Type-Based Static Analysis for Access Control in TypeScript

Alex Asch, Eric Huang, Nick Petrone
University of California, San Diego
San Diego, CA, USA
a.asch2020@gmail.com
eth003@ucsd.edu

***Abstract*—Static Application Security Testing (SAST) tools are used for defect detection and analysis as a catch all tool for finding security bugs in production code. With these tools used as compliance tools as well, it is exceedingly crucial that these tools provide usable insight to a wide range of developer users. However, many tools in reality have a very high false positive rate, leading to a poor signal to noise ratio that prevents any developer insights from being gained from their use. Many of these tools run into these usability issues since they need to be catch all, relying on pattern matching tools, and heavy slower tools like control flow graph construction. Additionally, rules engines and rules based exceptions apply at globally at codebase level, making it harder to gain real insights or tuning into specific points in the code. To address the above concerns, we create the prototype system TypeSAST, by presenting a JSdoc based UI for development, we make it easy to tie specific code to rule enforcement. We then use type system level enforcement, by transpiling the code to a separate typescript project, in which the jsdoc comments are encoded in as types. We then evaluate the usability and limitations of this system, then introduce our symbolic execution pass leveraging the ExpoSE symbolic execution engine. By doing this, we are demonstrate a proof of concept for tuneable comment level SAST for specific defect classes, encoded in the language's type system, with a symbolic secondary pass.**

***Index Terms*—static analysis, security, TypeScript, taint tracking, developer tools,**

I. Introduction

SAST tools are software tools that analyze source code for security vulunerabilities without actually executing the code. Because application code does not need to be executed for analysis, SAST tools are in theory a great tool for CI/CD pipelines by catching bugs before automated testing or release. Indeed, they are commonly integrated into these pipelines and IDEs to catch bugs [1]. These tools can be simple as a regex match banning the use of `eval` in javascript. Although simple to implement, these simple SAST tools have a poor understanding of program semantics and cannot flag more complex bugs. More sophisticated tools may parse source code in ASTs to better understand program semantics, such as to differentiate a string `"do not use eval()"` vs an actual call to `eval()`. Although they can reason better about programs, they are still relatively local and cannot easily reason about how taints move through an entire program. Even more advanced tools use symbolic execution to reason about program call graphs, which allow for analysis on a larger class of bugs including those that including conditionals. Despite the appealing premise of SAST tools, most software engineers do not consider them as a priority when developing or maintaining code [1].

A major problem with existing tools is the high level that rules are applied. Rules lists and exception lists are applied globally across an entire codebase, which make it difficult to tune SAST analysis to specific parts of the codebase or encode developer intent at the point where they require it. When a developer writes a function that requires elevated access, that constraint is far from the code, making a bad user experience. Another problem are that tools are either too local, or too global, which cause either limited uncovered bugs or long analysis times.

We address both problems with TypeSAST, a SAST system targeting access control violations in TypeScript. Rather than applying rules globally, we allow rules to be put into code via JSDoc annotations, giving developers a way to express access requirements at the function and block level. These annotations are encoded into the TypeScript type system with source transformation, which allows the standard TypeScript type checker determine violations as type errors. Due to this, the static pass is very fast. Where the static pass finds false positives (will likely occur around complex control flows), we run a secondary symbolic execution pass using the ExpoSE engine at flagged sites with callsite backtracking. This keeps amount of code symbolically executed small, comapred to going through the entire codebase from the entrypoint. Thus, we get a tuneable, function level SAST pipeline with a symbolic execution layer without heavy program analysis tools.

II. Related Work

Several studies have been performed on the reception and effectiveness of static analysis tools in a professional software engineering environment. Johnson investigates why static analysis tools are underused by professional software engineers [1]. In their study, they interview a group of 20 software engineers and graduate students on their usage of static analysis tools. Among the subjects in the experiement, most understand how and why static analysis tools are useful because they can speed up development work and find bugs faster than by hand. However, those who have used these tools report several issues that make it harder to adopt these tools. One issue that people report include the large number

of false positive reports generated by static analysis tools. In a large software project, developers can be shown thousands of errors [1]. In addition, these tools typically have a clumsy user interface which make sifting through these errors much more difficult. When reading reports, 19 out of 20 subjects also report that it is difficult to interpret the meaning of errors. Error messages may be opaque which make it difficult to understand the cause and implement a fix. Finally, some users report that tools are poorly integrated in their development environment, forcing them to go out of their way to use, which leads to lack of adoption. We will focus on these drawbacks: false positive rate and user experience, when designing TypeSAST.

Despite this reports of static analysis being underutilized, there are some hugely successful static analysis tools that have changed the software industry [2]. Coverity was a software company (aquired) that developed properietary static analysis tools for industry use. Originally started as a research project at Stanford, it soon began to see real usage in the industry. [2] discusses how academics transitioned their research tool into a monetized product. They describe their experiences and "laws of bug finding," which are lessons they learned about static analysis tools used in industry vs in academia. One interesting problem they mention is that they are not able to access their customer's source code. Because of this, sales engineers must keep in their head customer issues, making it difficult to debug issues. Another factor important for adoption is false positive rate. While not as important in a research setting, the false positive rate in a commericial tool is one of the most important factors customers are influenced by. The engineers discovered that customers are very quick to distrust a tool after encountering false positives and unhelpful errors from the beginning. Therefore, a tool with a high false positive rate causes customers to distrust the tools, making adoption harder. Another variable worth mentioning is "churn." This means that a future version of a tool should never miss bugs an older version of the tool reports. Although less important for our project, an industry tool should never surprise a paying customer in its outputs.

It is clear from the above experiences that the most important factor when designing a useful static analysis tool is to reduce useage friction as small as possible. Whether this be reducing false positive rate, increasing error message clarity, or other factors, there is no use of having a static analysis tool if no one is willing to use it.

III. The Defect Class

For our tool, we chose to focus on one defect class in one language, to adhere to our prior notion of a more focused tool. We choose to focus on access control at a function/block level. We chose the type of defect, because it is not recoginzeable by only low scope local pattern matching, as access permissiblity must be checked at each callsite, across the code, instead of a local grep like pattern. Additionally, this class of defect is user defined and tuneable, unlike standard taint tracking/source sink analysis, we have user input and code level control encoded within the defect class itself. This makes this type of security error well suited for the type of system and usage we want to prototype, since it is fundamentally grounded in a user configuration, and cannot be easily detected by simple pattern matching. Finally, this type of error class is well suited to test the expressiblity of the type based analysis, since it implements more user expressability than simple flagged sources or sinks.

IV. Implementation

A. The JsDoc language

The ui for the v0 and v1 implementation of our tool are both entirely JsDoc based. We choose this method as one of the premises of this work is tying SAST scanning rules closer to code and configuration behavior, over global matching and rules. In the table 1 are the types of JsDoc comments we support, what they transform to, and any additional propagation information. Important to note for the v0 rules as established in table 1, are that role hierarchies are not lattice based. There can be one or more linear chains of role escalation, but since multiple inheritance is not natively supported by typescript in a way that can be used as part of the type checking pass, v0 only supports linear role chains.

An important note about the below JsDoc language is on it's limitations. While it is good at handling direction function calls and enforcing access routes in that way, functions being handled as first-class in typescript makes it so there are many ways that a function call can be passed that cannot be caught by this access postpending method. We currently support simple single block alias tracking for functions, alongside finding callsites and transforming the definitions. For example, the first flow can be handled by our static pass, but the second callback flow cannot.

```
/** @requiresRole l2 */
foo(roleContext: l2) {};

{
  let x = foo;
  x()
}
```

```
/** @requiresRole l2 */
foo(roleContext: l2) {};

{
  let x = foo;
  runner(x);
}

function runner(fn: (s: string) => void) {
    fn();
}
```

In addition, the becomesRole and raised rules require some more enforcement and source transforms as compared to requires and roles. Since both are intended to change the label in a scoped fashion, we apply these passes after the first raised

JSDoc	Transform	Description
@roles l1 < l2 < l3	`export class l2 extends l1 { cl2(): void {} }`	Defines roles through creation of classes per role. Empty functions to stop duck typing
@requiresRole l1	`requirerer(roleContext: l1)`	Role required to execute
@becomesRole l1	`raise(roleContext) const roleContextBecome_1: l1 = new l1();`	Create a new raised label, and propagate to callsites in the block
@raised l1	`const roleContextBecome_1: l1 = new l1();`	Elevate the role, propagate new role to callsites block

TABLE I
JSDOC ROLE SEMANTICS

pass, so that we can have roleContext parameters in functions before we do block level replacement with raised.

B. The Despot Language

We also created a DSL for configuring the roles used in TypeSast using the "Blazing Fast" Chevrotain framework. Besides the fact that creating a DSL (even a small one like Despot) is fun, the main gain of using a DSL over our old JsDoc only semantic is for the encoding of DAG of type access flow over the simple linear hierarchy of roles. Parsing, lexing, and interpreting this language guarantees that config files are well-formed. This langauge is only used in the v2 driver code, but it is compatible with v1 single-hierarchy relationships between roles, and is linked with a large code refactor from v1. Thus, some of the edge case handling is not fully checked for it, and it does not have a complementing symex driver generator.

The structure of the Despot language mirrors that of the existing JsDoc language, but it defines semantics for creating disjoint hierarchies between different roles. Roles are created using the `@role [roleA]` rule. By default, unless a hierarchy is defined between different roles, they are treated as disjoint and code that tries to flow between them will generate a TypeError unless the function uses the JsDoc `@becomesRole` annotation in the TypeScript source to change the role the function will assume. The default role can be named using the Despot `@defaultRole [roleB]` rule. By default, every role subsumes the default role which is assumed to have the least privilege. When we transform the TypeScript source, every unannotated function is typed to require this default role. Hierarchical relationships can be defined between roles using the `@hierarchy [roleA] > ... > [roleN]` rule. Interpreting this config file guarantees that the roles are well defined and that there are no loops, which, if present, could indicate confused responsibilities or a bug in the defined hierarchies, both of which we would like to prevent. The validated interpreted config file is then passed on to the ts-morph transformation pipeline and used to inform the roles available for our JsDoc comment language.

V. THE V0 TRANSFORMATION PIPELINE

For the static ts-morph tranformation pipeline, we utilize a multi-pass pipeline that generates an instrumented copy of the original typescript code, with added types being utilized to generate initial error diagnostics for the symbolic execution pass.

The first pass is a simply find replace based pass in order to sidestep some limitations of tsmorph with relation to JsDoc. Since JsDocs can only be attached to certain language constructs, and the `@raised` annotation is not neccessarily attatched to those, we append a `void 0;` after each raised tag to allow the parser to function.

Following that, we initialize the role hierarchy. To begin, we first run a parameter injection pass on every function within our codebase and its call types. For each function, we inject the roleContext parameter with its appropriate `@requireRole`, and for those without annotation, they get typed with the lowest role possible (and with = new l0) in their call signature, so existing callsites to them remain intact.

From each function we then do parameter injection to all call sites. Since functions are first-class in TypeScript it is not possible to annotate 100% of all call sites in this pass, so we handle four cases: direct calls, method calls, single-block aliased calls, and callback pass sites (e.g. `arr.map(fn)`). For the callback case, the function reference is rewritten into a wrapper that captures the ambient `roleContext` at the pass site, making the role requirement visible to the type checker without modifying the host function's signature.

Any well typed case not covered by these four interprocedural argument passing, object literal storage, destructuring, and return values will surface a typeerror at call time due to parameter mismatch. Of course, having the callback receiver receive an any type of plain function type will fail silently.

The next pass propagates the becomesrole tags. This pass collects all callsites of functions with raised, as found in the above, and ensures that a manual level raise is inserted after it, following that all subsequent functions with roleContext within it's parameters has the variable replaced by the newly created raised context

Finally, we propagate the manual raised pass. This is similar to the prior becomesrole propagation, but instead of having

JSDoc	Transform	Description

@roles l1 < l2 < l3	`class AnalysisState`
	`raise(role) {`
	`this.raisedRole = role;`
	`this.currentRole = role;`
	`}`
	`requirel1() {`
	`if (!["l1", "l2"].includes(this.currentRole)) {`
	`throw new Error();`
	`}`
	`}`
		Creates global state object for role tracking

@requiresRole l1	`state.requirel1();`	Assert or throw the current role label
@becomesRole l1	`raise(roleContext)` `state.raise("l1");`	Raise the running state to the named role at function level
@raised l1	`state.raise("l1");`	Raise the running state to the named role at comment sites

TABLE II
JSDOC ROLES FOR SYMBOLIC EXECUTION

to track function callsites, it only needs to track the level. Following that call, we again insert the new roleContext in following statements in the block.

To gain diagnostic insights from our tranform, we simply get the pre emit diagnostics from the transformed, code to gain a list of erroneous call sites.

1) ts-morph: We utilize ts-morph as our manipulation engine over a regex or grep based approach, because the AST awareness it gives exposes the type checker and symbol resolution, allowing us to use a easily usable API to perform code aware and reference aware rewrites.

2) v0 limitations: From our v0 pass, we found two notable limitations. One being the limitation in complex alias tracking for first-class functions, the second in a "high - low - high" pattern. The first static pass handles direct calls, method calls, and single block aliasing or callback passes. However, multi block tracking is not supported, and fails at type check, and most critically, if the receiver is any typed for multi block function passing, the checker fails silently.

A model of the high - low- high problem is as below. With our origininal static pass, we are unable to account for such flows, as the name variable effectively encodes the access level as called in bar, and since it is set from a high roled context and thus can only be reached that way is safe, but static function level tranforms cannot addres that control flow, without a manual raised within the if block.

However, since both cases will fail at pre-emit (other than if any types are used), we find that this pass is false positive prone, but in our testing did not render any false negatives at pre-emit. This makes it very well formed for being created as target locations for a symbolic execution verifier pass.

```
/** @requiresRole l2 */
baz(roleContext: l2) {
};
/** @requiresRole l1 */
bar(roleContext: l1) {
  if(name == "l2"){
    baz()
  }
  return

};
/** @requiresRole l2 */
foo(roleContext: l2) {
  name = "l2"
  bar(name);

};
```

VI. V1 SYMBOLIC EXECUTION PASS AND DRIVER GENERATION

Tackling the limitations above (but introducing some of their own) is the symbolic execution pass. This pass also features a transformation of the code, this time into compiled JS for the ExpoSE engine, along with smart back tracking to try to reduce potential path explosion problems. Since we are now executing through the code, we do not rely in parameter injection for the type checker, and instead create a dynamic state object, that will track the state of the current call, and throw an error on a mismatched access. In table 2 we can see how the annotations are transformed for the symbolic state check. Compared to the static check we don't need to propagate values across blocks, and instead can leave all the state handling to the internal state variable, and control flow to the actual running flow of the symbolic execution.

This allows us to address the prior high- low- high flow, since of course, any level changes are now encoded in the state and will cleanly remain in the higher state.

In order to run our specific symbolic execution, we also create driver code for the expoSE engine. After the v0 pass is complete, we fetch all functions which we want to make a driver for. To do this, we utilize localized backtracking. At each error site brought from the preemit type diagnostics of

the ts type checker, we then backtrack the callsites from this function, until we find a function that has the requisite role type for the function causing the mismatched error. By doing this, we can symbolically execute only paths where there is a true chain of low callers to higher callees, since we discard any paths that hit high in the backtrack. Additionally, this backtracking pass can also serve to diagnose potential false positives positives even before symbolic execution. If a higher typed function always sees a function of it's level or greater in it's backtrack, we then know that it is clearly a false positive error site. Even with the above optimizations, we are not able to verify symbolic execution on much other than small lightweight paths due to it's overall speed.

Additionally, we use the type awareness of parameters afforded by typescript to break object types down to their primitive typed parts so that ExpoSE js can support them well. This is of course the same for parameters with primitive values. By seeding the symex engine only with valid typed values, we reduce path wastage. In the below example, token and body are both string types. However, this instrumentation runs into restrictions with functions passed as typed parameters. However, this does not work with regard to functions passed as parameters.

```
function verify() {
  var S$ = require("S$");
  var { state } = require("");
  var { handleAdminRequest } = require();
  var out;
  var token = S$.symbol("token", "");
  var body = S$.symbol("body", "");
  out = handleAdminRequest(token, body);
}
verify();
```

VII. Future/in-progress work

A. Vscode extension

We began work on the Vscode extension midway between the rewrite from v1 to v2, leaving it in the state that it doesn't work for our v1 version, and it requires more development time to work for v2. Problems that arose with getting the extension working for v2 were that we needed to do source mapping on top of the existing code transformation, and while this is possible using magic-string, getting it working in a way that isn't janky is difficult. This is because magic-string is not syntax-aware, and when we make code transformations with ts-morph, ts-morph does not build in a way to encode the AST mutation into a source map. Two routes forward to getting this running as an extension would be to either use a diff tool similar to Git's to get approximate source file offsets to map to, or generate string representations that are as close as possible to the code transformations made by ts-morph to replicate with magic-string. The benefit of the first way is that since we're almost exclusively adding code, it makes it easier to compress our security type additions back to the original lines we transformed from. A drawback would be that many close annotations could cause the diffing algorithm to lose track in either the source or transformed file and cause incorrect line numbers to be reported. The same drawback exists for the magic-string approach, since we would be making string transformations that, while informed by the AST, will not match the ts-morph transformation perfectly, and could also desync lines.

VIII. V1 Limitations

Within the current implemetantion of the tool we have serveral notable limitaions both in the v0 pass and static pass. While some are "by design" like any types escaping any type level enforcement, others are remediable and are implementation limitations.

The main such limitation is callbacks and functions as parameters. While we have some small scope handling for this, when passed across functions, the static transformation layer is unable to then propagate through to the callees.

Additionally, our symbolic execution driver generator walks objects to create the symbolic driver for it, however, this path includes things inherited from the prototype, and thus generates non-instrumentable driver code. The two above issues mean that callbacks and functions as parameters fail both the v0 and v1 static and symbolic tools. In addition to this, the execution pass is highly brittle when applied to anything past very lightweight instrumented code. We were consistently able to generate "good" typescript that we could write drivers for, that would fail in the symbolic execution pass, separate from the analyser state.

IX. Results

Our evaluation demonstrates the key limitation of the static pass, demonstrating that control flow sensitivity is difficult to impelment in a purely types based approach, and is cumbersome, or in some cases unfeasible in addressing dynamic priviledge changes. We then proceed to demonstrate that symbolic execution is a feasible strategy for addressing errors raised from this pass, by implementing the first in a way that is aligned to throw false positives over passing quietly.

We evaluate our project on a set of known vulnerable, and known safe code to demonstrate the feasiblity of the above methods.

First, we demonstrate the efficacy of the transform language, showing annotated examples of our evaluation codebase, with roles of unauth, user, and admin. (Some arguments and format has been omitted for brevity, but this is all real transformed code from our tool)

A. v0 Transformed Code

1) Catching access error:

```
  /** @requiresRole unauth */
function handleRequest(...) {
    let x = handleAdminRequest;
    x("xd", "yz");
}
```

```
function
handleRequest(..., roleContext: unauth) {
    let x = handleAdminRequest;
    x("xd", "yz", roleContext); // TypeError
}
```

2) Catching access error with conditional raise:

```
/** @requiresRole unauth */
function handleRequest(...) {
  if (token.startsWith("user-")) {
    /** @raised user */

    handleAdminRequest(token, body);
    return handleUserRequest(token, body);
  }
}


function handleRequest(...,
roleContext: unauth) {
  if (token.startsWith("user-")) {
    const roleContextRaised_0: user = new user();

    handleAdminRequest(token, body,
      roleContextRaised_0);  // TypeError
    return handleUserRequest(token, body,
      roleContextRaised_0);
  }
}
```

3) Tracking role with mapping:

```
function deleteUser(...) {
}
function processItems(...) {
    const deleted = items.map(deleteUser);
}


function deleteUser(..., roleContext: admin)
{
}
function processItems(.., roleContext: user)
{
    const deleted =
      items.map((...) =>
      deleteUser(..., roleContext)); // TypeError
}
```

4) BecomeRole propagation:

```
function handle3Request(..) {
    ath()
    handleRequest(...)
    let x = handleAdminRequest;
    x(...)
}
/** @requiresRole user @becomesRole admin */
function ath() {
    return true
}
```

```
function handle3Request(..., roleContext: user) {
  ath(roleContext)
  const roleContextBecome_1: admin = new admin();
  handleRequest(..., roleContextBecome_1)
  let x = handleAdminRequest;
  x(..., roleContextBecome_1) // No TypeError
}
```

The above cases we can see are cleanly handled by our static pass, catching aliasing, callback rewriting, and block level role propagation. However, the next cases are false positives not handled cleanly by the v0 pass, requiring symbolic execution.

B. Alias mapping gap

```
/** @requiresRole admin */
function rwc(items: string[],
  cb: (s: string) => void) {
    items.forEach(cb);
}

/** @requiresRole admin */
function tag(items: string[]) {
    const handlers = { delete:
      deleteUser };
    runWithCallback(items, handlers.delete);
}

export function rwc(...,
  roleContext: admin) {
    items.forEach(cb);
}

export function tag(...,
  roleContext: admin) {
    const handlers =
      { delete: deleteUser };
    rwc(items, handlers.delete,
      roleContext);
}
```

C. High, low, High

```
/** @requiresRole admin */
export function hicall() {
    locall("hicalled")
}

export function locall(item: string) {
    if( item == "hicalled"){
        deleteUser("uname")
    }
}

function truesymcall() {
  locall("loww");
}

export function hicall(roleContext: admin) {
```

```
    locall("hicalled")
}

export function locall(...,
  roleContext: unauth = new unauth) {
    if( item == "hicalled"){
        deleteUser("uname", roleContext)
    }
}

export function tsm(
  roleContext: unauth = new unauth){
    void 0;
    locall("loww")

}
```

The above three examples show errors in the static pass. The first is a gap in alias mapping, and the second is the high, low, high call where the parameter should propagate the access, but that cannot be guaranteed statically. We can see that in the second call, the hicall route can be validated by simple backtracking, leaving only the tsm function to be used for creation of symbolic driver.

Examining the log generated from a run on the tool enabled code, we can see that the preemit diagnostic and backtracing pass can correctly find failure sites. From the below trimmed output, we can clearly see that for our v0 flagged cases, the backtrace logic in conjunction with handling for parameter mismatches returns a list of hard failures, and properly flags potential false positives for the symex driver.

```
[symex] 'handleRequest' needs 'admin'
| trace: [handleAdminRequest]
in 'symextests.ts' [cite: 187]
[hard fail] 'processItems' needs 'admin'
but no entry point found
| 'symextests.ts': [cite: 201, 207]
[hard fail] 'handle3Request' needs 'admin'
but no entry point found
| 'symextests.ts':  [cite: 208, 220]
[symex] 'triggerAliasGap' callback mismatch,
 no trace found | using 'triggerAliasGap'
[symex]
'locall' needs 'admin'
| unprotected root: 'tsm'
```

Finally, we demonstrate that the symbolic execution pass leveraging ExpoSE can validate the above callsites using our instrumented state tracking with limitations. The main limitation is that our parameter reconstruction pass does not handle passed functions well, thus, when creating the symbolic driver for the code, we end up with a large trace of unscopable error for the parameter matching issue for callbacks. This is a current limitation of both the static pass and the symbolic pass. However, we see that running the symbolic execution engine on the below driver passes in ExpoSE without error.

```
function verify() {
  /* Generated symbolic driver for truesymcall */
  var S$ = require("S$");
  var { state } = require("./state");
  var { truesymcall } = require("./symextests");
  var out;
  state.currentRole = "unauth";
  out = truesymcall();
  console.log("Symbolic output:", out);
}
verify();
```

X. Conclusion

From our above implementation of TypeSAST, bridging static types encoding roles to symbolic execution, we are able to demonstrate that certain classes of errors can be encoded within the type system with minimal transforms, by implementing validation of access control within the typescript type system. We addionally were able to demonstrate shortcomings and a failure case of the above system, which we were able to resolve using a targeted symbolic execution pass, demonstrating refinement of false positives in our existing static methodology using a lighter weight method than full code symbolic execution.

References

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *International Conference on Software Engineering*, 2013, pp. 672–681. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486877

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, p. 66–75, Feb. 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374