Message` *has a payload represented by* `v`*. The method* `isACK` *returns* `null` *when the payload is* `null`*. Otherwise it returns a* `new Object()`*. (Recall that we treat* `null` *as the boolean value* `true`*.)*

5.2 Operational Semantics

The operational semantics of AMJ is defined in terms of transitions between *configurations*, which represent the current state of a running program.

Definition 3. *A* configuration *is a tuple* (E, H, F) *where*

- $E : \mathbb{V} \rightharpoonup \mathbb{O} \cup \{\textbf{null}\}$ *is a variable environment mapping variables into object identifiers (*o*,* o'*), or* **null***;*
- $H : \mathbb{O} \rightharpoonup (C, E)$ *is a heap function mapping object identifiers into object records;*
- $F ::= \texttt{s} \mid \texttt{e} \mid o \mid \textbf{null} \mid \bullet$ *is a term.*

We write $E(x) = \bot$ *when* $x \notin dom(E)$. $E|_{\texttt{x}}$ *is the environment that assigns* $\bot$ *to* `x` *and otherwise behaves as* E.

A terminal configuration *is a configuration* (E, H, u) *where* u *is either a* value *(i.e., a pointer* o *or* `null`*), or the void element* $\bullet$.

The operational semantics of expressions and statements is presented in Appendix B. An expression `null` is reduced to the constant **null** (rule (EE-Null)), a variable `x` is evaluated to the value v provided by the variable environment E (rule (EE-Var)), while a field access `e.f` results in a value v if `e` can be evaluated to a pointer o, associated to a record $(\texttt{C}, E)$ such that (*i*) `C` is a class declaring a field `f` and (*ii*) E assigns v to `f`. As a side effect, the evaluation of `e` might result in a new heap H'.

A method invocation (EE-Mth) `e.m(ē)` consists in evaluating whether `e` (or a value o which `e` evaluates to) points to a record $(\texttt{C}, E)$ such that `C` declares a method `m`, with formal parameters $\bar{\texttt{x}}$ and body `s`.[3] Then, the n expressions of $\bar{\texttt{e}}$ (i.e.,

[3]Additionally, we require that the number of actual parameters is the same as that of the formal parameters, which amounts to require that $\bar{\texttt{x}}$ and $\bar{\texttt{e}}$ have the same length.

$\mathtt{e}_1, \ldots, \mathtt{e}_n$) are evaluated to obtain the corresponding values $v_1, \ldots, v_n$. Each evaluation can result in a transformation of the heap, therefore leading to the chain $H_1 \ldots H_n$. The body of `m` is evaluated after adding the mappings between the parameters' names $\bar{\mathtt{x}}$ and values $\bar{v}$ to E. The resulting configuration (E'_o, H', v) defines the heap H' and the value v of the result of the application of rule (EE-Mth). After applying the rule, the environment E'_o is dropped, and is replaced by the external environment E.

A constructor `new C(`$\bar{\mathtt{e}}$`)` follows rule (EE-New). This rule requires the evaluation of the n expressions of $\bar{\mathtt{e}}$ (as for method invocations). Then, it creates a new environment E_o by initializing the fields of `C` mapping parameter names to the values computed up to this point, and assigns a fresh pointer o to the reserved name `this`. Finally, it adds a new record $(\mathtt{C}, E_o)$ to the heap and evaluates the body `s` of the constructor under E_o, until termination (because no return value is allowed for constructors). The resulting configuration consists of the original environment E, the heap modified by the execution of `s` and the pointer o. The cast expression `(C)e` is defined through rule (EE-New). The behavior on `(C)e` is similar to that on `e`, except for the side condition requiring `e` to return a pointer associated to a subtype of `C`.

Rules for statements are shown in the second part of Appendix B. A `skip;` command does not change E and H and reduces to the terminal symbol $\bullet$. The `return e;` statement results in the evaluation of `e` and a final configuration where the resulting value v is returned. Conditional statements require the evaluation of the expressions in their guards. If the comparison between the returned values succeeds (SE-If$_{\mathrm{tt}}$), the conditional behaves like the first statement $\mathtt{s}_1$. Otherwise (rule (SE-If$_{\mathrm{ff}}$)) $\mathtt{s}_2$ is executed. Provided that `x` is defined in the current environment E, an assignment (SE-Asgn) creates a binding between `x` and the value obtained by evaluating `e`. Also, the assignment succeeds only if the declared type (function $DType$) of `x`, i.e., the type appearing in the declaration of the variable, is compatible with the actual type (function $VType$) of the value v. Details on the types and the subtyping relation are provided in Section 5.3 as well as in Appendix A. Field assignments (rule (SE-Fld)) behave similarly, with the exception that expression `e` must evaluate to a pointer which refers to a record of `C` declaring a field `f`. If this is the case, the record is updated with the new values v for `f`. A block (rule (SE-Blk)) behaves like its body `s`, with the additional side effect that a new definition for `x` is added to E when entering the block, and then removed when leaving it.[4] Sequences consist of the execution of $\mathtt{s}_1$, possibly followed by $\mathtt{s}_2$. If $\mathtt{s}_1$ reduces to $\bullet$ (rule (SE-Seq$_c$)), then $\mathtt{s}_2$ is executed. Otherwise (rule (SE-Seq$_r$)) the value generated by $\mathtt{s}_1$ is returned.

Example 2. *Consider the class* `Message` *from Example 1. We show the execution of statement* $\mathtt{s} \triangleq$ `Message x in { x = new Message(null); }` *starting from the configuration* $(\emptyset, \emptyset, \mathtt{s})$.

$$\dfrac{\dfrac{\dfrac{([\mathtt{x} \setminus \mathbf{null}], \emptyset, \mathtt{null}) \rightarrow ([\mathtt{x} \setminus \mathbf{null}], \emptyset, \mathbf{null}) \quad (A)}{([\mathtt{x} \setminus \mathbf{null}], \emptyset, \mathtt{new\ Message(null)}) \rightarrow ([\mathtt{x} \setminus \mathbf{null}], H'_o, o)}}{([\mathtt{x} \setminus \mathbf{null}], \emptyset, \mathtt{x = new\ Message(null);}) \rightarrow ([\mathtt{x} \setminus o], H'_o, \bullet)}}{(\emptyset, \emptyset, \mathtt{s}) \rightarrow (\emptyset, H'_o, \bullet)}$$

[4]Operator $|_{\mathtt{x}}$ must remove all the entries for `x` created inside the body of the block from E.

where (A) *stands for the execution of the body of the constructor of* `Message`*, that is:*

$$\dfrac{\begin{array}{c}([\mathtt{u} \setminus \mathbf{null}] \circ E_o, H_o, \mathtt{this}) \rightarrow ([\mathtt{u} \setminus \mathbf{null}] \circ E_o, H_o, o) \\ ([\mathtt{u} \setminus \mathbf{null}] \circ E_o, H_o, \mathtt{u}) \rightarrow ([\mathtt{u} \setminus \mathbf{null}] \circ E_o, H_o, \mathbf{null})\end{array}}{\begin{array}{l}([\mathtt{u} \setminus \mathbf{null}] \circ E_o, H_o, \mathtt{this.v = u;}) \rightarrow \\ \quad ([\mathtt{u} \setminus \mathbf{null}] \circ E_o, [o \setminus (\mathtt{Message}, [\mathtt{v} \setminus \mathbf{null}] \circ E_o)] \circ H_o, \bullet)\end{array}}$$

To simplify presentation, we use abbreviations $E_o \triangleq [\mathtt{this} \setminus o, \mathtt{v} \setminus \mathbf{null}]$, $H_o \triangleq [o \setminus (\mathtt{Message}, E_o)]$ *and* $H'_o \triangleq [o \setminus (\mathtt{Message}, [\mathtt{v} \setminus \mathbf{null}] \circ E_o)] \circ H_o$.

Following the rules of the operational semantics, a configuration can get *stuck*, i.e., it is not a final configuration, and it admits no further reductions according to the next definition:

Definition 4. *A configuration* (E, H, F) *is said to be* stuck *if* F *is not a value and* $\nexists\, E', H', F'$ *such that* $(E, H, F) \rightarrow (E', H', F')$. *We denote a stuck configuration as* $(E, H, F) \not\rightarrow$.

5.3 AMJ Type System

In this section we present the type system of AMJ. We begin by defining the basic elements:

Definition 5. *Types are defined as follows.*
Expression types: $\tau, \tau' ::= \mathtt{C} \mid \top$
Statement types: $\sigma, \sigma' ::= \tau \mid \mathtt{void}$
Method types: $\mu, \mu' ::= \tau_1 \times \ldots \times \tau_n \rightarrow \tau$

The type of an expression can be either `C` or the value $\top$. Apart for the expression types, statements can also be typed to `void`. Methods admit arrow type from input types $\tau_1, \ldots, \tau_n$ to output type τ.

Typing judgements have the form $\Delta; \Gamma \vdash t : \omega$ where $t \in \{e, s\}$ and $\omega \in \{\tau, \sigma\}$. We use Γ and Δ to denote type environments for variables and classes, respectively. A variable environment can be either the empty environment ($\emptyset$) or the environment obtained by adding a mapping to an existing one (e.g., $\Gamma, x : \tau$). Instead, Δ consists of an immutable mapping between method and field names (unambiguously identified through their class name) and their declared type (we use functional types $\cdot \rightarrow \cdot$ for methods). For instance, $\Delta(C)(\mathtt{m}) = C \times D \rightarrow D$ denotes that class C declares a method `m` which has two arguments of type C and D and returns an object of type D.[5]

Typing rules for expressions and statements are presented in Appendix C. Expressions are typed as follows. The type of the `null` constant is the top element $\top$ (TE-Null), while the type of a variable x is provided by the current type environment Γ (TE-Var). The weakening rule (TE-Wkn) allows for typing an expression to τ' if it can be typed to τ, being a τ a subtype of τ'. Types of fields (TE-Fld) and methods (TE-Mth) are given by the definition of `C`, contained in Δ, as far as the base expression has type `C` (and the actual parameters of a method have compatible types). A constructor of `C` (TE-New) behaves similarly to a method invocation, except for the return type which is `C` itself. The cast operation (TE-Cst) types an expression

[5]For the sake of presentation, here we omit details on method typing. In general, we assume that the type that Δ associates to a method is always correct w.r.t. the method body. For more details, we refer the reader to [4].

Table 2: Rewriting rules syntax

```
ER ::= erew SX; EX | erew EX
EX ::= e | EX + EX
SR ::= srew SX
SX ::= s | SX + SX
```

to C if it can be typed to the subclass C'. Finally, (TE-Erew) states that if an annotated expression is typed to τ, then it can be typed to both its base expression and to the annotation (annotation typing is discussed in Section 5.4).

With respect to statements, a `skip;` command (TS-Skip) has type `void`, while a `return` statement (TS-Ret) has the same type of the returned expression. A conditional statement (TS-If) is typed to σ if both its branches are typed to σ. Assignments to a field (TS-Fld) or variable (TS-Asgn) are typed to `void` as far as the assigned expression has a type which is compatible with that of the identifier, i.e., `f` and `x`, respectively. The type of the identifiers is provided by the environment functions Δ and Γ, respectively. A block (TS-Blk) has the same type as the statement it contains (typed under an environment which defines variable `x`). The weakening rule (TS-Wkn) behaves similarly to the rule for expressions, i.e., it allows to type a statement to a more general type σ'. The rules for sequences behave as follows. Rule (TS-Seq$_1$) says that a sequence is typed to σ (with $\sigma \neq$ `void`) if so can be typed the two sub-statements. Also, rule (TS-Seq$_2$) states that a sequence is typed as its second statement if the first one is typed to `void`. Finally, rule (TS-Srew) says that we can type an annotated statement to σ whenever its annotation and base expression can be typed so.

Example 3. *Consider once again the class* `Message` *from Example 1. We type the statement* `s` *of Example 2 as follows.*

$$\text{(TE-New)}\ \frac{}{\Delta; x : \texttt{Message} \vdash \texttt{null} : \top}$$

$$\text{(TS-Asgn)}\ \frac{\Delta; x : \texttt{Message} \vdash \texttt{new Message(null)} : \texttt{Message}}{\Delta; x : \texttt{Message} \vdash \texttt{x} = \texttt{new Message(null);} : \texttt{void}}$$

$$\text{(TS-Blk)}\ \frac{\Delta; x : \texttt{Message} \vdash \texttt{x} = \texttt{new Message(null);} : \texttt{void}}{\Delta; \emptyset \vdash \texttt{s} : \texttt{void}}$$

When applying rule (TE-New)*, the typing procedure also verifies that* $\Delta(\texttt{Message})(\texttt{new}) = \texttt{Object} \to \texttt{Message}$.

An important property of typed terms is that they do not lead to stuck configurations. In fact, for all *closed* (i.e., containing no free variables) terms $t \in \{e, s\}$, if t is typed to ω, then t does not get stuck, and the value obtained when running t is of type ω (or a subtype of ω). In other words, typed terms do not cause faulty computations, and always return values of the expected type. These properties are formalized in Appendix A under Theorem 1.

5.4 Code Annotations

In this section we extend the type system of AMJ with rules for code annotations. The syntax of the annotations for defining rewriting rules is presented in Table 2.

Expression rewriting annotations `ER` can be either a statement rewriting term `SX` followed by an expression rewriting term `EX`, or simply an expression rewriting term `EX`. Expression rewriting terms `EX` can either be an expression `e`, or the union/choice of two sub terms `EX + EX`. Statement rewriting annotations only consist of a statement rewriting term `SX`. These terms can be either a simple statement `s`, or the union of two terms `SX + SX`.

Table 3: Annotation typing rules.

$$\text{(TER-Erew}_1\text{)}\ \frac{\Delta; \Gamma \vdash \texttt{EX} : \tau \qquad \Delta; \Gamma \vdash \texttt{SX} : \texttt{void}}{\Delta; \Gamma \rhd \texttt{erew SX; EX} : \tau}$$

$$\text{(TER-Erew}_2\text{)}\ \frac{\Delta; \Gamma \vdash \texttt{EX} : \tau}{\Delta; \Gamma \rhd \texttt{erew EX} : \tau}$$

$$\text{(TER-Exp)}\ \frac{\Delta; \Gamma \vdash \texttt{e} : \tau}{\Delta; \Gamma \rhd \texttt{e} : \tau}$$

$$\text{(TER-Sum)}\ \frac{\Delta; \Gamma \vdash \texttt{EX}_1 : \tau \qquad \Delta; \Gamma \vdash \texttt{EX}_2 : \tau}{\Delta; \Gamma \rhd \texttt{EX}_1 + \texttt{EX}_2 : \tau}$$

$$\text{(TSR-Srew)}\ \frac{\Delta; \Gamma \vdash \texttt{SX} : \sigma}{\Delta; \Gamma \rhd \texttt{srew SX} : \sigma}$$

$$\text{(TSR-Stm)}\ \frac{\Delta; \Gamma \vdash \texttt{s} : \sigma}{\Delta; \Gamma \rhd \texttt{s} : \sigma}$$

$$\text{(TSR-Sum)}\ \frac{\Delta; \Gamma \vdash \texttt{SX}_1 : \sigma \qquad \Delta; \Gamma \vdash \texttt{SX}_2 : \sigma}{\Delta; \Gamma \rhd \texttt{SX}_1 + \texttt{SX}_2 : \sigma}$$

To simplify presentation, we write `/*@ srew SX @*/s` instead of `/*@ srew SX + s @*/s`, and we use `/*@ sins SX @*/` as an abbreviation for `/*@ srew SX + skip;@*/skip;`.

Example 4. *We annotate class* `URemote` *from Section 2 as shown in Figure 3a. (To simplify exposition, we omitted field* `KeyPair kp` *in Section 2.) The annotated constructor can be modified to assign* `new RSAPair()` *(which represents an RSA key pair),* `new DHPair()` *(a Diffie-Hellman key pair), or* `new EmptyPair()` *(a placeholder that represents no key) to* `kp`*. The default behavior of method* `on` *is to initialize* `m` *with a device-specific payload. The annotation associated with this statement allows the meta-market to replace the payload with a message containing the user's PIN, obtained calling method* `askPIN()` *(not shown). Before receiving the confirmation message* `r`*, the code can be extended with a block of instructions. In the challenge-response protocol (Figure 1b) the universal remote signs a nonce received from the device. The corresponding code is made legal by a set of annotations that allow receiving the nonce, signing it, and handling possible errors.*

Extending the Type System. We complete the type system presented in Appendix C with the annotation typing rules reported in Table 3, which assign a type to each annotation. Rules (TER-Erew$_1$) and (TER-Erew$_2$) state that an expression rewriting annotation is typed to τ if it can be typed to its base term `EX`. Rule (TER-Erew$_1$) also requires `SX` to have type `void`. Rule (TER-Exp) reduces to typing expression `e`, while (TER-Sum) assigns type τ to $\texttt{SX}_1 + \texttt{SX}_2$ if the two sub expression scan be typed τ. The rules for statements behave similarly.

6. CODE VERIFICATION

In this section we show how a meta-market can verify the authenticity and validity of the code from the developer, and how the user verifies the same properties on the code and modifications received from the meta-market.

We use digital signatures to guarantee the authenticity of the data exchanged by the parties. Our instantiation

of the auxiliary information γ_D corresponds to a signature on all files that compose the application and the contract. Similarly, γ_M is a signature computed over γ_D and over all files that constitute the modification implemented by the meta-market.

A FLEX contract is a file containing all annotations from the app's source code. Each annotation is augmented with an absolute reference to the specific piece of code it applies to. mods consist of one or more AMJ source files, created by the meta-market according to the contract. Each file extends and overrides portions of the app's code via late binding.

Given a tuple D, the meta-market invokes $\mathsf{check}_{mm}(D, pk_{dev})$ to determine the validity of the contract. This function verifies that the signature on the app and the contract is correct. Then, it runs AMJ's type checking to determine if the contract can be honored, as detailed in sections 5.3 and 5.4. If the type system returns no errors and all signatures verify correctly, tuple D is accepted and $\mathsf{check}_{mm}(D, pk_{dev})$ returns 1. Otherwise, it returns 0.

A tuple M is checked by the user by invoking $\mathsf{check}_{user}(M, pk_{dev}, pk_{mm})$. First, this function extracts γ_D from γ_M and uses it to compute $\mathsf{check}_{mm}(\overline{D}, pk_{dev})$ where $\overline{D}$ is constructed from M as $\overline{D} = (\mathsf{app}, \mathsf{contract}, \gamma_D)$. If check_{mm} returns 1, then the user learns that the app has not been tampered with since it was issued by the developer. Next, check_{user} verifies the signature on γ_D and mods. A positive verification indicates that the modification from the meta-market have not been altered by an external adversary. Finally, check_{user} verifies mods against contract as discussed next. If the verification is successful, it compiles mods against app. The resulting binary incorporates all modifications from the meta-market, applied to the authentic app from the developer.

The correctness of code modifications is verified as follows. Each annotation `ER` (or `SR`) corresponds to a recursion-free finite language $\mathcal{L}_{\texttt{ER}}$ ($\mathcal{L}_{\texttt{SR}}$, respectively). Given an annotation `ER = erew EX`, $\mathcal{L}_{\texttt{ER}} = \mathcal{L}_{\texttt{EX}}$ where $\mathcal{L}_{\texttt{e}} = \{\texttt{e}\}$, and $\mathcal{L}_{\texttt{EX}_1+\texttt{EX}_2} = \mathcal{L}_{\texttt{EX}_1} \cup \mathcal{L}_{\texttt{EX}_2}$. Similarly, if `ER = erew SX; EX`, $\mathcal{L}_{\texttt{ER}} = \mathcal{L}_{\texttt{SX}} \cdot \mathcal{L}_{\texttt{EX}}$ (being $\cdot$ the sequence operator). Thus, verifying the compliance of the modifications and the contract amounts to checking whether a term, i.e., either an expression `e` or a statement `s`, belongs to the annotation language. In other words, we say that `/*@ ER @*/ e` (`/*@ SR @*/ s`, respectively) is legal if and only if $\texttt{e} \in \mathcal{L}_{\texttt{ER}}$ ($\texttt{s} \in \mathcal{L}_{\texttt{SR}}$). An app is legal if every annotation appearing in it is legal.

Example 5. *Fragments in figures 3b and 3c implement the protocols illustrated in figures 1b and 1c respectively. The two figures do not show annotations, presented in Figure 3a, and highlight code changes with "▶". The annotation in the constructor in Figure 3a defines the following language:*

$$\mathcal{L} = \left\{ \begin{array}{l} \texttt{kp = new RSAPair();}, \\ \texttt{kp = new DHPair();}, \\ \texttt{kp = new EmptyPair();} \end{array} \right\}$$

Both the constructors in figures 3b and 3c are valid, because they are obtained by replacing a statement in $\mathcal{L}$ with another statement in $\mathcal{L}$. A similar argument applies to method `on`.

To summarize, each annotation defines a finite language that the meta-market uses to perform modifications to the app. Given a contract (i.e., a list of annotations), the user can verify the membership of each modification to the language defined by the corresponding annotation. Violating the contract is equivalent to producing one or more modifications that are not in the language defined by annotations. Thus, the adversary cannot covertly provide illegal app modifications to the user without either violating, or altering the contract. Next, we discuss why the user can always determine if the contract has been altered by the adversary.

Security of FLEX. We argue that FLEX is secure against both internal and external adversaries, under the assumption that the underlying signature scheme is secure against existential forgeries. As discussed earlier in this section, given a tuple $M = (\mathsf{app}, \mathsf{contract}, \mathsf{mods}, \gamma_M)$, the user can verify that the modifications in mods comply with the contract. What we need to show next is that because of signatures γ_D and γ_M, our instantiation of FLEX is secure against Forging Application Attacks (Definition 1) and Application Poisoning Attacks (Definition 2).

Theorem 1. *Assuming that the underlying signature scheme is secure against existential forgeries, the instantiation of FLEX presented in this paper is* FAA*-secure.*

Proof Proof of Theorem 1 (Sketch). Assume that the adversary can construct, with non-negligible probability, a tuple $M^* = (\mathsf{app}^*, \mathsf{contract}^*, \mathsf{mods}^*, \gamma_{M^*})$, such that $D^* = (\mathsf{app}^*, \mathsf{contract}^*, \gamma_{D^*})$ was never issued by the developer. Because γ_{D^*} is a signature computed on both app^* and $\mathsf{contract}^*$, and γ_{D^*} was never computed by the developer, D^* represents a valid forgery for the underlying signature scheme. This contradicts our hypothesis. □

Theorem 2. *Assuming that the underlying signature scheme is secure against existential forgeries, the instantiation of FLEX presented in this paper is* APA*-secure.*

Proof Proof of Theorem 2 (Sketch). Assume that the adversary can win the APA experiment with non-negligible probability. Following the same argument as in the proof of Theorem 1, $D^* = (\mathsf{app}^*, \mathsf{contract}^*, \gamma_{D^*})$ must have been issued by the developer. Therefore, the adversary can win if and only if $M^* = (\mathsf{app}^*, \mathsf{contract}^*, \mathsf{mods}^*, \gamma_{M^*})$ was never issued by the meta-market. However, because γ_{M^*} is a signature computed on app^*, $\mathsf{contract}^*$, and mods^*, and the adversary has no access to the meta-market signing key, M^* represents a valid forgery for the underlying signature scheme. This contradicts our hypothesis. □

7. FLEX PROTOTYPE

In this section we provide further details on our prototype implementation of AMJ. The goal of our prototype is to show the feasibility of FLEX, and to provide a codebase that can be used and extended by the research community.

The AMJ interpreter consists of the following components: a lexer, a parser, an abstract syntax tree (AST) builder, an abstract semantic graph (ASG) constructor, a type checker, and an operational semantic executor. To build these components, we used Xtext [29] and Xsemantics [28]. Xtext is a framework for developing programming languages. It is based on the Eclipse Modeling Framework, and provide tools for building custom lexers, parsers and class models. We used Xtext to construct the lexer and parser used in FLEX from AMJ's grammar specifications. The AST resulting from parsing AMJ code is refined using Xtext to add cross-links between elements (e.g., method invocations

```
class URemote {

 Device d;
 KeyPair kp;

 URemote(Device dev) {
  this.d = dev;
  /*@ srew
   this.kp = new RSAPair();
   + this.kp = new DHPair();
  @*/
  this.kp = new EmptyPair();
 }

 void on() {
  Message m, r in {
   m = //@ erew this.askPIN();
     new Message(true);
   this.d.send(m);
   /*@ sins
    Message nonce, s in {
     nonce = this.receive();
     s = this.kp.sign(nonce);
     this.d.send(s);
    }
   @*/
   r = this.d.receive();
   if(!r.isACK()) {
    this.prompt(...);
   }
  }
 }
 // ...
}
```

(a) Annotated `URemote`.

```
class URemote {

 Device d;
 KeyPair kp;

 URemote(Device dev) {
  this.d = dev;



  this.kp = new EmptyPair();
 }

 void on() {
  Message m, r in {
   m =
▶     this.askPIN();
   this.d.send(m);









   r = this.d.receive();
   if(!r.isACK()) {
    this.prompt(...);
   }
  }
 }
 // ...
}
```

(b) Code corresponding to Fig. 1b.

```
class URemote {

 Device d;
 KeyPair kp;

 URemote(Device dev) {
  this.d = dev;



▶ this.kp = new RSAPair();
 }

 void on() {
  Message m, r in {
   m =
       new Message(true);
   this.d.send(m);

▶ Message nonce, s in {
▶  nonce = this.receive();
▶  s = this.kp.sign(nonce);
▶  this.d.send(s);
▶ }

   r = this.d.receive();
   if(!r.isACK()) {
    this.prompt(...);
   }
  }
 }
 // ...
}
```

(c) Code corresponding to Fig. 1c.

Figure 3: Annotation of `URemote` and two instantiations of modified code. Methods other than the constructor and `on` are omitted.

and corresponding method implementation). This process transforms the AST into the corresponding ASG.

Xsemantics is a plugin for Xtext that allows developers to build custom type systems. We used Xsemantics to implement the typing rules and operational semantic rules presented in Section 5. These rules are compiled by Xsemantics into the `AnnotatedMjTypeSystem` class, which maps judgements (e.g. type, or exec) to individual Middleweight Java methods with the corresponding name. Those methods take as input the environment (composed of heap environment H, variable environment E, and type environment Γ), and an AST element to type.

To check the validity of the `contract`, our prototype generates an ASG using Xtext, and uses the `AnnotatedMjTypeSystem` class to check for typing correctness, as well as operational semantics correctness of the ASG. It then outputs either *typing is successful*, or *typing has failed*. If typing is successful, all the statements and expressions can be typed, and therefore they comply with the `contract`.

Our prototype relies on the BouncyCastle [6] cryptographic library for signature generation and verification. The implementation of our prototype is available at [10].

8. CONCLUSION

In this paper we introduced FLEX, a framework for code authentication. FLEX allows a semi-trusted third party (e.g. an app meta-market) to perform limited modifications to a mobile app. The user can verify the authenticity of the application at each step of the modification process. In particular, after downloading an app authenticated using FLEX, the user can: (*i*) ascertain that the original app is authentic; (*ii*) check if the meta-market modifications comply with the developer's specifications; and (*iii*) determine if the modifications carried out by the semi-trusted third party have been tampered.

In order to test the practicality of FLEX, we instantiated it using AMJ—a language we designed to support code annotations. Because all verification steps are performed *before* the app is installed on the user's smartphone, FLEX introduces no additional overhead to the app at runtime.

By allowing the user to verify end-to-end authenticity of both developer's code and meta-market modifications, we believe that FLEX overcomes the major limitations of current approaches in this space. Moreover, code annotations do not add substantial complexity to the code development process because they do not affect the semantics of the code, making FLEX easy to use for the developer.

Although FLEX is meant for smartphone apps, it can be easily adapted to authenticate any *mobile code*, i.e., code sourced from a remote system and executed locally without explicit installation (e.g., JavaScript code included in HTML or PDF documents, Flash animations, etc.) [19]. As part of our future work, we will extend FLEX to support in-browser JavaScript authentication. Developers will be able to specify

which parts of their JavaScript code can be modified without invalidating the web page. Companies providing WiFi connectivity to the user (e.g., Gogo Inflight Internet [11], Starbucks [25], etc.), could then apply limited modification to the page's source. The user would still be able to authenticate (and possibly run) the original code. Because the contract would be entirely specified within comments, it would simply be ignored by legacy web browsers.

9. REFERENCES

[1] Apple App Store. http://www.apple.com/itunes/. Accessed: July 2015.

[2] Alessandro Armando, Gabriele Costa, Alessio Merlo, and Luca Verderame. Enabling BYOD Through Secure Meta-market. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec 2014, pages 219–230.

[3] Giuseppe Ateniese, Daniel Chou, Breno de Medeiros, and Gene Tsudik. Sanitizable Signatures. In *Proceedings of the European Symposium on Research in Computer Security*, volume 3679 of *ESORICS 2005*, pages 159–177. Springer Berlin Heidelberg.

[4] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An Imperative Core Calculus for Java and Java with Effects. Technical Report UCAM-CL-TR-563, University of Cambridge, 2003.

[5] Kai Bollert. On weaving aspects. In Ana M. D. Moreira and Serge Demeyer, editors, *ECOOP Workshops*, volume 1743 of *Lecture Notes in Computer Science*, pages 301–302. Springer, 1999.

[6] The Legion of the Bouncy Castle. https://www.bouncycastle.org/. Accessed: July 2015.

[7] Christina Brzuska, Marc Fischlin, Tobias Freudenreich, Anja Lehmann, Marcus Page, Jakob Schelbert, Dominique Schroder, and Florian Volk. Security of Sanitizable Signatures Revisited. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography*, volume 5443 of *PKC 2009*, pages 317–336. Springer Berlin Heidelberg.

[8] Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary Design of BML: A Behavioral Interface Specification Language for Java Bytecode. In *Proceedings of the Fundamental Approaches to Software Engineering*, volume 4422 of *FASE 2007*, pages 215–229. Springer Berlin Heidelberg.

[9] Firefox Marketplace. http://marketplace.firefox.com. Accessed: July 2015.

[10] FLEX Prototype. http://cl.ly/1L3n2P1i2F2d.

[11] Gogo Inflight Internet. http://www.gogoair.com.

[12] Google Play Store. http://play.google.com/. Accessed: July 2015.

[13] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. Homomorphic Signature Schemes. In *Proceedings of the Cryptographer's Track at the RSA Conference on Topics in Cryptology*, CT-RSA 2002, pages 244–262. Springer-Verlag.

[14] Gregor Kiczales. Aspect-oriented Programming. *ACM Computer Survey*, 28(4es), December 1996.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[16] Gary Leavens, Albert Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Proceedings of the Workshop on Formal Underpinnings of Java*, OOPSLA 1998.

[17] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM 2014, pages 93–104.

[18] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[19] Refik Molva and Françoise Baude. Mobile Code, Internet Security, and E-Commerce. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP 2000, pages 270–281.

[20] George Necula. Proof-carrying Code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1997, pages 106–119.

[21] OpenHAB project homepage. http://www.openhab.org/. Accessed: July 2015.

[22] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-Carrying Code (MCC): A New Paradigm for Mobile-code Security. In *Proceedings of the Workshop on New Security Paradigms*, NSPW 2001, pages 23–30.

[23] Signing Your Applications. http://developer.android.com/tools/publishing/app-signing.html.

[24] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[25] Starbucks. http://www.starbucks.com. Accessed: July 2015.

[26] Gregory Sullivan. Aspect-oriented Programming Using Reflection and Metaobject Protocols. *Communications of the ACM*, 44(10):95–97, October 2001.

[27] Wink Smart Hub. http://www.wink.com. Accessed: July 2015.

[28] Xsemantics. http://xsemantics.sourceforge.net/. Accessed: July 2015.

[29] Xtext. A framework for development of programming languages and domain specific languages. https://eclipse.org/Xtext/. Accessed: July 2015.

APPENDIX

A. TECHNICAL PROOFS

DEFINITION 6. *We define the function VType as follows:*

$$VType(v,H) = \begin{cases} \top & \text{if } v = \textbf{null} \\ \texttt{void} & \text{if } v = \bullet \\ C & \text{if } v = o \text{ and } H(o) = (C,E) \end{cases}$$

DEFINITION 7. *We write $E, H \models \Gamma$ if and only if:*

$$\forall x. VType(E(x), H) = \tau \wedge \tau \sqsubseteq \Gamma(x)$$

LEMMA 1. *For each expression e and statement s, and for all E, H, Γ such that $E, H \models \Gamma$, the following holds:*

$$\Delta; \Gamma \vdash e : \tau \wedge (E, H, e) \to (E', H', v) \Longrightarrow E', H' \models \Gamma$$
$$\Delta; \Gamma \vdash s : \tau \wedge (E, H, s) \to (E', H', v) \Longrightarrow E', H' \models \Gamma$$

PROOF. The property trivially holds for expressions. In fact, expressions have no effect on E (i.e., $E' = E$), and the types associated to the entries of H are immutable. With respect to statements, we proceed by induction over the structure of s:

- Case `skip;`. Trivially, $E' = E$ and $H' = H$.
- Case `return e;`. A direct consequence of the property for expressions.
- Case `x = e;`. Since the property holds for `e`, we know that $E, H' \models \Gamma$. Hence, we just need to show that $[\texttt{x} \setminus v] \circ E, H' \models \Gamma$, which is a consequence of the fact that $VType(v) \sqsubseteq DType(\texttt{x})$.
- Case `if (`$\texttt{e}_1$ `==` $\texttt{e}_2$`) { `$\texttt{s}_1$` } else { `$\texttt{s}_2$` }`. We iteratively apply the inductive hypothesis to $\texttt{e}_1$, $\texttt{e}_2$ and $\texttt{s}_1$ (or $\texttt{s}_2$, depending on the applied rule of the operational semantics).
- Case `e.f = e';`. We apply the inductive hypothesis to `e` and `e'` and we conclude by noticing that the operation does not change the type of the records in H.
- Case `C x in {s}`. A direct consequence of the inductive hypothesis applied to `s` and $[\texttt{x} \setminus \textbf{null}] \circ E$.
- Case $\texttt{s}_1\,\texttt{s}_2$. We conclude by applying the inductive hypothesis to $\texttt{s}_1$ and $\texttt{s}_2$, in this order.

□

LEMMA 2. *For each expression e and statement s and for all E, H, Γ such that $E, H \models \Gamma$ the following properties hold*

$$\Delta; \Gamma \vdash e : \tau \Longrightarrow \exists v, E', H'. (E, H, e) \to (E', H', v) \wedge VType(v, H') \sqsubseteq \tau$$
$$\Delta; \Gamma \vdash s : \sigma \Longrightarrow \exists v, E', H'. (E, H, s) \to (E', H', v) \wedge VType(v, H') \sqsubseteq \sigma$$

PROOF. We proceed by structural induction on e.

- Case `null`. By (TE-NULL), $\Delta; \Gamma \vdash \texttt{null} : \top$ and, by (EE-NULL) $(E, H, \texttt{null}) \to (E, H, \textbf{null})$. Trivially, by definition, $VType(\textbf{null}, H') = \top \sqsubseteq \top$.
- Case `x`. By (TE-VAR), $\Delta; \Gamma \vdash \texttt{x} : \Gamma(x)$ and, by (EE-VAR) $(E, H, \texttt{x}) \to (E, H, E(x))$. We conclude by noticing that by assumption $E, H \models \Gamma$, $VType(E(x), H) \sqsubseteq \Gamma(x)$.
- Case `e.f`. We instantiate rule (TE-FLD) and apply the inductive hypothesis to `e`. As a consequence, we have that $\Delta; \Gamma \vdash \texttt{e} : C$ (such that $\Delta(C)(\texttt{f}) = \tau$) and $(E, H, \texttt{e}) \to (E, H', o)$ with $H(o) = C' \preceq C$. We can conclude since $C' \preceq C$ implies $\Delta(C')(\texttt{f}) = \tau$ (fields cannot be redefined).
- Case `e.m(`$\bar{\texttt{e}}$`)`. We follow the same reasoning as above, but we apply rule (TE-MTH). Hence we obtain that $\Delta; \Gamma \vdash \texttt{e} : C$ (such that $\Delta(C)(\texttt{m}) = \bar{\tau}'' \to \tau$) and $(E, H, \texttt{e}) \to (E, H', o)$ with $H(o) = C' \preceq C$. Also, we iteratively apply the inductive hypothesis to all the elements of $\bar{\texttt{e}}$ starting from the configuration $(E, H_0, \texttt{e}_1)$ (with $H_0 = H'$).[6] We obtain that $\bar{\texttt{e}}$ are typed $\bar{\tau}^*$ and generate n values $\bar{v}$ such that $\forall i. VType(v_i, H_i) \sqsubseteq \tau_i^*$. Since $\bar{\tau}^* \sqsubseteq \bar{\tau}''$ we can conclude by applying rule (EE-MTH) (and the assumption that the method body is correctly typed w.r.t. Δ).
- Case `new C(`$\bar{\texttt{e}}$`)`. The proof follows the same reasoning as in the previous case.
- Case `(C)e`. By inductive hypothesis, we have $\Delta; \Gamma \vdash \texttt{e} : \tau$ and, by rule (TE-CST), we know that $\tau = C'$ such that $C' \preccurlyeq C$. The property holds, since $VType(o, H') \sqsubseteq C' \sqsubseteq C$.
- Case `/*@ ER @*/e`. Trivially from the inductive hypothesis.
- Case `skip;`. By (TS-SKIP), $\Delta; \Gamma \vdash \texttt{skip;} : \texttt{void}$ and, by (SE-SKIP) $(E, H, \texttt{skip;}) \to (E, H, \bullet)$. By definition, $VType(\bullet, H') = \texttt{void}$ which suffices to conclude.
- Case `return e;`. We conclude by applying the inductive hypothesis.
- Case `x = e;`. We start by applying the inductive hypothesis to `e`. Then, we apply the typing rule (TS-ASGN) and the operational semantics sule (SE-ASGN) and we conclude as in the previous case.
- Case `if (`$\texttt{e}_1$ `==` $\texttt{e}_2$`) {` $\texttt{s}_1$ `} else {` $\texttt{s}_2$ `}`. We apply the inductive hypothesis to $\texttt{e}_1$ and $\texttt{e}_2$ (in this order). Then, we have two symmetric cases depending on whether the guard evaluates to tt or ff. In both cases, we conclude by applying the inductive hypothesis (to $\texttt{s}_1$ and $\texttt{s}_2$, respectively).
- Case `e.f = e;'`. We follow the same reasoning applied for variable assignments. The only difference is that here we apply the inductive hypothesis to both `e` and `e'`.
- Case `C x in {s}`. We apply and assume the premise of rule (TS-BLK) to obtain $\Delta; \Gamma, x : C \vdash \texttt{s} : \sigma$. To apply the inductive hypothesis and conclude, we need to show that $[x \setminus \textbf{null}] \circ E, H \models \Gamma, x : C$. However, this trivially follows from $E, H \models \Gamma$ and $\top \sqsubseteq C$.
- Case $\texttt{s}_1\,\texttt{s}_2$. Here we have two sub-cases (depending on which typing rule is applied).
  - (TS-SEQ_1). Again we have two branches, one for rule (SE-SEQ_r) and one for (SE-SEQ_c). The first one simply requires to apply the inductive hypothesis to $\texttt{s}_1$. Instead, applying (SE-SEQ_c) we obtain that $(E, H, \texttt{s}_1) \to (E', H', \bullet)$. By Lemma 1 we know that $E', H' \models \Gamma$ which suffices to apply the inductive hypothesis to $\texttt{s}_2$ and conclude.

[6] Condition $E, H_i \models \Gamma$ is always satisfied due to Lemma 1.

– (TS-SEQ$_2$). In this case rule (SE-SEQ$_r$) does not apply (as $v \neq \bullet$ entails that $VType(v) \neq \texttt{void}$). Hence we consider rule (SE-SEQ$_c$) and we have $(E, H, \mathtt{s}_1) \to (E', H', \bullet)$. Again, we apply Lemma 1 and the inductive hypothesis to conclude.

- Case `/*@ SR @*/s`. Trivially from the inductive hypothesis.

□

THEOREM 1. *For all closed expressions e and statements s the following properties hold*

$$\emptyset, \Delta \vdash e : \tau \Longrightarrow \exists v, E, H.(\emptyset, \emptyset, e) \to (E, H, v) \wedge VType(v, H) \sqsubseteq \tau$$

$$\emptyset, \Delta \vdash s : \sigma \Longrightarrow \exists v, E, H.(\emptyset, \emptyset, s) \to (E, H, v) \wedge VType(v, H) \sqsubseteq \sigma$$

PROOF. A corollary of Lemma 2. □

B. OPERATIONAL SEMANTICS

NAME	RULE	SIDE
(EE-NULL)	$(E, H, \texttt{null}) \to (E, H, \mathbf{null})$	
(EE-VAR)	$(E, H, \mathtt{x}) \to (E, H, v)$	$E(\mathtt{x}) = v$
(EE-FLD)	$\dfrac{(E, H, \mathtt{e}) \to (E, H', o)}{(E, H, \mathtt{e.f}) \to (E, H', v)}$	$H(o) = (C, E_o)$ $\mathtt{f} \in Fields(C)$ $E_o(\mathtt{f}) = v$
(EE-MTH)	$\dfrac{\begin{array}{c}(E, H, \mathtt{e}) \to (E, H_0, o)\\ (E, H_0, \mathtt{e}_1) \to (E, H_1, v_1)\\ \vdots\\ (E, H_{n-1}, \mathtt{e}_n) \to (E, H_n, v_n)\\ ([\bar{\mathtt{x}} \setminus \bar{v}] \circ E_o, H_n, \mathtt{s}) \to (E'_o, H', v)\end{array}}{(E, H, \mathtt{e.m}(\bar{\mathtt{e}})) \to (E, H', v)}$	$H(o) = (C, E_o)$ $(\mathtt{m}, \bar{\mathtt{x}}, \mathtt{s}) \in Methods(C)$
(EE-NEW)	$\dfrac{\begin{array}{c}(E, H, \mathtt{e}_1) \to (E, H_1, v_1)\\ \vdots\\ (E, H_{n-1}, \mathtt{e}_n) \to (E, H_n, v_n)\\ ([\bar{\mathtt{x}} \setminus \bar{v}] \circ E_o, H', \mathtt{s}) \to (E'_o, H'', \bullet)\end{array}}{(E, H, \mathtt{new\ C}(\bar{\mathtt{e}})) \to (E, H'', o)}$	$(\bar{\mathtt{x}}, \mathtt{s}) \in Constructor(C)$ o fresh in H_n $\bar{\mathtt{f}} = Fields(C)$ $E_o = [\mathtt{this} \setminus o, \bar{\mathtt{f}} \setminus \mathbf{null}]$ $H' = [o \setminus (C, E_o)] \circ H_n$
(EE-CST)	$\dfrac{(E, H, \mathtt{e}) \to (E, H', o)}{(E, H, (\mathtt{C})\mathtt{e}) \to (E, H', o)}$	$H(o) = (D, E)$ $D \preccurlyeq C$
(SE-SKIP)	$(E, H, \mathtt{skip};) \to (E, H, \bullet)$	
(SE-RET)	$\dfrac{(E, H, \mathtt{e}) \to (E, H', v)}{(E, H, \mathtt{return\ e};) \to (E, H', v)}$	
(SE-IF$_{\mathrm{tt}}$)	$\dfrac{\begin{array}{c}(E, H, \mathtt{e}_1) \to (E, H', v)\\ (E, H', \mathtt{e}_2) \to (E, H'', v')\\ (E, H'', \mathtt{s}_1) \to (E_1, H_1, v_1)\end{array}}{(E, H, \mathtt{if\ (e_1 == e_2)\ s_1\ else\ s_2}) \to (E_1, H_1, v_1)}$	$v = v'$
(SE-IF$_{\mathrm{ff}}$)	$\dfrac{\begin{array}{c}(E, H, \mathtt{e}_1) \to (E, H', v)\\ (E, H', \mathtt{e}_2) \to (E, H'', v')\\ (E, H'', \mathtt{s}_2) \to (E_2, H_2, v_2)\end{array}}{(E, H, \mathtt{if\ (e_1 == e_2)\ s_1\ else\ s_2}) \to (E_2, H_2, v_2)}$	$v \neq v'$
(SE-ASGN)	$\dfrac{(E, H, \mathtt{e}) \to (E, H', v)}{(E, H, \mathtt{x\ =\ e;}) \to ([\mathtt{x} \setminus v] \circ E, H', \bullet)}$	$\mathtt{x} \in Dom(E)$ $VType(v) \sqsubseteq DType(\mathtt{x})$
(SE-FLD)	$\dfrac{\begin{array}{c}(E, H, \mathtt{e}) \to (E, H', o)\\ (E, H', \mathtt{e}') \to (E, H'', v)\end{array}}{(E, H, \mathtt{e.f\ =\ e}';) \to (E, [o \setminus (C, E')] \circ H'', \bullet)}$	$H''(o) = (C, E_o)$ $\mathtt{f} \in Fields(C)$ $E' = [\mathtt{f} \setminus v] \circ E_o$
(SE-BLK)	$\dfrac{([\mathtt{x} \setminus \mathbf{null}] \circ E, H, \mathtt{s}) \to (E', H', v)}{(E, H, \mathtt{C\ x\ in\ \{s\}}) \to (E'|_{\mathtt{x}}, H', v)}$	
(SE-SEQ$_c$)	$\dfrac{\begin{array}{c}(E, H, \mathtt{s}_1) \to (E', H', \bullet)\\ (E', H', \mathtt{s}_2) \to (E'', H'', v)\end{array}}{(E, H, \mathtt{s}_1\mathtt{s}_2) \to (E'', H'', v)}$	
(SE-SEQ$_r$)	$\dfrac{(E, H, \mathtt{s}_1) \to (E', H', v)}{(E, H, \mathtt{s}_1\mathtt{s}_2) \to (E', H', v)}$	$v \neq \bullet$

C. TYPING RULES FOR EXPRESSIONS AND STATEMENTS

NAME	RULE	SIDE
(TE-NULL)	$\Delta; \Gamma \vdash \texttt{null} : \top$	
(TE-VAR)	$\Delta; \Gamma \vdash \mathtt{x} : \tau$	$\Gamma(x) = \tau$
(TE-WKN)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : \tau}{\Delta; \Gamma \vdash \mathtt{e} : \tau'}$	$\tau \sqsubseteq \tau'$
(TE-FLD)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : C}{\Delta; \Gamma \vdash \mathtt{e.f} : \tau}$	$\Delta(C)(\mathtt{f}) = \tau$
(TE-MTH)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : C \qquad \Delta; \Gamma \vdash \bar{\mathtt{e}} : \bar{\tau}}{\Delta; \Gamma \vdash \mathtt{e.m}(\bar{\mathtt{e}}) : \tau}$	$\Delta(C)(\mathtt{m}) = \bar{\tau}'' \to \tau'$ $\bar{\tau} \sqsubseteq \bar{\tau}''$
(TE-NEW)	$\dfrac{\Delta; \Gamma \vdash \bar{\mathtt{e}} : \bar{\tau}}{\Delta; \Gamma \vdash \mathtt{new\ C}(\bar{\mathtt{e}}) : C}$	$\Delta(C)(\mathtt{new}) = \bar{\tau}'' \to C$ $\bar{\tau} \sqsubseteq \bar{\tau}''$
(TE-CST)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : C'}{\Delta; \Gamma \vdash (\mathtt{C})\mathtt{e} : C}$	$C' \preccurlyeq C$
(TE-EREW)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : \tau \qquad \Delta; \Gamma \rhd \mathtt{ER} : \tau}{\Delta; \Gamma \vdash \mathtt{/*@ER@*/e} : \tau}$	
(TS-SKIP)	$\Delta; \Gamma \vdash \mathtt{skip;} : \mathtt{void}$	
(TS-RET)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : \tau}{\Delta; \Gamma \vdash \mathtt{return\ e;} : \tau}$	
(TS-IF)	$\dfrac{\Delta; \Gamma \vdash \mathtt{s}_1 : \sigma \qquad \Delta; \Gamma \vdash \mathtt{s}_2 : \sigma}{\Delta; \Gamma \vdash \mathtt{if(e_1 == e_2)\ s_1\ else\ s_2} : \sigma}$	
(TS-FLD)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : C \qquad \Delta; \Gamma \vdash \mathtt{e}' : \tau}{\Delta; \Gamma \vdash \mathtt{e.f\ =\ e}'; : \mathtt{void}}$	$\Delta(C)(\mathtt{f}) = \tau$
(TS-ASGN)	$\dfrac{\Delta; \Gamma \vdash \mathtt{e} : \tau}{\Delta; \Gamma \vdash \mathtt{x\ =\ e;} : \mathtt{void}}$	$\Gamma(x) = \tau$
(TS-BLK)	$\dfrac{\Delta; \Gamma, \mathtt{x} : \mathtt{C} \vdash \mathtt{s} : \sigma}{\Delta; \Gamma \vdash \mathtt{C\ x\ in\ \{s\}} : \sigma}$	
(TS-WKN)	$\dfrac{\Delta; \Gamma \vdash \mathtt{s} : \sigma}{\Delta; \Gamma \vdash \mathtt{s} : \sigma'}$	$\sigma \sqsubseteq \sigma'$
(TS-SEQ$_1$)	$\dfrac{\Delta; \Gamma \vdash \mathtt{s}_1 : \sigma \qquad \Delta; \Gamma \vdash \mathtt{s}_2 : \sigma}{\Delta; \Gamma \vdash \mathtt{s}_1\ \mathtt{s}_2 : \sigma}$	$\sigma \neq \mathtt{void}$
(TS-SEQ$_2$)	$\dfrac{\Delta; \Gamma \vdash \mathtt{s}_1 : \mathtt{void} \qquad \Delta; \Gamma \vdash \mathtt{s}_2 : \sigma}{\Delta; \Gamma \vdash \mathtt{s}_1\ \mathtt{s}_2 : \sigma}$	
(TS-SREW)	$\dfrac{\Delta; \Gamma \vdash \mathtt{s} : \sigma \qquad \Delta; \Gamma \rhd \mathtt{SR} : \sigma}{\Delta; \Gamma \vdash \mathtt{/*@SR@*/s} : \sigma}$	