Gradual Typing Embedded Securely in JavaScript

Nikhil Swamy
Juan Chen
Microsoft Research

Cédric Fournet
Pierre-Yves Strub
University of Maryland

Aseem Rastogi
INRIA

Karthikeyan Bhargavan
IMDEA Software Institute

Gavin Bierman

\{nswamy, fournet, juanchen, gmb\}@microsoft.com, aseem@cs.umd.edu, karthikeyan.bhargavan@inria.fr, pierre-yves@strub.nu

Abstract

JavaScript’s flexible semantics makes writing correct code hard and writing secure code extremely difficult. To address the former problem, various forms of gradual typing have been proposed, such as Closure and TypeScript. However, supporting all common programming idioms is not easy; for example, TypeScript deliberately gives up type soundness for programming convenience. In this paper, we propose a gradual type system and implementation techniques that provide important safety and security guarantees.

We present TS*, a gradual type system and source-to-source compiler for JavaScript. In contrast to prior gradual type systems, TS* features full runtime reflection over three kinds of types: (1) simple types for higher-order functions, recursive datatypes and dictionary-based extensible records; (2) the type any, for dynamically type-safe TS* expressions; and (3) the type un, for untrusted, potentially malicious JavaScript contexts in which TS* is embedded. After type-checking, the compiler instruments the program with various checks to protect the type safety of TS* despite its interactions with arbitrary JavaScript contexts, which are free to use eval, stack walks, prototype customizations, and other offensive features. The proof of our main theorem employs a form of type-preserving compilation, wherein we prove all the runtime invariants of the translation of TS* to JavaScript by showing that translated programs are well-typed in TS*, a previously proposed dependently typed language for proving functional correctness of JavaScript programs.

We describe a prototype compiler, a secure runtime, and sample applications for TS*. Our examples illustrate how web security patterns that developers currently program in JavaScript (with much difficulty and still with dubious results) can instead be programmed naturally in TS*, retaining a flavor of idiomatic JavaScript, while providing strong safety guarantees by virtue of typing.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics; D.4.6 [Operating Systems]: Security and Protection—Verification.

Keywords type systems; language-based security; compilers

1. Introduction

Writing secure JavaScript is notoriously difficult. Even simple functions, which appear safe on the surface, can be easily broken. As an illustration, consider the script below, simplified from actual scripts in the OWASP CSRFGuard (2010) and Facebook API (2013), security-critical JavaScript libraries that aim to impose access controls on the networking APIs provided by web browsers.

```javascript
function send(u,m) { /* Stand-in for XMLHttpRequest */
    function protect(send) {
        var whitelist = { "https://microsoft.com/mail": true, "https://microsoft.com/ova": true};
        return function (url, msg) {
            if (whitelist[url]) send(url, msg);
        };
    }
    send = protect(send);
    /* exploit */
    send("http://evil.com") = true;
    send("http://evil.com", "bypass!");
}
```

This is just one attack on protect; similar attacks are often the consequence of unintended callbacks to untrusted code (caused, for example, by implicit coercions, getters or setters, prototype hacking, and global object overwriting). Experimentally, we found and reported several such security flaws in the OWASP CSRFGuard and Facebook API scripts, suggesting that untrusted callbacks remain difficult both to prevent and to contain.

1.1 Attacks \( \equiv \) Type Errors Arguably, each of these attacks can be blamed on JavaScript’s lax notion of dynamic type-safety. Many runtime behaviors that are traditionally viewed as dynamic type errors (e.g., accessing non-existent properties from an object) are not errors in JavaScript. However, almost any such error while running a sensitive script can be exploited by a malicious context, as follows. Anticipating that the script will dereference an otherwise undefined property \( z \), which in JavaScript would just return the value undefined, a hostile context can define

```javascript
Object.defineProperty(Object.prototype, "z",
    { get: function() { /* exploit */ } });
```

then run the script and, as \( z \) is dereferenced and triggers the callback, access any argument on the caller stack. Thus, for protecting good scripts from bad ones, silent type errors in JavaScript are just
as dangerous as buffer overruns in C. Despite the numerous dynamic checks performed by the JavaScript runtime, some stronger notion of type safety is called for.

Other researchers have made similar observations. For example, Fournet et al. (2013) show several attacks on a similar piece of code and argue that carefully mediating the interaction between a script and its context is essential for security. They show how to compile \(^*\), a statically typed, ML-like source language to JavaScript in a fully abstract way, allowing programmers to write and reason about functions like \texttt{protect in ML}, and to let their compiler generate secure JavaScript automatically. This is an attractive design, but the sad truth is that millions of JavaScript programmers are unlikely to switch to ML. Meanwhile, Bhargavan et al. (2013) have developed DJS, a minimal, statically typed, secure core of JavaScript, primarily for writing first-order, string-processing functions using arrays of fixed size. DJS is suitable for writing security-critical code, like cryptographic libraries, and Bhargavan et al. prove that the behavior of programs accepted by the DJS type checker is independent of the JavaScript environment in which they may run. Many others propose analyses for subsets of JavaScript (Chugh et al. 2012; Guarnieri and Livshits 2009; Guha et al. 2011; Hedin and Sablefeld 2012; Swamy et al. 2011; Taly et al. 2011), although the guarantees provided only hold if the entire environment in which the program runs is also in the subset in question. Efforts like AdSafe (http://www.adsafe.org) and Caja (http://code.google.com/p/google-caja) address the problem of safely loading untrusted JavaScript into a program’s environment after analyzing and rewriting it. However, in order to do this reliably, various JavaScript features are forbidden, e.g., Politz et al. (2011) show that by ruling out many commonly used JavaScript features (e.g., \texttt{eval} or explicit prototype manipulation), AdSafe can safely sandbox untrusted code. Although we find the work on JavaScript subsets a promising line to pursue (indeed, even the ECMAScript standard is evolving towards the definition of subsets, such as the strict mode), assumptions about subsetting the environment are hard to validate in the presence of cross-site scripts.

### 1.2 TS*: a gradually type-safe language within JavaScript

This paper presents TS*, a source programming language that retains many of the dynamic programming idioms of JavaScript, while ensuring type-safety even in an untrusted JavaScript environment. TS* supports writing functions like \texttt{protect} exactly as shown, while a compiler from TS* to JavaScript ensures that the access control check in \texttt{protect} cannot be subverted.

Although significantly more flexible than \(^*\) or DJS, TS* still rules out many inherently unsafe features of JavaScript for the code we protect, thereby enforcing a stricter programming discipline and facilitating security code reviews. We intend TS* to be used to protect security-critical scripts—guarding, for instance, sensitive resources and capabilities—executed on web pages that also include dynamically loaded, untrusted, potentially malicious scripts. By necessity, most of the code running on these pages is provided by third parties; we leave such code unchanged (and unprotected). Nonetheless, by lightly rewriting the security-critical scripts, and gradually typing them (Siek and Taha 2006), we enforce a strong notion of dynamic type-safety. In addition, we protect our scripts using type-directed wrappers to shield them from adversarial JavaScript environments. This places them on a robust type-safe foundation, from which we can reason about their security. In comparison, other gradually typed extensions to JavaScript, like TypeScript (http://www.typescriptlang.org) and Closure (https://developers.google.com/closure/compiler), aim primarily to increase programmer productivity by using types for documentation and code completion, but provide no safety guarantee.

The concrete syntax of TS* extends JavaScript with a type-annotation language based on TypeScript. Our implementation works by first using TypeScript to infer types for every sub-term. It then uses the inferred types to type-check the program once again, this time using a more restrictive type system. If this step is successful, the program is compiled to plain JavaScript (instrumented with various runtime checks to enforce type-safety) for execution. The TS* type system itself has the following prominent features:

#### A statically typed core of functions, datatypes and records

The base type system of TS* includes primitive types like \texttt{bool, number, and string}; higher-order function types; recursive datatypes; and extensible records of fields with optional immutability annotations. Records are equipped with a structural subtyping relation. For example, the type \texttt{point} defined below is a subtype of a record that omits some of its fields, and function subtyping is, as usual, contravariant on the arguments and covariant on the results.

\[
x : \text{number};
y : \text{number};
setX : \text{const (number \rightarrow \text{unit})}
\]

#### Dynamically typed fragment

The type \texttt{any} is for dynamically typed TS* expressions. All the types from the statically typed core are subtypes of \texttt{any}, and all TS* terms whose subterms all have type \texttt{any} can always be given the type \texttt{any}. In the spirit of JavaScript, in the \texttt{any}-fragment, we view records as extensible dictionaries with string-valued keys. TS* supports the use of computed properties, e.g., in \texttt{any}-typed code, expressions like \texttt{p["set\"\+\('x\)'\]} or \texttt{whitelist[\url]} are legal ways to safely project the appropriate field from the underlying object, if it exists. Dynamically typed code and its interactions with statically typed code are instrumented by our compiler for safety. As far as we are aware, ours is the first gradual type system to soundly support dictionary-based mutable records and computed properties.

#### Runtime reflection over types

Case analysis on the runtime type of a value is a common idiom in JavaScript and other dynamically typed languages—Guha et al. (2011) present several typical uses of JavaScript’s \texttt{typeof} operator. TS* embraces this idiom and compiles programs with runtime type information (RTTI) to support introspection on all source types at runtime, e.g., \texttt{isTagPoint\+p} checks whether the RTTI of \texttt{p} is a structural subtype of \texttt{point}. In addition to providing an expressive source programming construct, RTTI also forms the basis of an efficient and simple enforcement mechanism for gradual typing, an alternative to prior proposals based on higher-order contracts (Findler and Felleisen 2002).

\texttt{un, the type of the adversary, mediated by wrappers} Finally, and most distinctively, TS* provides a second dynamic type, \texttt{un}. Any JavaScript term which cannot be typed either statically or dynamically can be given the type \texttt{un} and simply passes through our compiler without further analysis or modification. As such, \texttt{un} is the type of arbitrary, potentially adversarial JavaScript expressions. Our \texttt{un} type is reminiscent of types for adversaries, as proposed by Gordon and Jeffrey (2001). However, unlike prior uses of \texttt{un} in the context of secure compilers (e.g. Fournet et al. 2013), \texttt{un} is a first-class type in TS*: \texttt{un} values may be stored in records, used as arguments and results of functions, etc. The type \texttt{un} is incomparable to \texttt{any} in the subtyping relation and, in contrast with \texttt{any}, all operations on \texttt{un} values are mediated by wrappers that safely build coercions to and from \texttt{un} (as well as other types). The wrappers enforce a strict heap separation between the un-context and typed TS*, ensuring that adversarial code cannot break type invariants.

#### Evaluating TS*: theory and practice

We specify our compiler as a type-directed translation relation (§3). To formalize properties of the translated program, we give TS* a translation semantics to JS*, a dependently typed model of JavaScript developed by Swamy et al. (2013), which is in turn based on \texttt{λJS} by Guha et al. (2010). Precise monadic refinement types in JS* allow us to conveniently phrase our metatheory (§4) in terms of type-correctness of JS*, yielding three main properties:
Memory isolation: un-typed code (i.e., the adversary) cannot directly call, read, write, or tamper with typed values.

Static safety: statically typed code is safely compiled without any runtime checks, even in the presence of type-changing updates.

Dynamic safety: runtime type information is sound and at least as precise as the static type.

Experimentally, we evaluate TS* by programming and adapting several security-sensitive JavaScript web libraries (§6). Our examples include a OWASP reference library to protect against cross-site request forgeries (CSRF) (Barth et al. 2008); and an adaptation of secure login and JSON-validation scripts within the Facebook API.

In summary, our main contributions include:

1. a type system and runtime support for safely composing statically typed, dynamically typed, and arbitrary JavaScript; (§3)
2. a type safety theorem and its proof by translation to TS*; (§4)
3. a prototype implementation, including a protocol to ensure that our runtime support runs first on pages hosting compiled TS*, and securely initializes our type invariant; (§5)
4. security applications, illustrating a series of authorization and tacks, sample web-application deployments, and a technical report with the full formalization and proofs are all available from http://research.microsoft.com/fstar.

2. An overview of TS*

We begin by presenting the design of TS* informally, using several small examples for illustration. We use the concrete syntax of TypeScript with minor simplifications: for instance we write function types as \( t \to t' \) rather than TypeScript's \( (s:t) \Rightarrow t' \). We also extend the syntax of TypeScript with datatype declarations and immutability qualifiers.

2.1 Gradually securing programs by moving from un to any

While we envisage TS* as the basis of a full-fledged gradually typed web-programming language, we initially consider JavaScript programs willing to harden safety- and security-critical fragments of their code. They can start by giving their existing JavaScript code the type un in TS*, and then gradually migrating selected fragments to (dynamically) type-safe code in TS* using the type any.

This exercise is valuable since any code in TS* enjoys a memory isolation property, a robust foundation upon which to build secure sub-systems of a larger program. Memory isolation alone prevents many common attacks. For example, the prototype poisoning attack of §1 occurs because of a failure of memory isolation: the command `whitelist[url]` causes a prototype-chain traversal that ends with reading a field of `Object.prototype` which, unfortunately, is a reference to an object controlled by the adversary. Passing `protect` (unchanged) through the TS* compiler while giving `whitelist` the type any produces an instrumented version of `protect` with checks to prevent any property dereferences from `whitelist` from accessing `Object.prototype` (or any other un object), thereby foiling the attack. Specifically, from memory isolation, we can prove that every dereference of a field of any object in TS* will only read the immediate fields of that object, and will never access a prototype controlled by the adversary. This ensures that `whitelist[url]` returns true only if `url` is immediately defined in `whitelist`.

Undecorated TS* programs can generally be given the type any (as long as they are well-scoped). Every function parameter in an unannotated TS* program defaults to the type any; every var-bound variable is given the type of its initializer. Under this convention, in the program from §1, the type of `protect` is any \( \to (\text{any,any}) \to \text{any} \), which is a subtype of any. When deploying a TS* program, we assume that the JavaScript global object (the `window` object in most browsers) and all objects reachable from it are under control of the attacker. Thus, it is not safe to simply store `protect(send)` into `window.send`, since that would break memory isolation and leak a value of type any to un-safe code—our type system prevents the programmer from doing this by mistake.

Instead, TS* provides wrappers to safely export values to the context. The TS* expression `wrapUn(wp(send))` wraps the `protect(send)` closure and yields a value of type un, indicating that it is safe to hand to any JavaScript context while preserving memory isolation. Dually, for e:un, the expression `wrapAny(e)` safely imports e from the context and gives it the type any.

Providing a JavaScript implementation of `wrap` is non-trivial. We base our implementation on wrappers defined by Fournet et al. (2013). Their wrappers are designed to safely export statically typed values from the translation of an F* program (roughly, a simply typed subset of ML) to its JavaScript context; and to import untyped values from the context into F* at specific types. For example, Fournet et al.'s `downwp` exports a pair of translated F* values \((v_1,v_2)\) of type \((t\to u)\) to the context, by building a new object with two fields initialized to `down`, \((v_1)\) and `down`, \((v_2)\). A corresponding wrapper `upwp` does the converse, safely copying a pair from the context and building a value that is the translation of an F* pair of type \((t\to u)\). Fournet et al. provide `upwp` and `downwp` for types \(t\) including `unit`, `bool`, `string`, `number`, `pairs`, `recursive datatypes`, and functions. We extend their constructions to additionally build wrappers to and from the type any, and to handle cyclic data structures that can be constructed using records with mutable fields.

To illustrate wrappers in action, we elaborate on our first example. Suppose we wished to protect `window.send` (a fictitious but simpler stand-in for JavaScript's `XMLHttpRequest object`) with an access control check. To support this, the standard library of TS* provides a facility to read fields from, and write fields to, the global object by including the following safe interface to the `window` object implemented in JavaScript. The `win` object, whose interface is shown partially below, shadows the fields of the `window` object, safely reading and writing it within a wrapper to ensure that the attacker-controlled `window` does not break type safety. Using `win` within a TS* program, we can safely import `window.send`, protect it, and export it back to the context using the following snippet of code, typed in a context where the `win` object has mutable un-typed fields. Of course, the attacker may `a priori` obtain a copy, and even redefine `window.send` before our code has the chance to protect and update it, but this is an orthogonal problem, solved once for all TS* programs—§2.3 and §5 present our mechanisms to ensure that our scripts run first on a web page.

```plaintext
interface win {
    send : un;
    ...}

win.send = wrapUn<(protect<wp(send))>

Wrappers are expensive, since they deeply copy the contents of objects back and forth, and—by design—they are not necessarily semantics-preserving. (For instance, they sanitize values, filter out some properties, and prevent some aliasing.) Thus, in an attempt to minimize the amount of copying, we may rewrite `protect`, by adding a few types, as shown below.

```
Intuitively, the \( \texttt{msg} \) argument in the closure returned by \texttt{protect} is treated abstractly, that is, our script does not directly access it. Thus, there is no need to import that argument from the context (potentially performing a deep copy). On the other hand, the \( \texttt{self} \) argument is not abstract—it is used to project a field from the original, and, as such, it had better be a string. The type system of \( \texttt{TS} \) gives us the flexibility to express exactly what should be imported from the context, helping us find a good balance between security and performance. The explicit use of \( \texttt{un} \) and \texttt{wrap} are advances of \( \texttt{TS} \) relative to prior languages such as \( \texttt{P} \) or, for that matter, any prior gradually typed programming language.

2.2 Expressing invariants with assertions over runtime types

As can be expected of gradual typing, a \( \texttt{TS} \) program migrated from \( \texttt{un} \) to \( \texttt{any} \) then can, with some effort, be made increasingly statically typed. Static types can improve runtime safety, robustness of code, modularity, as well as provide better IDE support. Static types in \( \texttt{TS} \) also improve performance relative to any-typed code and, relying on RTTI, can enforce data invariants. This is enabled by static safety and dynamic safety, two properties (in addition to memory isolation) provided by \( \texttt{TS} \).

Static safety \( \texttt{TS} \) ensures that, at runtime, no failures happen during the execution of statically typed parts of the source program. Since there are no runtime checks in the compiled JavaScript for such parts, as a bonus, the performance of statically typed \( \texttt{TS} \) code will approach that of native JavaScript (and potentially exceed it, if the type information can be communicated to the VM).

Dynamic safety Every \( \texttt{TS} \) value \( v : t \) (where \( t \neq \texttt{un} \)) is compiled to JavaScript with runtime type information (RTTI) that initially reflects \( v \)'s static type \( t \). \( \texttt{TS} \) ensures that, while \( v \)'s RTTI may evolve during execution (e.g., as fields are added to an extensible record), it is always (a) a subtype of \( v \)'s static type \( t \), and (b) a sound approximation (supertype of \( v \)'s most-precise current type, i.e., the RTTI of \( v \) may evolve only monotonically with respect to the subtyping relation. We call this property dynamic safety.

As an illustration, consider the example below, which codes up a lightweight form of objects with extensible records and closures in \( \texttt{TS} \), where \texttt{point} is the type defined in §1.

```javascript
function Point(x, y) {
    var self = {};
    self.x=x;
    self.y=y;
    self.setTag= function(d:number) { self.x = d; };
    return setTag<point>(self);
}
```

The function \texttt{Point} creates a new point. It allocates a new empty record and stores it in the local variable \texttt{self}, then it adds three fields \( x, y, \) and \texttt{setX}. The static type of \texttt{self} is just the empty record. However, \( \texttt{TS} \) allows us to add more fields to \texttt{self} than those documented in its static type. As such, the static type of a record only describes a subset of the fields in the term, as is usual with width-subtyping. Deleting fields from records is also possible—we discuss this in more detail in §3.

In the last line of \texttt{Point}, \texttt{setTag<point>} checks at runtime if the content of \texttt{self} is compatible with the \texttt{point} type, and fails otherwise. The term \texttt{setTag<point>} has static type \texttt{point}, although the static type of \texttt{self} remains unchanged.

Assertions like \texttt{setTag} allow source programmers to safely update RTTI while maintaining the runtime type invariant. Once a value has been tagged as a \texttt{point}, then it is guaranteed to always remain a point. A programmer may choose to add fields to a \texttt{point}, or to further update its type information (e.g., turning it into a colored\texttt{point}), but it will always contain at least the fields of a \texttt{point}.

Any attempt to delete, say, the \( x \) field, or to change it in a type-incompatible way (using for instance a dynamically typed alias to the point) will cause a runtime error.

In contrast, statically typed code raises no such errors. \( \texttt{TS} \) infers that function \texttt{Point} has type \( \texttt{any} \rightarrow \texttt{point} \), so the code below is statically type safe and does not require any runtime checks.

```javascript
var o = Point(0,0) ; o.setX(17);
```

As another example, consider that popular web frameworks, like Dojo, provide implementations of the JSON Schema standard. This allows programmers to validate JSON data, writing verbose schemas for them also as JSON objects. In \( \texttt{TS} \), data invariants can be expressed and enforced directly using types, rather than via schemas. For example, to check that a JSON string can be parsed into an array of user identities, we can write the \texttt{TS} code below, assuming that \texttt{JSON.parse} has type \( \texttt{string} \rightarrow \texttt{any} \). (See online materials for a \texttt{TS} implementation of a JSON parser.)

```javascript
type users = {id:number; user:string}[]

function check(j:string) : users {
    var o = JSON.parse(j);
    if (canTag<users>(o)) return setTag<users>(o);
    else return [];
}
```

The schema is captured by the type \texttt{users}. We parse a string \( j \) as JSON using \texttt{JSON.parse}, then use the \texttt{TS} type operator \texttt{canTag\<t\>} to check that \( o \)'s contents are consistent with \( \langle \texttt{t} \rangle \). If the check succeeds, we stamp \( o \) as a valid \texttt{users} object and return it.

2.3 Reliable primitive operations Since the global \texttt{window} object is shared with the adversary, all objects reachable from \texttt{window} may be compromised. This includes all built-in objects provided by the VM, e.g., \texttt{Object.prototype}, \texttt{Array.prototype}, the default \texttt{String} object, and others. In order to ensure memory isolation, translated \texttt{TS} programs should never read from any of those objects. This is remarkably difficult to arrange in JavaScript, since several primitive operations, e.g., reading and writing fields, depend on base prototypes, as illustrated in §1. Thus, in the face of attacker-controlled prototypes, even simple manipulations of objects are unreliable.

Thankfully, there is a relatively simple way out. JavaScript (since ECMAScript 5) provides a function \texttt{Object.create}, such that \texttt{Object.create(arg)} initializes to \texttt{arg}. As a special case, \texttt{Object.create(null)} creates an object with a \texttt{null} prototype, meaning that the created object does not inherit the attacker-controlled \texttt{Object.prototype}. Our compiler uses \texttt{Object.create(null)} to create new objects in the translation of a \texttt{TS} program, allowing manipulations of object fields simply as \( \texttt{o.f = v} \) or \( \texttt{["f"] = v} \), without worrying that a traversal of \( o \)'s prototype chain will reach an attacker-controlled object.

There is one exception, however. Function objects cannot be allocated using \texttt{Object.create}—they must inherit from the attacker-controlled \texttt{Function.prototype}. However, by an invariant of the translation, we can ensure that the only field ever accessed of a non-un-typed function \( f \) is \( \text{\texttt{ritti}} \). So, as long as \texttt{Function.prototype} contains a safe, immutable \texttt{ritti} field, accessing \( f.\text{\texttt{ritti}} \) will never trigger adversarial code.

The correctness of our translation then requires a reliable way to call \texttt{Object.create(null)} and to initialize \texttt{Function.prototype.ritti}. To achieve this, compiled \texttt{TS} programs are linked with a library called \texttt{boot.js}. This library is intended to be the first piece of JavaScript that runs on a page—§5 discusses how to make a script run first, reliably, before any adversarial script. \texttt{boot.js} takes a clean copy of \texttt{Object.create} and stores it in an immutable field. Later, the translated \texttt{TS} code accesses \texttt{Object.create} from this field, rather than as \texttt{window.Object.create}, which is an attacker-controlled path. We show a fragment of \texttt{boot.js} below, simplified slightly to use

1 Based on \url{http://davidwalsh.name/json-validation}.

2 \url{http://tools.ietf.org/html/draft-zyp-json-schema-03}
Then it checks that the self object contains values in those three fields whose RTTIs are compatible with the requested types `number`, `number`, and `→ unit`, respectively. Once this check succeeds, `setTag` updates the `rtti` field of `self` to `[point]`. An invariant of our translation is that the `rtti` field of every object evolves monotonically with respect to the subtyping relation. That is, `self.rtti` was initially `[]` and evolves to `[point]`, where `point <: ()`. The RTTI of `self` may evolve further, but it is guaranteed to always remain a subtype of `point`. Line 12: we add an `rtti` field to the `Point`. Finally, lines 13 and 14: we see the translation of a statically typed fragment of `TS`. Pleasantly, the translation there is just the identity.

As shown, the translated program does not interact with its context at all. However, the programmer can choose to export certain values to the context by writing `export function Point(x,y)(...)` instead. This instructs the compiler to wrap and export `Point` to the context by inserting the following code after Line 14.

The listing below shows the translation of the `TS` function `Point` to JavaScript. The translated code is placed within a single enclosing function to introduce a fresh local scope. Without this precaution, `TS` definitions would implicitly leak into the global un-typed object.

> The type annotations in `TS` are all erased in JavaScript.

This section formalizes `TS` by presenting its type system and type-directed translation to JavaScript. We describe in particular our runtime representation of types and the JavaScript implementations of `wrap`, `setTag` and related functions that manipulate translated terms and their types. We conclude this section with a detailed comparison of `TS` with prior gradual type systems.

### 3.1 Syntax

Figure 1 presents our source syntax. To aid in the readability of the formalization, we employ compact, λ-calculus style notation, writing for example `λ.x.e` instead of `function(x)(r)`. We also write `[]` for a sequence of expressions `e1,...,en`, `f` for the application `f(e1,...,en)`, and so on. Our formal syntax does not cover the un-typed fragment, since its typing and compilation are both trivial, although, of course, our theorems specifically address the composition of compiled `TS` code with arbitrary JavaScript.

Values `v` include variables `x`, Booleans, typed `λ`-abstractions, and data constructors `D` applied to a sequence of values. For conciseness, we exclude primitives like numbers and strings, since they can in principle be encoded using data constructors. In practice, our implementation supports JavaScript primitives, and so we use them in our examples. This requires some care, however, since some operations on primitive types can be overwritten by the adversary. In such cases, we cache reliable versions of those primitive operations in the `boot.js` file, instead. This instructs the compiler to wrap and export `Point` to the context by inserting the following code after Line 14.

```javascript
win.Point = Clean.wrap((any,any) → point), (un),Point;
```

Field names such as `rtti`, instead of the longer names (less likely to clash with source code) used in our implementation.

```javascript
1 function boot() {
2 var Clean = Object.create(null);
3 var o = Point(0,0);
4 o.setX(17);
5 ;

let self = Object.create(null);

let t = (function(x,y) {
3 var self = Clean.create(null);
4 var self.rtti = [];;
5 write(self, "x", x);
6 write(self, "y", y);
7 var tap = function(d) {write(self, "x", d);}
8 tap rtii = [number → unit];
9 write(self, "setX", tap);
10 return Clean.setTag([(],(point),self);
11 }
12 Point.rtti = [(any,any) → point];
13 var o = Point(0,0);
14 o.setX(17);
15 ;

Line 3: the source empty record `{}` is compiled to a new null-prototype object. Line 4: we set the `rtti` field of `self` to the translation of a source empty record type. Lines 5 and 6: we use the macro `write` to add two properties to the `self` object. This macro (defined in §3) checks that the RTTI of the assigned field (if any) is compatible with the RTTI of the assignee. In this case, since the `rtti` field of `self` is just the empty record, it does not constrain the contents of any of its fields, so these assignments succeed and the fields are added to `self`. Line 7: we translate `setX` and, line 8: we tag it with an `rtti` field recording its source type. We then add it to `self` using `write`. Line 10: the call to `Clean.setTag` checks whether `self`, whose static type is represented by `[(]`, contains a valid representation of a source point. For this, it examines the representation of the type `(point); notice that the type requires three fields, `x`, `y`, and `setX; then it checks that the `self` object contains values in those three fields whose RTTIs are compatible with the requested types `number`, `number`, and `→ unit`, respectively. Once this check succeeds, `setTag` updates the `rtti` field of `self` to `[point]`. An invariant of our translation is that the `rtti` field of every object evolves monotonically with respect to the subtyping relation. That is, `self.rtti` was initially `[]` and evolves to `[point]`, where `point <: ()`. The RTTI of `self` may evolve further, but it is guaranteed to always remain a subtype of `point`. Line 12: we add an `rtti` field to the `Point`. Finally, lines 13 and 14: we see the translation of a statically typed fragment of `TS`. Pleasantly, the translation there is just the identity.

As shown, the translated program does not interact with its context at all. However, the programmer can choose to export certain values to the context by writing `export function Point(x,y)(...)` instead. This instructs the compiler to wrap and export `Point` to the context by inserting the following code after Line 14.

```javascript
win.Point = Clean.wrap((any,any) → point), (un),Point;
```
Figure 2. A type-directed translation of $\text{TS}$ to JavaScript

cation; and conditionals. Finally, we have RTTI-based query operations $q(t/e)$, and coercions $c(t/e)$.

Types, $t$, include a number of primitive types (bool for boolean values, and any and un for dynamic values), abstract data types ranged over by $T$, and records. Record types are defined using the shorthand $\{f_1 : t_1, \ldots, f_n : t_n\}$ to denote the type $\{f_1, \ldots, f_n\}$ where the $f_i$ are distinct and the $a_i$ are accessibility annotations: $t$ for read-only, and $w$ for mutable. We also write $u \bowtie t$ for the record type $\{f_1 : t_1, \ldots, f_n : t_n\}$, where $r = \{f_1, \ldots, f_n\}$ and $u = \{f_1 : t_1, \ldots, f_n : t_n\}$.

The type system is given with respect to a signature $S$ which maps data constructors $D$ to their type signature, written $D : T$. In places we need to refer to all the data constructors for a given abstract data type $T$ in the signature $S$. We use the shorthand $[S]$ which is defined as $[D : T | S(D) = T \rightarrow T]$. We also have a standard type environment $\Gamma$ binding variables to their types.

Although we have data constructors, matching patterns in $\text{TS}$ is not primitive. Instead, it can be encoded in terms of the other constructs, as defined below. Note that we freely use $\text{as}$ and other Boolean operators, as well as $\text{as} \equiv \text{typeof}$, physical equality on $\text{TS}$ values:

```
match e with $D_{f_1 \ldots f_n} t \rightarrow e_1$ else $e_2$ $\triangleq$
  let $y = \text{in } [S]$ in $e_1$ else $e_2$
```

3.2 Type system and translation Figure 2 defines the judgment $\Gamma \vdash t : T \Rightarrow s$, which states that in an environment $\Gamma$ (along with an implicit signature $S$), the expression $e$ can be given the type $t$ and be translated to the JavaScript program $s$. We present the type system declaratively. In practice, our type checker analyzes a fully decorated AST produced by the type inference algorithm of TypeScript, so implementing the system in Figure 2 is straightforward. (A precise description of this type inference algorithm is beyond the scope of this paper.)

At a high level, the type system is designed to enforce the following three properties mentioned in §2:

**Static safety** $\text{TS}$ programs have no failing dynamic checks during the execution of statically typed sub-terms. We achieve this via two mechanisms: (a) the rules prefixed by $(\text{T})$ enforce the static typing discipline and they never insert any runtime checks when compiling the program; and (b) when a value $v$ is any is passed to a context that expects a precise type, e.g. $\text{point}$, the compiler inserts instrumentation to ensure that $v$ is indeed at least a $\text{point}$. Instrumentation inserted elsewhere in dynamic code also ensures that $v$ henceforth remains at least a $\text{point}$. This protects statically typed code from future modifications to $v$. In the other direction, the type system allows for $v$ to be passed to any-typed context via subtyping.

**Dynamic safety** The RTTI of $\text{v}$ is always a subtype of $t$ and a sound approximation of $v$’s most precise type—by two mechanisms: (a) $v$’s RTTI initially reflects $t$ and the $\text{setTag}$ operation ensures that RTTI always evolves towards the more precise types per subtyping, and (b) the rules prefixed by $(\text{A}\cdot)$ instrument the translation of the any-typed parts of the source to enforce that modifications to $v$ respect its RTTI. (We envision that an IDE can highlight uses of $(\text{A}\cdot)$ rules to the programmer as potential failure points.)

**Memory isolation** un-typed code cannot directly access an object reference that $\text{TS}$ code may dereference; this is enforced by ensur-
ing that the un type is treated abstractly. The only way to manipulate un values is via defensive wrappers, which means that typed code never dereferences an un-typed memory location, and that any-typed references are never be handed directly to the adversary. The subtyping rules are designed to prevent masking the presence of un-values in records using width-subtyping or subtyping to any.

We now turn to describing each of the rules in detail. The first rule in the judgment, \( \text{T-Sub} \), is a subsupposition form which shows that a use of subtyping in \( T' \) does not change the translation of a term. The subtyping relation \( S \models t <: t' \) (also in Figure 2) is mostly standard. Depth subtyping on records is permitted only for immutable fields. The penultimate rule allows all types that do not contain the un type to be a subtype of any. (The auxiliary predicate unFree detects occurrences of un in a type.) Allowing un <: any would clearly break our invariants. Allowing (\( \text{if:un} \) <: any) is also problematic, since if a value \( v: (\text{if:un}) \) could be promoted to v: any, then \( v[\text{un}] \) would also have type any, even though it produces an untrusted value. The last subtyping rule provides width-subtyping on records, forgetting the fields to weaken \( t \) to \( u \), only so long as \( t \) contains no occurrences of un.

The rule \( \text{T-X} \) for typing variables is standard. \( \text{T-Rec} \) introduces a record at type \( u \), such that \( u \) includes all the un-fields (necessary for compatibility with subtyping). Its compilation allocates a new object, safely sets the fields \( f \) to \( s \), and finally adds an \( \text{rtti} \) field containing \( \{u\} \). The rule \( \text{T-D} \) for typing data constructors is similar. The typing of functions with \( \text{T-Lam} \) is standard; however, the translation to JavaScript is a bit subtle: it defines a JavaScript function tagged with an \( \text{rtti} \) field, whose body \( s \) is preceded by declarations of all the \( \text{let} \)-bound variables in \( e \), the source function body. These (and other) rules use the JavaScript form \( (\mathcal{F}) \), which evaluates every \( e_i \) in \( \mathcal{F} \) and returns the last one.

The rules \( \text{T-Wr}, \text{T-Rd} \), and \( \text{T-App} \) are standard. One slight wrinkle in \( \text{T-Ir} \) is that we rely on JavaScript implicitly converting \( u \) to a boolean—this implicit conversion is a safe primitive operation. One could imagine a stricter variant of \( \text{T-Ir} \) with a runtime check to ensure that \( u \) is a boolean, without applying implicit conversions. However, \( \text{T-Ir} \) as shown is more idiomatic of JavaScript. On the other hand, our treatment of local variables in \( \text{T-Let} \) deviates from JavaScript’s \( \text{var} \)-statements—local variables are always immutable in \( T' \). We make this choice since mutability, if desired, can be encoded by boxing the let-bound value in a mutable record field. In contrast, encoding immutable local variables given only mutable ones is impossible (even with immutable record fields).

The rules \( \text{T-Q} \) and \( \text{A-C} \) cover queries \( q(i) \) and coercions \( c(i)e \). In each case, we have an expression \( e' \) compiled to \( x \), and we apply \( q \) or \( c \) at type \( t \), so long as \( t \) and \( t' \) are compatible. Type compatibility is a simple reflexive, symmetric, and non-transitive relation, in the spirit of Siek and Taha (2006). We discuss the implementation of these operations in the next subsection.

The remaining (A-) rules instrument the translated programs to ensure safety. In \( \text{A-App} \), we first check that \( s \) is a function. Then, before calling the function, we tag the argument with the type of the function’s parameter. \( \text{A-Rd} \) simply checks that \( s_1 \) is an object and that \( s_1 \) has field \( s_2 \). \( \text{A-Wr} \) checks that \( s_1 \) is an object. It then checks that \( s_1 \)’s RTTI allows for field \( s_2 \) to be written. If \( s_1 \)’s RTTI does not contain \( s_2 \), it is treated as a new property addition—deleting a property if it is not present in the RTTI is also straightforward, although we do not cover it here. Otherwise, it should contain a mutable \( s_2 \) field, and before writing, \( s_3 \) is tagged with the type expected by \( s_1 \)’s RTTI.

### 3.3 Implementing RTTI-based coercions and wrappers

As described in §2.3, runtime support for compiled \( T'S \) programs is provided by the immutable \texttt{Clean} object installed in the global namespace by the first-starter script \texttt{boot.js}. This section discusses in detail the \texttt{setTag} and \texttt{wrap} operations provided by the runtime.

#### Updating RTTI with setTag

The form \( \text{setTag}(\{e\}(t e')) \) is compiled to \texttt{Clean.setTag(}\{\\(t e'\\)\}, where \( e \) compiles to \( s \). In this case, the first argument \( t e' \) is redundant, since it can be recovered from the RTTI of \( s \) (which, by dynamic safety, must be a refinement of \( t' \))—we keep the first argument for uniformity with wrappers, as discussed below. The call \texttt{clean.setTag(}\{\\(t e'\\)\}, \texttt{a}) boils down to \texttt{setTagAux(}\{\text{a}\\(t e'\\)false\}), whose implementation is shown in Figure 3 (on the left). This code tries to update the RTTI of \( s \) to be some subtype of \( \{t e'\} \) and fails otherwise. There are two main concerns that the code addresses: (1) the code should not diverge if \( s \) is a cyclic data structure (unless \texttt{a} is called); (2) if the tag of \( s \) is updated, then \( s \) must represent a value of type \( t \) and the new tag must be a subtype of both the old tag and \( t \). These two concerns are complementary, since certain types (e.g., non-recursive types or types without mutable fields) specifically require all their values to be acyclic. The main idea of \texttt{setTagAux(}\{\text{a}\\(t e'\\)false\}) is to traverse the objects reachable from \( x \) in a depth-first manner, at each point checking that each field required by \( t \) is present in the object at the expected type (lines 11 and 21). Each object encountered in the traversal is temporarily marked (line 7) as visited by assigning \texttt{new(curTag, x)} to \( x.tap \). The assigned value is the greatest subtype of both \( curTag \) and \( t \) (which may not exist, in which case the operation fails by calling \texttt{die}). Cycles are only permissible when traversing mutable object references and cycles are detected by looking for a marker in \( x.tap \). If the traversal succeeds, the temporary marker is made permanent by updating \( x.rtti \) (line 4); if it fails, a fatal error is raised by calling \texttt{die}, which exhausts the JavaScript stack. This is drastic but effective; friendlier failure modes are feasible too.

Coercions based on \texttt{setTag} can be overly conservative, particularly on higher-order values. For example, trying to coerce the identity function \texttt{id} : any \( \rightarrow \) any to the type \texttt{bool} \( \rightarrow \) bool using \texttt{setTag} will fail, since \texttt{any} is a \texttt{universal}. As such, RTTI-based coercions are most effective when working with mutable first-order objects. One way to broaden the scope of RTTI-based coercions is, of course, to enrich the type language, e.g., to include polymorphism, or even refinement types—we plan to explore this in the future. Additionally, we provide operations to query RTTI, \texttt{setQueryTag(}\{\text{a}\\(t e'\\)false\}) or \texttt{canSetQueryTag(}\{\text{a}\\(t e'\\)false\}), that (respectively) allow programmers to query the current tag of \texttt{e} and test whether or not a \texttt{setTag} operation on \texttt{e} would succeed. When RTTI-based coercions are inapplicable (e.g., on un-typed values) or overly conservative (e.g., on functions), the \texttt{wrap} form is handy—we describe it next.

#### Wrappers

When \( et' \) compiles to \( s \), \texttt{wrap(}\{\text{a}\\(t e'\\)false\}) compiles to \texttt{Clean.wrap(}\{\\(t e'\\)\}, \texttt{a})—Figure 3 shows part of its implementation (on the right). Line 2 is the case where both \( \{t e'\} \) and \( t' \) are function types. As expected, we apply a higher-order cast to \( s \), wrapping the argument, applying \( x \), and then wrapping the result. Thus, \texttt{wrap(}any \( \rightarrow \) any\(), \texttt{bool} \( \rightarrow \) bool\(), \texttt{id}\) succeeds where the corresponding \texttt{setTag} fails.

#### Securely importing from un

The more interesting case of \texttt{wrap} is when the source type is \texttt{un} and the target type is \texttt{any} (line 7), which occurs when importing an un-typed value from the adversary into the typed fragment of \( T'S \). At line 8, we use JavaScript’s \texttt{typeof} operator to examine the simple type of \( x \), the un-typed value being coerced. If \( x \) has a primitive type, then the coercion is just the identity. If \( x \) is an object, we aim to enumerate all its fields, then wrap and copy each of those fields into a newly allocated \texttt{any}-typed object. This requires some care, however, since directly enumerating or accessing the fields of an un-typed objects causes callbacks to the un-typed adversary, potentially breaking memory isolation by leaking any-typed objects on the stack to the context. As a defense, we build on an idea from Fournet et al. (2013), as follows. At line 14, we allocate a new object \( r \) into which we will wrap and copy the fields of \( x \). Next, we build a closure (\texttt{atub at line 15}) which captures
Securely exporting to `un`  
The implementation of `wrap(id, (un))` for `t` not `un` exports `x` to the context. This case is somewhat simpler, since `x` is a typed value with RTTI which can be safely probed. If `s` is a primitive, it can be exported as is. If `s` is an object, `wrap` creates an empty object `x`, enumerates all the fields of `s`, exports them recursively and adds them to `x`. For functions, it uses Fournet et al.’s `downfun` to export it at type `un` to `un`.

3.4 Discussion and related work on gradual typing  
Languages that mix static and dynamic types date back at least to Abadi et al. (1991) and Bracha and Griswold (1993). Gradual typing is a technique first proposed by Siek and Taha (2006), initially for a functional language with references, and subsequently for languages with various other features including objects. Several others have worked in this space. For example, Flanagan (2006) introduces hybrid typing, mixing static, dynamic and refinement types; Wadler and Findler (2009) add blame to a gradual type system; Herman et al. (2010) present gradual typing with space-efficient wrappers; Bierman et al. (2010) describe type dynamic in C2; and Igra and Igarashi (2011) add gradual typing to generic Java.

Our system is distinct from all others in that it is the first to consider gradual typing for a language embedded within a larger, potentially adversarial environment via the type `un`. We are also, to the best of our knowledge, the first to consider gradual typing as a means of achieving security.

To compare more closely with other systems, let us set aside `un` for the moment, and focus on the interaction between `any` and statically typed `t`. Previous type systems mediate the interactions between static- and any-typed code by implicitly attaching casts to values. Higher order casts may fail at a program point far from the point where it was inserted. To account for such failures, blame calculi identify the cast (with a label to indicate the term or context) that causes the failure—Siek et al. (2009) survey blame calculi based on the errors they detect, points of failures, and casts they blame. In contrast, Interactions between static- and any-typed `t`s is based primarily on RTTI. Casts (wrappers in our terminology) are never inserted implicitly, although they are made available to the programmer. This design has the following advantages.

Preservation of object identity  
Object identity in JavaScript is a commonly used feature. Since `T^S` does not implicitly attach casts to values, it never implicitly breaks object identity in the source during compilation. Previous gradual type systems with implicit casts would always break object identity.

Space efficiency  
Casts can pile up around a value, making the program inefficient. Herman et al. (2010) introduce a novel cast-reduction semantics to gain space-efficiency. Our approach is also space efficient (there is only one RTTI per object) but does not require cast-reduction machinery.

Static safety and eager failures  
In contrast to our RTTI-based mechanism, statically typed code in other gradual type systems could fail (although blame would help them ascribe it to the any-typed code). Consider the following `T^S` example.

```
let v: any = (f: true) in \( \lambda: f = \text{int} \). r.f v
```

Compiling this term using `(A->APP)` introduces a `setTag` on the argument `v` at type `(f: `int`). The `setTag` operation, at runtime, recursively checks that `v` is `f: `int`, and expectedly fails. Thus, the failure happens prior to the application, a failure strategy called `eager` in prior works. Herman et al. (2010) also argue that their system can provide eager failures, but transposed to their notation (with the record replaced by a `ref any`), the failure occurs at the property read within the statically typed `A` term, breaking static safety. When eager runtime checking seems too strict, `T^S` wrappers provide an escape hatch. Arguably, for our security applications, a predictably uniform eager-failure strategy is a suitable default.

Dynamic safety and blame  
With no failures inside statically typed code to explain at runtime, blame seems less useful with RTTI-based coercions. However, because we enforce dynamic safety

```
 function setTagAux(x, t, cyclelock) {
  if (typeof x !== "object" || x.tmp) die();
  var f = function (y) {
    if (typeof y !== "Any") return y;
  };
  f.rtti = x.rtti;
  var stub = function () {
    return x;
  };
  var any2un = function (x) {
    var r = Clean.create(null);
    return stub();
  };
  return ok();
}
```

```
function wrap (src, tgt) {
  if (src.t === "arrow" && tgt.t === "arrow") {
    var f = function (y) {
      return wrap(tgt.ret, y);
    };
    f.rtti = tgt; return f;
  }
}
```

```
var curTag = tagOf(x);
if (curTag.t !== "record") die();
setTagAux(x["field", ft.typt, ft.mut]);
return ok();
} else if (src.t === "un" && tgt.t === "any") {
  switch (typeof x) {
    case "undefined":
      case "string":
        var f = function (x) {
          return wrap(any2un, (x,[]));
        };
      callUn(stub); return r;
      case "function":
        var un2any = function (x) {
          return wrap(un, (x,[]));
        };
        var f = upfun(any2un, un2any)(x);
        f.rtti = {any: any}; return f;
        default: die();
    }
}
```
Gradual typing for monotonic objects

Independently, in an unpublished manuscript, Siek et al. (2013) investigate gradual typing for a Python-like language (based on an earlier abstract presented at the STOP workshop in 2012). Like us, they notice that constraining the type of a mutable object to evolve monotonically at runtime enables statically typed code to be compiled without runtime checks, leading to improved performance and safety for that fragment. There are some important differences, however. First, Siek et al. require monotonicity with respect to an intentionally naïve subtyping relation. For example, (transposed to our notation) the type of an object can evolve from \( t = \{ \text{t : \text{number}} \} \) to \( t' = \{ \text{t : \text{any}} \} \), although \( t' \) is not a subtype of \( t \). This means that, in their system, writing \( o.t = \text{true} \) can fail at runtime, even if \( o \) has static type \( t \), which is prevented by \( \text{TS} \). This difference stems perhaps from differing perspectives on static safety: Siek et al. are willing to tolerate runtime errors in code whose typing somewhere employs the type \( \text{any} \); in our system, runtime errors may occur only when any-typed values are eliminated (i.e., in our (\( \lambda \)) rules only). Syntactically localizing errors in this manner does not appear as straightforward with naïve subtyping. Additionally, Siek et al. do not aim to model an adversarial context—there is a single dynamic type, not two (i.e., only any, no un). Exploring how to adapt our type un to defend against adversarial Python contexts would be an interesting line of future work.

4. Metatheory

This section formally establishes memory isolation, static safety, and dynamic safety for \( \text{TS} \) programs translated to JavaScript. Clearly, such a proof requires a formal semantics for JavaScript—we rely on \( \text{JS} \), a translation semantics for JavaScript developed by Swamy et al. (2013), which is in turn based on \( \lambda \text{JS} \) (Guha et al. 2010). We provide a brief review of \( \text{JS} \), define the central invariants of our translation, and describe our main theorem. A full formalism, including all proofs, is available online.

4.1 A review of \( \text{JS} \) and our high-level proof strategy

\( \text{JS} \) is a subset of \( \text{F} \) (Swamy et al. 2011), a programming language whose type system allows expressing functional-correctness properties of higher-order effectful programs. Prior work defines a semantics for JavaScript via translation to \( \text{JS} \). The semantics is implemented by a tool, called \( \text{JS2JS} \), that translates JavaScript concrete syntax to \( \text{JS} \). This semantics for JavaScript has been used previously both as a means of verifying JavaScript source programs (after translation to \( \text{JS} \)) as well as in Fournet et al.’s proof of full abstraction from \( \text{F} \) to JavaScript. At its core, \( \text{JS} \) provides a mechanically verified library called \( \text{JSVerify} \) that tries to faithfully model most security-relevant details of JavaScript, including, for example, its object model and its calling convention. The metatheory of \( \text{TS} \) is stated in terms of its translation to \( \text{JS} \), i.e., programs that can be type-checked against the \( \text{JSVerify} \) API. The validity of our theorem depends on \( \text{JS} \) being a faithful model of JavaScript, an assumption that can be checked separately, e.g., by semantics testing.

To set up the formal machinery, we develop a model of our compiler by transposing the translation judgment in Figure 2 to instead generate \( \text{JS} \) code. The relationship among these translations is depicted alongside. The translation from \( \text{TS} \) to \( \text{JS} \) can be seen as the composition of the translation from \( \text{TS} \) to JavaScript, and then from JavaScript to \( \text{JS} \). Our main theorem is stated as a type-preservation result from \( \text{TS} \) to \( \text{JS} \), where the types in \( \text{JS} \) are precise enough to capture our desired invariants, i.e., static safety, dynamic safety, and memory isolation.

Monadic computation types with heap invariants

All computations in \( \text{JS} \) are typed in a state monad of predicate transformers, \( \text{IDST} \), which is parametrized by a heap-invariant predicate \( \text{HeapInv} \) and a heap-evolution predicate \( \delta \). The type \( \text{IDST} \) \( a \) \( WP \) is the type of a computation, which for any post-condition \( post \), when run in an initial heap \( h \), may diverge or else produce a result \( va \) and a final heap \( h' \) that satisfy the formula \( \text{post} v a h' \wedge \text{HeapInv} h' \wedge \delta h h' \), so long as \( \text{HeapInv} h \wedge WP \) post \( h \) is valid. Additionally, all intermediate heaps in the computation satisfy \( \text{HeapInv} \), and every intermediate heap is related to all its successors by \( \delta \). That is, in \( \text{IDST} \) \( a \) \( WP \), \( a \) is the result type, and \( WP \) is a predicate transformer that computes a pre-condition for the computation with respect to \( post \), any desired post-condition; \( \text{HeapInv} \) is an invariant on a heap, and \( \delta \) is a reflexive and transitive relation constraining how the heap evolves. Our online materials also account for exceptions and fatal errors; however, we gloss over them here for lack of space.

The main idea behind our proof is that a \( \text{TS} \) term \( e t \) is translated to a \( \text{JS} \) term \( e' \) of type \( \text{IDST} \) \( WP \) \( h' \), where

\[
WP \; e' = \text{Apost} \; \lambda h. \; \text{Ok} \; (\text{loc}(e)) \; h \wedge \forall v h'. \; ([v] \; v \; h') \Rightarrow \text{post} \; v \; h'
\]

This ensures that, if \( e' \) is run in an initial heap \( h \) satisfying \( \text{HeapInv} \) \( h \) and \( \text{OK} \) \( (\text{loc}(e)) \) \( h \) (meaning that all free-variables of the source-term \( e \) are correctly promoted to the heap in \( \text{JS} \))
then either (1) it will terminate with a result \( v \) and a final heap \( h' \) satisfying \( ([v] \; v \; h') \) or (2) it will diverge. The code may also raise an exception or throw a fatal error, but all the while during the execution of \( e' \), \( \text{HeapInv} \) will be true, and the heap will evolve according to \( \delta \). This result holds even when \( e' \) is linked with arbitrary JavaScript code—adapting the universal typability lemma of Fournet et al. (2013). JavaScript code can always be typed in \( \text{JS} \) at a type corresponding to \( un \).

Our main task then is to carefully define \( \text{HeapInv}, \; \delta \) and \( [v] \) such that they capture our desired invariants, and then to prove that translated programs are well-typed in \( \text{JS} \).

4.2 Invariants of the translation

To prove memory isolation, \( \text{JS} \) provides a partitioned heap model. Every object reference \( l \text{oc} \) carries a tag, \( l \text{ag} \), which records the name of the compartment into which the reference points, i.e., each compartment is a disjoint fragment of the domain of the heap. There are six compartments in the heap model. The \( \text{Ref} \) compartment holds objects corresponding to \( \text{TS} \)’s records, datatypes, and RTTI; the \( \text{Abs} \) compartment holds function objects; the \( \text{Clean} \) compartment holds the \( \text{Clean} \) object initialized and then frozen by \( \text{boot} \), \( \text{ja} \); and the \( \text{Un} \) compartment belongs to the adversary. We focus primarily on properties of these first four compartments. The remaining two compartments, \( \text{Inv} \) and \( \text{Stab} \), are inherited unchanged from Fournet et al. (2013)—the former is for maintaining local variables, and the latter for tracking function objects used to make safe callbacks to the attacker.

Refined type dynamic

All JavaScript values (including the translation of \( \text{TS} \)) are represented as \( \text{JS} \) values of type \( \text{dyn} \), defined below.

We show only three representative cases. \( \text{d} = \text{str} \) is an injection of \( \text{string} \) into type \( \text{dyn} \), where the refinement \( \text{TypeOf} \) \( \text{d} = \text{str} \) recalls the static type. For object references \( \text{Obj} \) \( l \text{oc} \), the refinement is \( l \text{oc} \)’s tag, i.e., \( \{ \text{Ref}, \text{Abs}, \text{Un}, \text{Clean} \} \). Finally, for functions, \( \text{Fun} \; f \) builds a value of type \( \text{dyn} \) from a function closure \( f \) and the JavaScript object \( o \) for that closure. Its refinement is the predicate transformer of \( f \).
Guarantees that the result is an invariant of the value at runtime.

δ, the heap evolution invariant The full definition of δ has 4 clauses; we show the main one below: δ ensures that, for all objects in the Ref and Abs heaps, their “rtti” fields only evolve monotonically downward, according to the subtyping hierarchy.

A relatively easy lemma derivable from these definitions implies our static safety property. In particular, Lemma 1 guarantees that if a value v (potentially a location to a heap-resident record) is typeable at type t in some initial heap h0 then as the program’s heap evolves according to δ, v remains typeable at t. This ensures that is safe for TS* to ascribe a value a static type, since that type is an invariant of the value at runtime.

Lemma 1 (Static safety: δ preserves the interpretation of types). For all values v, heaps h0 and h1 such that HeapInv h0, HeapInv h1 and δ h0 h1, if for some t we have [Γ] h0 v then [Γ] h1 v h.

Finally, our main theorem, as promised, is a type preservation result that guarantees memory isolation, dynamic safety and static safety for TS* programs translated to JS*. In the hypothesis, the relation Γ ⊢ e : t → e′ is the formal translation of TS* to JS* (the index f is the name of the current function object in e′, a technical detail). The judgment in the conclusion of the theorem asserts that, in a translated environment, e′ has a type that is described by the predicate transformer corresponding to the source type t. As explained in §4.1, this ensures that the translated program respects the heap invariants and, if it terminates normally, produces a t-type result.

Theorem 1 (Type preservation). Given a TS* context Γ, an expression e and a type t, if Γ ⊢ e : t → e′ for some JS* expression e′ and function object f, then Γ, f, dyn, Γloc(e) ⊢ e′ : DST dyn WP[Γ[0]].

We conclude our formal development with a few remarks on the scope of our theorem and the style of its proof. First, our result is applicable to the translation of TS* to JavaScript only inasmuch as JS* is an accurate model of all of JavaScript—Fouquet et al. (2013) argue for how JS* is adequate for all security-relevant features of JavaScript. Regardless, the availability of these semantics together with its program logic is what made our proof feasible. Given an operational semantics, but lacking a program logic, our proof would have been mired in tedious inductions over the operational semantics. With JS*, we were able to carry out our proof as an induction over the compilation relation, and use the type system of JS* to structure and prove our invariants.

Second, our result includes within its trusted computing base the correspondence of boot.js to the CleanSpec predicate in the last clause of HeapInv. While it is perhaps standard for compiler and verification projects to rely on a small amount of trusted code, we
would like to do better. In particular, we aim to use the JavaScript verification toolchain developed by Swamy et al. (2013) to verify boot.js for compliance with CleanSpec—at the time of writing, this was still incomplete. More substantially, we would also like to build a translation validation pipeline for our compiler implementation, reflecting the generated JavaScript back into TS* for verification, i.e., we would like our compiler implementation to be also formally type-preserving.

5. Securely deploying TS* programs

The guarantees of TS* depend on boot.js being the first script to run on a web page. Many prior works on implicitly assumed that scripts are always executed in the order in which they appear on the page (Jim et al. 2007; Magazinius et al. 2010; Taly et al. 2011), but, as we explain, this is a naïve view. Instead, we develop a standards-based mechanism that ensures that our scripts run first.

5.1 Securely bootstrapping the TS* runtime

Suppose a script s is lexically the first element in the header of a page located at a URL $U = \text{http://W.com/page.html}$; one may expect that it will be guaranteed to run first on any window loaded from $U$. However, this intuition is correct only if the page has not been loaded programmatically from JavaScript by another web page, e.g., within another frame. On a page loaded initially from $U$, the script $s$ will indeed run first. Still, a malicious script running later on the page, or on a different page with the same origin \texttt{http://W.com}, may open a window or frame at $U$, and modify all the essential primitives before $s$ begins to run inside the new window frame. This execution order is consistent with the HTML standard (Berjon et al. 2013) and we have confirmed it experimentally on all mainstream browsers.

Hence, any simple first-starter implementation that relies on lexical ordering of script elements will fail if other scripts on the same origin are allowed to open windows or frames. Indeed, the web browser only provides security guarantees at the granularity of an origin (Barth 2011); finer-grained privilege separation between good and bad scripts within the same origin require application-level mechanisms, such as restricting all untrusted scripts to a sub-language like SESlight (Taly et al. 2011); loading them in sandboxed iframes with fewer privileges (Akhawe et al. 2012); or modifying the browser to allow the first-starter script to certify all other scripts running on the page (Jim et al. 2007).

Rather than restrict the functionality of untrusted scripts, we propose a mechanism that ensures that our scripts run first. For a given website, we use two distinct origins:

- \texttt{http://W.com}, used primarily as the service origin; it does not serve any resource.
- \texttt{http://start.W.com}, that serves HTML pages, including scripts compiled from TS*, but where the first two \texttt{script} elements on every page are as follows:

```html
<script src="http://start.W.com/boot.js"></script>
<script document=domain = "W.com"></script>
```

The crucial step here is that, after boot.js has loaded, the page sets \texttt{document.domain} to the parent domain \texttt{W.com}. This is a standards-based mechanism (Berjon et al. 2013, 5.3.1) by which the page voluntarily gives up its rights to the \texttt{http://start.W.com} origin for client-side same-origin access across frames and windows. Instead, it adopts an \texttt{effective script origin} of \texttt{http://W.com}.

All subsequent scripts on the page are unrestricted except that they can only read or write into frames or windows that have an effective script origin of \texttt{http://W.com}, and hence they cannot tamper with pages on \texttt{http://start.W.com}, even if such pages are loaded programmatically into other frames or windows. In all other ways, their functionality is unimpeded, without the need for expensive translations or messaging protocols, as in previous approaches.

More generally, by placing other trusted scripts after boot.js and before the assignment to \texttt{document.domain}, we can run scripts that grab reliable copies of built-in libraries, such as JSON and XMLHttpRequest, for use by subsequent code.

5.2 Loading scripts with embedded secrets

Our first-starter protocol reliably allows boot.js to build a trustworthy environment for our compiled scripts. Conversely, we sometimes need a way for scripts to be able to verify that their environment is trustworthy. This is particularly important when compiled scripts contain secret tokens embedded within them, e.g., to authenticate themselves to other servers. Embedding secrets as constants within program text may seem like an elementary mistake, but this is the predominant way of distributing these tokens in a JavaScript setting. Secrets within scripts must first be protected from malicious websites that may try to load our scripts, and second from malicious scripts on our own website \texttt{W.com}. In this threat model, many simple countermeasures one may think of are inadequate.

Even if we require a login cookie for authentication before serving the script, a malicious website that an innocent user visits when logged into \texttt{W.com} can make a cross-site script request and obtain the script (an attack sometimes called JavaScript hijacking). If we inline the scripts into our page, malicious scripts can read their source code and obtain the token. Even if they are not inlined but served from \texttt{http://start.W.com}, malicious scripts can perform an \texttt{XMLHttpRequest} to obtain their source code and then read them. Indeed, these are all methods commonly used by cross-site scripting attacks (e.g., the Samy worm) to break token-based security protections on the web.

To protect our scripts from same-origin attackers, we use a third distinct origin to serve our scripts:

- \texttt{https://src.W.com}, the secure source server, only serves GET requests for scripts that may embed secret tokens to be shared between the server and the script. The server refuses cross-origin requests and returns scripts as \texttt{text/javascript}, so attackers on \texttt{https://start.W.com} can execute these scripts but not read its source, due to the same-origin policy. We recommend the use of HTTPS for any pages that contain secrets, but HTTP is adequate if we exclude network adversaries from our threat model.

To protect our scripts against other websites, we need an additional check. Every script served from \texttt{src.W.com} is prefixed by a condition on the current webpage location, that is, before making any use of its secret token, the script checks that \texttt{window.location.href} actually begins with \texttt{http://start.W.com}. This ensures that the script has a reliable \texttt{Clean object} on that page, introduced by boot.js.

Experimentally, we found that checking the current location of a script is quite error-prone. Some scripts try to read \texttt{document.domain} (see e.g., OWASP CSRFGuard in §6.1) or \texttt{document.location}; others rely on \texttt{window.location.href} but then use regular expression or string matching to check it against a target origin. All these techniques lead to attacks because a malicious website could have tampered with its \texttt{document} object or with the regular expression libraries. We found and reported such attacks to vendors.

Notably, many browsers allow properties like \texttt{document.domain} and \texttt{window.location.origin} to be overwritten. Our origin check relies on the \texttt{window.location.href} object which is specified as unforgeable in the HTML specification (Berjon et al. 2013, 5.2). In practice, however, we found that some browsers incorrectly allow even supposedly unforgeable objects like \texttt{window.document} and \texttt{window.location} to be shadowed. We have reported these bugs to various browsers and are in discussions about fixes. If the unforgeability of \texttt{window.location.href} turns out to be too strong an assumption, we advocate the use of the origin authentication protocol of Bhargavan et al. (2013).
6. Secure web programming with TS*

We have evaluated our compiler by gradually migrating JavaScript sources to TS*, while ensuring that the migrated code (after compilation) exports the same API as the original. We have also written from scratch various utilities like a JSON parser and a reference monitor for HTML5 localStorage. All the code is available online.

We have yet to conduct a thorough performance analysis of our compiler, and to implement further optimizations. But, as mentioned previously, statically typed TS* should incur little, if any, runtime overhead, while un-typed code is unchanged. Understanding and optimizing the performance profile of any-typed code is left as future work.

In this section, we describe how TS* can be used to gradually secure existing access control patterns, as deployed in popular JavaScript APIs. We focus on client-sided, language-based security, relegating most other details to the online materials (notably a description of same-origin policies, protocol- and browser-specific security assumptions, and current attacks against them; see also e.g., Bhargavan et al. 2013).

6.1 OWASP CSRFGuard  We first consider the task of securing client code that performs an XMLHttpRequest to the page’s server.

**Cross-Site Request Forgeries (CSRF)** Suppose a website `W` has an API available from JavaScript. Authenticating and authorizing access using cookies ensures that only requests from a logged-in user’s browser are accepted. Conversely, if the user has any another websites open in her browser, their scripts also get access to the API, and can thus steal or tamper with the user’s data on `W`. Such request forgery attacks are persistently listed in OWASP Top 10 vulnerabilities, and have a serious impact on a variety of websites.

**CSRF Tokens** As a countermeasure to CSRF, a website `W` can inject a fresh, session-specific, random token into every page it serves, and only accept requests that include this token. Other websites cannot see `W`’s pages (thanks to the same origin policy) hence cannot forge requests. Additionally, cookies and tokens can be protected while in transit by using the HTTPS protocol. OWASP CSRFGuard 3 is the most widely-referenced CSRF protection library. As an advanced feature, it provides a token- injection script that transparently protects AJAX calls by intercepting any XMLHttpRequest. (Similar protections exist for frameworks like Django and Ruby-On-Rails.) Crucially, the token must be kept secret from other websites, and also from other scripts loaded on the rest of the library) can tamper with the execution.

**CSRFGuard in TS** Following the approach of §2, we migrate to TS* the OWASP proxy that uses the token to provide authentication RPCs to the rest of the library. This small script is typed in TS*, guaranteeing that no malicious script that runs alongside (including the rest of the libraries) can tamper with its execution. The TS* proxy, listed below, takes three string arguments: the target URL, the API function name, and the JSON-formatted arguments. It checks that the URL is well-formed and belongs to the current site (to avoid leaking the token to any other site), then it serializes the request as a query string, attaches the token, and makes an AJAX call. Once wrapped, it exports the same interface as before to any (untrusted) scripts loaded on the page. Additionally, it could be directly used by further TS* code.

```javascript
var csrfToken: string = "GENERATED_TOKEN"
var targetOrigin: string = "TARGET_ORIGIN"
function Rpc(url:string,apiFun: string, args: string): any {
  if (String.startsWith(url,String.concat(targetOrigin,"/")) && String.noQueryHash(url)) {
    var m = (method:apiFun, args: args, token: csrfToken);
    var request = String.concat("?",QueryParams.encode(m));
    var response = xhrGet(String.concat(url,request));
    return QueryParams.decode(response);
  }
  else return "unauthorized URL";
}
```

The first two lines define string literals, inlined by the server as it generates the script—the TS* compilation process ensures, via lexical scoping, that these two strings are private to this script. The `Rpc` function is our secure replacement for xhrGet, which performs the actual XMLHttpRequest. Compared with the original JavaScript, it includes a few type annotations, and uses either safe copies of built-in libraries, such as xhrGet and String, or typed TS* libraries, such as QueryParams (outlined below). Relying on memory isolation and secure loading from TS*, a simple (informal) security review of this script lets us conclude that it does not leak csrfToken.

Experimentally, we modified the OWASP library, to isolate and protect the few scripts that directly use the token (such as the proxy above) from the rest of the code, which deals with complex formatting and browser-specific extensions, and is kept unchanged and untrusted. The modified library retains its original interface and functionality, with stronger security guarantees, based on strict, type-based isolation of the token. Its code and sample client- and server-side code are available online. To our knowledge, it is the first CSRFGuard library that provides protection from untrusted scripts.

6.2 Facebook API Taking advantage of Cross-Origin Resource Sharing (CORS), Facebook provides a client-side JavaScript API so that trusted websites may access their personal data—one the user has opted in. Interestingly, Facebook also provides a “debug-mode” library, with systematic dynamic typechecks somewhat similar to those automated by TS*, to help programmers catch client-side errors. We focus on two aspects of their large API: the encoding of strings, and cross-domain messaging.

**QueryString encoding** We give a TS* implementation of the QueryString module (mentioned above) for the REST message format used in the Facebook API.

```javascript
function decode (s:string) : any {
    var res = {};
    if (s === "") return res;
    else {
        var params = String.split(s, ";");
        for (var k in params) {
            var kv = String.split(params[k], "=");
            res[kv["0"]] = kv["1"];
        }
        return res;
    }
}
```

(The encode function is dual.) Our function illustrates support for arrays provided by our compiler. Importantly, this code may be used to parse untrusted messages; our wrapper for `un` to `string` is straightforward—if the argument is already a string, it is just the identity. Hence, one can write efficient TS* that calls `decode` to parse messages received from the adversary; this coding style is idiomatic in JavaScript, while the checks performed by our type system and runtime prevent many pitfalls.

Another TS* sample illustrates the usage of Rpc and our typed JSON library (generalizing QueryString) to program a higher-level, statically typed API. It shows, for instance, how to program a client-side proxy for the “/me” method of the Facebook API, which retrieves the user profile; this TS* function has the return type:

```javascript
function profile = {
    id: string; email: string; age_range: {min: number}; ...
}
```
Cross-Domain Messaging  The Facebook API is meant to run on any website and protects itself from a malicious host by using iframes. For example, if the website calls FB.login, the API loads an iframe from facebook.com that retrieves the current user’s access token and then only sends it to the host website (via postMessage) if the host is in a list of authorized origins.

Bhargavan et al. (2013) report attacks on a prior version of this authorization code that were due to typing errors (and have now been patched). We re-implement this code in TS* and show how programmers can rely on typing to avoid such attacks.

The checkOrigins function below is given the current host origin and verifies it against an array of authorized origins. The proxyMessage function uses this check to guard the release of the token to the parent (host) website, using a safe copy of the primitive postMessage function.

```javascript
function checkOrigins(given: string, expected: array string): bool{
  for (var k in expected) {
    if(given === expected[k]) return true;
  }
  return false;
}
function proxyMessage(h: string, tok, origins: array string) {  
  if(checkOrigins(h, origins)) postMessage('parent', tok, h);  
}
```

In a previous version of the Facebook API, proxyMessage was accidentally called with an origins parameter of type string, rather than array string. This innocuous type error leads to an attack, because the untyped version of the code succeeds with both strings and arrays, but with different results. To see the core problem, consider a call to checkOrigins where given = ‘h’ and expected = "http://W.com". The for loop iterates over each character of expected, and hence succeeds, when it should not. In TS*, iteration is well-typed only for arrays and, unlike JavaScript, this enumerates only the “own” properties of the array. Thus, in our code, this error is caught statically (if the incorrect call to proxyMessage is local) or dynamically (if the call is from another iframe); the check fails in both cases and the token is not leaked.

7. Conclusions and prospects

This paper aims to broaden the scope of gradual typing: not only it is useful for migrating dynamically type-safe code to more structured statically typed code, it is also useful for moving from unsafe code, vulnerable to security attacks, to a robust mixture of dynamically and statically type-safe code.

Within the context of JavaScript, we have presented TS*, a language with a gradual type system, a compiler, and runtime support that provides several useful safety and confinement properties. Our preliminary experience suggests that TS* is effective in protecting security-critical scripts from attacks—without safety and confinement, such properties are difficult to obtain for JavaScript, and indeed security for such scripts has previously been thought unobtainable in the presence of cross-site scripts.

Even excluding the adversary, TS* develops a new point in the design space of gradual typing, using an approach based on runtime type information. This has several useful characteristics, including a simple and uniform failure semantics, and its applicability to a language with extensible objects and object identity.

In the future, we plan to develop TS* along several dimensions. On the practical side, we expect to integrate our ideas in an experimental branch of the open source TypeScript compiler, targeting the construction of larger secure libraries. On the theoretical side, we plan to explore the formal certification of our compiler and runtime. We also hope to develop our preliminary ideas on new notions of blame to explain runtime failures in TS*.

Acknowledgments  Many thanks to Martin Abadi, Antoine Delignat-Lavaud, Jeremy Siek, Phil Wadler, and all the anonymous reviewers.

References


A. Barth. The web origin concept, 2011. IETF RFC6454.


OWASP CSRFGuard. CSRFGuard 3 user manual, 2010.