UNIVERSITY OF CALIFORNIA,
IRVINE

SecureDart: Trusting Client-side Code

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Engineering

by

Bin Xu

Thesis Committee:
Professor Brian Demsky, Chair
Professor Guoqing(Harry) Xu
Professor Rainer Dömer

2016

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Brian Demsky for the continuous support of the related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor.

My sincere thanks also goes to Prof. Harry Xu, who provided a lot of guidance on my research. Without his precious support it would not be possible to conduct this research.

I would like to thank Prof. Rainer Dömer for being one of my committee member and giving me insightful comments.

ABSTRACT OF THE THESIS

SecureDart: Trusting Client-side Code

By

Bin Xu

Master of Science in Computer Engineering

University of California, Irvine, 2016

Professor Brian Demsky, Chair

Modern web sites make extensive use of client-side code, but this code runs on untrusted machines and thus the server-side code must validate all client-side requests. Errors in code that validates requests open the server to attacks by adversaries that send malicious requests. SecureDart extends the Dart language and runtime to support writing trustable client-side code. Requests from trusted client-side code contain a certificate that the server uses to validate the request's authenticity.

We have implemented SecureDart as an extension to the Dart compiler. We have evaluated SecureDart on web application benchmarks and were able to secure the applications against client-side attacks with minimal overhead.

Chapter 1

Introduction

Symantec reports that 76% of the websites they scanned had vulnerabilities, of which 20% were critical [29]. Researchers have paid much attention to the problem of protecting web browsers from attacks by potentially hostile web sites. However, clients may also be malicious and web sites may be targets of attacks themselves. This paper focuses on the following problem — how can the server trust client-side code run by end users who are potentially attackers?

Problems Even though client-side code typically only generates well-formed requests to the server, adversaries can easily replace the requests generated by client-side code with hand-crafted malicious requests. Thus, developers generally must manually write server-side validation code to explicitly verify all client requests. Missing validation checks can open the server to attacks. Validation is often performed twice: once on the client to provide feedback to the user and a second time on the server to prevent attacks.

Previous work has focused on static analysis to detect potential missing checks [27] or on specific classes of attacks such as SQL injection attacks [32, 26]. While these analyses are

useful for protecting applications from the kinds of attacks for which they are designed, they have limited generality. Furthermore, the execution environment in which an attack occurs is highly dynamic; a purely static approach can fail to prevent attacks due to missing runtime information.

Remote attestation [25] provides the ability to trust computations on remote computers. It requires support from the hardware, operating system, and browser, making it difficult to deploy in real systems as web sites may be reluctant to risk losing customers. Moreover, client-side implementation bugs can compromise these systems. Fully homomorphic encryption (FHE) [15] combined with garbled circuits can be used to verify remote computations [14]. However, FHE has extremely high overhead, preventing its usage in practical systems.

Swift [9] pioneers language-based techniques to secure web applications. Swift requires developers to write one single Java-like program annotated with information flow policies. The compiler automatically partitions the program into JavaScript code running on the client and Java code running on the server. There are two practical problems with using the Swift approach on real web applications: (1) A Swift server maintains program state for each client for the duration of the computation. This can be problematic as the server must assume that clients it has not heard back from are slow to reply and thus it must keep the states. Our approach only requires the server to maintain a single key per client and even this could be reduced to a counter. (2) Requiring the developer to write one single program as the starting point does not reflect how real-world web applications are developed. It also forces the developer to cede control over when the application communicates and what is communicated, potentially making it challenging to meet performance requirements.

Basic Idea We present a *language-based approach*, called SecureDart, that enables developers to write client-side components that the server can trust without needing hardware support. SecureDart is an extension of Dart (`https://www.dartlang.org/`), a web language

developed by and used widely in Google. Dart compiles client-side code to JavaScript that can be executed by any modern web browser. While Dart is a dynamically-typed language, it supports static typing, enabling SecureDart to type check programs for security.

At the heart of SecureDart are (1) an *annotation-based type system* and (2) a *record and replay runtime*. SecureDart allows developers to write client and server code separately in a natural way with annotations, and combines static and dynamic approaches to verify *authenticity* and *confidentiality*.

Our key idea is to have the server replay client executions to validate if a request received from the client matches its replay result. A mismatch indicates that the client has been tampered and thus the server discards the request and drops the connection. Hence, SecureDart not only solves the missing sanitization problem, but also checks many implicit constraints that the developer may not realize are important.

SecureDart provides a set of annotations for both client- and server-side development. These annotations and the restrictions enforced by them enable us to record and replay only a security-critical component of the client, leading to reduced implementation complexity (*e.g.*, GUI-related functionalities are not recorded and thus no browser modification is needed) and runtime overhead (*i.e.*, less instrumentation is needed).

We have developed a compiler and runtime for SecureDart. The server-side of SecureDart runs on an unmodified Dart VM, but could also be compiled to JavaScript and run on a JavaScript VM. We have used SecureDart to secure five Dart applications — requiring only a small number of annotations and incurring acceptable overheads.

Chapter 2

SecureDart Overview

SecureDart is designed to secure the server-side component of web applications against attacks. It protects the server from security vulnerabilities that arise from the fact that the data received from the client may not be actually generated by a valid client-side execution. In this section, we first describe our threat model. Then, we overview SecureDart's architecture and how it mitigates these vulnerabilities.

2.1 Threat Model

SecureDart assumes the following threat model. First, the adversary has complete control of the client-side code, including source code, library code, and executable code. The adversary can generate arbitrary requests with or without executing the code. Second, the adversary does not have any control over the server's execution, but can communicate with the server through client-side requests. Third, other avenues of attacking the server have been secured — we focus on attacks to the server via tampered client requests.

2.2 Architecture Overview

Ensuring *integrity* and *secrecy* properties are key concerns for the server side of web applications. The standard approach to ensure these properties is for the developer to write code that validates each client-side request. As an example of an integrity check, one can verify whether the number of items the user requests to purchase exceeds the total number of items in stock. As an example of a privacy property, one can verify that a user only requests information about their own account.

However, such checks are specific to an application and attack. Missing a single check may open the application to attacks. To solve the problem, we provide web developers with the abstraction of a *trusted client-side component*. SecureDart implements this abstraction via *replay*—when the client-side execution generates a request, it records the execution trace that produces the request, and sends the trace together with the request to the server; the server then re-executes the client code using the trace and verifies that the re-execution produces the same result as the request.

While this approach is general enough to protect the application from various kinds of attacks, naïvely validating the entire client-side computation can incur extra runtime overheads, potentially limiting its practicality for some applications. To effectively reduce overhead, SecureDart replays only partial client executions, based on the observation that, in most real-world web applications, the client-side code naturally divides into two components, one that may generate *security-critical* requests, which can be altered by the adversary to make the server mistakenly change its state (*e.g.*, writing to a database or modifying objects created at the server) or reveal secret information to the wrong user, and a second that does not generate such requests. For example, code that manipulates critical data to be sent to the server (*e.g.*, the total price of items) belongs to the former, while code that handles interactions with the user (such as the GUI) belongs to the latter.

While these two components are naturally separated, we need to restrict the way they interact with each other to provide security guarantees. To do so, we create a set of annotations that allow the developer to mark these two components. They are allowed to interact only via interface APIs explicitly defined using our annotations; other types of interactions are prohibited. Note that this restriction is consistent with how these components would interact normally — e.g., an interface API can be a `login` method in the core component that is invoked from the GUI when the user clicks the login button on the webpage. Since the security-sensitive requests generated by the first component are always checked, we refer to it as the *trusted component*. The second component, referred to as the *normal component*, is guaranteed to not issue any request that can change the server's state.

While record and replay can be expensive, it is efficient in SecureDart for two reasons. First, Dart does not support multi-threading, and hence, no inter-thread dependencies need to be captured. Second, the normal component does not need to be instrumented and its execution is thus not recorded; the recording of the trusted component needs to be done only at the interface, that is, we only need to intercept calls to the interface APIs to record their signatures and arguments.

The remainder of this section is structured as follows: we first present the core security properties provided by SecureDart, next discuss liveness properties for non-malicious executions, and finally describe how we ensure that trusted component inputs are validated.

Security Not all client requests have security implications. Many requests only retrieve public information from a database (*e.g.*, such as the price of an item), and can be safely called by the client code without any restrictions. SecureDart uses a relatively standard, *server-side* information-flow type system to identify requests that may have integrity or secrecy implications. In the server-side type system, any request that may affect persistent storage locations is considered to affect integrity. For secrecy properties, we rely on the

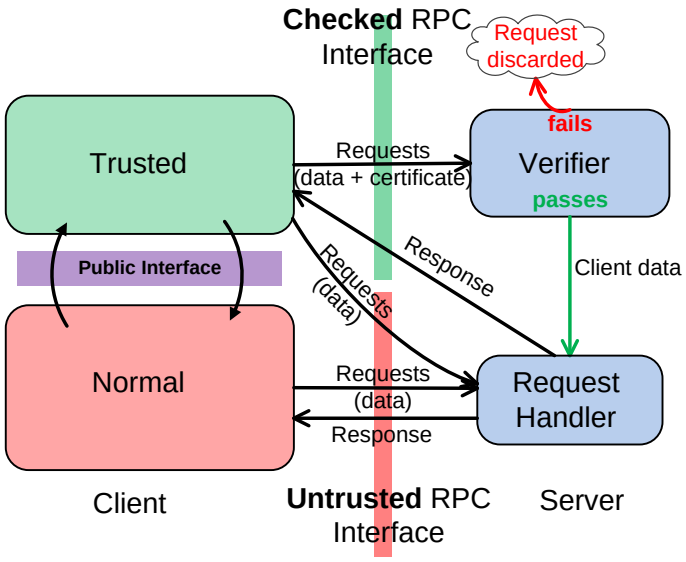


Figure 2.1: SecureDart's architecture: only the trusted component can send requests through the **Checked** interface while both components can send requests through the **Untrusted** interface; the server validates requests from the **Checked** interface by replaying the client-side execution.

developer to label locations that may hold secret values.

Figure 2.1 shows SecureDart's architecture. The server exposes their APIs to the client via remote-procedure-call (RPC) interfaces. The server and client are restricted to only communicate via these interfaces. SecureDart partitions RPC methods into two categories: *checked* and *untrusted*. The SecureDart server-side type system ensures that the request handlers that may either affect persistent locations or return secret values are declared as part of the checked interface.

As show in Figure 2.1, client requests going to the checked interface may only be issued by the trusted component and must be verified by the server before handling. Server validation of the request is implemented by replaying the sequence of invocations of the interface APIs in the trusted component that ultimately produced the request. The server-side request validation is designed such that it can be implemented in a sandbox without requiring communication. The validation process compares the data in each request and the result of replay based on a

certificate that encodes the API trace of the execution and an additional set of signatures. Requests that fail validation will be discarded.

The combination of the server-side type checking and server-side runtime checks provides the following property:

> **If a request affects persistent server state or returns information from secret locations, then it must contain a certificate with a sequence of calls that, when made to a non-tampered trusted component, would generate that exact request.** (2.1)

A second issue is ensuring that SecureDart properly identifies checked call sites. The SecureDart type checker ensures:

> **Checked requests are only made from the trusted component.** (2.2)

This property is dynamically enforced by the server-side validation procedure.

Liveness The client-side type system is designed to ensure that the requests produced by non-tampered executions always pass the server-side validation. As the replay process only replays interface API calls, hidden interactions between the normal and trusted component have the potential to cause legitimate executions to fail validation (because the replay will not reproduce those interactions).

Thus, SecureDart's client side type system ensures:

All interactions between the trusted and normal component should occur via the interfaces declared via annotations. (2.3)

This prevents hidden interactions between the trusted and normal components. For example, aliasing may cause object sharing and the normal component can thus write into objects created by the trusted component. Thus, the type system also prohibits hidden interactions via aliasing and global variables.

Note that SecureDart's security guarantees do *not* rely on the *type safety* of client code since the adversary can arbitrarily modify the executable code (after the source code passes the type check and gets compiled). Tampering with executable code would only make client-side executions generate invalid certificates, causing requests to fail the server validation. This compromises liveness, not security. However, type checking on client code does prevent validation failures.

Boundary Safety Since we do not record/replay the normal component, it is important to make sure that the boundary of the two components is safe, that is, the public interface APIs designed by the developer never mistakenly trust API arguments passed from the normal component that may have potentially been tampered. While it is the developer's responsibility to validate these API inputs (by either writing code to check them explicitly or leveraging domain knowledge), SecureDart includes a number of checks designed to flag type errors if the validation is missing.

The following property helps developers avoid APIs that mistakenly trust values from the normal component:

Values that originate from the server or the trusted component cannot flow from the normal component to the trusted component. (2.4)

The insight here is that dataflows that (1) originate from a potentially trusted source (*e.g.*, the trusted component or the server), (2) flow through the normal component (thus are open to attacks), and (3) then flow back to the trusted component risk mistakenly trusting code in the normal component. The developer could correct such problems by ensuring that such dataflows never leave the trusted component.

The following property ensures that the developer validates such values before use:

Values that originate from the normal component must be validated by either the trusted component or the server before affecting persistent server state. (2.5)

After the developer ensures that these values are indeed safe, she can mark them with a special annotation such that further uses of these values in the trusted component or the server would not flag type errors any more.

Example To summarize the value flow constraints, Figure 2.2 (a) illustrates four kinds of dataflows that are forbidden by our type system. Flow (a) represents the interaction between the trusted and normal components via non-interface channels (*e.g.*, object mutation through aliases), which violates Property 2.3. Flows (b) and (c) go through the normal component and thus violate Property 2.4 because the adversary may corrupt the execution by tampering with the normal component, which may make the tampered request mistakenly pass the

server validation because the recorded trace does not contain calls in the normal component. Flow (d) makes a checked RPC call in a normal component, which violates Property 2.2. The server-side type system forbids dataflow (e), which violates Property 2.1 — a value that comes from an untrusted RPC interface goes to a memory location that stores the server's persistent state or checked values. These properties are discussed in detail in §3.

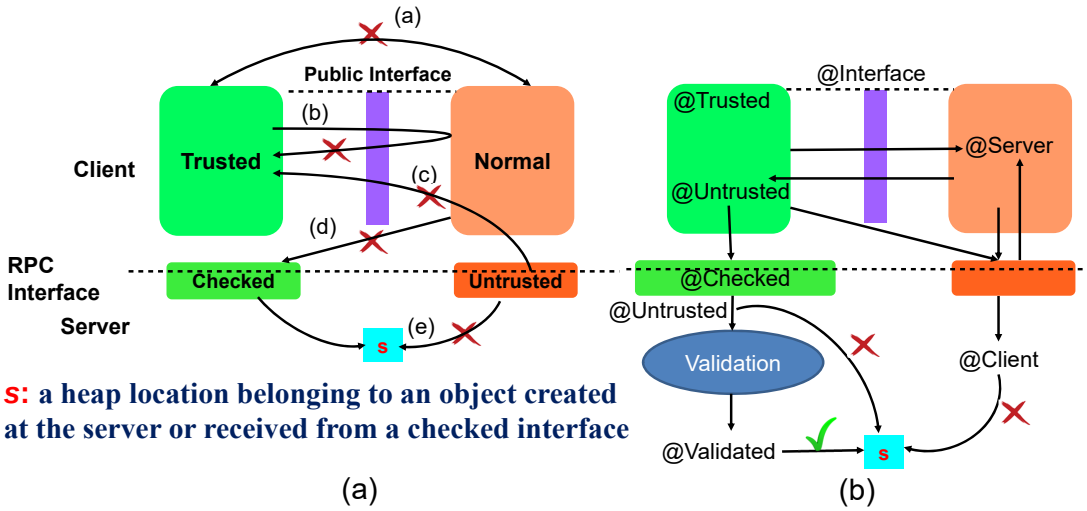


Figure 2.2: Dataflows (a) restricted and (b) *w.r.t.* our annotations.

Summary Now we present the big picture of how SecureDart prevents a malicious client to provide a fake price.

After SecureDart is applied, the function used to calculate the total price of the items in the shopping cart is placed in trusted component. Therefore, the execution will be replayed at the server-side. They type system ensures that all the executions to calculate the total price are recorded and no extra executions are involved. These guarantees are provided by the restrictions of the information flow and aliasings, which means all the legitimate executions that modify the value of the total price will be recorded. Hence, all the legitimate executions will not be rejected by server side validation, which provides the liveness property. Once the client sends a check-out request to the server, the server will validate all the executions that

calculate the total price by replay. All the requests with different total prices from the replay result will be considered as tempered and discarded.

Chapter 3

The SecureDart Language

This section describes how SecureDart protects integrity and confidentiality. We begin by presenting SecureDart's set of annotations and then discuss how these annotations ensure the properties from §2 using a simple Dart application.

Annotations Table 3.1 presents SecureDart's annotations. Among the twelve annotations, four are designed for interface declarations; three are aimed to enforce value flow properties; and the other five are used for both. By default, code is considered to be part of the normal component, *i.e.*, untrusted. Developers can explicitly declare a method or a class to be part of the trusted component with the `@Trusted` annotation. Methods in the trusted component that can be called from the normal component must be annotated with `@Interface`. The `@Checked` annotation is used to define a checked RPC interface, which can only be invoked by the trusted component.

We use `@Untrusted` and `@Validated` to annotate, respectively, variables whose values come from the normal component and those that, while from the untrusted world, have been validated by the server or trusted component and thus become trusted. The type system

Annotation	Type	Role
`@Trusted`	Interface	Defines a trusted class, method, or global variable
`@Interface`	Interface	Interface methods in trusted component callable from normal component
`@Delay`	Interface	RPC interfaces with delayed requests
`@CanBeUnique`	Interface	Interface methods that can be called by a UniqueReference receiver
`@Checked`	Interface	Defines a checked RPC interface
`@Client`	Interface/Flow	Server variables whose values come from normal component
`@Untrusted`	Interface/Flow	Unvalidated values from trusted component
`@Validated`	Interface/Flow	Validated values from the trusted component
`@UniqueReference`	Interface/Flow	Reference-type variables with unique referencences
`@P`	Interface/Flow	Parameterized type declaration
`@Server`	Flow	Values originated from server or trusted component
`@Secret`	Flow	Server values that can be only retrieved by the Checked interface
`@PC`	Flow	Set the program counter label type

Table 3.1: SecureDart annotations.

isolates values with `@Untrusted` and `@Validated` types — values cannot be moved between locations with different types unless the developer indicates a validation check via an explicit cast.

The `@Client` annotation is similar to `@Untrusted`, but it is used only to modify variables at the server side that contain values from the untrusted RPC interface. The main differences between `@Untrusted` and `@Client` can be seen from Figure 2.2 (b), which shows dataflows *w.r.t.* our annotations. An `@Untrusted` value can flow from the normal component into the trusted component and then the server via the checked RPC interface. It becomes a `@Validated` value after being validated. The `@Client` annotation further constrains the usage of values — values with the `@Client` annotation can never be used to modify the server's state.

Interface annotations on methods constrain the allowed types for the method's parameters. The `@Interface` declaration forces all of the parameters to be declared as `@Untrusted`, ensuring that all values that originate from the normal component will be properly labeled. Adding an untrusted method to the RPC interface forces that method's parameters to be annotated `@Client`, ensuring that such untrusted values cannot affect the server's persistent state.

Methods often need to be called in different contexts. For example, a side-effect-free method on the server side may be invoked from both checked and untrusted request handlers. To allow developers to create one single version of a method for these different contexts, we design the `@P` annotation that provides basic support for parameterized types. A variable of any (`@Client`, `@Validated` or `@Untrusted`) flow type can be passed as an argument into a method with a `@P`-annotated parameter. However, for type safety, we require that code that manipulates variables annotated with `@P` satisfy the constraints of the most restrictive flow type, `@Client`.

The `@UniqueReference` annotation restricts a variable to reference an object that is referenced exclusively by the variable. It is applied to constructors. An Object that constructed by a UniqueReference constructor cannot share references with other objects.In other words, the variable must not alias another variable. All the fields of the UniqueReference object are considered as UniqueReference as well. All reference-type parameters of an `@Interface` method must be `@UniqueReference`. SecureDart makes an exception and treats String objects as primitives as they are immutable and thus shared String objects cannot create a hidden communication channel.

To avoid aliasing through method calls on a uniquereference receiver object, we introduce a new annotation, `@CanBeUnique`. Only methods with `@CanBeUnique` annotation can be called by a UniqueReference receiver object. The CanBeUnique methods have the following property: `this` pointer are not allowed to be obtained by other pointers through assignments, method calls or return statements. The risk of leaking unique reference through `this` pointer is eliminated. Since the receiver object is UniqueReference, all the fields of the object are considered as UniqueReferences. Type errors will be thrown if they are referred twice in CanBeUnique methods. Therefore, no aliasing will occur through fields of UniqueReference objects.

Once a UniqueReference variable is accessed, it must be destroyed immediatedly so that it

cannot alias with other variables. We tried to provide flexibility to allow accessing different fields of a UniqueReference Object. We introduce a new annotation, `@Temporary` to define temporary variables in a particular scope. A UniqueReference variable that assigned to temporary variable can be destroy later so that we access multiple fields of that UniqueReference Variable.

To prohibit temporary variables from sharing references with objects out of scope, temporary variables can only be used in particular scopes. No method calls are allowed inside the scope. Temporary variables will be considered as invalid when it is used out of scope. All the checks and the formal type rules will be listed in Section 5.

The `@Server` annotation modifies a variable in the normal component, representing that its value comes from either the server or the trusted component. Values annotated with `@Server` in the normal component can never flow to the trusted component (*e.g.*, preventing the flows (b) and (c) in Figure 2.2 (a)). The `@Secret` annotation preserves *confidentiality* — server values with this annotation can never flow to the client through the untrusted RPC interface. Finally, the `@PC` annotation enables the compiler to check implicit flows so that these flows can never be exploited to bypass type checks.

Running Example Figures 3.1– 3.5 present a simple Dart messaging program. The application supports functionality such as logging in, sending messages, and checking for messages. Here we discuss a few important functions to illustrate SecureDart: Figure 3.1 and Figure 3.2 presents the normal and trusted components, respectively; Figures 3.3 presents the server-side code. Figure 3.5 defines an RPC interface.

The normal (GUI) component interacts with the trusted component by calling its interface methods (annotated with `@Interface`), which then invoke methods defined in the RPC interface (Line 3, Line 5, Line 7, and Line 8) to send requests to the server. The user-declared

```
class Graphic {
  static RPC rpc = new RPC();
  static void check_username(){
    String username = username_field.value;
    rpc.isValid(username).then((mes){
        login_info.text = mes;
    });
  }
  static void login() {
    String usr = username_field.value;
    String pwd = password_field.value;
    action.login(usr, pwd).then((result) {
      if (result == 'valid user') {
        login_div.hidden = true;
      }
    });
  }
  static void showMessages(
    @Server String messages) {
   //..show messages
  }
  static void send_message() {
    String to = destination.value;
    String msg = content.value;
    action.send_msg(to, msg);
  }
  static void showLoginInfo(@Server String msg){
    login_info.text = msg;
  }
  static void get_messages() {
    action.get_msgs();
  }
}
```

Figure 3.1: Client-side normal component.

RPC interface (Figure 3.5) contains four method declarations, three of which are checked. The SecureDart compiler generates stubs for these methods, each of which invokes the appropriate server method.

When a server method is called by the RPC interface (*e.g.*, in Figure 3.3), it processes the request and returns the result. Potential attacks on this application include forging messages from other users and checking others' messages.

```
@Trusted WebSocket ws = new WebSocket("ws://localhost:3333/ws");↩

@Trusted class Action {
  String _user_id = null;
  RPC rpc = new RPC();
  @Interface Future<String> login(
    String username,String password) {
    var completer = new Completer();
    rpc.login(username, password).then((result){
      if (result == 'valid user') {
        Graphic.showLoginInfo('Logged in as ' +
          username);
         _user_id = (@Validated) username;
      } else {
        Graphic.showLoginInfo(
          'Invalid user or password');
      }
      completer.complete(result);
    });
    return completer.future;
  }
  @Interface void send_msg(String to,
    String msg) {
    if (_user_id != null) {
      rpc.send_msg(_user_id, to, msg);
    }
  }
  @Interface void get_msgs() {
    if (_user_id != null) {
      rpc.get_msgs(_user_id).then((result) {
        Graphic.showMessages(result);
      });
    }
  }
}
```

Figure 3.2: Client-side trusted component.

To protect code using SecureDart, the developer first identifies the trusted and normal components. This is easy to do — the client-side code implements two main functionalities: the messaging API and GUI. The messaging API is potentially security-critical since it performs the actual messaging functionality. Hence, we annotate the `Messenger` class as `Trusted`. The `Graphic` class is a normal component since it does not have a class-level

```
KeyValueStore users = new KeyValueStore();
KeyValueStore messages = new KeyValueStore();
class MessagerClient {
  WebSocket _socket;
  MessagerClient(WebSocket ws){
    _socket = ws;
    _socket.listen(RPC.parseRPC);
  }
  void isValid(@Client String username) {
    @Client String result =
      users.containsKey(username).toString();
    RPC.write(result);
  }
  void login(@Untrusted String username,
    @Untrusted String password) {
    String result = '';
    if (users.get((@Validated)username) ==
      (@Validated)password){
      result = 'valid user';
    }
    RPC.write(result);
  }
  void send_msg(String from,
    @Untrusted String to,@Untrusted String msg){
    @Secret List<String> content =
      [from,(@Validated)msg];
    @Secret var user_msg =
      new List<List<String>>();
    if (messages.containsKey((@Validated)to)) {
      ser_msg = messages.get((@Validated)to);
    }
    user_msg.add(content);
    messages.put((@Validated)to, user_msg);
    RPC.write("message sent");
  }
  void get_msgsHandler(String username) {
    @Secret List<List<String>> user_msg =
      messages.get(username);
    @Secret String result =
      secure_encode(user_msg);
    RPC.write(result);
  }
}
```

Figure 3.3: Server-side messaging socket.

```
1 class KeyValueStore {
2   @Secret dynamic get(var key){...}
3   void put(var key, @Secret var value) {...}
4   void delete(var key) {...}
5   bool containsKey(var key) {...}
6 }
```

Figure 3.4: Key-Value Store.

```
1 interface RPC {
2   library messager;
3   String isValid(String username)
4   @Checked String login(@Untrusted String
5     username, @Untrusted String password)
6   @Checked String send_msg(String from,
7     @Untrusted String to, @Untrusted String msg)
8   @Checked String get_msgs(String user_id)
9 }
```

Figure 3.5: The RPC interface.

annotation.

All values in the normal component are untrusted and thus do not need to be annotated. On the server side, variables have a default `@Validated` annotation. Only `@Interface` methods defined in the trusted component can be invoked by the normal component and all their parameters have the `@Untrusted` type. Thus it is impossible for attackers to generate calls from the normal component into internal trusted component methods to bypass the validation of arguments. However, the developer can cast a `@Untrusted` variable explicitly to a `@Validated` variable in the trusted component to mark that the developer has written a check that the variable does not contain malicious values.

For example, Line 12 in Figure 3.2 casts the type of the variable `user`, which is originated from the normal component and thus `@Untrusted`, to `@Validated`, because the value of the variable has been validated by the server when Line 18 is executed. An `@Untrusted` value can flow to the server, but can only flow to locations that are allowed by the type system.

Server-side Security Checks SecureDart must validate all server requests that may modify persistent server state or may return private information. At the server side, a flow type system is employed to identify such requests. Values that come to the server through an untrusted RPC interface have `@Client` types. SecureDart prevents such values from flowing to locations that do not have the `@Client` type. Persistent locations (*e.g.*, data structures created by the server) are not annotated; therefore, the type system prevents untrusted values from flowing to these data structures.

An adversary may also attempt side-effect-free requests that improperly return confidential information he/she is not supposed to know. SecureDart protects confidentiality by using the `@Secret` annotation to ensure that confidential information is only accessed through the intended mechanism. An example usage scenario is to ensure that an attacker cannot generate requests to access the confidential information of other users. In Figure 3.4, the return type of the `get` method in the `KeyValueStore` class is annotated as `@Secret` (Line 2), ensuring that the return value cannot flow to an untrusted RPC interface. This value can only go through a checked interface into the trusted component.

Our type system handles both explicit and implicit flows — if the execution branches on variables with `@Untrusted` annotations, the program counter is set to `@Untrusted`. While we allow primitive-typed values to flow from trusted locations to untrusted locations, reference values require more careful treatment. If we allow an alias to a trusted object to flow to an untrusted location, the alias could be used to bypass the type system. Thus, we prohibit assignments that copy trusted object references to untrusted locations. Flow (e) in Figure 2.2 shows an example: a value from an untrusted request flows to a location s that may contain values from the trusted request.

Returning to the running example, note that a user has to login (checked by Line 23 and Line 28 in Figure 3.2) before being able to send and receive any message. After a successful login, the `@Untrusted` variable `username` that contains a value from the normal component

has been validated (Line 12 Figure 3.2) and the user name is then written into field `_user_id` of the trusted class `Action`. It is impossible for the adversary to change this field because it can only be written by method `login` after the server validation of the untrusted user name. Neither can the adversary alter the execution or send forged requests to the server because these requests are part of the checked interface and thus will be validated by the server.

Property 2.2 guarantees that the client can only call the checked API from the trusted component. The set of legal call sites is determined at compilation time and baked into the code SecureDart generates for the server-side RPC interface. If a server-state-changing request is sent via an untrusted RPC interface, it would either be discarded or cause a runtime error because a handler for such requests is prohibited by the type system and thus would not exist.

Flow (d) in Figure 2.2 shows such an example. In our program, the communication between the trusted component and the server is always through the RPC methods with the `@Checked` annotation (*i.e.*, `login`, `send_msg`, and `get_msg` in Figure 3.5). Certificates are always generated along the requests made through these methods.

Liveness Checks In this section, we describe how SecureDart ensures that well-behaved executions always pass the server-side validation procedure.

Recall that the server-side validation only replays the interactions between the trusted and normal components that occur via methods annotated with `@Interface`. If the components interact via hidden channels, *e.g.*, reading/writing of shared objects, legitimate executions may potentially fail the validation process. Flow (a) in Figure 2.2 is an example of a hidden interaction. Thus, the client-side type system prohibits *object sharing* between the trusted and normal components. SecureDart does so by performing two separate checks: (1) one that ensures references to the same object from both components cannot simultaneously exist and

(2) a second that prevents object sharing via global variables.

The first check is implemented with the `@UniqueReference` annotation. For each assignment that involves variables with the `@UniqueReference` annotation, our compiler automatically instruments a statement that nullifies the source variable. Reference-type variables accepted at an interface method as arguments between the trusted and normal components must be annotated with `@UniqueReference`. When the method is called, the caller nullifies each `@UniqueReference` argument, and hence, the argument would contain null after the call.

By default, global variables are owned by the normal component. The trusted component is prohibited from accessing regular globals. The only exception is for global variables annotated with `@Trusted`, such as `ws` at Line 1 in Figure 3.2; such a variable is exclusively accessible by the trusted component. These annotations ensure that the trusted and normal components only interact via declared interface methods, and thus validations of legitimate executions will always succeed.

Boundary Safety While the checks in the previous sections ensure that the trusted component cannot be tampered with, SecureDart contains type checks that ensure input is validated.

Recall that a concern is that the normal component may receive values from the trusted component or the server and the developer may mistakenly trust such values and pass them back into the trusted component as they originated from a trusted source, forgetting that the adversary can modify these values. Flow (b) in Figure 2.2 (a) is an example of such a flow.

The SecureDart type checker implements this check by prohibiting a variable annotated with `@Server` from being passed into a call to any `@Interface` method in the trusted component. In our example, since method `showLoginInfo` in Figure 3.1 is invoked from the trusted component, its parameter `msg` has a `@Server` annotation, which means `msg` can never be

passed back into the trusted component.

A second concern is that the developer may forget to validate the inputs to the trusted component. In our example, the trusted component API only takes strings as input and all possible strings are valid. Thus we leave the type as `@Untrusted` until the server actually checks the user's password. We cast the types at the password check to avoid forcing the program counter location to have the type `@Untrusted`.

Parallel Requests Requests from different clients can naturally be processed in parallel by the server. In some cases, it can be desirable for a single client to issue multiple requests in parallel. We add an annotation `@Delay` to support this feature. Instead of sending the requests immediately after they are ready, the RPC interfaces annotated with `@Delay` will buffer the requests. Once an RPC interface without `@Delay` is invoked, the buffered requests will be sent to the server together with the new request, and the server will process the entire set.

Chapter 4

Server Validation

The client encodes information for validating checked requests from the trusted component into a certificate, which is sent along with the request and validated by the server. To reduce overhead, SecureDart *incrementally* records and replays client-side executions between two consecutive validated requests. To do this, the certificate contains (1) a serialization s of the start state of the recorded execution, (2) a hash-based signature value e from the server that validates the start state s, and (3) a trusted API call trace for replay.

At each trusted API call site, SecureDart records two pieces of information: the identifier of the target method and the values of the arguments. SecureDart generates customized serialization code that serializes objects to strings. The serialization code works much like Java serialization, but replaces the use of reflection with calls into compiler-generated accessor methods. It supports serializing arbitrary object graphs.

To validate a request, the server first checks that the start state s matches the hash-based signature e. If they match, the server starts replay using the start state s. After replay, the server generates a new signature e' for the final state and sends this signature back to

the client. When the client receives e', it serializes its own current state into s'. The new signature e' and state s' will then be used as e and s for the next request sent via a checked RPC interface. For any legitimate execution, e and s should match, because e represents the final state of the last request-generating execution, which is exactly the start state of the execution producing the current request. If the adversary attempts to change the state of the trusted component between these two requests, the second request cannot pass the server validation due to the mismatch of s and e.

SecureDart uses a Hash-based Message Authentication Code ($HMAC$) [7] to sign states due to the efficiency and security it provides. The HMAC takes as input a message m and a secret key K. In SecureDart, the key K is only known to the server so that a client is not able to forge a valid signature.

The trusted component may also send a request to the server via an untrusted RPC interface. Since these requests are not validated, allowing the trusted component to send a mix of checked and untrusted requests creates a challenge for validating checked requests — when the trusted component sends a checked request, we must ensure that the client cannot lie about the server's responses to previous untrusted requests. A naïve solution is to force the server to replay untrusted requests as well, which would incur overhead. To eliminate this need and reduce overhead, we have the server respond to each untrusted request with a signature e that encodes the combination of the request and the response. The client's runtime then records, for each untrusted request, both the request and the response as well as the signature e from the server.

When the client later makes a checked request r, it sends *all of the untrusted requests* since the last checked request as part of r's certificate. During the validation process, the server first verifies the signature for each untrusted request; the validation process fails if the signature does not match its corresponding request and response. Once all the signatures are verified, the server replays the sequence of the trusted API calls stored in the trace in r's certificate.

4.1 Security Analysis

SecureDart ensures that any request that the server executes and that updates the server's state could have been generated by an untampered execution of the trusted component.

Since after each validation SecureDart sends a certificate that certifies that a given state has been validated, there is the potential for a malicious client to perform a *replay attack* and reuse a certificate multiple times. To protect web applications from replay attacks, the server maintains a per-client signature generation key. After each successful replay validation, the server changes the per-client key. By updating this key, the server effectively revokes the previous signatures it had sent the client. Thus a given signature can only be used by a client at most once. In deployments where replay attacks are not a concern, the server can eliminate the key management overheads by not maintaining per-session keys.

One might imagine that attackers would seek to subvert the server-side replay process for use in an attack. However, the replay process is designed in such a way that it can be performed in a sandboxed process and that this process never needs to contact remote machines or machine local resources outside of the VM. We can determine ahead of time all of the necessary state and prevalidate that the recorded communications have not been tampered with. The certificate mechanism then allows the replay process to validate requests without needing to recontact the remote machine.

It is beyond SecureDart's design goal to defend against attacks that are not produced by modifying the client code. For example, if an adversary fills in a form with malicious values, SecureDart cannot tell whether the request is malicious. Such attacks can be caught by inserting SQL sanitizers. Another limitation of SecureDart is that an adversary may launch denial of service (DoS) attacks by sending many tampered requests to generate excessive load on the server.

Chapter 5

SecureDart Type System

This section summarizes the SecureDart type system.

5.1 Server-side Type Checking

The server-side type system ensures that incoming requests that (1) potentially generate side effects or (2) potentially return secret information are always validated. We achieve this via type labels that restrict information from flowing from untrusted requests to memory locations that store the server's persistent state and type labels that label secret information.

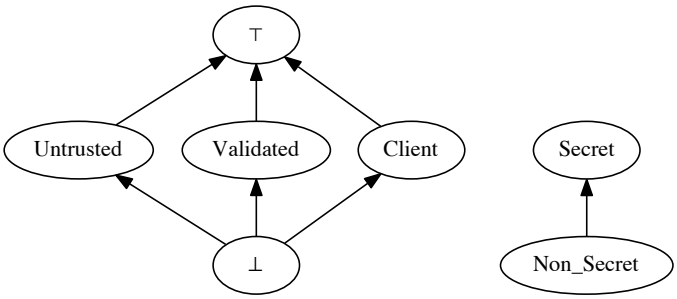
The server side has four annotations: `@Client`, which indicates untrusted values that may originate from the normal component of the client; `@Untrusted`, which indicates untrusted values that originated from the trusted component of the client; `@Validated`, which indicates validated values from the trusted component of the client; and `@Secret`, which indicates confidential information that can only be returned via a checked interface. *The SecureDart compiler forces the parameters of untrusted server interfaces to be annotated* `@Client`.

In order to restrict implicit flows, each statement has an implicit program counter (pc) label type. The default pc label type for each method entry is the highest label type unless it is explicitly set by the developer using the `@PC` annotation.

Server Lattice Structure We arrange the label types into a lattice. Values are only allowed to flow from variables with lower label types in the lattice to those with equal or higher label types. The lattice operators $\sqcup$ and $\sqsubseteq$ are defined in standard ways to capture the set of legal flows. Figure 5.1 presents the lattice structure for the server-side types. There are two lattices, which are checked orthogonally to preserve different properties. As mentioned in the previous section, to maintain soundness in the presence of aliasing, we must further restrict flows for reference types. For this purpose, we extend the partial order $\sqsubseteq$ to the partial order $\sqsubseteq_S$ that defines order for both primitive- and reference-type variables:

$$L_x \sqsubseteq_S L_y \textit{ iff } (L_x \sqsubseteq L_y \wedge P(x)) \vee L_x = L_y$$

where $P(x)$ returns true if x is a primitive-type Dart variable. Recall that SecureDart special cases String objects as primitives as they are immutable.



(a) Untrusted-validated lattice (b) Secret lattice

Figure 5.1: Type lattices for the server.

Type Rules The server-side type system uses a relatively standard implementation of an information flow type system. Each type τ is comprised of two dimensions, which follow the

lattices in Figure 5.1. Type checking passes only if both dimensions of the type follow the rules.

We define the following notations. The judgment $\Gamma \vdash e : \tau$ states that the expression e has the type τ in the environment Γ. $\Gamma[pc = v]$ represents the same environment Γ except that the program counter pc is bound to a new value v. Each type τ is comprised of three dimensions, which follow the lattices in Figure 5.1 respectively. Figure 5.2 presents SecureDart's server-side type rules. The type checking passes only if all the three dimensions of the type follow the rules.

$$\text{LITERAL}\frac{}{\Gamma \vdash \mathit{literal} : \mathit{top}}$$

$$\text{VAR}\frac{\Gamma(x) = \tau}{\Gamma \vdash e : \tau}$$

$$\text{OP}\frac{\Gamma \vdash e_1 : \tau_1 \; \Gamma \vdash e_2 : \tau_2 \; \tau = \tau_1 \sqcap \tau_2}{\Gamma \vdash e_1 \odot e_2 : \tau}$$

$$\text{LOAD}\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e.f : \tau}$$

$$\text{VARASS}\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash x : \tau_x \quad \tau_x \sqsubseteq_S \tau \quad \tau_x \sqsubseteq \Gamma(pc)}{\Gamma \vdash x = e}$$

$$\text{STORE}\frac{\Gamma \vdash e : \tau_e \quad \Gamma \vdash e' : \tau_{e'} \quad \tau_e \sqsubseteq_S \tau_{e'} \quad \tau_e \sqsubseteq \Gamma(pc)}{\Gamma \vdash e.f = e'}$$

$$\text{IF}\frac{\Gamma \vdash e : \tau \quad \Gamma[pc = \Gamma(pc) \sqcap \tau] \vdash s_{i \in \{1,2\}}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

$$\text{LOOP}\frac{\Gamma \vdash e : \tau \quad \Gamma[pc = \Gamma(pc) \sqcap \tau] \vdash s}{\Gamma \vdash \texttt{while}(e)\ s}$$

$$\text{CALL}\frac{mSig(C, m) = (pc_m, \overrightarrow{x_i : T_i}, \tau_r) \quad \Gamma(pc_m) \sqsubseteq \Gamma(pc) \quad \overrightarrow{\Gamma \vdash z_i : \tau_i} \quad \overrightarrow{T_i \sqsubseteq_S \tau_i} \quad \Gamma(x) \sqsubseteq_S \tau_r \quad \Gamma(x) \sqsubseteq \Gamma(pc)}{\Gamma \vdash x = y.m(\vec{z_i})}$$

Figure 5.2: Server-side type checking.

Literal: Every literal value has the highest label type top in the lattice.

OP: Rule OP infers the type of arithmetic expressions of the form $e_1 \odot e2$. The inferred label type is the greatest lower bound of the types of the operands e_1 and e_2.

VARASS: Rule VARASS checks each variable assignment to ensure that the LHS's label

type is lower than both the RHS's label type and the label type of the statement's *pc*. The check on *pc* guarantees that implicit data flows never occur from a lower label type variable to a higher label type variable via a conditional branch. Details of the implicit flow handling will be discussed shortly.

LOAD and STORE: These two rules type check loads and stores, respectively. For a load $e.f$, field f inherits the label type from its base expression e. For example, the expression $e.f$ has a `Client` type if e has it. When a store occurs, we prohibit the write of a value that may potentially come from the untrusted interface to persistent storage in the server, thereby preventing harmful values from corrupting the server's state.

IF and LOOP: Conditional expressions in an `if` statement or a loop may create implicit data flows. To illustrate, consider the following example, `if (a>b) x=1; else x=0;` In this simple code snippet, the value of `x` is determined by the result of the comparison of `a` and `b`.

The rules **IF** and **LOOP** use a special program counter label to prohibit implicit flows. In these rules, the program counter label of an `if` or a `while` statement is updated to the greatest lower bound of the *pc*'s original label type and the label type of the predicate e. Thus, implicit flows from the predicate e to higher types are prohibited. This is because (1) the label type of e is used to update the *pc* label and (2) the label type of the destination of a store or variable assignment must not be higher than that of *pc*.

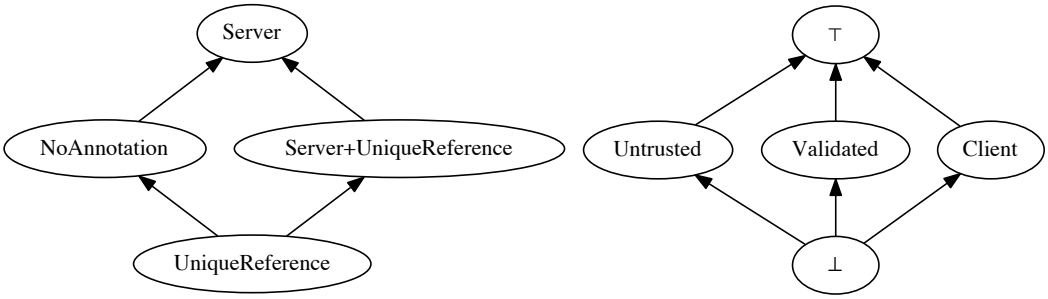
In the above example, the *pc* of the `if` statement is set to the label type of expression `a<b`. Hence, the label type of `x` must not be higher than that of expression `a<b` based on Rule **VARASS**.

CALL: Rule **CALL** checks method invocations. The label types of the (actual) arguments must not be lower than the label types declared for their corresponding (formal) parameters. Similarly, the declared label type of the return value must be higher than or equal to the type of the location that stores the return value. The declared label type of the method's pc

should be lower than or equal to the label of the pc at the call site that invokes the method, which ensures that the rules for implicit information flow (**IF** and **LOOP**) correctly prevent implicit flows from crossing method boundaries.

5.2 Client-side Type Checking

The client-side type system (1) ensures that requests from legitimate executions of the trusted component will always validate and (2) prohibits information flows that likely correspond to missing validation checks in the trusted component.



(a) Client lattice (b) Untrusted-validated lattice

Figure 5.3: Type lattices for the client.

Figure 5.3 presents the lattice structure for the client side. Similar to the server side, the client-side type system restricts values to only flow from lower label types to higher (or equal) label types. Two lattices are applied to client-side types. The label type `Server+UniqueReference` indicates that the annotated value has both `@Server` and `@UniqueReference` annotations. Note that only two of the four types in this lattice, `NoAnnotation` and `Server`, are valid for primitive-type variables as the other types refer only to reference-type variables.

Type Rules We next discuss the client-side type system. We next present the client-side type system rules. We only show the rules for checking the first dimension of the type since the checking the Untrusted-Validated property is similar to the server-side. We first introduce

some new notations. We use n_m to denote the type of the component in which a method is declared and n to indicate the type of the method that contains the current statement. They both have the same value domain $\{$*Trusted*, *Normal*$\}$. Figure 5.4 presents the definitions of a few auxiliary functions.

We define a new partial order $\sqsubseteq_{SU}$ to track label type propagation along heap accesses. It is used only for load and store operations. Its usage is always in the form of $\tau_{e.f} \sqsubseteq_{SU} \tau_e$, indicating that $e.f$ "inherits" from e all its label properties. For example, for an expression e with both `@Server` and `@UniqueReference` annotations, a dereference expression $e.f$ on a reference-type field f has a label type `Server+UniqueReference`, denoting that the `Server` and `UniqueReference` properties of e are propagated to $e.f$. `Server+UniqueReference` is not a new label type, but a combined type denoting the two properties a location has. If f is a primitive-type field, only the `Server` property of e is propagated to $e.f$ as $e.f$ cannot contain reference.

Function TN($\tau, flag$) checks if a variable with label type τ can be used as an argument to a method call. The parameter $flag$ is used to check that the method is annotated with `@Interface`. When the program calls a trusted method from a normal component, we need to check that variables with a Server label cannot be passed as arguments – this is done by the first two cases. Additionally, the second case ensures that only references with a UniqueReference label type can be passed as parameters to avoid aliasing. When a normal method is called from the trusted component, references used as method parameters must be unique references and all arguments must have the Server type label.

The Access function checks that an access to a global variable is consistent with the declared type.

Figure 5.5 presents a subset of the client-side type checking rules. We have omitted the rules that are the same as their server counterparts.

$$\tau_x \sqsubseteq_{SU} \tau_y \Leftrightarrow \begin{cases} \tau_x = \text{NoAnnotation} \\ \qquad \text{if } P(x) \wedge \tau_y = \text{UniqueReference} \\ \tau_x = \text{Server} \\ \qquad \text{if } P(x) \wedge \tau_y = \text{Server+UniqueReference} \\ \tau_x = \tau_y \\ \qquad \text{if } \neg P(x) \wedge \text{Server+UniqueReference} \sqsubseteq \tau_y \\ \tau_x \sqsubseteq \tau_y \\ \qquad \text{Otherwise} \end{cases}$$

$$\text{TN}(\tau_x, \text{flag}) \Leftrightarrow \begin{cases} \tau_x = \text{NoAnnotation} \wedge \text{flag} = \text{@Interface} \\ \qquad \text{if } P(x) \wedge (n_m, n) = (\text{Trusted,Normal}) \\ \tau_x = \mathit{UniqueReference} \wedge \text{flag} = \text{@Interface} \\ \qquad \text{if } \neg P(x) \wedge (n_m, n) = (\text{Trusted,Normal}) \\ \tau_x \sqsubseteq \text{Server+UniqueReference} \\ \qquad \text{if } (n_m, n) = (\text{Normal,Trusted}) \\ \text{true} \\ \qquad \text{Otherwise} \end{cases}$$

$$\begin{aligned} \mathit{Access}(x, \tau_x) = (&\tau_x = \text{Trusted} \wedge n = \text{Trusted}) \\ &\vee (\tau_x \neq \text{Trusted} \wedge n = \text{Normal}) \end{aligned}$$

Figure 5.4: Auxiliary functions.

$$\text{LOAD}\frac{\Gamma \vdash e : \tau \quad \Gamma_e(f) = \tau_f \quad \tau_f \sqsubseteq_{SU} \tau}{\Gamma \vdash e.f : \tau_f}$$

$$\text{VARASS}\frac{\Gamma \vdash x : \tau_x \quad \Gamma \vdash e : \tau \quad \tau_x \sqsubseteq \tau \quad \tau_x \sqsubseteq \Gamma(pc)}{\Gamma \vdash x = e}$$

$$\text{STORE}\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \quad \Gamma \vdash e' : \tau'_e \quad \Gamma_e(f) = \tau_f \\ \tau_f \sqsubseteq \tau_{e'} \quad \tau_f \sqsubseteq_{SU} \tau_e \quad \tau_f \sqsubseteq \Gamma(pc)\end{array}}{\Gamma \vdash e.f = e'}$$

$$\text{FCALL}\frac{\begin{array}{c}\Gamma \vdash x : \tau_x \quad \Gamma \vdash y : \tau_y \quad \overrightarrow{\Gamma \vdash z_i : \tau_i} \quad \overrightarrow{\tau_i \sqsubseteq \Gamma(pc)} \\ \mathit{Sig}(C, m) = (pc_m, \text{flag}, n_m, \overrightarrow{x_i : T_i}, \tau_r) \quad \mathit{CSig}(s) = n \\ \tau_x \sqsubseteq \tau_r \quad \tau_x \sqsubseteq \Gamma(pc) \quad \Gamma(pc_m) \sqsubseteq \Gamma(pc) \\ (n_m, n) \neq (\text{Normal, Trusted}) \quad \overrightarrow{T_i \sqsubseteq \tau_i} \quad \overrightarrow{\text{TN}(\tau_i, \text{flag})}\end{array}}{\Gamma \vdash x = y.m(\vec{z_i})}$$

$$\text{PCALL}\frac{\begin{array}{c}\Gamma \vdash x : \tau_x \quad \Gamma \vdash y : \tau_y \quad \overrightarrow{z_i : \tau_i} \quad \overrightarrow{\tau_i \sqsubseteq \Gamma(pc)} \\ \mathit{Sig}(C, m) = (pc_m, \text{flag}, n_m, \overrightarrow{x_i : T_i}, \tau_r) \quad \mathit{CSig}(s) = n \\ \Gamma(pc_m) \sqsubseteq \Gamma(pc) \quad \overrightarrow{T_i \sqsubseteq \tau_i} \quad \overrightarrow{\text{TN}(\tau_i, \text{flag})}\end{array}}{\Gamma \vdash y.m(\vec{z_i})}$$

$$\text{GLOBAL}\frac{\Gamma(x) = \tau \quad \text{Access}(x, \tau)}{\Gamma \vdash x : \tau}$$

Figure 5.5: Client-side type checking.

LOAD and STORE: A heap load $e.f$ inherits the label type `Server` from its base expression e because all fields of an object with the label type `Server` must have type `Server`. The label type `UniqueReference` will also be propagated if field f is a reference type.

VARASS: Variable assignment follows the flow rules based on the lattice. The value of the expression can be assigned to another memory location as long as the label type of the expression is *higher than or equal to* that of the destination memory location. The label type of the destination variable x must be lower than or equal to that of the program counter to constrain implicit flow, as discussed in §5.1.

FCALL and PCALL: At the client side, SecureDart's type system prohibits the trusted component from calling a normal method with a return value because the execution of the normal component will not be validated by the server. Therefore, we split the rules for method calls to two separate rules–**FCALL** states the rules for method calls with a return value while **PCALL** presents the rules for method calls without a return value. The function *Sig* returns the signature of a method with a return value, and pc_m denotes the method's pc label type. T_i denotes the label type of the ith parameter and τ_r represents the label type of the return value.

The client-side type system checks the arguments to method calls from the normal component to trusted component. It first checks that such calls cannot introduce aliasing that could potentially introduce hidden interactions between components that could cause false validation failures. This check ensures that only references with a `UniqueReference` label type can be passed as parameters to avoid aliasing. It then checks that method calls do not pass values in the normal component that originated from the server or trusted component (*i.e.*, values with the `Server` label) back into the trusted component. The client-side type system also checks that accesses to global variable are consistent with the declared type.

SecureDart uses the `UniqueReference` label type to guarantee that objects passed between

the trusted and normal components do not create side channels. SecureDart uses a simple linear type system to ensure that variables and fields with the `UniqueReference` labels are the only references to their respective pointee objects. Our compiler generates code that overwrites (1) the RHS of an assignment or store that copies a unique reference and (2) any arguments to method calls that have the `UniqueReference` label type.

In general, we have the following checks on `UniqueReference` to guarantee that no side channels are created:

- Only UniqueReference objects can be passed through the interfaces between trusted and normal components.
- UniqueReference variables must be destroyed right after it is used, unless it is assigned to a `Temporary` variable.
- The fields of an object are not allowed to be defined as UniqueReference. All the fields of a UniqueReference Object are considered as UniqueReference.
- Globals are not allowed to be defined as UniqueReference.
- When a UniqueReference object is used as the receiver of a method call, the method must be annotated with `@CanBeUnique`.
- When allocating a UniqueReference Object, the constructor of the class must annotated with `@CanBeUnqiue`, unless using the default constructor.
- Inside a `CanBeUnique` method, *this* pointer can not be used in RHS of assignments. All the fields are considered as UniqueReference and need to follow UniqueReference checks.
- A *Temporary* variable is used to be hold the reference of a UniqueReference object temporarily. It can only be used in a defined scope. No method calls are allowed inside the scope. A *Temporary* variable is not allowed to be returned.

- `this` cannot be used as a method parameter or a value to be returned.

$$\text{METHOD}\frac{\begin{array}{c}(\Gamma[\texttt{this}: Unique, z_{u_i}: Unique_{i=0\ldots k}, z_{n_i}: NonUnique_{i=0\ldots l}] \vdash e \wedge \Gamma(m) = Unique) \\ \vee(\Gamma[\texttt{this}: NonUnique, z_{u_i}: Unique_{i=0\ldots k}, z_{n_i}: NonUnique_{i=0\ldots l}] \vdash e \wedge \Gamma(m) = NonUnique) \\ \texttt{declared\_Unique}(m, u_i)_{i=0\ldots k} \quad \texttt{declared\_NonUnique}(m, n_i)_{i=0\ldots l} \\ |\{u_0, u_1, \ldots, u_k\} \cup \{n_0, n_1, \ldots, n_l\}| = |\vec{z_i}| \wedge \{u_0, u_1, \ldots, u_k\} \cap \{n_0, n_1, \ldots, n_l\} = \emptyset\end{array}}{\Gamma \vdash m(\vec{z_i})\{e\}}$$

$$\text{ASSIGN UNIQUE}\frac{\Gamma \vdash e : Temp \quad \Gamma \vdash e_1 : Unique}{\Gamma \vdash e = e_1 : Unique}$$

$$\text{ASSIGN NONUNIQUE}\frac{\Gamma \vdash e_1 : NonUnique \quad \Gamma \vdash e : NonUnique}{\Gamma \vdash e = e_1 : NonUnique}$$

$$\text{NULLIFY}\frac{\begin{array}{c}\Gamma \vdash e = x.f_1.f_2 \ldots f_i \quad \Gamma \vdash x.f_1.f_2 \ldots f_j = null \\ j \leqslant i \quad \Gamma \vdash x : Unique \quad \Gamma \vdash e : NonTemp \\ x \neq \texttt{this} \vee j >= 1\end{array}}{\Gamma \vdash e = x.f_1.f_2 \ldots f_i; x.f_1.f_2 \ldots f_j = null}$$

$$\text{FIELD}\frac{\Gamma \vdash x : Unique}{\Gamma \vdash x.f : Unique}$$

$$\text{GLOBAL}\frac{\texttt{Global}(x)}{\Gamma \vdash x : NonUnique}$$

$$\text{METHOD CALL}\frac{\begin{array}{c}\Gamma(Temp) = \emptyset \quad \texttt{this} \notin z_i \vee \Gamma(this) = NonUnique \\ (\Gamma(m) = Unique \wedge \Gamma \vdash y : Unique) \vee \Gamma \vdash y : NonUnique \\ \Gamma \vdash z_{u_i} : Unique_{i=0\ldots k} \quad \Gamma \vdash z_{n_i} : NonUnique_{i=0\ldots l} \\ Sig(C, m) = (\text{flag}, n_m, \overrightarrow{x_i : T_i}, \tau_r) \quad T_{u_i} = Unique_{i=0\ldots k} \quad T_{n_i} = NonUnique_{i=0\ldots l} \\ \Gamma \vdash e : \tau_e \quad \tau_e \prec \tau_r \\ |\{u_0, u_1, \ldots, u_k\} \cup \{n_0, n_1, \ldots, n_l\}| = |\vec{z_i}| \wedge \{u_0, u_1, \ldots, u_k\} \cap \{n_0, n_1, \ldots, n_l\} = \emptyset\end{array}}{\Gamma \vdash e = y.m(\vec{z_i}); z_{u_0} = null; z_{u_1} = null; \ldots z_{u_k} = null}$$

$$\text{ALLOCATE}\frac{\begin{array}{c}\Gamma(Temp) = \emptyset \quad \texttt{this} \notin z_i \vee \Gamma(this) = NonUnique \\ (\Gamma(T) = Unique \wedge \Gamma \vdash e : Unique) \vee \Gamma(T) = NonUnique \\ \Gamma \vdash z_{u_i} : Unique_{i=0\ldots k} \quad \Gamma \vdash z_{n_i} : NonUnique_{i=0\ldots l} \\ Sig(C, T) = (\text{flag}, n_T, \overrightarrow{x_i : T_i}, \tau_r) \quad T_{u_i} = Unique_{i=0\ldots k} \quad T_{n_i} = NonUnique_{i=0\ldots l} \\ \Gamma \vdash e : \tau_e \quad \tau_e \prec \tau_r \\ |\{u_0, u_1, \ldots, u_k\} \cup \{n_0, n_1, \ldots, n_l\}| = |\vec{z_i}| \wedge \{u_0, u_1, \ldots, u_k\} \cap \{n_0, n_1, \ldots, n_l\} = \emptyset\end{array}}{\Gamma \vdash e = \texttt{new}\ T(\vec{z_i}); z_{u_0} = null; z_{u_1} = null; \ldots z_{u_k} = null}$$

$$\text{RETURN}\frac{\Gamma \vdash e : NonTemp \quad \texttt{this} \notin e}{\Gamma \vdash \texttt{return}\ e}$$

Figure 5.6: UniqueReference checking.

The formal type rules are listed in Figure 5.6. We first define a new relation between two types $\tau \prec \tau_1$, which means τ_1 is compatible to τ when the corresponding variable with type τ_1 is used as method parameters :

$$\tau \prec \tau_1 \ \textit{iff}\ (\tau_1 = Unique) \vee (\tau = NonUnique \wedge \tau_1 = NonUnique)$$

τ cannot be $Temp$ since it is the declared type of a method parameter. We define the type of the assignment $e = e_1 : T$ when $e_1 : T$ holds. We use $e : Unique$ or $e : Temp$ to indicate that the expression e is `UniqueReference` or `Temporary` type. While $e : NonUnique$ or $e : NonTemp$ indicate that the expression e is not UniqueReference or Temporary respectively. $\Gamma(m) : Unique$ indicates that the method m is annotated as `CanBeUnique`. $\Gamma(T) : Unique$ means the constructor of class `T` is annotated with `CanBeUnique`. $\Gamma(temp)$ denotes the set of live temporary variables. $\Gamma[x : T]$ denotes that the type environment is bounded to the type condition $x : T$. `Global(x)` means x is a global variable. `declared_Unique`(m, u_i) indicates the u_ith parameter of method m is declared as Unique, while `declared_NonUnique`(m, n_i) indicates the n_ith parameter of method m is declared as NonUnique

These type rules describe the checks described above. SecureDart will check client-side code together with the rules in Figure 5.5 to preserve the liveness property.

Dynamic Features and Other Issues Dart includes dynamic language features. For example, Dart allows programmers to type a variable using a dynamic type *var*. SecureDart implements table-based dynamic checks for dynamically typed variables and collections. A second issue is that the Dart static type system allows potentially unsound assignments. SecureDart compiles Dart code in a checked mode that dynamically detects unsound assignments.

Benchmark	LoC (app)	LoC (lib)	Avg. Delay w/o replay	Avg. Delay w/ replay	Overhead	Public API Calls	Base Payload Size	SecureDart Payload Size	Num. of Anno.	LoC in RPC decl.	Replay Overhead on Stress Test
IssueMover	1861	23796	792.26 ms	911.24 ms	15.0%	3	4.3 KB	7.8KB	22	11	N/A
DartChat	1231	798	4.75 ms	6.48 ms	36.4%	2	622 bytes	893 bytes	19	7	19.4%
E-Commerce	2114	4445	77.82 ms	94.01ms	20.8%	4	390 bytes	1146 bytes	27	14	6.9%
Forum	1907	5075	49.09ms	63.58ms	29.5%	4	1.4KB	3.9KB	64	35	13.5%
Turnin	1304	26893	18.0ms	20.2ms	12.2%	3	82 Bytes	457 Bytes	46	29	14.7%

Table 5.1: Various static and dynamic statistics about our applications.

Chapter 6

Experience

We report our experience with SecureDart on five applications: IssueMover for GitHub [4], an application that moves issues between GitHub repositories; DartChat [3], a multi-user chat room web application; E-Commerce [2], an e-commerce application ported from PHP; Forum, an online forum developed using the AngularDart Framework [1]; and Turnin, a tool for managing student git repositories for courses.

A major challenge in the evaluation was finding Dart applications. Dart is used primarily to develop custom, in-house web applications. The open source community has released many Dart libraries, but **there are no large-scale, openly available Dart applications.** While Dart is used by many companies to develop large, deployed systems, these systems are not available to us. We were only able to find two client-server based Dart applications with properties that could be checked, IssueMover and DartChat. Thus, we augmented the two available Dart programs with three that we developed ourselves, E-Commence, Forum, and Turnin.

For each application, Table 5.1 reports the number of annotations added, the amount of

Benchmark	Label Type Anno.	UniqueReference Anno.	RPC Anno.
IssueMover	13	9	3
DartChat	13	4	2
Ecommerce	17	5	5
Forum	48	6	10
Turnin	35	3	8

Table 6.1: Number of annotations based on categories

refactoring effort to use SecureDart, and its performance. It is important to note that the **LoC(lib)** section only includes the numbers of lines of library code that is checked by our compiler. Hence, for each program, the sum of its **LoC(app)** and **LoC(lib)** gives the total number lines of compiler checked Dart code.

We also reports the total number of annotations applied to each benchmark. Basically the annotations are classified as three types: Lable type annotations, annotations for UniqueReference and RPC annotations. Table 6.1 shows the number of annotations in each category.

Performance is evaluated on both regular workloads and stress tests. We compiled the client-side code to JavaScript and ran the server-side code on the Dart VM to simulate the intended deployment. To generate the regular workload, we started a server on one remote machine and launched requests from a browser. The server code was executed on a 3.4GHz Intel Core i7-3770 running Ubuntu. The client-side code was executed on a 2.7GHz Intel Core i5 running OS X on a *different network* to simulate a typical deployment environment.

Stress tests were generated by sending computer-generated requests, as discussed shortly. For each workload, we measured the average response delay before and after the SecureDart run time was added, as well as the average number of bytes transferred between the server and the client.

6.1 Overall Performance

Table 5.1 reports statistics for the benchmarks. Both the data and time overheads are very small for IssueMover because the replay cost is dwarfed by the communication delay between the server and the GitHub repository backend. The overheads for the other three applications are larger but still quite acceptable, which demonstrates the effectiveness of our incremental re-execution algorithm. The developer's effort for adding annotations is insignificant: the ratio between the total number of annotations and the total application lines of code summed across all our benchmarks is 1.1%.

To measure SecureDart's performance under heavy load, we conducted stress tests on four benchmarks. We excluded IssueMover because stress testing would potentially generate a denial-of-service attack on the GitHub server. For the other four benchmarks, we used one machine to act as the server and six other machines to act as clients. For each benchmark, we launched 96 processes (across 6 client machines) to send requests to the server. All the machines have 3.5GHz Intel(R) Xeon(R) CPU E3-1246 v3 CPUs and run Ubuntu.

The last column of Table 5.1 reports the results of the stress tests on three benchmarks. The replay overhead of each benchmark is between 6.9% – 19.4%, which is acceptable. The overhead percentages of the stress tests are smaller than those of the normal tests (Column *Overhead*) because the distribution of requests differs between the stress tests and normal tests. DartChat has a relatively high overhead as the server computation for DartChat does very little work, and thus the small fixed overhead of SecureDart takes a relatively larger fraction of the total time. In general, replaying requests did not introduce much extra overhead on the server.

6.2 Potential Attack, Porting Experience, and Defense

1. IssueMover To use IssueMover, the user specifies the issue to move and the destination repository. IssueMover then retrieves the specified issue and all the corresponding comments from the GitHub server, copies them to the destination repository, and closes the original issue. We launched an attack before we applied SecureDart on it. When we tried to move an issue from the issue mover repository to our own repository, we modified the contents of the issue to be moved before we sent the issue-moving request to the server. Since the server did not validate requests after a successful login, the tampered issue was posted with the tag "copied from Repository `Issue Mover of GitHub`" although the contents were modified.

Next, we modified the original application to use SecureDart constructs. During porting, it was easy for us to identify the trusted and normal component — the code for user interactions, which was straightforward to find, was placed into the normal component; this is basically code that operates on HTML elements. The core component that moves an issue from one repository to the second was annotated as the trusted component. We then changed the communication to use SecureDart RPC interfaces.

SecureDart prevented this attack by validating the executions of the client-side trusted component. The issue-moving request was sent to the server through a `Checked` interface. The certificate in the request contained the sequence of method calls used to construct the request. When validating the `checked` request, the server detected that the contents of the issue generated by the replay routine were not the same as the contents sent along with the request. Therefore, the server rejected the request and disconnected the client.

2. DartChat DartChat is a multi-user chat room application written in Dart, which uses web sockets to communicate between client and server. To use the chat room, a client first signs up for a user name. The server checks whether the user name is in a valid format and has not been used before. The client starts to send and receive messages if the user name is

authorized. When sending a message, a client has two options: send a message to one specific user in the chat room or to all active users. The server distributes the message to the targets selected by the client.

We launched an attack to forge messages from other users against the non-SecureDart version. Before a message is sent, the text that the user types is packed into a `Message` data structure together with the username and the message destination. We modified the username stored in the data structure before it was sent. The server then believed the message was from a different user and sent the forged message. SecureDart effectively blocks this attack.

To port DartChat to SecureDart, we modified the application to route all communications through the RPC interfaces. All data structures used to store messages were defined in the trusted component. Porting DartChat is relatively easier because it has fewer RPC interfaces. After porting, the server replayed the construction of the data structure used to store the user information and the message. Since the data structure generated from replay contained a different username than the one received from the client, the server discarded the request and disconnected the client.

3. E-Commerce E-Commerce is an online shopping application ported from PHP. It implements basic functions (*e.g.*, purchasing items in the cart) for an E-Commerce system. The cart implementation is in the trusted component while the normal component contains mainly GUI code that lists the quantities and prices of items.

Since the calculation of the total price for the items in the cart is conducted at the client side, to launch an attack against the non-SecureDart version, we simply set the calculated total price to zero and sent it to the server. The server then sold a TV to the client for free. SecureDart prevented the adversary from launching this attack. The type system ensures that the total price must be sent to the server via a checked interface. The SecureDart runtime re-calculated the total price and detected the forged request during the validation process.

4. Forum Forum is an online forum system we developed directly in SecureDart on top of the AngularDart Framework. It implements basic functions including writing and reading posts, adding and removing comments to posts, and sending messages between users. After one logs into the system, she could only edit her own posts, messages, and comments. We let the trusted component contain code for adding and editing posts, comments, and messages since all these actions create side effects on the server.

We launched an attack against the non-SecureDart version— we wanted to delete a comment of another user (*e.g.* Bob). To do this, we forged the request to tell the server that the sender of the request was Bob. Forum's server responded by deleting Bob's posts due to a missing validation check. Under SecureDart, the runtime re-executed the construction of the comment-removing request at the server side. The server then discarded the tampered request because the replay result was different from the request.

5. Turnin Turnin is a repository system we developed in SecureDart. It stores and manages code submitted by students in courses. It implements functions such as creating a repository and adding or removing group members. After logging in, a student can only access the projects he owns so that he cannot copy code from others. Only the leader of a group or the instructor can add or remove group members. The trusted component contains the code that creates and deletes repos, adds and removes collaborators, and changes group leaders since these functionalities have side effects on the server.

We launched an attack against the Dart version of this application. We forged a request to lie to the server that the user logged in was the instructor of the course. Then the sender could do everything on the file system such as removing others' repositories. Under SecureDart, this attack did not work because the server reproduced the executions that generated the repo-removing request. The validation checked if the user logged in was authorized to remove the repo. The server then discarded the request since it failed the validation.

Annotation Effort To create a SecureDart program, the developer can start from an existing Dart program, annotate its code, and declare RPC interfaces. The last two sections in Table 5.1 report the #LoC that we wrote to use SecureDart. For example, IssueMover required 16 annotations while E-Commerce required 17. The total size of the RPC interfaces for all of the benchmarks is 93 lines of Dart code.

Most of the porting time was spent understanding the original code. While SecureDart provides a programming model to the developer, this model serves to gather security-critical code and state in one place and to isolate it from the rest of the system. This can be a good development methodology regardless of whether or not SecureDart is used.

We believe that developing new applications in SecureDart is no more difficult than Dart applications. We developed E-Commerce, Forum, and Turnin from scratch directly in SecureDart. We used the RPC communication model from the very beginning. When we wrote the code, we simply used the expected architecture, and thus it did not require extra work.

Chapter 7

Related Work

Several previous works focused on JavaScript security issues [16, 31, 33]. The work of Guarnieri *et al.* detects client-side vulnerabilities using a static taint analysis for JavaScript, which scales to large programs. Their experimental results show that they find many vulnerabilities in client-side JavaScript code. However, pure static analysis suffers from imprecision problems and many other vulnerabilities related to the server-side code cannot be detected by their analysis. Our technique focuses on designing a combined static and dynamic approach to avoid the imprecision problem.

Type systems have been used to control information flow and avoid vulnerabilities in previous work [21, 22, 28, 13]. We use an information flow type system to identify potentially harmful types of requests that must be validated. Researchers have developed language support for designing hardware that provably enforces information flow policies[20, 34].

Aliasing [8, 11, 6, 10, 12, 24, 19, 5] is a challenge for building object-oriented programs. One solution to constrain aliasing is to assign each object a unique reference [10]. In SecureDart, we use a similar mechanism to prevent aliasing of objects between trusted and normal

components.

MrCrypt [30] provides a programming environment for targeting homomorphic encryption schemes that guarantees data confidentiality for computations run on untrusted machines. This tool provides programming support for targeting existing homomorphic encryption schemes. However, efficient homomorphic encryption schemes are limited to few operations and conversions between schemes must be performed by trusted computers and require shuffling the data back and forth.

Proof-carrying code [23] provides a mechanism to validate the safety of programs from an untrusted source. The code recipient checks a set of rules that guarantee safe behavior of programs, and the code producer generates a certificate that proves that a source program adheres to the safety policies. SecureDart's goal differs in that it seeks to prove that the results of an execution on an untrusted device are legitimate.

Accountable Virtual Machines (AVM) [17] are VMs that provide users with the capability to audit a program's execution comparing its log with a log of a known-good execution. PeerReview [18] is a system that provides accountability for distributed systems — it guarantees that Byzantine faults observed are eventually detected and linked to a faulty node by maintaining a secure record of messages sent and received by each node and comparing it to a reference implementation. While SecureDart also uses a comparison-based approach, SecureDart focuses on web applications and is language-based.

Chapter 8

Conclusion

Developing web applications requires careful consideration for security as the client-side components are untrusted. SecureDart is a promising approach for securing the server code of web sites against attacks by malicious users.

In summary, SecureDart has the following novel properties:

- Safety: All the client requests that have side effects on the server will be validated by re-executing the client-side code. Tempered requests will be rejected and discarded.
- Liveness: All the legitimate requests will pass the server-side validation

The contributions of SecureDart include:

- We designed a annotation based type system to partition the web application code, which will be used to provide the liveness property.
- We presented a runtime system that will validate all the client-side requests that will have side effects on server.

- We implemented SecureDart as an extention of Dart.
- We evaluated SecureDart on real world web applications. Our experiments shows that SecureDart can prevent tempered request attacks with minimal overhead.

Bibliography

[1] AngularDart: a web framework for Dart. `https://angulardart.org/`.

[2] Dart-ECommerce: A simple electronic commerce application. `https://github.com/neoben/eCommerce`.

[3] DartChat: A Dart chatroom. `http://jamesslocum.com/download/dart_websockets.tar.gz`.

[4] Issue mover for GitHub: A tool that makes it easy to move issues between GitHub repos. `https://github.com/google/github-issue-mover`.

[5] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2002.

[6] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, 1997.

[7] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proceeding of the 1996 International Cryptology Conference*, 1996.

[8] J. Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 2007 ACM Symposium on Operating Systems Principles*, 2007.

[10] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *Proceedings of the 2003 European Conference on Object-Oriented Programming*, 2003.

[11] D. G. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2002.

[12] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *Proceedings of the 2006 European Conference on Object-Oriented Programming*, 2006.

[13] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, 2014.

[14] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, 2010.

[15] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, 2009.

[16] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 ACM International Symposium on Software Testing and Analysis*, 2011.

[17] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[18] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 175–188, 2007.

[19] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 2010 European Conference on Object-Oriented Programming*, 2010.

[20] X. Li, M. Tiwari, T. Sherwood, V. Kashyap, V. R. Rajarathinam, B. Hardekopf, J. K. Oberg, R. Kastner, and F. T.Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 2014 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[21] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[22] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedfings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[23] G. Necula. Proof-carrying code. In *Proceedings of the 1997 ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 1997.

[24] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proceedings of the 1998 European Conference on Object-Oriented Programming*, 1998.

[25] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[26] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: Detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, 2013.

[27] S. Son, K. S. McKinley, and V. Shmatikov. FixMeUp: Repairing access-control bugs in web applications. In *Proceedings of the 2013 Network and Distributed System Security Symposium*, 2013.

[28] A. J. Summers and P. Muller. Freedom before commitment-a lightweight type system for object initialisation. In *Proceedings of the 2011 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2011.

[29] Symantec. Internet security threat report volume 20, 2015.

[30] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. MrCrypt: Static analysis for secure cloud computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[31] O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proceedings of the 2014 ACM International Symposium on Software Testing and Analysis*, 2014.

[32] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[33] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[34] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the 2015 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.