Checking Cryptographic API Specifications in JavaScript

Duncan Mitchell

Submitted in fulfillment for the degree of
Doctor of Philosophy

Department of Computer Science
Royal Holloway, University of London

Declaration of Authorship

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Computer Science as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Duncan Mitchell
September 2019

Acknowledgments

I would like to thank my supervisor and academic mentor Johannes Kinder for his advice and guidance throughout this journey. His faith in me and drive for me to succeed have been an invaluable source of encouragement.

I would like to thank all my lab mates for creating an enjoyable and congenial place to undertake this PhD. In particular, to Blake Loring for our fruitful collaborations and his work on ExpoSE, and to Thomas van Binsbergen for being so welcoming and generous with his time. I would also like to thank Claudio Rizzo and James Patrick-Evans for proof-reading and technical discussions, but most importantly good company.

To Dave Cohen, Carlos Matos, Jasper Lyons and Matthew Hague: thank you for the opportunity to enrich my research life through teaching. I would like to thank Calum and Dan for taking a close interest in my work and their enthusiasm for my future.

To my parents, for fostering my love of science, technology and mathematics, for their unfailing support and for every opportunity that bought me to this point. Finally, to Holly, my rock: without you, I would not be writing this.

Abstract

Increased awareness of privacy concerns on the Internet has encouraged developers towards implementing strong cryptography by default, in a trend dubbed "ubiquitous encryption". For instance, web applications for messaging platforms routinely implement client-side cryptography in JavaScript for true end-to-end encryption. The standardization of cryptographic APIs in JavaScript through the W3C Web Cryptography API, *WebCrypto*, has made strong cryptography available to non-expert web developers. However, cryptographic APIs are often hard to use correctly; the clash between the agile mindset of JavaScript developers and the requirements of secure software engineering leads to lingering problems. Further, JavaScript's dynamic types and often surprising semantics make it difficult to spot subtle security bugs, and such errors do not lead to failing test cases or visible errors.

In this thesis, we propose an automatic mechanism for the detection of incorrect usage of cryptographic APIs in JavaScript. We introduce a system of Security Annotations: type-like tags which express security properties of values, e.g., whether a value is a ciphertext, or a cryptographically secure random value. Security Annotations are transparent to client code until they encounter an error, in which case the program under test fails. We formalize the notion of Security Annotations in a small lambda calculus, and use this to motivate the design of Security Annotations within an executable formal semantics for JavaScript. We construct a specification of

the WebCrypto API and prove security guarantees in this setting. Finally, we implement Security Annotations within full JavaScript via source code instrumentation and use this to analyze both hand-crafted examples and real-world JavaScript applications.

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

1.1 Context

1.1.1 Cryptography

In the modern world, cryptography is ubiquitous. Even unawares, almost every individual encounters it every day—from instant messaging to baby monitors or nuclear power plant control systems. Cryptography itself encompasses a diverse range of use cases: despite its root as simply *"secret writing"* (from the ancient Greek), cryptography has grown to encompass all elements of secure communication, and is even used further afield, e.g., in database integrity.

Within this thesis, we consider a cryptographic mechanism to be any mechanism which aims to ensure that two (or more) parties can communicate information without an adversary gaining knowledge about (or successfully manipulating) the information itself. These mechanisms comprise primitives, which describe low-level algorithms (e.g., the AES symmetric key encryption function [74]), and protocols, which use these primitives as building blocks to describe how to enable secure communication within particular settings (e.g., TLS [84]).

In the most abstract setting, cryptography protects messages Alice wishes to send to Bob, without an adversary, Eve, gaining information about (or

tampering with) these messages. Cryptographic mechanisms for *authentication* seek to ensure that Bob can verify any message received is indeed from Alice, and not from Eve masquerading as Alice. Second, *confidentiality* ensures any message sent by Alice to Bob cannot be read by Eve. Finally, *integrity* ensures any message sent by Alice to Bob has not been tampered with by Eve.

The Snowden revelations [101] were a watershed moment in public understanding of, and demand for, privacy online. Between the diverse settings for applications requiring cryptography, and the constant evolution of cryptographic mechanisms to meet increasing security requirements, there is an ever-changing range of available cryptographic primitives and protocols. However, as computation power increases, old primitives become insecure [109, 96, 63], and over time, new attacks are found against systems thought secure [15, 4, 8]. To the average developer, selecting the correct primitive can be challenging; beyond that, they can be even more challenging to implement correctly.

1.1.2 Implementing Cryptography

The addition of cryptography to an application does not automatically make it safer or more secure for users. For example, in certain safety-critical systems, increased computation times involved with cryptographic operations may construct latency issues [111]. Cryptography itself can be circumvented by insecure system design: for example, TLS protects data in transit but, upon data sent by a client to a server being received, weaknesses in the server could reveal the data to an attacker. Importantly, the introduction of cryptography to an application requires that the developer attempting to enable privacy-as-a-feature for end users must utilize safe primitives and correctly implement trusted protocols in order to offer real security gains.

Developers are prone to mistakes, and errors in the implementation of cryptographic routines can fundamentally undermine their best intentions, introducing critical security weaknesses. For example, the end-to-end encrypted open source chat client Cryptocat suffered from critically weak security for over a year due to a type coercion bug in the use of private keys [102]. Where the protocol required an array of 15-bit random integers, the implementation instead provided a string of decimal characters, which was quietly coerced into the correct type by the JavaScript interpreter, leading to a vastly reduced key space of just 27.1 bits, down from 128 bits. Attacks such as FREAK [9] have demonstrated that even when individual components of an application are correct, the combination of these could introduce critical security weaknesses—or allow attackers to bypass this security entirely.

In order to rule out common errors in the implementation of cryptography, modern programming languages offer APIs (application programming interfaces) for primitives, and in some cases, protocols. They vary from comprehensive APIs written and verified by cryptographic experts (e.g., HACL* [116]) to smaller APIs are written by developers for specific purposes (e.g., bcrypt, a JavaScript library for password hashing [23]). These APIs prevent developers from handling the inner workings of low-level primitives, reducing the risk of developers incorrectly translating these algorithms to code. However, these APIs do not guarantee correct implementation: they are often flexible and many allow the use of insecure algorithms. For example, the Java Cryptography Architecture supports the insecure ECB (Electronic Codebook) mode of encryption [76]. Further, these APIs are flexible to allow for a myriad of use cases, which enables the introduction of misuse bugs. For example, in JavaScript's WebCrypto API [110], it is possible to use a constant, non-random value as the IV (initialization vector) for encryption. For AES-CBC (the AES cipher in Cipher Block Chaining mode, which is a mechanism allowing multiple blocks of

data to be encrypted together), this could reveal that two messages carry the same prefix, potentially leading to more damaging attacks. Indeed, it has been suggested that cryptographic APIs are still too low-level and present developers with challenges in correctly implementing secure cryptography in their application [68, 54].

Security weaknesses in applications built using cryptographic APIs are still widespread. A study of Android applications found large-scale failure of implementations to correctly implement cryptography: 88% of analyzed applications were found to violate core cryptographic tenants by misusing APIs, e.g., using constant encryption keys [32]. Such studies have been extended, and the problem of insecure implementations due to cryptographic misuse persists [53].

1.1.3 Detecting Cryptographic Errors

The core challenge addressed in this thesis is that cryptographic implementations, traditionally, have proved difficult to analyze. This means that bugs within both these implementations, and applications which use cryptography, are difficult to detect. Importantly, to a developer, such bugs also have no visible effect on runtime. For example, ciphertexts are designed to, on inspection, appear indistinguishable from random no matter the underlying keying material. This means that simple inspection by a developer will not reveal if the encryption is correctly implemented in general. Similarly, fixed test inputs (e.g., checking the ciphertext for a fixed plaintext input) check only that the encryption is correct on a single test vector. These tests are necessarily limited; they will not reveal, for example, that initialization vectors are not reused.

Bugs in cryptographic implementations often have subtle causes. This is exacerbated by the gap between theory and practice; developers using cryptography are not experts in underlying security proofs for pro-

tocols [68]. The core issue here is that security proofs are often heavily caveatted to specific (even idealized) settings or require specific preconditions on inputs. Such assumptions are often highly technical in nature and limit the applicability of the protocol. Since developers using these protocols are not experts—and indeed, should not be expected to be—these caveats and preconditions may be violated, leading to potential security bugs. Although efforts have been made to construct standardized cryptographic APIs which limit the extent to which developers can violate these assumptions [76, 116, 110], these errors often involve subtle requirements which cannot be ruled out by the API alone, e.g., the correct chaining together of specific API calls in a specific sequence. When developers make such errors, and if the program still executes, there is no obvious way to detect such an error.

The causes of these cryptographic bugs are therefore largely orthogonal to the observable outputs of a program. Since it is intimately related to the preconditions of cryptographic primitives, a core challenge of detecting cryptographic misuse is that we must distinguish between the origin of values in a program. For example, a cryptographic primitive may require that a key be the output of a cryptographically secure pseudorandom number generator (CSPRNG). We must distinguish such a key from another arbitrary string in the program. In general, for a single value, one cannot reason about how such a value was created. In this thesis we are interested in constructing mechanisms to establish the provenance of a value and therefore whether it satisfies the preconditions of a cryptographic primitive or protocol.

1.1.4 JavaScript and Cryptography

In this thesis, we are principally concerned with the application of cryptographic misuse detection to JavaScript programs. Despite its often cited

quirks, shortcomings, and outright design flaws, the popularity of JavaScript is unwavering [112]. It has long been an important building block of the world-wide web; but developments, like Node.js [73], have broken new ground by allowing its use as a general purpose programming language outside the web browser. The rise of major frameworks, such as Electron [40], has allowed the deployment of JavaScript for client-side applications, including mainstream communication applications, e.g., Slack[1], Discord[2] and Skype[3]. Over time, JavaScript's application domain has expanded from simple interactive web content to complex web applications, server-side frameworks, and embedded devices.

As a consequence, JavaScript code routinely handles security-sensitive data, such as financial or health details, even implementing cryptographic protocols. JavaScript cryptography has long been frowned upon [82, 5]; however, it has become increasingly popular—with a range of developer-led APIs [29, 66] and academic projects [94]. Cryptographic APIs have been used in JavaScript for end-to-end encryption in payment systems [75], off-the-record messaging[4], or to manage server certificates [97].

The demand for cryptography in JavaScript has led to standardization in the form of the W3C WebCryptography API [110] (henceforth, *WebCrypto*) and the Node.js module, `crypto` [70]. However, neither of these APIs provide any formal security guarantees, nor do they rule out unsafe usage patterns. Historical approaches to verifying cryptography in JavaScript predominantly rely on restriction to subsets of the language [52, 12]. Such languages are designed to enable the implementation of cryptographic protocols, and forbid access to external APIs, e.g., WebCrypto. In this thesis, we aim to address the challenge of checking existing crypto-

[1] `https://slack.com/`
[2] `https://discordapp.com/`
[3] `https://www.skype.com/`
[4] `https://github.com/arlolra/otr`

graphic implementations which use these standardized APIs.

1.2 The Problem

In this section we describe the scope of this thesis and give a problem statement. Given a program P, which utilizes some cryptographic API or library, L, we seek to automatically determine if P is secure. Of course, such a statement is very broad, and the notion of *secure* is vague. As such, we narrow our scope to a more tractable problem, which allows us to check that usage of L does not violate the underlying assumptions of the API (e.g., preconditions of cryptographic primitives).

The first restriction is to require that P is a JavaScript program. This provides a natural target for checking usage of recently standardized cryptographic APIs. In addition, JavaScript's dynamic type system and often surprising semantics make it difficult to spot subtle security bugs as long as they do not directly impact functionality. This means that JavaScript poses an interesting target for security-centric program analysis. However, we believe the approach described in this thesis to be applicable to languages beyond JavaScript; for example, Chapter 4 describes the application of the approach to a functional language.

Second, we assume that L is, in some sense, *trusted*. Both WebCrypto and the node.js library `crypto` provide access to cryptographic primitives, e.g., AES and SHA-256. Within this context, we assume that the implementation of these primitives is correct and without implementation bugs. From these correct primitives, we construct specifications for safe usage of these API functions, S, which describes the pre- and postconditions of safe usage for L.

Given these restrictions, this thesis seeks to address the problem of automatically checking that given a specification S of L, a JavaScript program

P obeys S. We have not defined how S is obtained; in practice, we will describe the form of these specifications, and explain how cryptographic experts can describe one for an arbitrary cryptographic API. This restriction of the scope means that we no longer seek full program verification; instead, we check that implementation bugs in the usage of L are ruled out.

In particular, we seek an analysis which encapsulates the following properties:

- The analysis should be *automatic*; this allows developers to obtain feedback on their use of cryptographic without needing to be an expert in the interplay between cryptographic primitives.
- The analysis should be, in some sense, *agnostic*. By this, we mean that the mechanisms by which we perform the analysis should be applicable to distinct APIs (and languages). Further, in the case of APIs, it should be easily *extensible* to new APIs as they are developed.
- The analysis should be *independent* of the implementation. That is, the developer should not have to modify their existing codebase (e.g., by writing in a strict subset of the language) to benefit.

1.3 The Approach

Despite several initiatives to verify not just cryptographic protocols but reference implementations in software [7, 14], it is not yet feasible to prove the correctness of mainstream implementations. Dynamic languages like JavaScript are notoriously difficult to formally reason about: the dynamic type system and extensive use of reflection thwart the application of expressive type systems or theorem provers without significant manual intervention by cryptographic experts [52, 10].

There is currently little support to help non-expert developers avoid introducing critical security bugs into applications built on full JavaScript. In this thesis, we describe a new approach for checking the use of cryptographic APIs in client code that is compatible with the dynamic type system and common design patterns of JavaScript. Our goal is to arrive at an automatic testing methodology that does not require manual analysis by cryptographic experts.

In this thesis, we introduce a mechanism which rules out misuse of trusted APIs in JavaScript code through runtime enforcement. Our core contribution is to introduce a system of Security Annotations that is orthogonal to the existing type system. Security Annotations themselves are described in detail in Chapter 3. Security Annotations express security properties of values, e.g., whether a value is a key of a certain length, or a cryptographically secure random value. They are propagated alongside regular type tags but follow their own semantics. Security Annotations are enforced at the boundaries of function calls, and are transparent to client code until they encounter an error (e.g., failing this enforcement), in which case the test program fails. This allows for the specification of trusted cryptographic APIs via Security Annotations, e.g., a specification for the WebCrypto API is given in Section 5.7. These specifications can then be checked in a system which supports Security Annotations.

We break down the research program described in this thesis into three phases. This ensures that the design of Security Annotations is sensible and correct, by providing first formal systems within smaller languages and security guarantees within those settings before full implementation. First, we demonstrate the type safety of our approach within a lambda calculus, which avoids the complexities of JavaScript but allows us to ensure the underlying mechanisms allow for checking of the intended properties (Chapter 4). Second, we extend a partial runtime semantics for JavaScript [81] with Security Annotations (Chapter 5). This allows us to ex-

plore the relationship between Security Annotations and many complex features of JavaScript and solve the natural problems that arise within this formal setting. Alongside this, we are able to provide limited security guarantees for a given API specification and program for which no control flow path results in a function contract violation.

Finally, this thesis describes an automatic testing methodology for JavaScript programs which utilize Security Annotations (Chapter 7). We implement Security Annotations for full JavaScript using code instrumentation [92] within the dynamic symbolic execution (DSE) engine ExpoSE [60, 61]. DSE is an effective strategy for test generation allowing the systematic enumeration of feasible control flow paths in a program, which has been successful for generating tests and finding crash bugs in real-world C and machine code [22, 42]. On one hand, the dynamic nature of DSE means that we will not obtain any security proofs over the entire program unless DSE terminates and all paths are explored; on the other hand, all paths executed are guaranteed to be real (feasible) paths with respect to the execution environment. Along each feasible control flow path, Security Annotations are either respected or we have found a possible security bug.

1.4 Analyzing String Manipulating Programs

Many programs which utilize cryptography must also necessarily perform string manipulations. In JavaScript it is most idiomatic to do this through regular expressions, e.g., for processing signed ciphertexts received across a network. It is therefore a core requirement for any JavaScript analysis targeted at cryptographic code to support regular expressions. Regular expressions are popular with developers for matching and substituting strings and are supported by many programming languages. For instance,

in JavaScript, one can write `/goo+d/.test(s)` to test whether the string value of `s` contains `"go"`, followed by one or more occurrences of `"o"` and a final `"d"`. Similarly, `s.replace(/goo+d/,"better")` evaluates to a new string where the first such occurrence in `s` is replaced with the string `"better"`.

Several testing and verification tools include some degree of support for regular expressions because they are so common [57, 107, 90, 104, 60]. In particular, SMT (satisfiability modulo theory) solvers now often support theories for strings and classical regular expressions [58, 59, 1, 2, 104, 115, 113, 16, 114, 38], which allow expressing constraints such as $\texttt{s} \in \mathcal{L}(\texttt{goo+d})$ for the `test` example above. Although any general theory of strings is undecidable [17], many string constraints are efficiently solved by modern SMT solvers.

SMT solvers support regular expressions in the language-theoretic sense, but "regular expressions" in languages like Perl or JavaScript—often called *regex*, a term we also adopt in the remainder of this thesis—are not limited to representing regular languages [3]. For instance, the expression `/<(\w+)>.*?<\/\1>/` parses any pair of matching XML tags, which is a context-sensitive language (because the tag is an arbitrary string that must appear twice). Problematic features that prevent a translation of regexes to the word problem in regular languages include capture groups (the parentheses around `\w+` in the previous example), backreferences (the `\1`, which refers to the capture group), and greedy/non-greedy matching precedence of subexpressions (the `.*?` is non-greedy). In addition, any such expression could also be included in a lookahead `(?=)`, which effectively encodes intersection of context sensitive languages. In tools reasoning about string-manipulating programs, these features are usually ignored or imprecisely approximated.

In the context of dynamic symbolic execution (DSE)—which, as discussed in Section 1.3, we intend to use for test case generation—this lack

of support can lead to loss of coverage or missed bugs where constraints would have to include membership in non-regular languages. The difficulty arises from the typical mixing of constraints in path conditions—simply *generating* a matching word for a standalone regex is easy (without lookaheads). To date, there has been only limited progress on this problem, mostly addressing immediate needs of implementations with approximate solutions [90, 91]. In this thesis, we present a comprehensive model for the specification of regexes as described in ECMAScript 2015 (ES6) [31], which enables the analysis of JavaScript programs featuring string manipulation (Chapter 6).

1.5 Contributions

In this thesis, we first discuss the necessary background (Chapter 2) then make the following contributions:

- We present a system of Security Annotations and design operators on them which allow for the expression of cryptographic API specifications (Chapter 3). This chapter provides an extended look at the underlying structure of Security Annotations discussed in *PEPM 2018* [65] and *ESORICS 2019* [64].

- We formalize a statically-checked mechanism for Security Annotations within a small lambda calculus, λ_{SA}, and provide safety guarantees (Chapter 4). This chapter comprises an extended version of work first published in *PEPM 2018* [65].

- We define a runtime semantics for Security Annotations within an existing partial formal semantics for JavaScript, $S5_{SA}$ and provide security guarantees for a particular API with a Security Annotation

specification (Chapter 5). This chapter comprises an extended version of work first published in *ESORICS 2019* [64].

- We discuss a particular challenge involved in the dynamic symbolic execution of JavaScript, providing a sound semantics for the modeling real-world regular expressions (Chapter 6). This chapter is based on work published in *SPIN 2017* [60] and *PLDI 2019* [61].
- Finally, we implement Security Annotations in full JavaScript, describe how it relates to the guarantees provided for formal semantics, and demonstrate its usage through case study (Chapter 7).

2 Background

In this chapter we outline the necessary background information and survey related literature to the project. We begin by describing state-of-the-art approaches to cryptographic verification in general (Section 2.1). We discuss work on the analysis of JavaScript code (Section 2.2) and then narrow this to current approaches to JavaScript security analysis (Section 2.3). Finally, we give an overview of Dynamic Symbolic Execution (DSE) as a method for test generation and describe one such DSE engine for JavaScript, ExpoSE (Section 2.4).

This chapter draws on the related work discussed in the author's published work as a first author [65, 64]. Section 2.4 draws on the author's other published work [60, 61] to describe ExpoSE, a dynamic symbolic execution engine for JavaScript designed by Blake Loring and Johannes Kinder.

2.1 Analyzing Cryptographic Implementations

Here, we describe the analytical toolbox for generic cryptographic applications. We begin by discussing generic typing approaches to program analysis and their relation to this thesis. We then discuss approaches to the checking of security properties and explain their relationship to cryptographic applications. Next, we describe cryptographic protocol verifi-

cation. Finally, we discuss the most closely related work to this thesis: checking cryptographic API usage—which until now has focused primarily on Java code and Android applications.

Type qualifiers. Type qualifiers [34] extend a language's base type system via composable qualifiers representing properties of the program terms. The canonical example of a type qualifier is the ANSI C qualifier `const`, which indicates that a variable may be initialized but never updated. Type qualifiers support the qualification of types in the manner of initial type declarations, and this qualification is fixed for the runtime of the program.

Security property checking. We discuss approaches for analyzing programs to check security properties which are influential to this thesis. Type systems for F#, such as F7 [7, 13, 14] and F* [98], allow for the description of security properties of terms via dependent types which are checked statically. These systems utilize type refinement [37] to encode disjoint security properties in terms of complex logical expressions. Their work demonstrates the applicability of type-based approaches towards static verification of security properties in implementations.

Security type systems designed to enforce secure information flow have proved influential; we refer to the work of Sabelfeld and Myers for a comprehensive survey [86]. In these systems, types of terms in the language are augmented with information describing policies for safe usage of the term. The canonical example is to describe two policies: a term has a sensitivity level that is either *high* or *low*. A security type system seeks to eliminate both *explicit* flows from terms labeled high to those labeled low (e.g., by assigning data labeled high to a variable labeled low), but also *implicit* flows. To understand these, consider the simple program `h := i ; l := false; if (h) { l := true }`. In this example, `i` is some program input stored in a high-security variable `h`. Depending on this input,

the value of `l` may be changed: this is an implicit flow of high-security information to a low-security variable. This line of work seeks to detect such violations, and use of these systems can offer strong guarantees; unfortunately they have proved impractical in the JavaScript setting. We discuss the related concept of dynamic information flow monitoring, which has seen success in the setting of JavaScript, in Section 2.2.

Cryptographic protocol verification. There is a strong strand of work on cryptographic protocol verification in the literature; tools such as EasyCrypt [6] or FCF [79] seek to construct frameworks which allow for the construction of provably secure protocols through proof assistants. Such tools are targeted specifically towards protocol verification rather than implementations written by developers. This work is therefore orthogonal to the approach of this thesis, where we seek to check developer implementations.

Checking cryptographic API usage. We discuss the current state-of-the-art in checking developer use of cryptographic APIs; similar to this thesis, these works seek to rule out developer misuse. In pursuit of this aim, Lazar *et al.* sought to identify the origins of bugs in cryptographic software [54]. Across a survey of 269 CVEs related to cryptographic vulnerabilities, they found that 83% were due to misuse of cryptographic libraries. In another survey of the state of cryptographic APIs, Nadi *et al.* [68] argue that Java APIs are too low-level, expecting developers to possess an implicit understanding of the underlying cryptographic protocols.

In a foundational work, Egele *et al.* introduced CryptoLint [32], a lightweight static analyzer for Android applications which to detect cryptographic misuse. They found that of 11,748 applications analyzed, 88% violated at least one of six identified cryptographic rules [32]:

(i) Do not use ECB mode for encryption.

(ii) Do not use a non-random initialization vector (IV) for CBC encryption.

(iii) Do not use constant encryption keys.

(iv) Do not use constant salts for Password-based encryption (PBE).

(v) Do not use fewer than 1,000 iterations for PBE.

(vi) Do not use static seeds to seed `SecureRandom(seed)`.

The first two of these cover encryption modes to rule out deterministic ciphers: the former mode is known to be insecure, and in the latter the chaining mechanism of the blocks in CBC mode is undermined by using a deterministic IV [50]. Rules (iii) and (iv) ensure that it is not trivially easy to recover secret keys; (v) is a heuristic for secure PBE as defined in the standard [49]. Finally, rule (vi) is specific to Android's `SecureRandom` pseudo-random number generator. In this thesis, we go beyond these types of rules and develop specifications which rule out deeper classes of cryptographic failure, e.g., rule (iii) is extended to ensure keys have been properly generated by a trusted API.

In research contemporaneous with this thesis, Krüger *et al.* present *CrySL*, a domain specific language (DSL) for the specification of correct usage of cryptographic APIs, focusing on the Java Cryptography Architecture [53]. These specifications are used to statically analyze applications deploying these cryptographic APIs.

2.2 Analyzing JavaScript Programs

In this section we describe approaches from the literature to analyzing JavaScript code and making it safe. We start by discussing efforts to de-

velop formal semantics for JavaScript and the application of these to program verification. We then discuss existing typing approaches for JavaScript, and dynamic contract enforcement mechanisms for the language. At each step, we describe the relationship of these to the approach described in this thesis.

Formal Semantics for JavaScript. There is a plethora of work in the space of formal semantics for JavaScript [88, 77, 43, 81], taking differing approaches towards describing the language. λ_{JS} [43] and its successor S5 [81] provide small languages which model the key features of JavaScript. S5 targets EcmaScript 5.1 (ES5) [31], while λ_{JS} targets the older EcmaScript 3. Both languages remain close to the standard lambda calculus in order to make them amenable to extension [62] and analysis: for instance, Guha *et al.* describe an analysis to disallow use of the object `XMLHttpRequest` [43].

JSIL [88] aims to follow the standard line-by-line, designed for symbolic verification of JavaScript programs. By the authors' admission, it is ill-suited to high-level analyses of JavaScript programs like those of examining cryptographic API usage. KJS [77] is an executable formal semantics for JavaScript based on the K framework [85]. Formal semantics for fragments of JavaScript have also been constructed in order to demonstrate the static type-checking of popular tools Flow [25] and TypeScript [83]. However, these small fragments are incomplete and make no attempt to represent themselves as tested semantics aiming to construct JavaScript-like languages.

Verification of generic JavaScript programs We discuss the state-of-the-art for formal verification of JavaScript programs: if current verification approaches suffice to precisely ensure the correctness of programs then it would follow that there is no requirement to design a custom testing

framework for JavaScript cryptography.

Recent work has demonstrated the tractability of logic-based symbolic verification techniques towards JavaScript [87, 36]. Building upon previous work [39], the authors propose a toolchain for semi-automatic verification of JavaScript programs; this comprises an intermediate language along with a JavaScript-to-intermediate language compiler, and logical specifications of JavaScript internal functions. To verify individual programs, a user must manually write precise logical specifications for each function: this can provide strong guarantees at the cost of requiring significant manual intervention by developers.

Verifying the cryptographic properties of a program using this framework would require the developer to manually write the cryptographic specifications themselves for each function used in the program.

Type systems for JavaScript. There is a plethora of existing work on expressive type systems for JavaScript: both research tools [27, 108] and industry languages such as Flow [25] and TypeScript [83]. All of these approaches require the application developer to write within this dialect to leverage the benefits enabled by static checking. In this thesis, we aim to avoid such a requirement to enable the testing of existing implementations. Such approaches suffer the disadvantages inherent to static typing-based approaches, e.g., false positives.

Design-by-Contract systems. Contracts traditionally specify the pre- and postconditions of individual methods in a program, along with invariants about individual objects. Contracts are designed for runtime enforcement to ensure that programs behave correctly in their deployed environment; recent work has extended this concept to JavaScript [51, 30]. TreatJS [51] is a language-embedded higher-order contract system which works by enforcing these contracts via run-time monitoring; predicates (e.g. `TypeStr`

) are checked by custom JavaScript functions. This approach allows for the checking of simple properties, ones which can be checked inline by native JavaScript: the properties we wish to check - such as checking a key has been properly generated as a precondition - cannot be easily expressed in such an extension. Further, contracts specify invariants of objects and pre- and postconditions of functions, which mean they cannot express the relationships between values within control flows (e.g. a key generated by a proper function followed by an encrypt function). Encoding security properties within contracts would require a mechanism to check such meta-properties: one could naturally encode these properties between functions in JavaScript by adding additional type-like information to values—this is the approach described within this thesis. Alternatively, one could achieve a similar aim by introducing additional shadow variables. Ultimately, in the dynamic setting of JavaScript, this approach would be equivalent to a system of dynamic typing such as that introduced in this thesis.

Information-flow monitors. Work on monitors [44, 89] provide mechanisms for dynamic enforcement of information flow in JavaScript; work on information flow monitoring in the presence of libraries [45] extends the applicability of monitor-based approaches. COWL [95, 46] is an information flow control system for web browsers preventing third-party library code from leaking sensitive information, achieved via the labeling of browser contexts. This dynamic approach bears much similarity to the static approaches advocated by Security Type Systems as discussed in Section 2.1. These systems utilize a dynamic tag-based approach (e.g., the work of Hedin *et. al.* [44]). Values are labeled with *security labels* representing a security classification. At runtime, labels are propagated through the program and checked against particular security policies, e.g., that a high-security value is not leaked to the user. As the program runs, when

a computation involves (directly or indirectly) a high-security value, the result is also high-security. However, values can never have their security labels declassified (or lowered) by this approach: as such, high security values cannot become low security. Unfortunately, this is not true of all security properties, and particularly those discussed in this thesis, which require declassification. For example, consider a `Uint32Array` representing a cryptographic key: the alteration of a single element means the entire array can no longer be considered a valid key. Further, rather than general security policies, we wish to enforce specific properties on values at the boundaries of APIs; natural mechanisms for this are discussed in detail in Section 4.4.2 and Section 5.4.2.

2.3 JavaScript and Cryptography

We describe the relationship between JavaScript and cryptography. First, we give a brief overview of the cryptographic APIs in JavaScript. Second, we discuss the current state-of-the-art in cryptographic analysis for JavaScript, which focuses on verification of protocol implementations.

Cryptographic APIs for JavaScript There is a plethora of established cryptographic APIs available for client-side JavaScript. This diverse collection of APIs ranges from those written by experts (e.g. SJCL [94]) to those written by developers (e.g. CryptoJS [66]). However, the need for an established standard for cryptography in JavaScript led to the development of the W3C specification for the Web Cryptography API, or *WebCrypto* [110]. Although previous libraries remain in use (particularly in legacy applications), WebCrypto is now the de-facto standard for client-side applications.

We describe in more detail the WebCrypto API, which is the principal

example throughout this thesis. The API is available within web browsers through the `crypto` property of the `Window` object. This API provides both a `SubtleCrypto` interface for access to cryptographic primitives, accessed via `crypto.subtle` (so named because the use of these primitives is *subtle* and requires careful use), and a source of randomness through `crypto.getRandomValues`. The method `getRandomValues` takes as argument a JavaScript `TypedArray` object, e.g., an array of unsigned 16-bit integers, `Uint16Array`. The method overwrites the elements of the array with values which are considered to be random for cryptographic purposes, and returns this array.

The `SubtleCrypto` interface exposes a series of methods to the developer allowing the use of cryptographic primitives. They make extensive use of objects representing keys, `CryptoKey` and `CryptoKeyPair`. The latter of these describes a key pairing simply as an object with two properties, `privateKey` and `publicKey`, each of which is a `CryptoKey` object. The `CryptoKey` object itself allows for interaction with the key but does not directly expose the raw key to the developer. It has four accessible properties:

- `type`, which is either `"secret"` (in the case of symmetric algorithms), `"private"` or `"public"` (in the case that the key is stored as part of a `CryptoKeyPair` object).

- `extractable`, which is **`true`** if the key can be extracted using the API and **`false`** otherwise. That is, if this is **`true`**, the developer is able to directly access the raw bytes of the key. Otherwise, the key is only accessible through the `CryptoKey` object.

- `usages`, an array of strings governing what the key can be used for. Possible elements are `"encrypt"`, `"decrypt"`, `"sign"`, `"verify"`, `"deriveKey"`, `"deriveBits"`, `"wrapKey"`, `"unwrapKey"`. If a key is

passed to a method not in this list, the use of the key is considered invalid.

- `algorithm`, which is an object describing the parameters of the algorithm the key is valid for.

Table 2.1 describes the algorithms supported by the various WebCrypto APIs exposed by `SubtleCrypto`. We note that WebCrypto does not support certain known insecure encryption modes (e.g., ECB mode as discussed in Section 2.1), in a move that demonstrates progress in API design since the work of Egele *et al.* [32]. When specifying the algorithm as an argument to any of these API calls, one must provide an object which specifies both the argument and any special parameters required for use of the primitive. For example, to encrypt a message with AES-GCM, one must provide an object with a `name` property (the string `"AES-GCM"`), and an `iv` property (which should be distinct for each encryption). We do not describe the individual methods in Table 2.1 in detail.

All `SubtleCrypto` APIs return JavaScript `Promise` objects. A promise, first introduced in ES5, is "an object that is used as a placeholder for the results of a deferred (and possibly asynchronous) computation" [31]. For example, calling `encrypt` returns a promise to a buffer of the resulting ciphertext. In this way, WebCrypto makes extensive use of asynchronous code to defer cryptographic routines, only executing them when necessary. The handling of the result of a promise `p` is achieved through the syntax `p.then(f, r)`: if the computation succeeds, `f` is called. If it fails, `r` is called. A promise is considered to be in one of the following states:

(i) *fulfilled* if the computation succeeds, and `f` is called.

(ii) *rejected* if the computation fails, and `r` is called.

(iii) *pending*, otherwise.

```
1  let p = function() {
2    return new Promise(function(f, r){
3      setTimeout(function() { f('resolved'); }, 5000)
4    });
5  };
6
7  let g = async function() {
8    console.log('entering g');
9    let res = await p();
10   return res;
11 }
12
13 console.log(g());
```

Listing 2.1: A program using `async/await`

A promise is *resolved* if it is either fulfilled or rejected. Promises provide the important guarantee that neither `f` nor `r` will be called until `p` is resolved. Throughout this thesis, we use the `async/await` syntax to abstract the complexity of chaining multiple promises together. Consider the example in Listing 2.1: first, `g` is marked `async`. This allows `await` expression on line 9 inside the function. Here, execution is paused until this promise is resolved; execution of `g` is then resumed. In this case, the function `p` returns a promise which waits for 5 seconds before resolving to the string `'resolved'`. As a result, this program prints `'entering f'` and then prints `'resolved'` five seconds later.

Finally, we discuss those APIs which allow for extracting and generating keys. The `generateKey` allows the user to specify the valid usages of the key to be generated: first, the user specifies which algorithm the key is for, and the parameters of this algorithm, e.g., the name of the elliptic curve to use in Elliptic Curve-based Diffie Hellman Key Exchange (ECDH). Second, the user declares whether the raw bytes of the key can

Table 2.1: Cryptographic algorithms supported by WebCrypto.

WebCrypto API Call	Supported Algorithms
`sign()`, `verify()`	RSASSA-PKCS1-v1_5, RSA-PSS, ECDSA, HMAC
`encrypt()`, `decrypt()`	RSA-OAEP, AES-CTR, AES-GCM, AES-CBC
`wrapKey()`, `unwrapKey()`	RSA-OAEP, AES-CTR, AES-GCM, AES-CBC, AES-KW
`digest()`	SHA-1, SHA-256, SHA-384, SHA-512
`deriveKey()`, `deriveBits()`	ECDH, HKDF, PBKDF2

be extracted; third, the user declares possible uses of the key. As with the other methods of WebCrypto, this returns a promise to a `CryptoKey` object containing the new key. The `exportKey` API takes as argument a data format (e.g., `jwk`, for a JSON web key), and a `CryptoKey` argument. It then returns a promise to a buffer containing the key stored in the specified format. In this way, for example, a public key can be broadcast across a network. The dual of this method, `importKey`, allows data to be stored in a `CryptoKey` object so that it can be used as a cryptographic key in future WebCrypto API calls.

Analyzing cryptographic implementations. There is a strong line of work involving the restriction of JavaScript to subsets in order to analyze cryptographic implementations. In particular, this work seeks to leverage cryptographic experts for semi-automatic verification of protocol implementations.

Bhargavan *et al.* proposed a statically typed subset of JavaScript for the protection of cryptographic protocols, Defensive JavaScript (DJS) [11, 12]. The core aim of DJS is to guarantee behavioral integrity of JavaScript even when loaded in a hostile environment. DJS is designed for use by cryptographic experts: guarantees of security properties are governed by proofs

in the ProVerif protocol analyzer [19]. This analysis requires the developer to hand-write models and the automated extraction of these models from DJS also requires the program be loop-free. Further, DJS itself is a restrictive subset of JavaScript which heavily limits developer freedom. For instance, DJS forbids access to methods of native objects, such as `String.length`, or access to DOM variables, such as `document.location`. Such restrictions greatly limit the practicality of DJS for developers.

ProScript [52] is a DSL for the development of verified protocol implementations which can be executed within JavaScript applications, and also compiles to a formal model which allows for symbolic analysis. With intervention from a cryptographic expert, this in turn can be used to construct computational proofs in CryptoVerif [18]. This has been successfully used to verify implementations of both TLS 1.3 Draft-18 [10] and a variant of the Signal protocol [52]. This line of work is complementary to this thesis; in describing a new language allowing for verification, developers are able to partner with experts to fully verify new implementations. Our work focuses on existing implementations written in full JavaScript and does not require cryptographic expert intervention to detect many classes of cryptographic errors.

2.4 Test Generation for JavaScript

In this section we outline a particularly effective strategy for test generation which allows us to systematically enumerate feasible control flow paths of a program. We first provide a primer on the technique of dynamic symbolic execution (Section 2.4.1). We then describe the architecture of ExpoSE (Section 2.4.2), a dynamic symbolic execution engine for JavaScript.

2.4.1 A Primer on Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) has proved a successful technique for automatically finding bugs in real world software. Traditionally, DSE engines mostly targeted C, Java, or binary code [42, 78]. However, implementations of DSE for JavaScript have shown promising results for testing of JavaScript code [90, 55, 60].

In DSE, some inputs to the program under test are made *symbolic* while the rest are fixed. Starting with an initial concrete assignment to the symbolic inputs, the DSE engine executes the program both concretely and symbolically and maintains a *symbolic state* that maps program variables to expressions over the symbolic inputs. Whenever the symbolic execution encounters a conditional operation, the symbolic state's evaluation of the condition, or its negation, are added to the *path condition*, depending on the concrete result of the operation. Once the execution finishes, the path condition uniquely characterizes the executed control flow path. By negating the last constraint of the path condition (or one of its prefixes), the DSE engine generates a constraint for a different path. A constraint solver is then called to check feasibility of that path; if this path is feasible, a satisfying assignment for the symbolic input variables is returned. This satisfying assignment can then be used as a new test case for the program, executing a distinct control flow path.

An example. Consider the artificial example in Listing 2.2. We want to analyze this function with DSE to see if a combination of inputs can result in an error. To do this, we first mark both inputs (line 1), `x` and `y` as symbolic—with symbols X and Y respectively—and provide an initial concrete input for each, say `x=0` and `y="a"`. The program is then run with these concrete inputs, and at each program branching point, we record and add the constraint to the path condition.

```
1  function f(x, y) {
2    var z = x + 2;
3    if (z > 10) {
4      if (y == "err") {
5        throw "Error";
6      } else {
7        console.log("Success");
8      }
9    } else {
10     console.log("Success");
11   }
12 }
```

Listing 2.2: A simple branching JavaScript program.

For these concrete inputs, at line 3, `z=2`, so `z>10` is false, and we instead enter the **`else`** branch (line 9). Collecting up the path constraint and simplifying it in terms of our symbolic program input variables, this control flow path is characterized via the path constraint $pc_1 = X + 2 \leq 10$, since this is equivalent, after simplifying and describing in terms of our symbols, to $\neg(z > 10)$. We then negate the final (in this case only) constraint of pc_1, obtaining $X > 8$, and pass this to a constraint solver, which finds that this is satisfiable and returns a satisfying model, say `x=9, y="a"`. There are no constraints on `y`, so the constraint solver could offer any string as a valid assignment; in this case, the assignment is unmodified.

We then execute our function again with this new satisfying model as input, again collecting the constraints as we go: this time `z>10` is true, so we take this branch (line 3) and then perform an equality check on `y` (line 4). This fails, so we execute the else branch (line 6) and the program terminates. We collect up constraints in terms of our symbols as before and obtain $pc_2 = X > 8 \wedge Y \neq$ `"err"`. Negating the final constraint of pc_2, and

passing this to a constraint solver yields a satisfying model `x=9, y="err"`. Using these as concrete inputs reaches line 5, triggering the bug. Since the negating constraints within the path condition for this execution results in a path we have already triggered, we also know we have covered all control flow paths within the function.

Concretization. The joint symbolic and concrete execution is a core advantage of DSE: when an external function cannot be analyzed, or an operation lies outside the constraint solver's theory, the DSE engine can concretize parts of the symbolic state without sacrificing soundness, at the cost of reducing the search space. Nevertheless, avoiding excessive concretization is important to allow effective test generation. This is one of the reasons why low-level or byte-code languages are popular DSE targets, while keyword-rich languages with large standard libraries are supported less frequently [21]. In order to avoid excessive concretization in JavaScript, constructing robust models for such standard libraries is important. In Chapter 6 we describe the construction of a model for a particularly complex part of the ECMAScript 2015 standard [31]: the regular expression API (see Section 6.1 for details of this API).

2.4.2 ExpoSE: Practical DSE for JavaScript

ExpoSE [60, 61] is a DSE engine for JavaScript programs. In this section we give an overview of ExpoSE, describing both its general architecture and approach to test case selection.

Architecture ExpoSE takes a JavaScript program and a symbolic unit test (a test harness) and generates test cases until it has explored all feasible paths or exceeds a time bound. ExpoSE consists of two main components, the *test executor* and the *test distributor*, as shown in the overview in Fig-

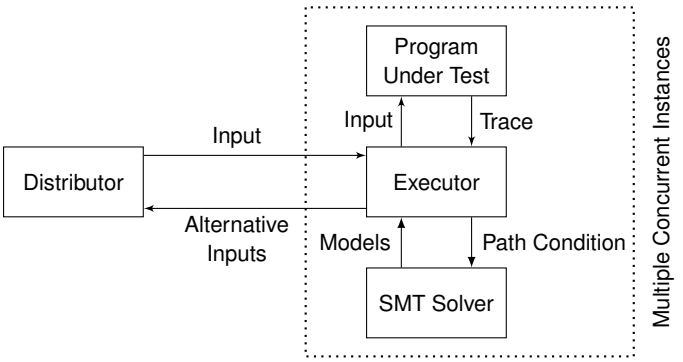

Figure 2.1: The ExpoSE architecture [60].

ure 2.1. The distributor manages the global state of the exploration, aggregates statistics, and schedules test cases for symbolic execution. The ExpoSE framework allows for the parallel execution of individual test cases in separate test executor processes, aggregating coverage and alternative path information as each test case terminates. This parallelization is achieved by executing each test case as a unique process allocated to a dedicated single core; as such the analysis is highly scalable.

The test case executor instruments the program under test to perform DSE and detect any bugs during execution. ExpoSE uses the Jalangi2 [92] framework for instrumentation of JavaScript software in order to create a program trace. It inserts callbacks for all JavaScript syntax, including in code dynamically created by **`eval`** and `require`. As the program terminates, ExpoSE calls the SMT solver Z3 [28] to identify all feasible alternate test-cases, in the form of models, from the trace. These models are then added to a queue of test cases.

Test case selection. The next test case to be executed is selected from the queue for execution in the manner of generational search [42]. The strategy for test case selection is similar to the CUPA strategy proposed

by Bucur *et al.* [21]. Program fork points are used to prioritize unexplored code: each expression is given a unique identifier and scheduled test cases are sorted into buckets based upon which expression was being executed when they were created. The next test case is selected by choosing a random test case from the bucket that has been accessed least during the analysis; this prioritizes test cases triggered by less common expressions.

3 Structure of Security Annotations

We describe the mathematical structure underpinning Security Annotations. In order to avoid obscuring this presentation with the complications of language mechanisms, in this chapter we divorce Security Annotations from the context of a single language. We begin by describing their design and the types of security properties that can be modeled in this setting; we explore the necessity of hierarchical Security Annotations and their combination via composition through example (Section 3.1). Next, we formalize these notions: we first define a mechanism for the declaration of Security Annotations for a given program, and we describe the partial ordering on Security Annotations induced by combining security properties. We discuss a novel `cut` operator for the removal of Security Annotations (Section 3.3). Finally, we discuss related work (Section 3.4).

This chapter details the underlying foundations of Security Annotations. Some results and definitions contained in this chapter (e.g., the definition of `cut` in Section 3.3) have been previously discussed in work for which the author of this thesis was first author [65]. However, the full presentation of the concept of Security Annotations and accompanying proofs are unique to this thesis.

3.1 What are Security Annotations?

Following their introduction in Chapter 1, we delve further into what Security Annotations are, and the security properties they represent. Through example, we illustrate the relationship between Security Annotations and programs.

Security properties. The notion of a security property is somewhat vague but we use it to ground the more concrete notion of Security Annotations. For a given program (or program term), a security property is simply any property relating to the security of the program (term). Of course, we have not defined what we mean by security: within the context of this thesis we confine ourselves to properties underpinning secure and verified communication. In this sense, we consider predominantly cryptographic properties of programs and terms.

Security properties can be considered as preconditions for cryptographic functions and are then valid properties of results of evaluation of these functions (postconditions). For example, negotiating a Diffie-Hellman Key Exchange with another party allows us to establish clear security guarantees about encryption performed with the resulting secret key, under the assumption that the secret keys were properly generated and sufficiently large, and that the underlying group and corresponding generator for that group are chosen properly. If this is the case, we can say that the resulting shared secret is indeed known only to these two parties.

Such properties allow us to make specific security guarantees about overall programs: for example, if a secure key exchange was combined with encryption via AES-CBC with a sufficient block size, one can argue that the resulting messages can be read only by the intended recipient. However, in order to express these relationships between different cryptographic primitives, we need a mechanism for describing the properties of

a program term—either statically or during program evaluation. Otherwise, we cannot ensure these implicit contracts are respected: encryption may be correctly performed, but if the key exchange was undermined in some way, then the resulting encryption does not offer any meaningful security guarantees.

Defining Security Annotations. Security Annotations represent the security properties which are valid on program terms (or, in the case of dynamic languages, individual objects and values). For example, the return value of a trusted key generation API is a valid cryptographic key, so could carry an annotation `CryptKey`. Such Security Annotations, which represent only a single security property are *atomic*. Annotations are composable to represent multiple distinct properties: for example, if a value is a key and has been generated as a cryptographically secure random value (CSRV), then it must be associated to both the corresponding annotations since it can be used in operations requiring either of these. The composition of these, via the operator $*$, is written `CSRV * CryptKey`. Security properties are also hierarchical: for example, `PrivKey` is more specific than `CryptKey`. Security Annotations therefore have a notion of subannotation judgments, e.g., `PrivKey` $\prec:$ `CryptKey`.

Different libraries have different security properties, and to prevent reasoning about unused security annotations, we propose that each library declare the Security Annotations used. This avoids reasoning about every possible annotation in each program, and also allows for the construction of new Security Annotations within a library when required.

A concrete set of Security Annotations. We describe a set of Security Annotations which allow us to express the security properties required for the JavaScript WebCrypto `encrypt` API to produce a secure ciphertext. Listing 3.1 describes a specification for this API in terms of Security An-

notations. In particular—and most importantly for illustrating the structure of security annotations—lines 1-3 describe the Security Annotations required to construct this specification. Line 1 defines the base Security Annotations used in the specification. Line 2 defines the subannotations of `CryptKey` and line 3 the subannotations of `Message`. These Security Annotations represent the following underlying security properties:

- `CSRV` represents cryptographically secure randomly generated values: any value or term annotated with this is assumed to have been generated by a random number generator with sufficient entropy that it can be used in cryptographic operations.

- `Message` is a Security Annotation on values which, in of itself, does not have any specific notions of security. However, it has two subannotations, `Plaintext` and `Ciphertext`. The former is used to represent unencrypted values, and the latter represents values which have been encrypted.

- `CryptKey` represents the security property that a value is a valid cryptographic key. The subannotations `PrivKey`, `PubKey` and `SymKey` represent specific types of cryptographic key: public, private and symmetric, respectively, which are utilized for different cryptographic algorithms.

In this chapter we will not describe in detail how the actual enforcement mechanisms work to define a specification for `encrypt` API; details of the runtime manipulations of Security Annotations in JavaScript are deferred to Chapter 5. However, we describe how the structural features of Security Annotations are used to define this specification. First, lines 6-8 describe a check of an atomic Security Annotation—namely, that the first argument to the API is an object containing a property `"iv"`, which is annotated

with `CSRV`—that is, the initialization vector to be used for the encryption has been generated from an API providing a sufficient source of entropy.

We seek to check that the key provided for the encryption has been properly generated and is of the correct type. The check that the key argument is properly generated is performed on line 7—we simply check the argument is a valid cryptographic key. WebCrypto supports both symmetric encryption schemes (via AES) and asymmetric encryption schmes (via RSA); we differentiate between these two cases. In order to do this, we recall the encoded hierarchy of Security Annotations relating to keys (line 2); this enables us to perform a more granular analysis of the supplied key. Line 10 ensures that when a symmetric algorithm is chosen, the supplied key is derived from a valid key derivation scheme producing a shared symmetric key (`SymKey`). This prevents, for example, the user mistakenly using a private key to encrypt the data rather than the shared secret. Similarly, line 12 ensures that when an asymmetric algorithm is selected, the key argument is a public key, per the requirements of RSA.

Finally, lines 18-19 encodes the security postconditions of the API in terms of Security Annotations. First, we remark that any security property of the original data to be encrypted remains true of the ciphertext, so we use **`cpAnn`** expression to duplicate these annotations onto the return value. The result is a valid encrypted message, so we describe this via the `Ciphertext` annotation (added to the annotations on the value via the **`as`** expression), and because of this, we must discard the dual of this annotation, `Plaintext`, if it was valid security property of the original data copied across (via the **`drop`** expression).

```
1  SecAnn <CSRV * Message * CryptKey>;
2  SecAnn <PrivKey * PubKey * SymKey> Extends <CryptKey>;
3  SecAnn <Plaintext * Ciphertext> Extends <Message>;
4
5  window.oldCrypto = window.crypto;
6  const encShim = async function(alg :S ["iv", <CSRV>],
7                                 key : <CryptKey>,
8                                 data) {
9    if (/AES/.test(alg.name)) {
10     (function(arg : <SymKey>) {})(key);
11   } else if (/RSA/.test(alg.name)) {
12     (function(arg : <PubKey>) {})(key);
13   } else {
14     throw FailedSecurityCheck;
15   }
16   var res = await
17     window.oldCrypto.subtle.encrypt(alg, key, data);
18   return (cpAnn(data, res) drop <Plaintext>)
19     as <Ciphertext>;
20 }
21 Object.defineProperty(window.crypto.subtle,
22                       "encrypt",
23                       {value: encShim});
```

Listing 3.1: An annotated shim of WebCrypto's `encrypt` API.

3.2 Formalizing Security Annotations

In this section we describe a lattice of Security Annotations, which provide a natural structure for the security properties in this thesis. We define a mechanism for introducing atomic annotations and their composition of these annotations, and then describe their lattice structure. We generalize this definition to allow for user-defined hierarchies of atomic Security Annotations and describe the construction of a lattice for this generic approach to defining Security Annotations.

Defining atomic Security Annotations. Security Annotations representing a single security property are defined on a per program basis. Across languages, the statement **SecAnn** a defines a Security Annotation with the name a. We call annotations defined in this way *atomic* since they encode only a single property. Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be the set of all names of atomic Security Annotations defined in this way in a program P. Throughout this thesis, we assume $\mathcal{A}$ is finite. We will frequently abuse notation in this thesis and write **SecAnn** $a_1 \star \ldots \star a_n$, to represent the sequence of statements **SecAnn** a_1**;** $\ldots$ **SecAnn** a_n.

Composition of annotations. As discussed in Section 3.1, program terms may possess multiple Security Annotations, composed via an operator $*$. To describe this composition, we first describe a structure for Security Annotations which allows for a natural ordering of composed annotations. Recall that the set $\mathcal{A}$ describes the names of all atomic Security Annotations in a program. The set of all possible annotations in the program is therefore $\mathcal{P}(\mathcal{A})$, the power set of $\mathcal{A}$. In this way, for $\mathcal{A}$ defined as above, the atomic annotations are simply the singleton sets $\{a_i\}$ for $i \in \{1, \ldots, n\}$. Throughout this thesis we will frequently drop this singleton set from notation and simply write a_i for these annotations. Since Security Annota-

tions are more specific when they describe more security properties, we order $\mathcal{P}(\mathcal{A})$ by reverse inclusion, i.e., if $S, S' \in \mathcal{P}(\mathcal{A})$, $S \prec: S'$ whenever $S' \subseteq S$. This structure gives a natural lattice on $(\mathcal{P}(\mathcal{A}), \prec:)$ with:

- Greatest lower bound (denoted $\sqcap$): set union ($\cup$)
- Least upper bound ($\sqcup$): set intersection ($\cap$)
- Top element (`Top` or $\top$): $\emptyset$
- Bottom element (`Bot` or $\bot$): $\mathcal{P}(\mathcal{A})$.

This structure is a variation of the standard powerset lattice where we have reversed the ordering (Example A.1, [69]), and gives us a practical ordering of Security Annotations. The composition of two annotations, S and S' is simply greatest lower bound, i.e., $a * b$ is simply syntactic sugar for $\{a\} \sqcap \{b\}$.

Hierarchies of atomic annotations. Unfortunately, the notion of our Security Annotation lattice is not quite so simple in practice. As illustrated in Listing 3.1, atomic annotations may be ordered into atomic hierarchies. To encode this, we introduce a new ordering $\prec:_{\mathcal{A}}$ on $\mathcal{A}$ with an initial axiom that for all $a \in \mathcal{A}$, $a \prec:_{\mathcal{A}} a$.

We introduce hierarchies in programs through the language-agnostic statement **`SecAnn`** a **`Extends`** b, which first defines a by adding it to $\mathcal{A}$ and then constructs the axiom $a \prec:_{\mathcal{A}} b$ which orders a and b[1]. We extend this relation to its transitive closure, i.e., if $a \prec:_{\mathcal{A}} b$ and $b \prec:_{\mathcal{A}}$ we add the axiom $a \prec:_{\mathcal{A}} c$. By definition, $\prec:_{\mathcal{A}}$ is a partial order on $\mathcal{A}$.

We abuse notation and write **`SecAnn`** $a_1 * \ldots * a_n$ **`Extends`** b to represent **`SecAnn`** a_1 **`Extends`** b`;` $\ldots$ **`SecAnn`** a_n **`Extends`** b. Motivated by our concrete

[1]We forbid declarations of the form **`SecAnn`** a **`Extends`** b where a is already defined, which ensures that the resulting ordering is antisymmetric (i.e., one cannot construct a circular ordering $a \prec:_{\mathcal{A}} b \prec:_{\mathcal{A}} a$).

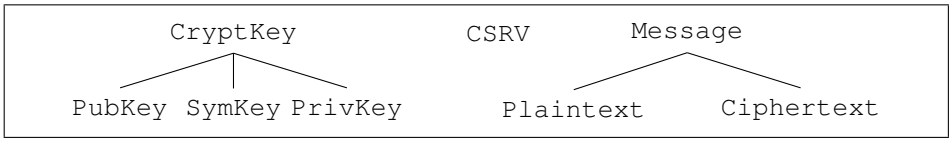

Figure 3.1: The atomic Security Annotation hierarchy for Listing 3.1.

link to security properties, we do not allow the definition of atomic annotations to be subannotations of multiple atomic annotations: this would not make sense. Similarly, we do not allow an atomic annotation to be defined as a subannotation of the composition of annotations, since it is not clear what this would mean for the underlying security properties.

An illustrative atomic hierarchy. The atomic Security Annotation hierarchy for the program in Listing 3.1 is given concretely in Figure 3.1. The atomic hierarchy in this program is governed by three distinct trees of annotations, note that `CSRV` has no parent or child annotation. The set $\mathcal{A}$ comprises all annotations given in the diagram.

The set of Security Annotations. From our partial ordering on $\mathcal{A}$, we construct a new, smaller, lattice of Security Annotations, with natural ordering and composition operator. We consider our set of annotations to be the power set of $\mathcal{A}$ without any annotations which would contain more than one member of an atomic hierarchy. Explicitly, we define the set of annotations as:

$$L := \mathcal{P}(\mathcal{A}) \setminus \{S \mid \{a, b\} \subseteq S \wedge a \prec:_{\mathcal{A}} b\}.$$

A partial ordering on Security Annotations. Since this is simply a subset of the power set of $\mathcal{A}$, we inherit its partial ordering of reverse subset inclusion. However, we extend this definition to incorporate our partial

ordering on atomic hierarchies. We say that $S_1 \prec: S_2$ whenever:

(i) $S_2 \subseteq S_1$, or

(ii) $\forall b \in S_2, \exists a \in S_1$ such that $a \prec:_{\mathcal{A}} b$.

That is, we check first if the elements are ordered by reverse set inclusion. If not, intuitively, $S \prec: S'$ only if S is more specific than each singleton $\{b\}$ for each $b \in S'$. This is true if we can find an $a \in S$ such that a and b are ordered by our partial ordering on atomic annotations.

Lemma 3.1. The relation $\prec:$ as defined above is a partial order on L.

Proof. Reflexivity is immediate since $S \subseteq S$. Next, we prove transitivity, i.e., that if $S_1 \prec: S_2$ and $S_2 \prec: S_3$ then $S_1 \prec: S_3$. There are four cases:

(i) Suppose both $S_2 \subseteq S_1$ and $S_3 \subseteq S_2$, then we are done immediately since reverse set inclusion is a partial order.

(ii) Suppose $S_2 \subseteq S_1$ and $\forall c \in S_3, \exists b \in S_2$ such that $b \prec:_{\mathcal{A}} c$. Then since $S_2 \subseteq S_1$ we know that each $b \in S_2$ is also in S_1 and so we are done.

(iii) Suppose $S_3 \subseteq S_2$ and $\forall b \in S_2, \exists a \in S_1$ such that $a \prec:_{\mathcal{A}} b$. Then since all $b \in S_2$ are in S_3 and there are no other elements of S_3, we have that $\forall b \in S_3, \exists a \in S_1$ such that $a \prec:_{\mathcal{A}} b$ and we are done.

(iv) Suppose $\forall b \in S_2, \exists a \in S_1$ such that $a \prec:_{\mathcal{A}} b$ and $\forall c \in S_3, \exists b \in S_2$ such that $b \prec:_{\mathcal{A}} c$. Then, for all $c \in S_3$, we can chose some b in S_2 with $b \prec:_{\mathcal{A}} c$, and for that b, there is some $a \in S_1$ such that $a \prec:_{\mathcal{A}} b$. Since $\prec:_{\mathcal{A}}$ is a partial order, we have $a \prec:_{\mathcal{A}} c$ and are done.

Finally, we prove antisymmetry, i.e., that whenever $S_1 \prec: S_2$ and $S_2 \prec: S_1$ we must have $S_1 = S_2$. There are three cases:

(i) Suppose both $S_2 \subseteq S_1$ and $S_1 \subseteq S_2$, then we are done immediately since reverse set inclusion is a partial order.

(ii) Suppose $\forall b \in S_2, \exists a \in S_1$ such that $a \prec:_{\mathcal{A}} b$ and $\forall a \in S_1, \exists b' \in S_2$ such that $b' \prec:_{\mathcal{A}} a$. Chaining these definitions, we get that for all $b \in S_2$ $b' \prec:_{\mathcal{A}} a \prec:_{\mathcal{A}} b$ for some $b' \in S_2$ and $a \in S_1$. But since $S_2 \in L$, $b = b'$ and then since $\prec:_{\mathcal{A}}$ is a partial order, $a = b$. Therefore $S_1 \subseteq S_2$. The other inclusion is identical, and so we have $S_1 = S_2$.

(iii) Finally, suppose $S_1 \subseteq S_2$ and $\forall b \in S_2, \exists a \in S_1$ such that $a \prec:_{\mathcal{A}} b$. Then since $S_1 \subseteq S_2$, we know that the $a \in S_1$ with $a \prec:_{\mathcal{A}} b$ is also in S_2, but since $S_2 \in L$ we must have $a = b$. But then $S_2 \subseteq S_1$ and we are done since reverse set inclusion is a partial order.

□

The greatest lower bound, $\sqcap$. The greatest lower bound operator requires similar modification. We define $\sqcap$ for $(L, \prec:)$ as follows:

$$S_1 \sqcap S_2 := (S_1 \cup S_2) \setminus \{b \mid \{a, b\} \subseteq S_1 \cup S_2 \wedge a \prec:_{\mathcal{A}} b\}.$$

Intuitively, we compose Security Annotations, constructing a more expressive annotation, and prune those atomic annotations which are represented by more expressive subannotations. Considering again Figure 3.1, the greatest lower bound of {`PubKey, CSRV`} and {`CryptKey, CSRV`} is {`PubKey, CSRV`} since `PubKey` $\prec:_{\mathcal{A}}$ `CryptKey`.

Lemma 3.2. The definition of $\sqcap$ above coincides with the greatest lower bound for $(L, \prec:)$.

Proof. For S_1, S_2, let $R = (S_1 \cup S_2) \setminus \{b \mid \{a, b\} \subseteq S_1 \cup S_2 \wedge a \prec:_{\mathcal{A}} b\}$. We first prove that $R \prec: S_1$, the case that $R \prec: S_2$ is similar. Suppose $b \in S_1$, then either $b \in R$ (since $R \subset S_1 \cup S_2$), or there is some a such that $a \prec:_{\mathcal{A}} b$ and

$a \in R$. In either case, there is some $c \in R$ (either a or B) with $c \prec:_{\mathcal{A}} b$. But this is precisely clause (ii) of the definition of $\prec:$ and therefore $\sqcap$ is a lower bound.

Second, we prove that for any $T \in L$ with $T \prec: S_1$ and $T \prec: S_2$ then $T \prec: R$. Suppose, for contradiction, that $T \not\prec: R$. Then $r \in R$ such that there is no $t \in T$ with $t \prec:_{\mathcal{A}} r$. Pick any $x \in R$ which is not in T (this must exist otherwise $T \prec: R$), since $R \subseteq S_1 \cup S_2$, suppose without loss of generality that $x \in S_1$. Given $x \notin T$, we know $S_1 \not\subseteq T$. But $T \prec: S_1$, so $\forall b \in S_1, \exists t \in T$ such that $t \prec:_{\mathcal{A}} b$. But then there is some $t \in T$ with $t \prec:_{\mathcal{A}} x$, which is a contradiction. □

The least upper bound operator, $\sqcup$. Our least upper bound is considerably more complex. Note that in this thesis, we will not regularly use the least upper bound, preferring instead an operation that allows to remove invalid Security Annotations (Section 3.3). Intuitively, the least upper bound of two annotations is the intersection of the two sets (as in a standard powerset lattice ordered by reverse inclusion). However, we must account for our atomic hierarchies: where two distinct annotations of the same hierarchy are present in this intersection, we should 'traverse up' the tree induced by $\prec:_{\mathcal{A}}$ to find an atomic annotation which has both as common children. For example, considering Figure 3.1, the least upper bound of the singletons $\{\texttt{SymKey}\}$ and $\{\texttt{PrivKey}\}$ is their immediate parent $\{\texttt{CryptKey}\}$. However, since our annotations do not contain multiple elements of the same atomic hierarchy we must reincorporate them to compute the least upper bound. For $S_1, S_2 \in L$, let

$$R_1 := S_1 \cup \{c \mid c \in \mathcal{A} \wedge \exists a \in S_1 : a \prec:_{\mathcal{A}} c\}$$
$$R_2 := S_2 \cup \{c \mid c \in \mathcal{A} \wedge \exists b \in S_2 : b \prec:_{\mathcal{A}} c\}.$$

Intuitively, these sets are the Security Annotations, along with all atomic parents of members the Security Annotation. For example, for $S_1 = \{\texttt{SymKey}\}$, this extended set is given by $R_1 = \{\texttt{SymKey}, \texttt{CryptKey}\}$. Clearly, in general, $R_1, R_2 \notin L$; however, we will correct this in our definition of $\sqcup$ for $(L, \prec:)$ as follows:

$$S_1 \sqcup S_2 := (R_1 \cap R_2) \setminus \{d \mid \{c, d\} \subseteq (R_1 \cap R_2) \wedge c \prec:_{\mathcal{A}} d\}.$$

Here, we take the intersection of these two extended sets, in the manner of the least upper bound for a powerset lattice order by reverse set inclusion, but then we must ensure that the resulting set is in L. To do this we take all pairs in this set and, whenever they are ordered hierarchically, keep only the atomic subannotation. We first prove that this is indeed a member of L.

Lemma 3.3. $S_1 \sqcup S_2 \in L$.

Proof. Let $R = (R_1 \cap R_2) \setminus \{d \mid \{c, d\} \in (R_1 \cap R_2) \wedge c \prec:_{\mathcal{A}} d\}$ with R_1, R_2 as defined previously. Suppose otherwise, then $\exists a, b \in R : a \prec:_{\mathcal{A}} b$. Then $a, b \in R_1 \cap R_2$ but $a, b \notin \{d \mid \{c, d\} \in (R_1 \cap R_2) \wedge c \prec:_{\mathcal{A}} d\}$. But this a contradiction, since $\{a, b\} \in R_1 \cap R_2$ and $a \prec:_{\mathcal{A}} b$. □

Second, we prove that this is indeed the least upper bound.

Lemma 3.4. The definition of $\sqcup$ coincides with the least upper bound for $(L, \prec:)$.

Proof. Let $R = (R_1 \cap R_2) \setminus \{d \mid \{c, d\} \in (R_1 \cap R_2) \wedge c \prec:_{\mathcal{A}} d\}$ with R_1, R_2 as defined previously. We first show R is an upper bound. We show $S_1 \prec: R$, the case where $S_2 \prec: R$ is similar. It suffices to show that $\forall r \in R, \exists a \in S_1 : a \prec:_{\mathcal{A}} r$. Since by definition $R \prec: R_1 \cap R_2$, for all $r \in R$, $r \in R_1$. This means that $r \in S_1$ or $r \in \{c \mid c \in \mathcal{A} \wedge \exists a \in S_1 : a \prec:_{\mathcal{A}} c\}$. In either case, there is some $a \in S_1$ with $a \prec:_{\mathcal{A}} r$, but then we are done.

Second, we show R is the least upper bound. We want to show that if $T \in L$ and $S_1 \prec: T, S_2 \prec: T$ then $R \prec: T$. Since $S_1 \prec: T$, pick any $t \in T$, then there is an $a \in S_1$ with $a \prec:_{\mathcal{A}} t$. Therefore, $t \in R_1$ by definition; similarly, $t \in R_2$, so $T \subseteq R_1 \cap R_2$. Since $t \in R_1 \cap R_2$, we know that either $t \in R$ or $t \in \{d \mid \{c, d\} \subseteq (R_1 \cap R_2) \wedge c \prec:_{\mathcal{A}} d\}$. But then in either case there must be some $r \in R$ with $r \prec:_{\mathcal{A}} t$, so $R \prec: T$ as required. □

A lattice for Security Annotations. Now that we have bounds operators on L we can define top and bottom elements, which follow naturally from the annotation lattice without atomic hierarchies. Since the $\emptyset \prec: S$ for any S, the top element remains $\emptyset$. The bottom element of this lattice, $\bot$, is not simply $\mathcal{A}$—from our definition of L, we must keep only the most specific annotation in each disjoint atomic hierarchy, i.e., if $a \prec:_{\mathcal{A}} b$, then $b \notin \bot$. We therefore obtain

$$\bot := \mathcal{A} \setminus \{b \mid \{a, b\} \subseteq \mathcal{A} \wedge a \prec:_{\mathcal{A}} b\}.$$

For example, considering the atomic hierarchy in Figure 3.1. In this case,

$$\bot = \{\texttt{PubKey}, \texttt{PrivKey}, \texttt{SymKey}, \texttt{CSRV}, \texttt{Plaintext}, \texttt{Ciphertext}\}.$$

Although a value may possess a Security Annotation with such seemingly inconsistent atomic annotations, such annotations may still make sense under certain contexts, allowing the value to satisfy either requirement (through subtyping).

Putting the results of this section together we obtain a lattice of Security Annotations. As before, the composition of annotations, $*$, coincides with the greatest lower bound of L.

Theorem 3.5. The tuple $(L, \prec:, \sqcap, \sqcup, \emptyset, \bot)$ is a lattice.

3.3 The cut Operator

In this section, we describe an additional operator on Security Annotations, `cut`, which allows the removal of specified Security Annotations without removing superannotations. Recall that we use the composition operator $*$ to simplify presentation and that $a * b$ is syntactic sugar for $\{a\} \sqcap \{b\}$.

The `cut` operator is motivated by the situation where as a result of an API call, a security property which previously held is no longer valid. For example, after a value is decrypted, it can no longer be considered ciphertext. We therefore require an operator which allows us to discard annotations from values or terms. Unfortunately this operator does not coincide with the least upper bound, since none of the underlying discarded security properties should remain valid. For example, consider the lattice in Figure 3.1. The result of `cut(PrivKey * CSRV,PrivKey)` is `CryptKey * CSRV`: the most specific annotation such that `PrivKey` is no longer valid. However, there is one key subtlety here: when pruning a Security Annotation, any superannotations of this annotation should remain valid, e.g., `CryptKey` should remain valid: it has not been explicitly discarded. For the general case, with arbitrary annotations, we ensure that none of the atomic annotations discarded are subannotations of the resulting Security Annotation; this is because the intuition here is that we are arguing none of the discarded properties remain valid.

Recall that we assume $\mathcal{A}$ is finite; the finiteness of this lattice allows us to define `cut` as follows:

Definition 3.6 (The `cut` operator)**.** `cut`(S_1, S_2) is the annotation R such that[2]:

[2] This definition is adapted from the work of Mitchell et al. [65] in two ways: first, an additional clause clarifies the definition when S_2 is the empty set. Second, in case (ii), we clarify the condition when S_2 is non-singleton, and the clause stipulating minimality

(i) if $S_2 = \top$, then $\texttt{cut}(S_1, \top) = \top$;

(ii) otherwise, R is the annotation with $S_1 \prec: R$ and $R \not\prec: \{b\}$ for all $b \in S_2$ such that for any other T satisfying these properties, $R \prec: T$.

Armed with a formal definition of this we define an explicit procedure for this operator and demonstrate that it satisfies the definition. This illustrates that this operator both exists for any two annotations, and also that it is uniquely defined. To do this, we use a similar trick to the description of a least upper bound operator: our base Security Annotation, S_1, is enlarged with all parent atomic annotations, and the annotation we want to remove, S_2 is enlarged with all the atomic children. We then remove this extended S_2 from the extended S_1, and then add a correction to ensure that the result is a member of L.

Proposition 3.7. Let $S_1, S_2 \in L$ be Security Annotations, and $S_2 \neq \top$. Let:

$$R_1 := S_1 \cup \{c \mid c \in \mathcal{A} \wedge \exists a \in S_1 : a \prec:_{\mathcal{A}} c\}$$
$$R_2 := S_2 \cup \{d \mid d \in \mathcal{A} \wedge \exists b \in S_2 : d \prec:_{\mathcal{A}} b\}$$
$$R' := R_1 \setminus R_2$$

Then $R := R' \setminus \{f \mid \{e, f\} \subseteq R' \wedge e \prec:_{\mathcal{A}} f\}$ coincides with $\texttt{cut}(S_1, S_2)$.

Proof. We need to show (i) $S_1 \prec: R$, (ii) $R \not\prec: \{b\}$ for all $b \in S_2$ and (iii) if T is satisfies (i) and (ii), then $R \prec: T$.

The proof of (i) is straightforward by this construction: clearly, $R \subseteq R' \subseteq R_1$. But then for any $r \in R$, $r \in S_1$ or $\exists a \in S_1$ such that $a \prec:_{\mathcal{A}} r$. But this is precisely the definition of $S_1 \prec: R$.

To prove (ii), we need to prove that for each $b \in S_2$, then $\forall r \in R, r \not\prec:_{\mathcal{A}} b$. Pick any $b \in S_2$, and suppose for contradiction that there is some $r \in R$

has been modified to remove the requirement that S is distinct and $R \not\prec: S'$ since $\prec:$ is a partial order.

with $r \prec:_{\mathcal{A}} b$. Then either $r \in S_2$ or $r \in \{d \mid d \in \mathcal{A} \wedge \exists b \in S_2 : d \prec:_{\mathcal{A}} b\}$ by definition. But then $r \in R_2$, and so $r \notin R$ which is a contradiction.

To prove (iii), suppose otherwise. Then $\exists t \in T$ such that $\forall r \in R, r \not\prec:_{\mathcal{A}} t$. Since $S_1 \prec: T$, we know that either $t \in S_1$ or $\exists a \in S_1 : a \prec:_{\mathcal{A}} t$, which precisely means that $t \in R_1$. Then $T \subseteq R_1$. Consider $T \cap R_2$; if this is non-empty, then for some $t \in T \cap R_2$ there is some $b \in S_2$ with $t \prec:_{\mathcal{A}} b$. But then this contradicts $R \not\prec: \{b\}$. This intersection must therefore be empty, and we need only consider the reduction step from R' to R in a similar manner to Lemma 3.4. In particular, we therefore know that $T \subseteq R'$ and so either $t \in R$ or $t \in \{f \mid \{e, f\} \subseteq R' \wedge e \prec:_{\mathcal{A}} f\}$. But then in either case there must be some $r \in R$ with $r \prec:_{\mathcal{A}} t$, so $R \prec: T$ as required. □

The final property we prove is to demonstrate that `cut` respects the annotation hierarchy, which is necessary in order to demonstrate annotation safety for a lambda calculus with Security Annotations (Theorem 4.3).

Lemma 3.8. Let R, R' be Security Annotations such that $R \prec: R'$. Then $\texttt{cut}(R, S) \prec: \texttt{cut}(R', S)$.

Proof. By definition, $R' \prec: \texttt{cut}(R', S)$. By transitivity, we obtain $R \prec: \texttt{cut}(R', S)$. We also know by definition that $\texttt{cut}(R', S) \not\prec: S$. Applying the second clause of Definition 3.6, we must have $\texttt{cut}(R', S) \not\prec: \texttt{cut}(R, S)$ and $\texttt{cut}(R, S) \prec: \texttt{cut}(R', S)$ as required. □

3.4 Related Work

The description of security properties enforced by Security Annotations as discussed in this chapter draws on similar notions discussed in other works which address security properties of program terms [7, 54, 32, 53, 95]. In the manner of these, we do not formally define what it means for a

program property to be a security property, but rather construct machinery to allow for description of a wide variety of useful properties. For example, F7 [7] describes machinery for arbitrary refinement types and describes how to use these for specifying programs obeying security properties.

The design of a lattice of Security Annotations follows largely standard constructions [69]. The partial order on Security Annotations draws from a range of subtyping approaches. In particular, the structure of Security Annotations induced by composition are inspired in part by record typing [80]. For example, subtyping for records defines that records are more specific when they carry more fields. This mirrors the intuition that a Security Annotation is more precise when it describes more security properties.

Type qualifiers [34], as discussed in Section 2.1, provide inspiration for the design patterns of Security Annotations. In particular, the underlying construction of type qualifiers is that of a qualification lattice, similar to our induced lattice on Security Annotations. In both cases, this provides a natural mechanism for composition of additional type specifications. However, Section 3.3 describes operators that we use to allow reclassification of a term's Security Annotations, e.g., `cut`. In contrast, type qualifiers of a term cannot be reclassified, which has contributed to the observation that type qualifiers are ill-suited to untyped scripting languages [48].

A Lambda Calculus of Security Annotations

4

In this chapter we introduce a model for Security Annotations within the context of a small language. We use as our framework a simple lambda calculus and extend both types and values with Security Annotations. We first explore the design decisions behind Security Annotations (Section 4.1). We then describe the details of λ_{SA}: first, the syntax (Section 4.2) and runtime semantics (Section 4.3), then typing judgments for Security Annotations (Section 4.4). Next, we describe the manipulation of Security Annotations (Section 4.5) and describe safety guarantees for the Lambda Calculus (Section 4.6). Finally, we discuss related work (Section 4.7).

This chapter contains an extended version a paper presented at *PEPM 2018* for which the author of this thesis was first author [65]. In this thesis, we extend the discussion of underlying design decisions, and augment the language with the **`cpAnn`** operator. Finally, we provide full proofs of annotated type safety for λ_{SA}[1].

4.1 Design Decisions for λ_{SA}

In Section 3.1, Security Annotations were introduced as add-ons to the type system of a language representing security properties, which can be

[1] An accompanying reference implementation is available at `https://github.com/ltbinsbe/lambda-security-tags`.

manipulated during evaluation of the program. In this chapter we formalize this concept by designing a lambda calculus with Security Annotations. We describe, in this section, the core design decisions facilitating such a calculus and describe alternatives to the presented design, comparing their relative merits. In this chapter our broad notational choices and description of the calculus follow that of Pierce [80].

A traditional typed lambda calculus enforces types statically and provides guarantees that well-typed programs are indeed valid: they will evaluate to a single value. We want to mirror this notion: although our eventual target is dynamic languages, we wish to be able to comment on what it means for a program with Security Annotations to be valid. This gives us our central design goal: we want a similar notion to type safety to be provable for Security Annotations, i.e., if a program respects the Security Annotation contracts, then it is safe. We further wish to ensure that the process of adding Security Annotations to the calculus—and the means to manipulate them—does not break the existing safety properties of the language.

In order to leverage similar guarantees to those of type safety, we propose the static enforcement of Security Annotations within our extended lambda calculus. This ties into the idea that Security Annotations for JavaScript act as add-ons to the type system: attaching Security Annotations to the type of a term retains the central design paradigm outlined in Section 1.3. However, in adopting this approach, have three design challenges to resolve:

(i) How extensive should our language be so as to capture the subtleties of Security Annotations?

(ii) How can we provide the guarantees of static typing of Security Annotations without losing sight of the dynamically-typed target language?

(iii) Since Security Annotations are manipulated at runtime, how should these modifications be conveyed within the static system?

In answering challenge (i), we observe that the core intention of this chapter is to demonstrate the mechanisms of manipulation of Security Annotations themselves are safe. We therefore want to focus on those areas where the design significantly differs from that of standard simply-typed lambda calculus. Ultimately, there is therefore little value in adding additional standard values and terms to the language (e.g., integers), when their interactions with Security Annotations would be the same as that of other types of values (e.g., booleans). We therefore opt for a minimal calculus, which offers just booleans and lambda abstractions as values and conditional, application and let bindings as terms.

Ultimately, we answer challenges (ii) and (iii) together, noting that both concern themselves with what the shape of terms and types in the calculus should be. The shape of types with annotations in λ_{SA} is a straightforward decision—we introduce annotated types, A, which are a pair comprised of a pretype T and Security Annotation S, and write T<S>. However, the shape of values in the calculus is more complex, since we want program terms, such as t **as** S to allow for the runtime manipulation of Security Annotations and this must be accounted for in the static typing system. There are three possibilities for this:

(i) Maintaining an explicit memory between evaluation steps of the term, and retain the shape of values as in a standard lambda calculus.

(ii) Label all terms in the calculus, keeping the shape of values the same as in a standard lambda calculus.

(iii) Change the shape of values such that they retain a dynamic copy of their Security Annotation—each value becomes a pair of a prevalue and Security Annotation, written w<S>.

The first approach is similar to approaches such as hybrid type checking [33]. For our purposes, this approach has two disadvantages: first, it is more complex than the other two approaches, obscuring the core mechanisms behind Security Annotations. Second, it would not reflect the model of the target language, so the extra machinery would not provide a benefit in terms of constructing a safe framework to adapt to JavaScript.

The second approach has merit: the labeling of terms allows the description of the shape of evaluation and express all of Security Annotations within the static type system. The approach allows the typing of the same value at different program points differently: for example, the value `true` may have type `Bool<`S`>` at label 0 but type `Bool<`S'`>` at label 1. However, such an approach is ultimately not type-safe. The natural evaluation rule for the expression $[[\texttt{true}]^0 \ \texttt{as}\ S]^1$ is to evaluate to $[\texttt{true}]^1$. However, in order to type this, we need to introduce a rule of the form

$$\frac{\Gamma \vdash [[t]^{i_1} \ \texttt{as}\ S]^{i_2} : T\texttt{<}S\texttt{>}}{\Gamma \vdash [t]^{i_2} : T\texttt{<}S\texttt{>}}$$

which allows to 'look back' through the prior evaluation of the program. Unfortunately, such a rule allows any program to type with any Security Annotation S. Consider the program $[\texttt{true}]^1$; we can produce a typing derivation of this as `Bool<`S`>` for any S through the application rule above. As such, this labeled approach is not a suitable direction.

The final option, of values carrying their annotations at runtime, is a natural representation of the tag-typing approach of JavaScript. This offers significant advantages: first, static semantics mirror the dynamic semantics in a natural way (e.g., the static and dynamic semantics for the **as** binary operator in Figure 4.6). Second, this approach enables an easier transition to the dynamic typing approach of JavaScript—in essence, it allows us to 'read-off' the evaluation rules as the design pillars for Se-

curity Annotations. In this way, the static typing system becomes a proof framework for type safety, allowing us to verify the core concepts prior to a natural translation to semantics for JavaScript. A formal translation to JavaScript is described in detail in Chapter 5, which uses these core runtime semantics as a guideline.

4.2 Syntax of λ_{SA}

The syntax for the lambda calculus extended with Security Annotations, λ_{SA}, is given in Figures 4.1 and 4.2. The syntax differs from that of a standard lambda calculus in several ways. Firstly, values in this calculus comprise a prevalue, corresponding to the values of a traditional lambda calculus, and a security annotation S. This allows us to represent security properties on individual values in the runtime semantics. Annotations are manipulated via the **as**, **drop** and **cpAnn** keywords. The term **as** adds annotations, representing newly valid security properties, while **drop** removes the annotations, representing security properties that are no longer valid. **cpAnn** allows the copying of Security Annotations from one value to another.

Programs are prepended by a series of *annotation declarations*, of the form `SecAnn` and `SecAnn ... Extends ...`; these define the annotations available, inducing the lattice of annotations for the program. Annotation declarations define the lattice as follows: first, the program's annotation lattice is initialized with `Top`. Second, any annotation declarations of the form `SecAnn` α define a new atomic annotation in the lattice with the relationship $\alpha \prec:$ `Top` and no hierarchical relationship to any other Security Annotations. Next, we consider annotation declarations of the form `SecAnn` α_1 `Extends` α_2; here, we require that α_2 has been previously introduced. This defines a new Security Annotation α_1 with a hierarchical

P ::=		*programs*:
	D . . . D t	annotation declarations and term
t ::=		*terms*:
	x	variable
	v	annotated value
	λx : T<S>.t	abstraction
	t t	application
	if t **then** t **else** t	conditional
	let x = t **in** t	let binding
	t **as** S	annotation introduction
	t **drop** S	annotation removal
	cpAnn t_1 t_2	annotation copy
w ::=		*prevalues*:
	λx : T<S>.t	abstraction prevalue
	true	true prevalue
	false	false prevalue
v ::=		*annotated values*:
	w<S>	annotated prevalue

Figure 4.1: Syntax of λ_{SA}: programs, terms, values and prevalues.

relationship $a_1 \prec: a_2$. The combination of these annotation declarations, D, together with the composition operator, $*$, induce the full lattice of Security Annotations for the program, L_P. In the following sections we assume that for any given program P, the corresponding annotation lattice L_P has been constructed, and that any annotations in semantic judgments are well-defined in L_P.

D ::=		*annotation declarations*:
	SecAnn a	new annotation
	SecAnn a **extends** a	annotation inheritance
S ::=		*security annotations*:
	a	security annotation
	$S * S$	annotation composition
	`Top`	empty annotation
T ::=		*pretypes*:
	`Bool`	pretype of Booleans
	$A \to A$	pretype of functions
A ::=		*annotated types*:
	T<S>	annotated type
Γ ::=		*contexts*:
	$\varnothing$	empty context
	$\Gamma, x : T<S>$	term variable binding

Figure 4.2: Syntax of λ_{SA}: types, environments and Security Annotations.

4.3 Dynamics

We present the runtime semantics for the core of λ_{SA} in Figure 4.3. These broadly follow those of a standard simply-typed lambda calculus [80]; the key alteration is that we add Security Annotations and develop mechanisms to ensure that such annotations are sensibly propagated. As standard, $[x \mapsto v]\ t$ denotes the substitution of the value v in place of the free occurrences of variable x in term t; we denote small-step evaluation via $\to$.

We describe here only those evaluation rules which do not directly manipulate Security Annotations; this allows us to focus on how Security

Annotations affect the standard runtime of the lambda calculus (we discuss their manipulation in Section 4.5). The reduction rules [E-APP1] and [E-APP2] are unchanged from a standard simply-typed lambda calculus, as are the rules governing the `let` construct. This is simply because these rules cover the evaluation of terms down to values, and the substitution of values in place of variables; they do not concern themselves with Security Annotations or annotated types of values. The rule [E-APPABS] performs the runtime evaluation of functions by substituting the value supplied as an argument inside the body of the abstraction. The bound variable of the lambda abstraction is annotated with an annotated type which serves to allow us to assume any application of this abstraction is with a value of that annotated type. However, this rule does not make any comment on the compatibility of this annotation guard and the provided argument; this is dealt with in the static annotated type system described in Section 4.4. We do not add annotation enforcement here in order to retain symmetry with the base type system: in a standard simply-typed lambda calculus the type guard of the abstraction is not enforced at runtime, and since we consider annotations as add-ons to our type system, we make the decision to avoid additional runtime overhead in checking them dynamically as well as statically.

Finally, we discuss the evaluation judgments governing the `if` construct. In particular, to ensure Security Annotations are transparent to control flow, we appeal directly to prevalues in order to govern which branch should be taken in these judgments. This reflects the paradigm of Security Annotations that they provide an add-on to the existing type system but should not result in modifications to the program evaluation. We highlight the `if` construct specifically here since it serves a general design pattern for Security Annotations in any language: unless a construct is specifically designed to manipulate Security Annotations, the runtime semantics of the language should be transparent to them.

$$\text{[E-App1]}\quad \frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

$$\text{[E-App2]}\quad \frac{t_2 \to t_2'}{t_1\ t_2 \to t_1\ t_2'}$$

$$\text{[E-AppAbs]}\quad (\lambda x : T\langle S\rangle.t)\langle S'\rangle\ v \to [x \mapsto v]\ t$$

$$\text{[E-LetV]}\quad \texttt{let}\ x = v\ \texttt{in}\ t \to [x \mapsto v]\ t$$

$$\text{[E-Let]}\quad \frac{t_1 \to t_1'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \to \texttt{let}\ x = t_1'\ \texttt{in}\ t_2}$$

$$\text{[E-IfTrue]}\quad \texttt{if true}\langle S\rangle\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \to t_2$$

$$\text{[E-IfFalse]}\quad \texttt{if false}\langle S\rangle\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \to t_3$$

$$\text{[E-If]}\quad \frac{t_1 \to t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \to \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}$$

Figure 4.3: Runtime semantics for λ_{SA}.

4.4 Statics

Figures 4.4 and 4.5 present inference rules defining the static type system for λ_{SA}. Following standard notation, $\Gamma \vdash t : \mathcal{A}$ means that under the typing environment Γ, the term t is of annotated type $\mathcal{A}$, which we make explicit in this presentation as base type T and Security Annotation S. We describe first a notion of subtyping in this calculus based on the lattice of Security Annotations (Section 4.4.1) and then discuss the remainder of the static semantics of the calculus (Section 4.4.2).

4.4.1 A Subtyping Relation

Recall that in Chapter 3, we embedded the hierarchy of Security Annotations into a lattice via a partial ordering based on reverse set inclusion.

$$\text{[T-SUB]}\quad \frac{\Gamma \vdash t : A_1 \qquad A_1 <: A_2}{\Gamma \vdash t : A_2}$$

$$\text{[S-SUBANN]}\quad \frac{S_1 \prec: S_2}{T\langle S_1\rangle <: T\langle S_2\rangle}$$

$$\text{[S-ARROW]}\quad \frac{B_1 <: A_1 \qquad A_2 <: B_2}{(A_1 \to A_2)\langle S\rangle <: (B_1 \to B_2)\langle S\rangle}$$

$$\text{[S-TRANS]}\quad \frac{A_1 <: A_2 \qquad A_2 <: A_3}{A_1 <: A_3}$$

$$\text{[S-REFL]}\quad A <: A$$

Figure 4.4: Annotated subtyping rules for λ_{SA}.

We extend this notion of hierarchical Security Annotations to a subtyping relation for the annotated types of λ_{SA} in the natural way.

Definition 4.1 (Induced subtyping)**.** The rules in Figure 4.4 induce a direct subtyping relationship for annotated types from the lattice of Security Annotations. Whenever A is a *subtype* of B, we write $A <: B$.

These rules, in practice, are simply standard subtyping semantics enriched with annotations. The rule [S-SUBANN] describes the core concept of inducing a subtyping relationship from our lattice of annotations, by directly inheriting from this hierarchy whenever pretypes match. The rule [S-REFL] follows immediately immediately from the lattice of Security Annotations; we present this form here for clarity. It is worth noting transitivity does not follow immediately from the definition in the case of function types, hence our need for a separate semantic in [S-TRANS]. Note that this formulation is distinct from that presented in the work this chapter is based on [65]; the construction of an explicit subtyping judgment for annotated types considerably simplifies some of the necessary inversion lemmas for proving type safety in Section 4.6.

$$\text{[T-VAR]}\quad \frac{x : T\langle S\rangle \in \Gamma}{\Gamma \vdash x : T\langle S\rangle}$$

$$\text{[T-TRUE]}\quad \texttt{true}\langle S\rangle : \texttt{Bool}\langle S\rangle$$

$$\text{[T-FALSE]}\quad \texttt{false}\langle S\rangle : \texttt{Bool}\langle S\rangle$$

$$\text{[T-ABS]}\quad \frac{\Gamma, x : T_1\langle S_1\rangle \vdash t : T_2\langle S_2\rangle}{\Gamma \vdash (\lambda x : T_1\langle S_1\rangle.t)\langle S_3\rangle : (T_1\langle S_1\rangle \to T_2\langle S_2\rangle)\langle S_3\rangle}$$

$$\text{[T-APP]}\quad \frac{\Gamma \vdash t_1 : (T_1\langle S_1\rangle \to T_2\langle S_2\rangle)\langle S_3\rangle \qquad \Gamma \vdash t_2 : T_1\langle S_1\rangle}{\Gamma \vdash t_1\ t_2 : T_2\langle S_2\rangle}$$

$$\text{[T-LET]}\quad \frac{\Gamma \vdash t_1 : T_1\langle S_1\rangle \qquad \Gamma, x : T_1\langle S_1\rangle \vdash t_2 : T_2\langle S_2\rangle}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2\langle S_2\rangle}$$

$$\text{[T-IF]}\quad \frac{\Gamma \vdash t_1 : \texttt{Bool}\langle\rangle \qquad \Gamma \vdash t_2 : T_2\langle S_2\rangle \qquad \Gamma \vdash t_3 : T_3\langle S_3\rangle}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T_2\langle S_2 \sqcup S_3\rangle}$$

Figure 4.5: Static semantics for λ_{SA}.

4.4.2 Annotated typing rules for λ_{SA}

The static semantics for λ_{SA} are given in Figure 4.5; rules for typing the manipulation of Security Annotations are discussed in Section 4.5. In general, these static semantics are simple extensions of the standard typing rules for a simply-typed lambda calculus [80]; the typing of these judgments in view of Security Annotations often mirrors the dynamic semantics, since annotations are dynamically propagated and manipulated. For example, both the [T-TRUE] and [T-FALSE] rules reflect this: the underlying type of the value is simply the type of the prevalue (`Bool`), and the static Security Annotation is the same as the annotation directly associated to the prevalue in the term. The rules for variables, [T-VAR] and [T-LET], are

the natural extensions of their equivalent rules in the standard calculus; we simply add annotations to these rules. For example, in [T-LET], we enforce that the Security Annotations of the term bound to the variable matches the assumed Security Annotation of the variable in the typing of the second term, t_2.

The rules for typing function abstractions demonstrate the point at which Security Annotations are directly enforced: [T-ABS] is a small extension, describing the annotated type of an abstraction. Note that since abstractions are values within this calculus they are associated to Security Annotations. The arrow type of the abstraction is between two annotated types as one would expect, mirroring the construction of an abstraction type from the standard calculus. The semantic [T-APP] governs the typing of the application of a value to an abstraction, it is here that Security Annotations are enforced, reflecting our intention that Security Annotations are enforced on entry to functions (specifically, API calls). This rule ensures that the supplied argument to the function meets the annotated type-guard of the variable declared in the abstraction. Note that the annotation associated to the abstraction S_3 is not enforced, since this represents security properties of the abstraction value itself. Combined with the subsumption judgments of Figure 4.4, this ensures that Security Annotation contracts at entry of functions are enforced.

The typing of the **`if`** construct merits discussion: we want to retain the maximal amount of information possible without re-working and over-complicating our annotation lattice (e.g., via sum-like annotations). It is intuitive to consider the most expressive annotation valid along both branches (i.e. the $\sqcup$ operator); consider the trivial example in Listing 4.1. Since both `C` and `D` are not valid along the **`else`** branch, they cannot be subannotations of the resulting annotation. We therefore type this as `Bool <D*B ⊔ A*B> = Bool<A*B>`. The static Security Annotation will be over-approximate as a result: the result of evaluation will possess the annota-

```
1 SecAnn A
2 SecAnn B
3 SecAnn C extends A
4 SecAnn D extends C
5
6 if true<> then true<D*B> else false<A*B>
```

Listing 4.1: Typing the **if** construct.

tion of precisely the taken branch, which is a subannotation of the static Security Annotation.

We further remark on the contrast between our semantics for **if** and the judgments in traditional security type systems. In these systems, the resulting security label of the term is affected by the security label of the guard term; this design is primarily motivated by implicit flows from high to low security terms (see Section 2.1). In contrast, we do not make any comment on the Security Annotations of the guard term's effect on the overall type; one could extend this calculus to enable such propagation, however this is unnecessary for cryptographic properties. We explore the propagation of Security Annotations in JavaScript programs in Chapter 7, and design extensions allowing for the handling of these use cases.

4.5 Manipulating Security Annotations in λ_{SA}

We describe the mechanisms in λ_{SA} by which we manipulate Security Annotations based on the validity of security properties of terms in the program. We first describe the process of adding Security Annotations (explicit upcasting of terms), then removing Security Annotations (downcasting) and finally discuss the copying of Security Annotations from a term to another.

$$\text{[E-AsV]} \quad w\langle S_1\rangle \ \mathbf{as}\ S_2 \rightarrow w\langle S_1 * S_2\rangle$$

$$\text{[E-As]} \quad \frac{t \rightarrow t'}{t\ \mathbf{as}\ S \rightarrow t'\ \mathbf{as}\ S}$$

$$\text{[T-As]} \quad \frac{\Gamma \vdash t : T\langle S_1\rangle}{\Gamma \vdash t\ \mathbf{as}\ S_2 : T\langle S_1 * S_2\rangle}$$

Figure 4.6: Adding Security Annotations within λ_{SA}.

Adding Security Annotations via the `as` operator. Figure 4.6 provides both the runtime and static semantics for the adding of Security Annotations to terms in λ_{SA} via the **as** binary operator. The first argument to **as** is the term to coerce, and the second a Security Annotation which should be valid on the term. For example, in the term `true`$\langle S_1\rangle$ **as** S_2, we upcast the term `true`$\langle S_1\rangle$ to also satisfy the annotation S_2. The annotation S_1 remains valid; therefore the resulting annotated type of the term must have the base type of the first operand and annotation the composition of both the annotation of the first operand and also the second operand: this is precisely the composition of the two annotations, $S_1 * S_2$. This intuition governs the typing judgment for this upcasting, [T-As], and also the runtime judgment for when the left operand is a value, [E-AsV]. Finally, noting that the right operand is a Security Annotation, we need [E-As], which evaluates the left operand to a value. We do this before evaluation of the **as** operator, in the same manner as [E-If]. For example, to evaluate the program `true`$\langle\rangle$ **as** S_1 **as** S_2, the first step of evaluation results in `true`$\langle S_1\rangle$ **as** S_2, and then `true`$\langle S_1 * S_2\rangle$.

Removing Security Annotations via the `drop` operator In the same manner as upcasting Security Annotations, we describe the mechanism for downcasting Security Annotations via the **drop** operator. The runtime

$$\text{[E-DROPV]} \quad w\texttt{<}S_1\texttt{>}\ \textbf{drop}\ S_2 \rightarrow w\texttt{<cut(}S_1, S_2\texttt{)>}$$

$$\text{[E-DROP]} \quad \frac{t \rightarrow t'}{t\ \textbf{drop}\ S \rightarrow t'\ \textbf{drop}\ S}$$

$$\text{[T-DROP]} \quad \frac{\Gamma \vdash t : T\texttt{<}S_1\texttt{>}}{\Gamma \vdash t\ \textbf{drop}\ S_2 : T\texttt{<cut(}S_1, S_2\texttt{)>}}$$

Figure 4.7: Removing Security Annotations within λ_{SA}.

```
1 SecAnn A
2 SecAnn B
3 SecAnn C extends A
4 SecAnn D extends C
5
6 true<D*B> drop C*B
```

Listing 4.2: Typing the **drop** operator.

and static semantics for this operator are given in Figure 4.7. As with the **as** operator, the binary **drop** operator takes two arguments, the left operand a term the resulting term will be a downcasted copy of, and the right operand the Security Annotation which should not be valid. For example, consider the program in Listing 4.2: it is clear that this program will evaluate to a downcasted copy of the value `true<D*B>` comprised of the prevalue `true` along with an annotation corresponding to the result of discarding `C*B` from the more specific annotation `D*B`. This annotation is precisely the result of applying the `cut` operation described in Definition 3.6: `cut(D*B, C*B)` is precisely `A`. This operator ensures that Security Annotations which are superannotations of those discarded annotations remain valid: they have not been discarded. The typing rule [T-DROP] and the evaluation rule [E-DROPV] both utilize this operator for that purpose. As in the case of the **as** operator, we also need [E-DROP] to ensure

$$\text{[E-CPANN1]}\quad \frac{t_1 \rightarrow t_1'}{\textbf{cpAnn}\ t_1\ t_2 \rightarrow \textbf{cpAnn}\ t_1'\ t_2}$$

$$\text{[E-CPANN2]}\quad \frac{t_2 \rightarrow t_2'}{\textbf{cpAnn}\ v_1\ t_2 \rightarrow \textbf{cpAnn}\ v_1\ t_2'}$$

$$\text{[E-CPANN]}\quad \textbf{cpAnn}\ w_1\langle S_1\rangle\ w_2\langle S_2\rangle \rightarrow w_2\langle S_1 * S_2\rangle$$

$$\text{[T-CPANN]}\quad \frac{\Gamma \vdash t_1 : T_1\langle S_1\rangle \qquad \Gamma \vdash t_2 : T_2\langle S_2\rangle}{\Gamma \vdash \textbf{cpAnn}\ t_1\ t_2 : T_2\langle S_1 * S_2\rangle}$$

Figure 4.8: Copying Security Annotations within λ_{SA}.

```
1 SecAnn Ciphertext
2 SecAnn Plaintext
3 SecAnn CryptKey
4 SecAnn Password
5
6 (let encrypt = λ x : Bool<>. λ y : Bool<CryptKey>
7   (((cpAnn x (internal.encrypt x y)) drop Plaintext) as
      Ciphertext)
8 in encrypt true<Password*Plaintext> false<CryptKey>
```

Listing 4.3: Typing the **cpAnn** operator.

the left operand is fully evaluated before evaluating the **drop** operator.

Handling polymorphism via the `cpAnn` operator Security properties valid as postconditions of functions are not necessarily independent of the properties of the arguments. We refer to such cases as functions which are *polymorphic* in their Security Annotations. For example, the JavaScript function `window.btoa` which takes a string and returns a Base64 encoded string: clearly, any security properties of the argument should be valid on the return value. Therefore, if value passed to `window.btoa` is $w\langle S\rangle$, we would expect the annotation of the result to be `cut(S, encoding)*Base64`,

because any previous description of the encoding is no longer valid, but all other security properties remain so. To achieve this without the need for annotation variables, we introduce an operator, **`cpAnn`**, which allows for the copying of annotations from one term to another. The runtime and static semantics for this operator are given in Figure 4.8.

Listing 4.3 demonstrates the usage of this operator through example. This example describes a simple encryption in the lambda calculus, where the call to `internal.encrypt` refers to some built-in function which encrypts the first argument with the second argument as key. In this case, the security property that the plaintext was a password is still valid on the argument to be encrypted afterwards—although the notion that the argument might be plaintext is not, and so we copy all annotation across to the result, and then use **`drop`** and **`as`** to modify the resulting annotation as required.

This example demonstrates the runtime semantics for this operator: first, each operand must be evaluated to a value (via the [E-CPANN1] and [E-CPANN2] judgments in Figure 4.8). Second, once the two operands are fully evaluated, then we can evaluate the operator itself: the result is the prevalue of the second operand, along with the accompanying annotation of the first operand. Much like the **`as`** operator, this does not invalidate previous security properties, so we use composition to ensure previous security annotations remain valid (the [E-CPANN] judgment in Figure 4.8). The operator is typed similarly: the annotated type of this expression has base type matching the base type of the second operand, and Security Annotation the composition of the annotations of the two operands.

4.6 Annotated Type Safety for λ_{SA}

This section concerns itself with adapting the classical notion of type safety to λ_{SA}. Type safety in this calculus ensures that the flexible system of Security Annotations introduced does not result in the ability of invalid programs to pass type checks. We formulate a notion of type safety based on the traditional notions of (i) *progress* and (ii) *preservation* [80], which we adapt to incorporate Security Annotations:

(i) A well-typed term is either a value, or can take a step of evaluation according to the rules in Figure 4.3 and Figures 4.6-4.8.

(ii) If a well-typed term takes a step of evaluation, then the resulting term is also well-typed, i.e. if t : T<S> and $t \rightarrow t'$, then t' : T<S>.

The non-standard notion of (ii) reflects the typing of `if`, however it remains impossible for programs to type as valid when they do not possess the necessary security properties. First, we formally define what it means for a term to be well-typed in the context of λ_{SA}.

Definition 4.2 (closed well-typed term)**.** A term is *well-typed* if there exists some pretype T and Security Annotation S such that $\Gamma \vdash$ t : T<S> for some Γ. Further, a well-typed term t is *closed* if there are no free variables within t, i.e., when Γ is $\varnothing$.

In the rest of this section, we first construct a proof of preservation (Section 4.6.1) and then of progress (Section 4.6.2). Finally, we provide a discussion of this guarantee in the context of security properties (Section 4.6.3).

4.6.1 Preservation for λ_{SA}

The statement of preservation for λ_{SA} encodes that after a step of evaluation, a well-typed term is still well-typed. We formalize this through the following statement.

Theorem 4.3 (Preservation)**.** If $\Gamma \vdash t : T\langle S\rangle$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T\langle S\rangle$.

To prove this theorem, we build a series of technical results, which adapt standard results to account for Security Annotations [80].

Lemma 4.4 (Inversion of the Subtyping Relation)**.** If $A <: (B_1 \rightarrow B_2)\langle S\rangle$ then A has the form $(A_1 \rightarrow A_2)\langle S'\rangle$ with $B_1 <: A_1$, $A_2 <: B_2$ and $S' \prec: S$.

Proof. By induction on subtyping derivations derived from the rules in Figure 4.4. There are four possibilities for the final rule applied in the subtyping derivation. First, if the final rule is [S-REFL], this is trivial since A is simply $(B_1 \rightarrow B_2)\langle S\rangle$ and we are done. In the case that the final rule applied is [S-ARROW], then we are done simply noting that A already has the desired form. Third, if the final rule applied is [S-SUBANN] then A already has the desired form, since $T = (B_1 \rightarrow B_2)$.

Finally, if the final rule applied is [S-TRANS] then we know that there exists some C such that $A <: C$ and $C <: (B_1 \rightarrow B_2)\langle S\rangle$. But then by induction, we know that there must be some C_1, C_2, S'' with $C = (C_1 \rightarrow C_2)\langle S''\rangle$ and $B_1 <: C_1$, $C_2 <: B_2$ and $S'' \prec: S$. We therefore know that $A <: (C_1 \rightarrow C_2)\langle S''\rangle$ by assumption, and then by induction we obtain an A_1, A_2, S' with $C_1 <: A_1, A_2 <: C_2$ and $S' \prec: S''$. By [S-TRANS] we obtain $B_1 <: A_1$ and $A_2 <: B_2$, and by transitivity of elements of the annotation lattice we obtain $S' \prec: S$ and we are done. □

Armed with an inversion of the subtyping relation we now invert the typing relation, which allows us, for a given well-typed term t, to examine the annotated types of subterms of t.

Lemma 4.5 (Inversion of the Typing Relation)**.** We can determine the types of terms via the following:

(i) If $\Gamma \vdash x : T\langle S\rangle$ then $x : T\langle S\rangle \in \Gamma$.

(ii) If $\mathbf{true}\langle S\rangle : A$ then $A = \mathtt{Bool}\langle S\rangle$.

(iii) If $\mathbf{false}\langle S\rangle : A$ then $A = \mathtt{Bool}\langle S\rangle$.

(iv) If $\Gamma \vdash (\lambda x : A_1.t_1)\langle S\rangle : (B_1 \to B_2)\langle S\rangle$, then $B_1 <: A_1$ and $\Gamma, x : A_1 \vdash t_1 : B_2$.

(v) If $\Gamma \vdash t_1\ t_2 : A$ then there are some A_1, S such that $\Gamma \vdash t_1 : (A_1 \to A)\langle S\rangle$ and $\Gamma \vdash t_2 : A_1$.

(vi) If $\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : T\langle S\rangle$, then $\Gamma \vdash t_1 : \mathtt{Bool}\langle S_1\rangle$, $\Gamma \vdash t_2 : T\langle S_2\rangle$ and $\Gamma \vdash t_3 : T\langle S_3\rangle$ for some S_1, S_2 and S_3 with $S_2 \sqcup S_3 = S$.

(vii) If $\Gamma \vdash t\ \mathbf{as}\ S_2 : T\langle S\rangle$ then $\Gamma \vdash t : T\langle S_1\rangle$ with for some S_1 with $S = S_1 * S_2$.

(viii) If $\Gamma \vdash t\ \mathbf{drop}\ S_2 : T\langle S\rangle$ then $\Gamma \vdash t : T\langle S_1\rangle$ for some S_1 with $S = \mathtt{cut}(S_1, S_2)$.

(ix) If $\Gamma \vdash \mathbf{cpAnn}\ t_1\ t_2 : T\langle S\rangle$ then $\Gamma \vdash t_1 : T\langle S_1\rangle$ and $\Gamma \vdash t_2 : T\langle S_2\rangle$ with for some S_1, S_2 with $S = S_1 * S_2$.

(x) If $\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : A$ then there exists some B with $\Gamma \vdash t_1 : B$ and $\Gamma, x : B \vdash t_2 : A$.

Proof. The vast majority of these cases are immediate from the typing judgments in Figure 4.5. However, case (iv) merits special attention for a single case of induction on typing derivations, when the final rule of the typing derivation is [T-SUB]. We let $t = (\lambda x : A_1.t_1)\langle S\rangle$, then we know $\Gamma \vdash t : (B_1 \to B_2)\langle S\rangle$ and since the last rule was [T-SUB], there must be some $C <: (B_1 \to B_2)\langle S\rangle$ with $\Gamma \vdash t : C$. Applying Lemma 4.4, we obtain that C has the form $(C_1 \to C_2)\langle S''\rangle$ with $B_1 <: C_1$, $A_2 <: C_2$ and $S'' \prec: S$. Applying induction and using [T-SUB] and [S-TRANS], we obtain the desired result. □

We also need a technical lemma covering the permuting and weakening of the typing context, allowing us to, when necessary, insert additional well typed variables into the context or permute the current context without altering the typing of a term.

Lemma 4.6 (Permutation and Weakening)**.** Let $\Gamma \vdash t : A$. Then:

(i) Whenever Δ is a permuation of Γ then $\Delta \vdash t : A$.

(ii) Whenever $x \notin \mathrm{Dom}(\Gamma)$ then $\Gamma, x : A' \vdash t : A$.

Proof. Both statements are trivial inductions on the typing judgments. □

The final technical result—which uses Lemma 4.5—ensures that when substituting variables for appropriately-typed terms into terms we do not break well-typedness.

Proposition 4.7 (Preservation of annotated types under substitution)**.** If $\Gamma, x : A' \vdash t : A$ and $\Gamma \vdash t' : A'$ then $\Gamma \vdash [x \mapsto t']t : A$.

Proof. We proceed by induction on the typing derivation of $\Gamma, x : A' \vdash t : A$, inspecting the final typing rule in the derivation.

[T-TRUE] Let $t = \texttt{true<}S\texttt{>}$; since there are no variables in t, then the annotated typing after substitution is immediate. The case [T-FALSE] is identical.

[T-IF] Let $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$; we know that:

- For some S_1, $\Gamma, x : A' \vdash t_1 : \texttt{Bool<}S_1\texttt{>}$, and
- For $i = 2, 3$, we have $\Gamma, x : A' \vdash t_i : T\texttt{<}S_i\texttt{>}$ with $A = T\texttt{<}S_2 \sqcup S_3\texttt{>}$.

By induction on each term, we obtain:

- $\Gamma \vdash [x \mapsto t']t_1 : \texttt{true<}S_1\texttt{>}$, and
- For $i = 2, 3$, $\Gamma \vdash [x \mapsto t']t_i : T\texttt{<}S_i\texttt{>}$.

Applying [T-IF], we can type $\Gamma \vdash [x \mapsto t']t : T\langle S_2 \sqcup S_3\rangle$ as required.

[T-AS] Let $t = t_1$ **as** S_2, then we have:

- $\Gamma, x : A' \vdash t : T\langle S_1 * S_2\rangle$ and
- $\Gamma, x : A' \vdash t_1 : T\langle S_1\rangle$ for some S_1.

By induction on t_1, we obtain $\Gamma \vdash [x \mapsto t']t_1 : T\langle S_1\rangle$, and then applying [T-AS] we are done.

[T-DROP] Let $t = t_1$ **drop** S_2, then we have:

- $\Gamma, x : A' \vdash t : T\langle \mathtt{cut}(S_1, S_2)\rangle$ and
- $\Gamma, x : A' \vdash t_1 : T\langle S_1\rangle$ for some S_1.

By induction on t_1, we obtain $\Gamma \vdash [x \mapsto t']t_1 : T\langle S_1\rangle$, and then applying [T-DROP] we are done.

[T-CPANN] Let $t =$ **cpAnn** $t_1\ t_2$, then we have, for some S_1, S_2:

- $\Gamma, x : A' \vdash t : T\langle S_1 * S_2\rangle$,
- $\Gamma, x : A' \vdash t_1 : T\langle S_1\rangle$, and
- $\Gamma, x : A' \vdash t_1 : T\langle S_2\rangle$.

By induction, we obtain:

- $\Gamma \vdash [x \mapsto t']t_1 : T\langle S_1\rangle$, and
- $\Gamma \vdash [x \mapsto t']t_2 : T\langle S_2\rangle$.

and then applying [T-CPANN] we are done.

[T-VAR] Let $t = z$, with $z : A \in (\Gamma, x : A')$. We have two subcases: in the first, $z = x$, and so $[x \mapsto t']z = t'$. Since the typing of this was an assumption, we are done. In the second case, z is a distinct variable and then $[x \mapsto t']z = z$ and again, we are done by our original assumption.

[T-ABS] Let $t = (\lambda y : B.t_1)\langle S\rangle$, and $A = (B \to B')\langle S\rangle$. We know that $\Gamma, x : A', y : B \vdash t_1 : B'$. We can assume $x \neq y$ and that y is not a free variable of x. Via weakening and permutation (Lemma 4.6), we can state that $\Gamma, y : B \vdash t' : A'$. By induction, we obtain $\Gamma, y : B \vdash [x \mapsto t']\, t_1 : B'$. Finally, applying [T-ABS] we get the required result.

[T-APP] Let $t = t_1\ t_2$, then we know that for some S and B that:

- $\Gamma, x : A' \vdash t_1 : (B \to A)\langle S\rangle$ and
- $\Gamma, x : A' \vdash t_2 : B$.

Applying induction we know that:

- $\Gamma \vdash [x \mapsto t']\, t_1 : (B \to A)\langle S\rangle$ and
- $\Gamma \vdash [x \mapsto t']\, t_2 : B$.

Applying [T-APP] to these two derivations, we are done.

[T-SUB] By assumption, we know that $\Gamma, x : A' \vdash t_1 : B$ with $B <: A$, and that $\Gamma \vdash t' : A'$. Applying induction, we know $\Gamma \vdash [x \mapsto t']\, t : B$. Using $A' <: A$, we can apply [T-SUB], and we are done.

□

Armed with the ability to substitute variables for terms, we are now in a position to prove Preservation.

Proof of Theorem 4.3. Again, we prove this by induction on the typing derivation of $\Gamma \vdash t : T\langle S\rangle$.

[T-TRUE], [T-FALSE], [T-ABS] Since t is already a value in each of these cases, there is nothing to prove.

[T-VAR] Since there are no evaluation rules for variables, there is nothing to prove here.

[T-IF] We have $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$ and know that $t : T\langle S\rangle$. Inverting the typing relation, we have

- $\Gamma \vdash t_1 : \texttt{Bool}\langle S_1\rangle$
- $\Gamma \vdash t_2 : T\langle S_2\rangle$
- $\Gamma \vdash t_3 : T\langle S_3\rangle$,

for some S_1, S_2, S_3 with $S = S_2 \sqcup S_3$. There are three possible cases.

First, if $t_1 = \texttt{true}\langle S_1\rangle$, then $t \to t_2$ via [E-IFTRUE]. Since$\Gamma \vdash t_2 : T\langle S_2\rangle$ and $S_2 \prec: S_2 \sqcup S_3 = S$, we are done.

Second, whenever $t_2 = \texttt{false}\langle S_1\rangle$, we apply [E-IFFALSE], and via the same reasoning we are done.

Third, if $t_1 \to t_1'$, then applying [E-IF], we get $t' = \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3$. By induction, $t_1' : \texttt{Bool}\langle S_1'\rangle$, and $S_1' \prec: S_1$. Applying [T-IF], we obtain $t' : T\langle S\rangle$ and we are done.

[T-AS] We have $t = t_1\ \texttt{as}\ S_2$, and via the inversion of the typing relation we know that $\Gamma \vdash t_1 : T\langle S_1\rangle$ for some S_1 with $S = S_1 * S_2$. We have two cases based on the two possible evaluation rules.

First, if $t_1 = w\langle S_1\rangle$ is a value, then we apply [E-ASV], and then $t \to t' = w\langle S_1 * S_2\rangle$. Depending on the prevalue w, we apply either [T-TRUE], [T-FALSE] or [T-ABS] and we are done.

In the second case, we suppose that there is some t_1' with $t_1 \to t_1'$, and so we can apply [E-AS]. By induction, we know that $\Gamma \vdash t_1' : T\langle S_1'\rangle$, for some S_1' with $S_1' \prec: S_1$. Via [T-AS] we get $\Gamma \vdash t_1'\ \texttt{as}\ S_2 : T\langle S_1' * S_2\rangle$. Applying the partial ordering on annotations, we know $(S_1' * S_2) \prec: S$ and so by [T-SUB] we are done.

[T-DROP] The case for [T-DROP] is almost identical to [T-AS], but uses the property that whenever $S_1' \prec: S_1$, then for any S_2, we must have $\texttt{cut}(S_1', S_2) \prec: \texttt{cut}(S_1, S_2)$ (per Lemma 3.8).

[T-CPANN] We have $t = \texttt{cpAnn}\ t_1\ t_2$ and by inverting the typing relationship we know that for some S_1, S_2 with $S = S_1 * S_2$ we have:

- $\Gamma \vdash t_1 : T\langle S_1\rangle$
- $\Gamma \vdash t_2 : T\langle S_2\rangle$

There are three possible cases.

First, if $t_1 \mapsto t_1'$ and [E-CPANN1] is applied, then by induction we know $\Gamma \vdash t_1 : T\langle S_1'\rangle$, where $S_1' \prec: S_1$. Applying [T-CPANN]. $\Gamma \vdash t' : T\langle S_1' * S_2\rangle$ and by the partial ordering on annotations we are done.

The second case, where [E-CPANN2] is applied, is almost identical.

Third, if both t_1 and t_2 are values, then we apply [E-CPANN]. Let $t_1 = w\langle S_1'\rangle$ and $t_2 = w\langle S_2'\rangle$, with $S_1' \prec: S_1$ and $\S_2' \prec: S_2$. Then $t \rightarrow w_2\langle S_1' * S_2'\rangle$. By our annotation partial ordering and—depending on the prevalue w_2—one of [T-TRUE], [T-FALSE] or [T-ABS] we are done.

[T-APP] We have $t = t_1\ t_2$, and we can type these two subterms as:

- $\Gamma \vdash t_1 : (B \rightarrow C)\langle S_1\rangle$
- $\Gamma \vdash t_2 : B$,

with $B = T_B\langle S_B\rangle$ and $C = T_C\langle S_C\rangle$. There are three distinct cases.

First, if [E-APP1] is applied, then by induction $t_1' : (B \rightarrow C)\langle S_2'\rangle$, with $S_1' \prec: S_1$. By applying [T-APP], we are done.

Second, if [E-APP2], is applied, then by induction we have $\Gamma \vdash t_2' : T_B\langle S_B'\rangle$ with $S_B' \prec: S_B$. By [T-SUB], we get $\Gamma \vdash t_2' : B$ and then by applying [T-APP], we are done.

Third, if the final application rule is [E-APPABS]. Then we need to prove that for $t' = [x \mapsto v]\ s : T_C\langle S_C'\rangle$ with $S_C' \prec: S_C$, given that:

- $t_1 = (\lambda x : B'.s)\langle S_1\rangle$

- $t_2 = v$.

From the inversion of the typing relation we know that $B <: B'$ and that $\Gamma, x : B' \vdash s : C$. Applying [T-SUB], we have $\Gamma \vdash v : B'$. Putting this together, along with the substitution lemma (Lemma 4.7), we obtain the desired result.

[T-LET] This case is similar to [T-APP].

[T-SUB] Inverting the typing relation for t, we have $\Gamma \vdash t : T\langle S' \rangle$ and $S' \prec: S$. Since $t \to t'$, by induction we know that $\Gamma \vdash t' : T\langle S' \rangle$ and immediately we are done.

□

4.6.2 Progress for λ_{SA}

The statement of progress for λ_{SA} intuitively describes that for any whenever a term can be typed, then it is either a value, or can take a single step of evaluation according to the runtime semantics described in Sections 4.3 and 4.5. We formalize this as follows:

Theorem 4.8 (Progress)**.** Let t be a closed, well-typed term. Then either t is a value or there exists some t' with $t \to t'$.

In order to prove this statement we need only add a *canonical forms* lemma to our existing machinery. This result describes what values look like based on the described annotated type.

Lemma 4.9 (Canonical forms)**.** Let v be a closed value with annotated type A, then:

(i) If A is `Bool`$\langle S \rangle$, then either v is `true`$\langle S' \rangle$, or `false`$\langle S' \rangle$ for $S' \prec: S$.

(ii) If A is $(T_1\langle S_1 \rangle \to T_2\langle S_2 \rangle)\langle S \rangle$, then v has the form $(\lambda x : T_1\langle S_1' \rangle . t_2)\langle S' \rangle$ with $S' \prec: S$ and $S_1 \prec: S_1'$.

Proof. The first of these statements is proved by observing that we assumed the pretype of v to be `Bool`, which means the prevalue must be either `true` or `false`. Since the assumed static Security Annotation of v is S, via the inversion of the typing relation the accompanying Security Annotation of v must be such that $S' \prec: S$.

The case for abstractions is similar: we assume that v has the annotated type of functions; applying via Lemma 4.5, we obtain that the only valid forms for values are those of the required form. □

We are now ready to prove progress (Theorem 4.8), again through induction. Along with the statement of preservation (Theorem 4.3), this proves the central notion of annotated type safety for this calculus, meaning that the addition of Security Annotations to a simply-typed lambda calculus respects the notion that properly typed programs cannot go wrong. The consequences of this are further discussed in Section 4.6.3.

Proof of 4.8. By induction on the typing derivation. Those cases which concern values ([T-TRUE] and [T-FALSE] and [T-ABS]) are immediate; [T-VAR] cannot occur since t is closed.

[T-IF] Let $t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3$, we know that

- $t_1 : \texttt{Bool}\langle S_1 \rangle$,
- $t_2 : T\langle S_2 \rangle$, and
- $t_3 : T\langle S_2 \rangle$.

By induction, either t_1 is a value or $t_1 \rightarrow t_1'$. In the latter case we can immediately apply [E-IF] and we are done. In the former, we have $t_1 = w\langle S \rangle$, and by canonical forms (Lemma 4.9), w is either `true` or `false`; in the former case we can apply [E-IFTRUE], and in the latter [E-IFFALSE].

[T-APP] We have $t = t_1\ t_2$. By induction:

- either t_1 is a value or $t_1 \rightarrow t_1'$, and
- either t_2 is a value or $t_2 \rightarrow t_2'$.

If $t_1 \rightarrow t_1'$, we can apply [E-APP1]; if t_1 is a value and $t_2 \rightarrow t_2'$, we can apply [E-APP2]. Finally, if both t_1 and t_2 are values, then by canonical forms (Lemma 4.9), t_1 has the form $(\lambda x : T_1 \langle S_1' \rangle . s) \langle S' \rangle$ and we can apply [E-APPABS].

[T-LET] Similar to the case [T-APP]: we have $t = \texttt{let}\ x = t_1\ \texttt{in}\ t_2$, by induction either t_1 is a value (in which case we can apply [E-LETV]), or t_1 can take a step of evaluation, in which case [E-LET] applies.

[T-AS] We have $t = t_1\ \texttt{as}\ S$, by induction if t_1 is a value, in which case [E-ASV] applies. Else, t_1 can take a step of evaluation and [E-AS] applies.

[T-DROP] We have $t = t_1\ \texttt{drop}\ S$, by induction if t_1 is a value, in which case [E-DROPV] applies. Else, t_1 can take a step of evaluation and [E-DROP] applies.

[T-CPANN] We have $t = \texttt{cpAnn}\ t_1\ t_2$. By induction, if t_1 is not a value then [E-CPANN1] applies. If t_1 is a value and t_2 is not, then [E-CPANN2] applies. Finally, if both t_1 and t_2 are values, then we can take a step of evaluation through [E-CPANN].

[T-SUB] This case follows directly from the induction hypothesis.

□

4.6.3 Discussion

In this section we discuss first the scope of annotated type safety in a language allowing ad-hoc coercions of annotations, and then discuss what

annotated type safety means within the setting of our core motivation of API checking.

Subverting annotated type safety. Since Security Annotations can be arbitrarily modified inline by the programmer, it is possible to circumvent the notion of annotated type safety by assigning annotations to terms to arbitrarily pass any type check; the developer would simply lose the benefits of using security annotations to enforce correct preconditions of secure APIs. For example, the term `(λx:Bool<A*B*C>. x)<> true<>` fails to type check; however, a developer may modify the program to change the applied term to `true<>` **`as`** `A*B*C`, at which point the program typechecks, although the annotations no longer faithfully models the security properties of the term.

What does annotated type safety mean for security properties? Security Annotations represent the validity of security properties on a value. Annotated type safety allows us to guarantee that the designed propagation of Security Annotations does not go wrong in two ways. First, that Security Annotations are transparent to the control flow of a program other than at enforcement of contracts in abstractions. Second, annotated type safety guarantees that the static enforcement of Security Annotations respects the dynamic evaluation: the static type of a program safely approximates the resulting type of the program after evaluation. However, the presence of a theorem of annotated type safety does not mean that security properties are verifiable inside this framework. Such verifiable properties are beyond the reach of annotated type safety: one must in addition verify that Security Annotation contracts between functions are correct with respect to enforcing the intended security property. Further, one must ensure that, per the previous discussion, upcasting via **`as`** and **`cpAnn`** are not used outside of these enforceable function contracts. In this setting, a type

safe program would successfully enforce these contracts.

4.7 Related Work

Type systems for security property enforcement have been advocated by Bhargavan *et al.* [7, 13, 35, 14]. The dependently typed languages F7 and F* [98] seek to statically verify security properties in F# code. Such languages allow for the use of expressive refinement types to check security properties in arbitrary implementations. Since we aim to check cryptographic misuse, we establish a more lightweight machinery which is designed specifically with such properties in mind.

Security type systems [86] augment types with annotations specifying policies for secure information flow. Our approach is similar to this, but heavily relies on annotations changing over time, which in security type systems only occurs during declassification. Security Annotations are designed to be particularly well-suited to dynamic languages: in allowing for their ad-hoc removal and addition to terms and values, they share similar paradigm as the base type system of such languages, e.g., JavaScript, which implicitly coerces types as required at runtime.

Another approach to checking cryptographic API usage is to check specifications written in domain specific languages (DSLs) [53]. One could encode the properties described within the DSL as Security Annotations and enforce these as pre- and postconditions in the manner described in this chapter. As discussed in Chapter 5, Security Annotations described in this chapter are designed to translate naturally to the dynamic nature of JavaScript, which is notoriously difficult to statically analyze.

A Semantics for Security Annotations in JavaScript

5

In this chapter we describe a formal semantics for Security Annotations in a JavaScript-like language. We construct a dynamic variant of Security Annotations, attached to values and objects via type-like information. We formalize this variant by extending S5 [81], an existing JavaScript semantics. We mechanize this language—which we call $S5_{SA}$—to obtain a reference implementation (Section 5.3-5.4). We extend the JavaScript-to-S5 desugaring relation to $S5_{SA}$ to obtain a reference interpreter for JavaScript with embedded Security Annotations (Section 5.5). Alongside this, we provide a specification for a fragment of the W3C WebCrypto standard and demonstrate how this specification can reveal security vulnerabilities in JavaScript code with the help of a case study within our reference interpreter (Section 5.7). We define a notion of safety with respect to $S5_{SA}$ and extend this to security guarantees for individual $S5_{SA}$ programs (Section 5.8). Finally, we discuss related work (Section 5.9).

This chapter comprises an extended version of a paper originally presented at *ESORICS 2019*, for which the author of this thesis was first author [64]. In particular, we extend our language formalization to include an annotation store (Section 5.5) and the accompanying case study (Section 5.7)[1].

[1] An accompanying reference implementation is available at `https://github.com/duncan-mitchell/SecAnnRefInterpreter`.

$$\text{[E-COMPAT]}\quad \frac{e \implies e'}{\sigma\Theta; \mathsf{E}\langle e\rangle \rightarrow \sigma\Theta; \mathsf{E}\langle e'\rangle}$$

$$\text{[E-ENVSTORE]}\quad \frac{\sigma; e \rightarrow^{\sigma} \sigma'; e'}{\sigma\Theta; \mathsf{E}\langle e\rangle \rightarrow \sigma'\Theta; \mathsf{E}\langle e'\rangle}$$

$$\text{[E-CONTROL]}\quad \frac{e \rightarrow^{\in} e'}{\sigma\Theta; \mathsf{E}\langle e\rangle \rightarrow \sigma\Theta; \mathsf{E}\langle e'\rangle}$$

$$\text{[E-OBJECTS]}\quad \frac{\Theta; e \rightarrow^{\Theta} \Theta'; e'}{\sigma\Theta; \mathsf{E}\langle e\rangle \rightarrow \sigma\Theta'; \mathsf{E}\langle e'\rangle}$$

Figure 5.1: The reduction relations for S5 [81].

5.1 Working with a semantics for JavaScript

S5 [81] is a lambda calculus-like language which reflects the semantics of the strict mode of EcmaScript 5.1 (ES5). S5 adopts the style of λ_{JS} [43], which targeted ES3, by reducing the language of JavaScript into a small core semantics. S5 is accompanied by a desugaring function, which takes native JavaScript source programs and translates them to S5 programs. S5 itself is described via small-step semantics, incorporating ES5 features such as getters, setters and eval. The language is not a complete reference implementation for the entire standard; however, S5 is tested against the official ECMAScript test suite, ensuring JavaScript's key features are handled correctly. This means it is the ideal target for describing the details of Security Annotations within JavaScript without unnecessary complication.

Terms in S5 are 3-tuples comprised of an expression, e, a store σ (mapping locations to values) and an object store Θ (mapping references to object literals); the evaluation context is denoted E. The reduction relation $\rightarrow$ is split into four parts dependent on which portions of the term are ma-

nipulated; their definitions are given in Figure 5.1. For example, the $\Longrightarrow$ relation, given by [E-COMPAT], does not modify either the object or variable store, instead operating entirely within the evaluation context. The relation given by [E-CONTROL], $\rightarrow^{e}$, governs certain control flow operators, like **`throw`**. For ease of reference, S5's syntax is given in Figure 5.2; full details of S5 are contained in the work of Politz et al [81].

5.2 A More Complex Language

Listing 5.1 describes a simple, abstract JavaScript program with Security Annotations[2]. This example constructs an object with two fields—one a boolean and the other a string—and describes Security Annotations on both the values stored in internal fields and on the overall object. This program evaluates to the object `{field1 : true<B>, field2: "val2" <C>}<A>`. In evaluating the expression stored in `field1` down to a value, since **`true`** is a boolean value, we can make use of our work in Chapter 4 and port over how the **`as`** expression evaluates. We can do this safely because we know this dynamic semantic mirrors the static enforcement of λ_{SA}; this enables us to keep evaluation of Security Annotations in some sense 'close' to our known safe language. In evaluating the expression stored in `field2`, we observe that the evaluation for booleans discussed in Chapter 4 can be carried over to strings and numbers as well.

Evaluating the coercion of the object's Security Annotation is more complex: objects in JavaScript are stored in an *object store* and a reference looks up this object in the store. To define sensible evaluation rules for JavaScript, then, we enrich S5 with Security Annotations in order to formalize the handling of such cases. Since S5 is close to a lambda calculus itself,

[2] In implementation, we use the delimiters < ! and ! >; in this thesis we match the semantic rules and use < and >.

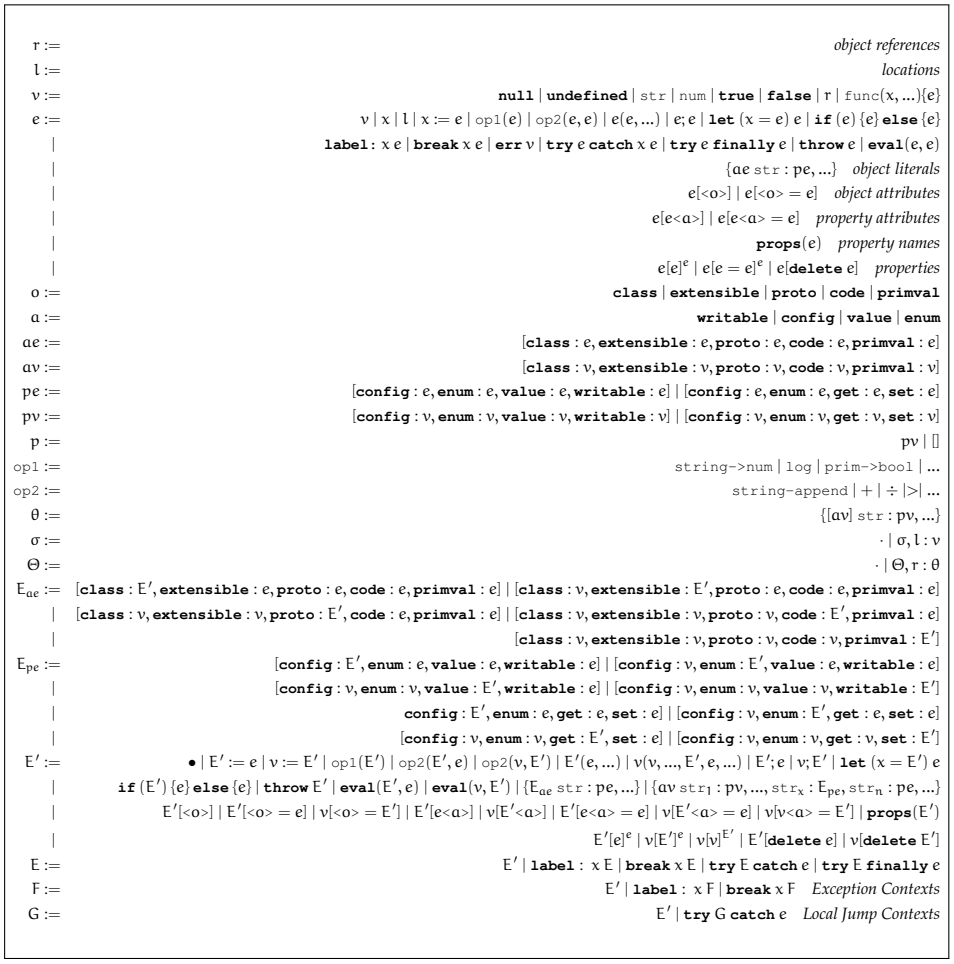

Figure 5.2: The syntax of S5 [81].

this allows for a natural translation between λ_{SA} and $S5_{SA}$ for most of expressions using Security Annotations; we need only semantics for those features unique to S5.

The $S5_{SA}$ program equivalent to Listing 5.1 is given in Listing 5.2. This is obtained from the original JavaScript program via a desugaring process

```
1 SecAnn <A * B * C>;
2 ({field1 : true as <B>, field2: "val2" as <C>}) as <A>
```

Listing 5.1: A simple JavaScript program using Security Annotations.

```
1  {let (%context = %nonstrictContext)
2    {let (#strict = false)
3      {SecAnn <!A * B * C!>;
4      { [#proto: %ObjectProto,
5        #class: "Object",
6        #extensible: true,]
7       'field1' : {#value (true as <B>),
8                 #writable true,
9                 #configurable true},
10      'field2' : {#value ("val2" as <C>),
11                 #writable true,
12                 #configurable true}
13     } as <A>}
14   }
15 }
```

Listing 5.2: The $S5_{SA}$ program corresponding to Listing 5.1.

$$
\begin{array}{rr}
a := & \textit{Atomic Annotations} \\
S := & a \mid S * S \mid \texttt{Top} \\
w := & \texttt{str} \mid \texttt{num} \mid \textbf{true} \mid \textbf{false} \\
w' := & \textbf{null} \mid \textbf{undefined} \mid \texttt{func}(x : S, \ldots)\{e\} \\
r := & \textit{object references} \\
l := & \textit{locations} \\
v := & w\texttt{<}S\texttt{>} \mid r\texttt{<Top>} \mid w'\texttt{<Top>} \\
e := & \ldots \mid e \ \textbf{as} \ S \mid e \ \textbf{drop} \ S \mid \textbf{cpAnn}(e, e) \\
\ldots & \\
o := & \{[av] \ \texttt{str} : pv, \ldots\} \\
\theta := & o\texttt{<}S\texttt{>} \\
\sigma := & \cdot \mid \sigma, l : v \\
\Theta := & \cdot \mid \Theta, r : \theta \\
\ldots & \\
E' := & \ldots \mid E' \ \textbf{as} \ S \mid E' \ \textbf{drop} \ S \mid \textbf{cpAnn}(E', e) \mid \textbf{cpAnn}(v, E')
\end{array}
$$

Figure 5.3: Syntax modifications to add Security Annotations to S5.

described in the work of Politz *et al.* [81] and extended to Security Annotations in Section 5.6. This process makes explicit the JavaScript context, as well as implicit properties of the defined object and the fields of the object. The desugaring process preserves syntax for manipulation of Security Annotations in S5, which remains the same as in JavaScript (and indeed, λ_{SA}).

5.3 Syntax of $S5_{SA}$

The additions and modifications to the syntax of S5 (Section 5.1) to extend the language to $S5_{SA}$ by incorporating Security Annotations are contained in Figure 5.3. We introduce atomic annotations α, as discussed in Section 3.2, and general annotations S, which are either `Top`, the least specific annotation, an atomic annotation, or the composition of two annotations, given by $*$. Although all values within $S5_{SA}$ are annotated, only certain prevalues w can possess arbitrary Security Annotations; values are written as $w\langle S\rangle$ which is syntactic sugar for the pair of a prevalue w along with its corresponding Security Annotation S. The prevalues **`undefined`** and **`null`** are considered non-annotatable precisely because it does not make sense to describe security properties of these special values. We attach `Top` to these prevalues to represent the lack of security properties, but arbitrary Security Annotations are invalid. Similarly, we do not allow annotations beyond `Top` on functions: they may act on values to alter security properties but in of themselves to do not possess security properties as described in this thesis (e.g., a function cannot be a cryptographic key or a ciphertext, see Section 3.1). An additional modification to the syntax reflects the addition of annotations to objects: we consider preobjects, o, which form objects when annotated with a Security Annotation S. We allow the arbitrary annotation of objects directly as opposed to their references (which, in the manner of w' possess only the annotation `Top`); properties within objects are annotated in the same manner as values. When an object is modified, previously valid security properties on the object are no longer guaranteed: modifying an object field should alter the annotations associated to the field, and also the annotations of the overall object.

Additional expressions, e, based on manipulating security annotations, cover the **`as`**, **`drop`** and **`cpAnn`** constructs. We add evaluation contexts, E', to cover these cases, where these are built in the same manner as in

S5 (recalling Section 5.1). Finally, enforcement of Security Annotations is added to functions via the form `func(x : S, ...)`; this does not require modification of the evaluation contexts.

5.4 Semantics for S5$_{SA}$

We formalize the extensions to S5 necessary to describe S5$_{SA}$. We describe mechanisms for manipulating annotations (Section 5.4.1), runtime enforcement (Section 5.4.2), and the rest of S5$_{SA}$ (Section 5.4.3).

5.4.1 Coercing Security Annotations

The evaluation judgments for coercion of annotations on values and objects are given in Figure 5.4, distinguished by case analysis on values. The expression v **as** S upcasts v to a more specific annotation, achieved by composing the previously valid annotation with S. Dependent on whether we treat w<S> (in [E-ASW]), or a reference ([E-ASR]), we make use of distinct reduction relations (Figure 5.1). In the former case, [E-COMPAT] is used to govern the evaluation. In the latter, [E-OBJECTS] is used to modify the object's annotation in the object store. Finally, we throw an error whenever w'<Top> is passed to one of these expressions treating coercion of annotations (e.g., [E-ASW']), since coercing the annotation of these values is prohibited. The case analysis for **drop** is similar; v **drop** S downcasts v to a less specific annotation. This is accomplished via the `cut` operator (Section 3.3) to prune the S from the annotation of v. As with **as**, the addition of newly valid annotations does not render previous annotations invalid, so composition unifies them; the evaluation rules are therefore similar in structure.

$$\text{[E-AsW]}\quad \frac{v = w\langle S\rangle}{v\ \mathbf{as}\ S' \implies w\langle S * S'\rangle}$$

$$\text{[E-AsR]}\quad \frac{v = r\langle \texttt{Top}\rangle \qquad \Theta(r) = o\langle R\rangle \qquad \Theta' = \Theta[^r/_{o\langle R * S\rangle}]}{\Theta; v\ \mathbf{as}\ S \to^{\Theta} \Theta'; v}$$

$$\text{[E-AsW$'$]}\quad \frac{v = w'\langle \texttt{Top}\rangle}{v\ \mathbf{as}\ S \implies \mathbf{throw}\ \texttt{NotAnnotatable}}$$

$$\text{[E-DropW]}\quad \frac{v = w\langle S\rangle}{v\ \mathbf{drop}\ S' \implies w\langle \texttt{cut}(S, S')\rangle}$$

$$\text{[E-DropR]}\quad \frac{v = r\langle \texttt{Top}\rangle \qquad \Theta(r) = o\langle R\rangle \qquad \Theta' = \Theta[^r/_{o\langle \texttt{cut}(R,S)\rangle}]}{\Theta; v\ \mathbf{drop}\ S \to^{\Theta} \Theta'; v}$$

$$\text{[E-DropW$'$]}\quad \frac{v = w'\langle \texttt{Top}\rangle}{v\ \mathbf{drop}\ S \implies \mathbf{throw}\ \texttt{NotAnnotatable}}$$

$$\text{[E-CpWW]}\quad \frac{v_1 = w_1\langle S_1\rangle \qquad v_2 = w_2\langle S_2\rangle}{\mathbf{cpAnn}(v_1, v_2) \implies w_2\langle S_1 * S_2\rangle}$$

$$\text{[E-CpWR]}\quad \frac{v_1 = w\langle S_1\rangle \quad v_2 = r\langle \texttt{Top}\rangle \quad \Theta(r) = o\langle S_2\rangle \quad \Theta' = \Theta[^r/_{o\langle S_1 * S_2\rangle}]}{\Theta; \mathbf{cpAnn}(v_1, v_2) \to^{\Theta} \Theta'; v_2}$$

$$\text{[E-CpRW]}\quad \frac{v_1 = r\langle \texttt{Top}\rangle \qquad \Theta(r) = o\langle S_1\rangle \qquad v_2 = w\langle S_2\rangle}{\Theta; \mathbf{cpAnn}(v_1, v_2) \to^{\Theta} \Theta; w\langle S_1 * S_2\rangle}$$

$$\text{[E-CpRR]}\quad \frac{v_1 = r_1\langle \texttt{Top}\rangle \qquad \Theta(r_1) = o_1\langle S_1\rangle \quad v_2 = r_2\langle \texttt{Top}\rangle \qquad \Theta(r_2) = o_2\langle S_2\rangle \quad \Theta' = \Theta[^{r_2}/_{o_2\langle S_1 * S_2\rangle}]}{\Theta; \mathbf{cpAnn}(v_1, v_2) \to^{\Theta} \Theta'; v_2}$$

$$\text{[E-CpW$'$V]}\quad \frac{v_1 = w'\langle \texttt{Top}\rangle}{\mathbf{cpAnn}(v_1, v_2) \implies \mathbf{throw}\ \texttt{NotAnnotatable}}$$

$$\text{[E-CpVW$'$]}\quad \frac{v_2 = w'\langle \texttt{Top}\rangle}{\mathbf{cpAnn}(v_1, v_2) \implies \mathbf{throw}\ \texttt{NotAnnotatable}}$$

Figure 5.4: Judgments for coercing annotations: **as**, **drop** and **cpAnn**.

$$\text{[E-App]}\quad \frac{\forall i \in \{1,\ldots,n\} : \mathrm{ann}(v_i) \prec: S'_i \qquad \sigma' = \sigma, l_1 : v_1, \ldots, l_n : v_n \text{ where } l_1 \ldots l_n \text{ fresh in } \sigma, e, v_1, \ldots, v_n}{\sigma\Theta; E\langle \texttt{func}(x_1 : S'_1, \ldots, x_n : S'_n)\{e\}(v_1, \ldots, v_n)\rangle \rightarrow \sigma'\Theta; E\langle e[^{x_1}/_{l_1}, \ldots, ^{x_n}/_{l_n}]\rangle}$$

$$\text{[E-AppFail]}\quad \frac{\exists i \in \{1,\ldots,n\} : \mathrm{ann}(v_i) \not\prec: S'_i}{\Theta; \texttt{func}(x_1 : S'_1, \ldots, x_n : S'_n)\{e\}(v_1, \ldots, v_n) \rightarrow^{\Theta} \Theta; \textbf{throw } \texttt{FailedSecurityCheck}}$$

Figure 5.5: Function application with Security Annotation enforcement.

5.4.2 Checking Security Annotations

Figure 5.5 codifies the enforcement of Security Annotations at function boundaries. To simplify the presentation of these rules we introduce a projection from values to annotations; note that this function returns the annotation of the corresponding object for references r, rather than `Top`:

$$\mathrm{ann}(v) := \begin{cases} \texttt{Top} & v = w'\langle\texttt{Top}\rangle \\ S & v = w\langle S\rangle \\ S & v = r\langle\texttt{Top}\rangle \wedge \Theta(r) = \theta\langle S\rangle \end{cases} .$$

[E-App] governs the case when arguments meet their annotation guards; in this case the function is evaluated as normal. This rule inspects the object store (to look up object annotations when arguments are references) and modifies the variable store (to bind arguments to the corresponding variables). We therefore use the standard reduction relation rather than the split components (Figure 5.1), which requires that we also explicitly make reference to the evaluation context E. To reflect the hierarchy of the annotation lattice, this rule bakes in subsumption, e.g., enforcement of `CryptKey`

would accept the more specific `PrivKey`. A common JavaScript paradigm is for non-annotatable values, e.g., functions, to be passed as arguments; their annotation guard must be `Top`, i.e., no security precondition, in line with the attached annotation. For any annotatable values, $w\langle S \rangle$, we insist S satisfies the guard S'. For references r, we look up the corresponding object and insist the annotation meets the guard. Direct checking of object properties and the **`this`** argument is achieved via source-to-source rewritings, described in Section 5.6. [E-APPFAIL] describes what happens when annotation-checking fails, i.e., whenever an argument carries a less precise annotation than its guard. `FailedSecurityCheck` is thrown to report the potential security vulnerability to the user, rather than simply halting evaluation.

5.4.3 Completing $S5_{SA}$

The rest of S5 remains largely unchanged. After object fields are manipulated, there is no guarantee the object annotation remains valid. For example, modifying the `keyUsages` field of a key object returned from WebCrypto's `generateKey` API may undermine the security of future operations involving the key. Valid security properties on the object are therefore unknown, so we associate `Top` to the object. Figure 5.6 includes judgments for field manipulation, including adding fields which do not exist and overwriting the values of existing fields which are writable ([E-SHADOWFIELD]). These semantics are transparent to annotations to allow prevalues to govern control flow, e.g., the `configurable` property must be **`true`** in [E-DELETEFIELD].

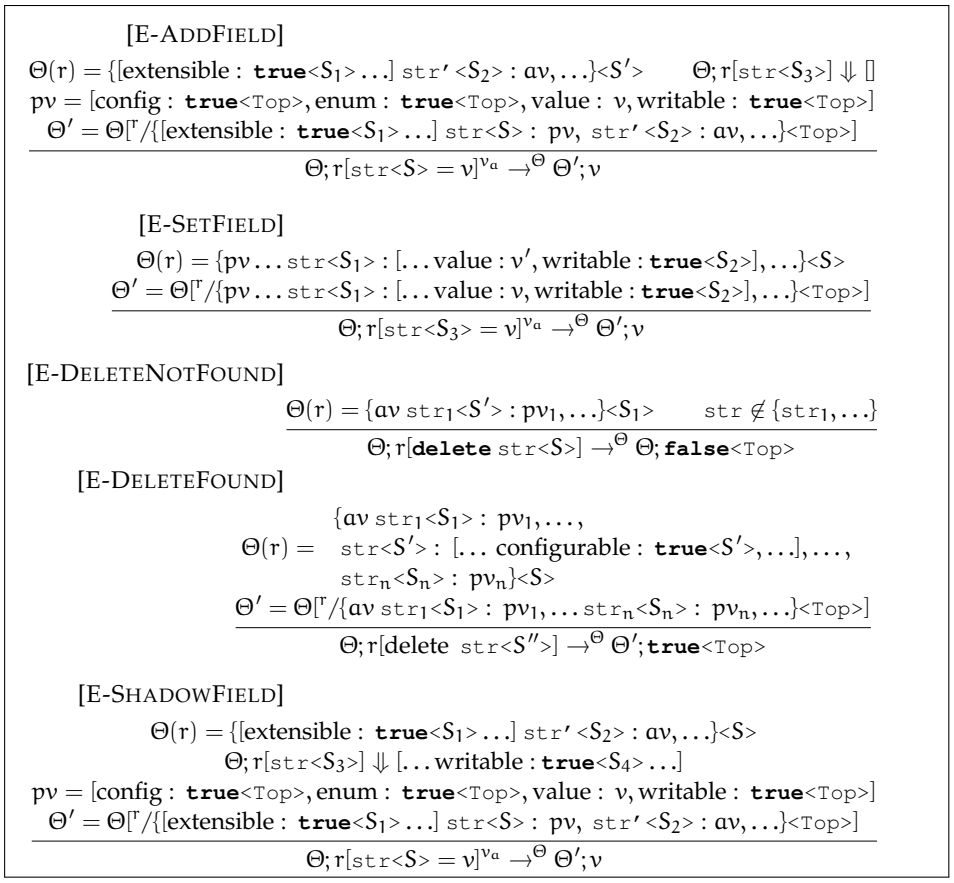

Figure 5.6: Judgments for setting, deleting and adding fields.

5.5 Implementing $S5_{SA}$

We mechanize Security Annotations on top of the existing reference implementation of S5 [81], closely following the rules described in Section 5.4. We describe the necessary extensions to our semantics to incorporate persistent annotations in a program, starting with declaration of annotations (Section 5.5.1) and an algorithm for deciding the subannotation relation within this framework (Section 5.5.2). We then discuss the adaptations to our core semantics this requires (Section 5.5.3-5.5.4).

5.5.1 Declaring Annotations

The only principal divergence from these rules is that alongside object and variable stores, we maintain a third *annotation store*, the lattice of valid annotations in the program. So far in this chapter we have not discussed the introduction of Security Annotations to a program; Security Annotations and their hierarchies are added via the expressions **`SecAnn`** S and **`SecAnn`** S **`Extends`** S described informally in Section 3.2. The former declares a new security annotation underneath `Top` but with no other hierarchical relationships, whereas the latter introduces a new Security Annotation as an immediate subannotation of the other. These expressions modify the annotation store to reflect additions to the lattice and evaluate to **`undefined`**. Using an annotation prior to declaration results in an exception.

Formally, the annotation store is a mapping from atomic annotations to sets of atomic annotations, $\mathcal{A} ::= \cdot \mid \mathcal{A}, a : \{a_1, \ldots, a_n\}$. Explicitly, whenever

$$\mathcal{A} : a \mapsto \{a_1, \ldots, a_n\}$$

then a has immediate subannotations $a_1, \ldots a_n$ (i.e., $a_i \prec: a$ and there is no b such that $a_i \prec: b$ and $b \prec: a$). In a similar manner to the reduction

$$\text{[E-SecAnnAtom]}\quad \frac{\mathfrak{a} \notin \mathrm{Dom}(\mathcal{A}) \qquad \mathcal{A}' = \mathcal{A}, \mathfrak{a} : \varnothing}{\mathcal{A}; \texttt{SecAnn}\ \mathfrak{a} \rightarrow^{\mathcal{A}} \mathcal{A}'; \texttt{undefined}}$$

$$\text{[E-SecAnnS]}\quad \frac{S = \mathfrak{a} * S' \qquad S' \neq \texttt{Top}}{\texttt{SecAnn}\ S \implies \texttt{SecAnn}\ \mathfrak{a}; \texttt{SecAnn}\ S'}$$

$$\text{[E-SecAnnTop]}\quad \frac{S = \mathfrak{a} * \texttt{Top}}{\texttt{SecAnn}\ S \implies \texttt{SecAnn}\ \mathfrak{a}}$$

$$\text{[E-SecAnnDef]}\quad \frac{\mathfrak{a} \in \mathrm{Dom}(\mathcal{A})}{\mathcal{A}; \texttt{SecAnn}\ \mathfrak{a} \rightarrow^{\mathcal{A}} \mathcal{A}; \texttt{undefined}}$$

Figure 5.7: Judgments for **SecAnn**.

relations (Figure 5.1), we introduce a new reduction relation which allows us to examine (and modify) this annotation store. This rule is expressed as:

$$\text{[E-Annotations]}\ \frac{\mathcal{A}; e \rightarrow^{\mathcal{A}} \mathcal{A}'; e'}{\sigma\Theta\mathcal{A}; \mathsf{E} < e > \rightarrow \sigma\Theta\mathcal{A}'; \mathsf{E} < e' >}.$$

We extend all other reduction rules to allow inspection (but not modification) of the annotation store.

Figure 5.7 details evaluation judgments for the **SecAnn** expression. In each of these figures, we use $\mathrm{Dom}(\mathcal{A})$ to denote the domain of $\mathcal{A}$, i.e., $\mathfrak{a} \in \mathrm{Dom}(\mathcal{A})$ requires that the mapping $\mathcal{A}$ is defined at the atomic annotation $\mathfrak{a}$. The rule [E-SecAnnAtom] defines a new atomic Security Annotation, adding it to the hierarchy with no immediate subannotations. The rule [E-SecAnnS] is applied when the Security Annotation declared is not atomic; in this case, we proceed recursively on the Security Annotation, decomposing it into atomics we can apply [E-SecAnnAtom] to. Finally, [E-SecAnnDef] handles the case where the Security Annotation has already been defined, in which case we simply evaluate to **undefined** and do not modify the annotation store.

Figure 5.8 describes the evaluation judgments for **Extends**; for the ex-

$$\text{[E-EXT]}\quad \frac{\mathcal{A}(a_2) = A \qquad a_1 \notin \mathrm{Dom}(\mathcal{A}) \qquad \mathcal{A}' = \mathcal{A}[^{a_2}/_{A \sqcap \{a_1\}}], a_1 : \varnothing}{\mathcal{A}; \textbf{SecAnn } a_1 \textbf{ Extends } a_2 \rightarrow^{\mathcal{A}} \mathcal{A}'; \textbf{undefined}}$$

$$\text{[E-EXT2]}\quad \frac{a_1, a_2 \notin \mathrm{Dom}(\mathcal{A}) \qquad \mathcal{A}' = \mathcal{A}, a_1 : \varnothing, a_2 : \{a_1\}}{\mathcal{A}; \textbf{SecAnn } a_1 \textbf{ Extends } a_2 \rightarrow^{\mathcal{A}} \mathcal{A}'; \textbf{undefined}}$$

$$\text{[E-EXTS]}\quad \frac{S = a_1 * S' \qquad S' \neq \texttt{Top}}{\textbf{SecAnn } S \textbf{ Extends } a_2 \implies \textbf{SecAnn } a_1 \textbf{ Extends } a_2; \textbf{SecAnn } S' \textbf{ Extends } a_2}$$

$$\text{[E-EXTSUBERR]}\quad \frac{a_1 \in \mathrm{Dom}(\mathcal{A})}{\mathcal{A}; \textbf{SecAnn } a_1 \textbf{ Extends } a_2 \rightarrow^{\mathcal{A}} \mathcal{A}; \textbf{throw } \texttt{SubAnnDeclared}}$$

$$\text{[E-EXTSUPERR]}\quad \frac{S_2 = a_1 * S' \qquad S' \neq \texttt{Top}}{\textbf{SecAnn } S_1 \textbf{ Extends } S_2 \implies \textbf{throw } \texttt{SuperNotAtomic}}$$

Figure 5.8: Judgments for **SecAnn Extends**.

pression **SecAnn** a_1 **Extends** a_2, we require that a_1 is previously undefined (in the case that it is, we apply [E-EXTSUBERR]). We distinguish by case when a_2 is defined (applying [E-EXT]) and when it is not (applying [E-EXT2]). In either case, we add a_1 to our annotation store (with no immediate subannotations) and augment the set mapped to by $\mathcal{A}(a_2)$ with $\{a_1\}$. The rule [E-EXTS] is similar to [E-SECANNS], whenever we extend a_2 with a non-atomic subannotation. Finally, the rule [E-EXTSUPERR] covers the case that a non-atomic subannotation is extended, which we do not allow (we prohibit atomic annotations from having multiple unrelated parents, see Section 3.2).

5.5.2 Deciding $\prec:$.

Algorithm 1 describes the mechanism we use in order to verify whether $S_1 \prec: S_2$ for arbitrary Security Annotations in implementation. The function *isSubAtomAtom* (lines 1-12) details the process of deciding if $a_1 \prec: a_2$ for two atomic Security Annotations. In this case, the results are immediate if either annotation is `Top`, since all annotations are a subannotation of `Top`, and `Top` $\prec:$ S only when S is precisely `Top`. Otherwise, we need to inspect the annotation hierarchy (lines 6-12); to do this, we look up a_2 in the annotation store, inspecting the elements of $\mathcal{A}(a_2)$. If $a_1 = a_2$, we are done by reflexivity. Whenever $a_1 \in \mathcal{A}(a_2)$, we are done, since a_1 is an immediate subannotation of a_2. Otherwise, we recurse on elements of $\mathcal{A}(a_2)$ (line 10) in order to examine the rest of the annotation hierarchy.

The function *isSubProdAtom* (lines 14-22) generalizes this to deciding whether $S \prec: a$ for a general Security Annotation S and atomic Security Annotation a. First, we consider the set of atomic Security Annotations which make up the atomic decomposition of S (line 14). Clearly, by our partial ordering, if a is part of this atomic decomposition, the $S \prec: a$. Otherwise, we check if there is some b in the atomic decomposition with $b \prec: a$ (line 20); if there is, then $S \prec: a$. If not, $S \not\prec: a$.

Armed with these two auxiliary functions, we check that S_1 is a subannotation of each annotation in the decomposition of S_2 (line 26). In the case that there is a b in the decomposition such that $S_1 \not\prec: b$, then S_1 cannot be a subannotation of S_2; if we exhaust all annotations in this decomposition, then $S_1 \prec: S_2$.

5.5.3 Adapting The Semantics of $S5_{SA}$

The addition of an annotation store does not meaningfully change the semantics detailed in Section 5.4; none of the semantics modify the annotation store, though they do inspect it. In particular, the annotation coercion

Algorithm 1: Deciding the subannotation relation.

Input : Security Annotations S_1, S_2 The annotation store, $\mathcal{A}$
Output: *true* if $S_1 \prec: S_2$, *false* otherwise

```
1  function isSubAtomAtom(a1, a2):
2      if a2 = Top then
3          return true
4      if a1 = Top then
5          return false
6      if a1 = a2 then
7          return true
8      else
9          foreach a ∈ A(a2) do
10             if isSubAtomAtom(a1, a) then
11                 return true
12         return false
13
14 function isSubProdAtom(S, b):
15     A := {a1, ..., an} where S = a1 * ... * an
16     if b ∈ A then
17         return true
18     else
19         foreach a ∈ A do
20             if isSubAtomAtom(a, b) then
21                 return true
22         return false
23
24 B := {b1, ..., bn} where S2 = b1 * ... * bn
25 foreach b ∈ B do
26     if !isSubProdAtom(S1, b) then
27         return false
28 return true
```

$$\text{[E-CompatAnn]}\quad \frac{\mathcal{A}; e \implies {}^{\mathcal{A}}\mathcal{A}; e'}{\sigma\Theta\mathcal{A}; \mathsf{E}\langle e\rangle \rightarrow \sigma\Theta\mathcal{A}; \mathsf{E}\langle e'\rangle}$$

$$\text{[E-EnvStoreAnn]}\quad \frac{\sigma\mathcal{A}; e \rightarrow^{\sigma\mathcal{A}} \sigma'\mathcal{A}; e'}{\sigma\Theta\mathcal{A}; \mathsf{E}\langle e\rangle \rightarrow \sigma'\Theta\mathcal{A}; \mathsf{E}\langle e'\rangle}$$

$$\text{[E-ControlAnn]}\quad \frac{\mathcal{A}; e \rightarrow^{\in} \mathcal{A}; e'}{\sigma\Theta\mathcal{A}; \mathsf{E}\langle e\rangle \rightarrow \sigma\Theta\mathcal{A}; \mathsf{E}\langle e'\rangle}$$

$$\text{[E-ObjectsAnn]}\quad \frac{\Theta\mathcal{A}; e \rightarrow^{\Theta\mathcal{A}} \Theta'\mathcal{A}; e'}{\sigma\Theta\mathcal{A}; \mathsf{E}\langle e\rangle \rightarrow \sigma\Theta'\mathcal{A}; \mathsf{E}\langle e'\rangle}$$

Figure 5.9: Reduction relations enriched with the annotation store.

expressions must inspect the annotation store to check that the annotations in the expression are defined. As mentioned previously, we construct natural extensions of the reduction relations (Figure 5.1); these additional rules, given in Figure 5.9, simply permit inspection of the annotation store.

In Figure 5.10, we give the rules for the **as** operator with reference to the annotation store. These make use of our extended reduction relations to ensure that the added Security Annotation has been properly defined prior to the coercion. In the case that it has not, we apply [E-AsUndec], and throw an error. Otherwise, we simply proceed by case analysis on v as before; note that [E-AsW′] remains unchanged.

5.5.4 Mechanizing Functions

The annotation store is also inspected in function application (Figure 5.5) to compare annotations with respect to subsumption. Figure 5.11 describes the extended evaluation rules for function application which first extend [E-App] by ensuring that all annotations have been declared, and extend the failure case to cover when specified annotations do not exist. The sub-

$$\text{[E-AsW-Store]}\quad \frac{v = w\langle S\rangle \qquad S' = a_1 * \ldots * a_n \qquad \forall i \in 1,\ldots,n, a_i \in \mathrm{Dom}(\mathcal{A})}{\mathcal{A}; v \texttt{ as } S' \implies {}^{\mathcal{A}}\mathcal{A}; w\langle S * S'\rangle}$$

$$\text{[E-AsR-Store]}\quad \frac{v = r \qquad \Theta(r) = o\langle R\rangle \qquad \Theta' = \Theta[{}^{r}/o\langle R * S\rangle] \qquad S = a_1 * \ldots * a_n \qquad \forall i \in 1,\ldots,n, a_i \in \mathrm{Dom}(\mathcal{A})}{\Theta\mathcal{A}; v \texttt{ as } S \rightarrow^{\Theta,\mathcal{A}} \Theta'\mathcal{A}; v}$$

$$\text{[E-AsUndec]}\quad \frac{S = a_1 * \ldots * a_n \qquad \exists i \in 1,\ldots,n, a_i \notin \mathrm{Dom}(\mathcal{A})}{\mathcal{A}; v \texttt{ as } S \implies \texttt{throw UndeclaredAnnotation}}$$

Figure 5.10: Judgments for **as** enriched with the annotation store.

annotation relation assertions are checked via the algorithm described in Section 5.5.2.

In $S5_{SA}$, we have described only functions in which each argument is checked against some annotation guard. In implementation, we retain enforcement-free functions and do not insist every argument has an annotation guard. In addition, instead of performing a single-step reduction on enforced functions (as in [E-App]), we split this reduction into two steps. First, we check the annotation guards against the arguments supplied; if any argument fails its annotation check, we throw `FailedSecurityCheck` as in [E-AppFail]. If all annotation checks succeed, we instead evaluate to an equivalent, enforcement-free function applied to the same arguments. This allows reuse of existing ES5 environment implementations described in the work of Politz *et al.* [81].

5.6 Executing JavaScript in $S5_{SA}$

We execute JavaScript code with Security Annotations by extending the JavaScript-to-S5 desugaring relation. We extend the syntax of JavaScript

$$\frac{\begin{array}{c}\text{[E-App]}\\ \forall i \in \{1,\ldots,n\} : (S'_i = a_{i,1} * \ldots * a_{i,n_i} \wedge \forall j \in \{1,\ldots,n_i\}, a_{i,j} \in \mathrm{Dom}(\mathcal{A}))\\ \forall i \in \{1,\ldots,n\} : \mathrm{ann}(v_i) \prec: S'_i\\ \sigma' = \sigma, l_1 : v_1, \ldots, l_n : v_n \text{ where } l_1 \ldots l_n \text{ fresh in } \sigma, e, v_1, \ldots, v_n\end{array}}{\begin{array}{c}\sigma\Theta\mathcal{A}; \mathrm{E}\langle \texttt{func}(x_1 : S'_1, \ldots, x_n : S'_n)\{e\}(v_1, \ldots, v_n)\rangle \rightarrow\\ \sigma'\Theta\mathcal{A}; \mathrm{E}\langle e[^{x_1}/_{l_1}, \ldots, ^{x_n}/_{l_n}]\rangle\end{array}}$$

$$\frac{\begin{array}{c}\text{[E-AppFail]}\\ \exists i \in \{1,\ldots,n\} : (S'_i = a_{i,1} * \ldots * a_{i,n_i} \wedge \exists j \in \{1,\ldots,n_i\}, a_{i,j} \notin \mathrm{Dom}(\mathcal{A}))\\ \vee\ \exists i \in \{1,\ldots,n\} : \mathrm{ann}(v_i) \not\prec: S'_i\end{array}}{\begin{array}{c}\Theta\mathcal{A}; \texttt{func}(x_1 : S'_1, \ldots, x_n : S'_n)\{e\}(v_1, \ldots, v_n)\\ \rightarrow^{\Theta\mathcal{A}} \Theta\mathcal{A}; \textbf{throw } \texttt{FailedSecurityCheck}\end{array}}$$

Figure 5.11: Function application enriched with an annotation store.

by adding Security Annotations and function guards, as well as the expressions **as**, **drop**, **cpAnn**, **SecAnn** and **SecAnn Extends**. Our desugaring rewrites these expressions into their S5 equivalents, which are then executed in the reference interpreter.

Checking object properties. We discuss how our implementation performs annotation checking of object internals through source-to-source transformations (Section 5.6). Recall Listing 3.1 provides a specification for WebCrypto's `encrypt` API, in which the `"iv"` property of the `alg` argument should be a cryptographically secure random value (i.e., annotated with `CSRV`). This demonstrates the need for checking properties of objects. We achieve this via source-to-source rewritings at the JavaScript level; these are simplified by an assert function:

```
let assert = function(arg, ann){
  (function(x : ann) {})(arg);
```

```
}.
```

There are three possible cases: first, `obj :S [prop, S]` checks that `obj[prop]` meets S. We check the specified property exists, and insist it satisfies the guard S:

```
if (typeof obj == 'object' &&
  Object.getOwnPropertyNames(obj).indexOf(prop) >= 0) {
    assert(x[prop], S);
} else { throw 'FailedSecurityCheck'; }
```

Second, `obj :A S` checks all properties meet the guard S; to achieve this we iterate over all object properties:

```
if (typeof obj == 'object') {
  let props = Object.getOwnPropertyNames(obj);
  for (let iter = 0; iter < props.length; iter++) {
    assert(obj[props[iter]], S);
  }
} else { throw 'FailedSecurityCheck'; }.
```

Finally, `obj :E [N, S]` checks at least N properties satisfy S. As before, we iterate over object properties, counting the number that meet the guard:

```
if (typeof obj == 'object') {
  let props = Object.getOwnPropertyNames(x), successes = 0;
  for (let iter = 0; iter < props.length; iter++) {
    try { assert(x[props[iter]], S); successes++; } catch (e
        ) { };
  }
  if (successes < N) { throw 'FailedSecurityCheck'; }
} else { throw 'FailedSecurityCheck'; }.
```

Checking `this`. Functions have an implicit **`this`** argument, the context object in which the current code is executing. For example, the following

evaluates to **true**:

```
let obj = {get: function() {return this.val;}, val: true};
obj.get();
```

In the manner of object property enforcement, we check this via the syntax `function(: S, ...) { body(); }` which is rewritten to `function (...){ assert(this, S); body(); }`, where `assert` is the helper function discussed previously.

5.7 Using S5$_{SA}$: A Case Study

We provide a reference implementation of Security Annotations for the correctness of future implementations in native JavaScript. Our interpreter translates a subset of Node.js programs into S5 programs; we demonstrate the scope of this reference interpreter by describing the modifications to programs necessary for execution. We outline how we envisage Security Annotations being used by developers to detect security vulnerabilities through case study within our interpreter.

A client-server application. We implement in Node.js a small chat application which takes as argument a confidential message a client wishes to transmit to a server. The server and client negotiate a key exchange, and an encrypted copy of this message is sent to the server, which decrypts it. We omit authentication from this case study for simplicity of presentation.

Execution in S5$_{SA}$. In order to execute the case study in S5, we must simulate the behvior of certain libraries; throughout this thesis, we call these copies of the libraries *mocks*. Library mocks are necessary to execute the case study in S5. S5 does not support asynchronous code, so we

construct a synchronous mock of the networking API, `net` [72]. An extension to asynchronous code is possible in principle based on an existing formalization of JavaScript promises [62]. Second, cryptographic operations are mocked as stub functions returning objects of the same underlying structure. Finally, S5 programs do not take input, so we explicitly declare `process.argv` to simulate this. WebCrypto is not implemented in Node.js, so we construct a synchronous mock using the Node.js `crypto` module [70].

A shim for a WebCrypto fragment. Listing 5.3 contains an annotated shim of a fragment of WebCrypto for use by developers. These method specifications follow the same structure as Listing 3.1 (line 47 refers to this shim). `getRandomValues` fills the supplied array with cryptographically secure random values, so this array is annotated with `CSRV`. Despite the lack of a return, the annotation on this array persists because the annotation is attached directly to the object. `generateKey` constructs a key (or key pair) object for the supplied algorithm; postconditions of this method are differentiated by case analysis. `deriveKey` is used to compute a shared secret key from the other party's public key and the private key. The contract for `decrypt` is similar to `encrypt`; we do not enforce `Ciphertext` against `data`—or that the IV is randomly generated—to allow decryption of messages received across a network. `importKey` allows public keys received across a network to be formatted for use with other WebCrypto APIs. This API allows the upcasting of arbitrary data; however, without `importKey`, it would be impossible to use WebCrypto across a network.

A security property violation. Listing 5.4 shows the developer's function used to construct the IV. The developer constructs an array to store the IV is constructed as an array of 16 unsigned 8-bit integers (line 2). This array is then used in the correct generation of an IV the same size as the

```
1  SecAnn <CSRV * Message * CryptKey>;
2  SecAnn <PrivKey * PubKey * SymKey> Extends <CryptKey>;
3  SecAnn <Plaintext * Ciphertext> Extends <Message>;
4
5  window.oldCrypto = window.crypto;
6  let wc = window.oldCrypto.subtle;
7
8  const grvShim = function(arr) {
9   window.oldCrypto.getRandomValues(arr);
10  arr as <CSRV>;
11 };
12
13 const gkShim = async function(alg, extractable, keyUsages) {
14  let key = await wc.generateKey(alg, extractable, keyUsages);
15  if (/RSA|ECD/.test(alg.name)) {
16   key.privateKey = key.privateKey as <PrivKey * CSRV>;
17   key.publicKey = key.publicKey as <PubKey * CSRV>;
18  } else if(/AES|HMAC/.test(alg.name)) {
19   key as <SymKey * CSRV>;
20  } else { throw FailedSecurityCheck; }
21  return key;
22 };
23
24 const dkShim = async function(alg :S ["public", <PubKey>],
25    masterKey : <PrivKey>, derivedKeyAlg, extractable, keyUsages) {
26  let key = await wc.deriveKey(alg, masterKey, derivedKeyAlg,
        extractable,
27    keyUsages);
28  return (key as <SymKey>);
29 };
30
31 const decShim = async function(alg, key, data) {
32  if (/AES/.test(alg.name)) {
33   (function(arg : <SymKey>) {})(key);
34  } else if (/RSA/.test(alg.name)) {
35   (function(arg : <PrivKey>) {})(key);
36  } else { throw FailedSecurityCheck; }
37  var res = await wc.decrypt(alg, key, data);
38  return ((cpAnn(data, res) drop <Ciphertext>) as <Plaintext>);
39 };
40
41 const ikShim = async function(type, key, alg, extractable, keyUsages) {
42  let pubKey = await wc.importKey(type, key, alg, extractable, keyUsages
        );
43  return (pubKey as <PubKey>);
44 };
45
46 const wcShim = {generateKey: {value: gkShim},
47  deriveKey: {value: dkShim}, encrypt: {value: encShim},
48  decrypt: {value: decShim}, importKey: {value: ikShim}};
49 defineProperty(window.crypto, "subtle", { value: wcShim});
50 defineProperty(window.crypto, "getRandomValues", {value: grvShim});
```

Listing 5.3: An annotated shim for a fragment of the WebCrypto API.

```
1 function getIV() {
2   var iv = new Uint8Array(16);
3   crypto.getRandomValues(iv);
4   for (var i = 0; i < iv.length; i++) {
5     iv[i] = iv[i] % 128;
6   }
7   return iv;
8 }
```

Listing 5.4: Case Study: constructing the IV.

```
1 function encrypt(store, str) {
2   var ivEnc = getIV();
3   var res = crypto.subtle.encrypt({
4     name:'AES-CBC',
5     length:128,
6     iv: ivEnc,
7   }, store.sharedSecret, str);
8   return "{ct:"+res[0]+",iv:"+res[1]+"}"
9 }
```

Listing 5.5: Case Study: the developer's encrypt function.

cipher block size (line 3). The developer then forces each element of the array forming the IV to be an integer in the interval $[0, 127]$ (line 5). This is intended to ensure that the IV can be directly encoded as an ASCII string. However, the developer in practice, this simply reduces the entropy to only 112 bits (from 128), less than the block size: a potential security flaw which does not visibly affect runtime behavior.

To detect such bugs, a developer includes our WebCrypto shim. When the developer calls their `encrypt` function (Listing 5.5), they first generate a fresh IV (line 2). As discussed, this IV is generated through a call to the WebCrypto API, and so is initally annotated with `CSRV` (per the specifi-

cation in Listing 5.3). However, the manipulation of the array drops the annotation (per [E-SETFIELD] in Figure 5.6). Since the `iv` property of the object passed as the first argument (line 6) is not annotated with `CSRV`, the call to `encrypt` fails, `FailedSecurityCheck` is thrown and this security flaw is reported to the developer. When the manipulation of the array in `getIV` is removed, no error is thrown; the security pre- and postconditions enforced in the shim are respected.

5.8 Properties of S5$_{SA}$ Programs

We discuss safety guarantees for S5 programs with Security Annotations (Section 5.8.1) and extend this to security guarantees (Section 5.8.2). Finally, we apply this to prove security of our case study (Section 5.8.3). Throughout this section, we assume all programs discussed terminate; since Security Annotations are enforced at runtime, for programs which do not terminate, we cannot comment on whether there are Security Annotation violations. That is, due to the nature of the analysis, for these programs, we cannot comment on the program since the analysis itself may not terminate.

5.8.1 Safety Guarantees

We adopt a relatively modest notion of safety: first, a program is safe if it does not evaluate to an exception as a result of a function argument failing to meet the annotation guard. Second, the program should not coerce the annotation of a non-annotatable value, e.g., **`null as`** `<CSRV>`. This gives us the definition:

Definition 5.1 (Annotation Safety)**.** An S5 program is *safe with respect to Security Annotations* (or, *annotation safe*) if the execution of the program does

not result in any of the `FailedSecurityCheck`, `UndeclaredAnnotation` or `NotAnnotatable` exceptions.

Although programs in S5 are deterministic, programs in JavaScript (or any meaningful language) are not: their execution depends on the DOM or user input. To consider this broader context, suppose $\mathcal{P}$ is a program expecting input, we extend Definition 5.1 as follows:

Definition 5.2 (Annotation Safety for Programs with Input)**.** A program which takes input $\mathcal{P}$, is *annotation safe* if no execution of the program results in any of the `FailedSecurityCheck`, `UndeclaredAnnotation` or `NotAnnotatable` exceptions.

Consider a family of S5 programs, Π, which are deterministic and simulate input by declaring a global variable `process.argv` assigned to an object containing N fields. For each field, f_i suppose there is an accompanying value v_i. For each v_i, we fix a base type and range over all possible prevalues (and **`undefined`**, which simulates a lack of input). If v_i is a reference to an object, we range over all possible objects θ. The resulting family of programs represents the space of possible executions for $\mathcal{P}$. For example, our case study (Section 5.7) takes only a single input (the message to send); Π describes this by fixing `process.argv` as an object with a single field which ranges over all possible strings and **`undefined`**. We can therefore reformulate Definition 5.2:

Lemma 5.3. Let $\mathcal{P}$ be an S5 program with input and Π the family of deterministic programs p describing all possible inputs for $\mathcal{P}$. Then $\mathcal{P}$ is annotation safe if and only if every program $p \in \Pi$ is annotation safe.

Proof. By construction, each execution of $\mathcal{P}$ is considered as a separate deterministic program P so the result is immediate. □

Since this family Π is very large, we formalize safety in terms of a subset of these programs. Let π be the set of all $p \in \Pi$ following exactly the same

sequence of evaluation judgments. This set of S5 programs corresponds to a single control-flow path of $\mathcal{P}$: so if any p is annotation safe, so are all programs in π. Since the union of all (clearly disjoint) possible paths π is equal to the overall family of programs Π, we can obtain a simpler notion of safety for $\mathcal{P}$:

Theorem 5.4. Let Π be the family of deterministic programs describing all possible inputs for $\mathcal{P}$. Consider all disjoint subsets $\pi \subseteq \Pi$ representing single control flow paths of $\mathcal{P}$, and for each, choose a single $p \in \pi$. Then $\mathcal{P}$ is annotation safe if and only if each p is annotation safe.

Proof. Suppose first that $\mathcal{P}$ is annotation safe. Then by Lemma 5.3, we know every $P \in \Pi$ is annotation safe. Since each $\pi \subseteq \Pi$, each p must be annotation safe as required. For the other direction, suppose each p is annotation safe. Pick one such p, and the subset of Π to which it belongs, π. Let p' be some other program in π, and suppose that p' is not annotation safe. Then the execution of p' results in either a `FailedSecurityCheck` or `NotAnnotable` exception. This means that the final evaluation judgment applied in the evaluation of p' is either [E-AppFail], [E-AsW'], [E-DropW'], [E-CpW'V] or [E-CpVW']. Since p and p' both belong to π, they follow the same sequence of evaluation judgments. But then p is not annotation safe, which is a contradiction. Thus each p in π is annotation safe, and extending this across all disjoint subsets π of Π, each program in Π must be annotation safe. Applying Lemma 5.3 again, we are done. □

This result says that if any set π is not safe, then some control-flow path in $\mathcal{P}$ violates the Security Annotation specification of the program, indicating a possible security vulnerability. This description of safety requires us to find these subsets π to obtain a guarantee. In practice, this is equivalent to enumerating all control flow paths of a program over all types of input values and objects, which makes our mechanism ideally suited for combination with feedback-directed fuzzing or dynamic symbolic execution [61].

For example, in our case study, there are two control flow paths—either the input is provided, or it is not. We therefore need only consider two programs to obtain Π-safety: where `process.argv` is **`undefined`** and one where the only field is an arbitrary string, say `'hi'`.

5.8.2 Security Guarantees

We extend the notion of safety to security guarantees for Security Annotations. Let α be the function annotating an arbitrary program supplied as argument (of course, in practice this is a manual process). Let L be a library and $\alpha(L)$ an annotated shim of this library (e.g., Listing 5.3); any security guarantees are conditional on the correctness of L, e.g., that WebCrypto itself is a correct implementation of cryptographic primitives. Let P be an S5 program which calls L, and suppose the developer of P in-lines this annotated shim in a program $P' = \alpha(L); P$. For the remainder of this section, we assume that P does not contain any expressions which manipulate Security Annotations. We can make the following (overapproximate) claim, which states that whenever P' is annotation safe, it respects the security properties enforced by the Security Annotation specifications of the methods in $\alpha(L)$.

Lemma 5.5. Suppose P' is annotation safe and that P does not contain any expressions which manipulate Security Annotations. Then the Security Annotation specifications described in $\alpha(L)$ are respected.

Proof. Suppose a Security Annotation specification in $\alpha(L)$ is not respected. Then some function precondition fails, so the judgment [E-APPFAIL] is evaluated, contradicting our assumption that P' is annotation safe. Since P does not involve the manipulation of Security Annotations, any annotations must be the postconditions of an API call in $\alpha(L)$; hence these specifications are respected. □

Analogously to Section 5.8.1, we extend this result to programs with input:

Theorem 5.6. Let $\mathcal{P}$ be a program with input and suppose $\mathcal{P}' = \alpha(\mathtt{L}); \mathcal{P}$ is annotation safe. Suppose further that that $\mathtt{P}$ does not contain any expressions which manipulate Security Annotations. Then the Security Annotation specifications described in $\alpha(\mathtt{L})$ are respected.

Proof. This follows immediately from the combination of Theorem 5.4 and Lemma 5.5. □

Annotation safety and approximation. The converse of this result is not true: if a program $\mathcal{P}'$ is not annotation safe this does not necessarily mean a security property is violated. The conservative nature of Security Annotations means that some valid Security Annotations may be discarded resulting in a program to fail to annotation check. For example, adding a field to the key object generated by WebCrypto's `deriveKey` API does not necessarily invalidate the fact that the object contains symmetric key, however the annotation would be discarded. The notion of annotation safety is therefore overapproximate with respect to the security properties Security Annotations express.

5.8.3 Security Guarantees in Practice

We use Theorem 5.6 to describe concrete security guarantees for the case study outlined in Section 5.7, which are conditional on the correctness of WebCrypto. Recall that after fixing the security vulnerability involving the ASCII-encoded IV, when a message supplied as argument, the program executes without error; if no message is provided the application simply reports this to the user and exits. Both control-flow paths of this program are annotation safe. With reference to the specifications described in our WebCrypto shim (Listing 5.3), there are two caveats to our claim; the first

assumes the developer does not leak keying material and the second relates to the omission of authentication from the case study.

Security Statement 5.7. Suppose that: (i) None of the initialization vector (IV), symmetric key nor either party's secret keys are leaked across the network, (ii), an attacker impersonates neither party and (iii) the attacker cannot break the AES-128 cipher in CBC mode. Then encrypted messages sent by the client can only be read by the server.

Analysis. The application does not manipulate annotations; when executed with a non-annotated copy of the library the program is annotation safe. As described above, both control-flow paths of the program are annotation safe with our annotated library in-lined, we can directly apply Theorem 5.6. It remains to demonstrate the specification enforced by the annotation library. The encryption—via AES-CBC with a 128-bit key—is secure only when the symmetric key has been securely derived, and the IV is a block-sized CSRV (Listing 3.1). Our WebCrypto specification enforces the CSRV portion of the contract directly: calling `getRandomValues` annotates the IV with `CSRV` (lines 8-10 of Listing 5.3), and this array is not subsequently modified, the annotation check on entry to `encrypt` passes.

Second, the symmetric key used for AES must be shared between the two parties secretly. The key is derived through an ECDH key exchange; both the server and client use `generateKey` (lines 13-22 of Listing 5.3) to compute a key pair. Public keys are exchanged, and validated it through `importKey` (lines 41-44). The client supplies their private key and the server's public key to `deriveKey` (lines 24-29). Neither key has been tampered with, so the client's key is annotated with `PrivKey` and the server's with `PubKey`. This satisfies the guard of `deriveKey`, and so the key for AES is computed, and annotated `SymKey`. The provenance of the secret key as derived from safe API calls can be confirmed, so the guard against the key in `encrypt` succeeds (line 10 of Listing 3.1). Therefore, only some-

one in possession of the private key corresponding to the server's public key can read the message supplied as `data` to this API. □

5.8.4 Limitations

We discuss the limitations of the approach described in the previous section, with particular reference to the annotation of libraries through the function $\mathfrak{a}$. As before, let $\mathtt{P}$ be a program calling a library $\mathtt{L}$ and $\mathtt{P}' = \mathfrak{a}(\mathtt{L}); \mathtt{P}$ be the program with an annotated copy of $\mathtt{L}$ in-lined. There are three principal limitations to our approach relating to the fact that $\mathtt{P}$ is an arbitrary piece of code, other than the fact that it does not directly manipulate Security Annotations:

(i) The process of annotating the library, $\mathfrak{a}$, may affect $\mathtt{L}$ or $\mathtt{P}$.

(ii) The program $\mathtt{P}$ may affect $\mathtt{L}$.

(iii) The program $\mathtt{P}$ may affect the annotations $\mathfrak{a}$.

In this section we describe our assumptions in detail, and describe the relation of these to each of the three principal limitations and their implications outside of an idealized setting.

Threat assumptions. As described throughout this thesis, the intended usage scenario of Security Annotations is within testing environments. We consider three actors in our assumptions: the library author, the library annotator and the program author. First, we assume that none of these actors is malicious. Second, we assume that the library author writes a trusted, correct implementation. We assume that the library annotator does not modify the library; in particular, we assume that $\mathfrak{a}(\mathtt{L})$ has the following structure: for each method $\mathtt{m}$ in $\mathtt{L}$, the shim $\mathfrak{a}(\mathtt{m})$ first enforces annotation guards on provided arguments, then calls $\mathtt{m}$ directly. Finally, the return

value of m is annotated through the **as**, **drop** and **cpAnn** expressions (in the manner of the shim given in Listing 5.3). That is, the logic of m is preserved. Finally we assume that the program author does not directly manipulate Security Annotations in P via the expressions **as**, **drop** and **cpAnn**. Other than this, the program P is considered arbitrary.

α **affecting** L **or** P**.** This limitation covers the effect of the library annotator causing a change in behavior to either L or P. As described in our assumptions, we assume that the result of evaluation of each exposed library method is, modulo annotations, unmodified by the shim α(L). That is, there should be no difference in the evaluation of the library method and the shimmed method other than the presence of Security Annotations. If this assumption is broken, the behavior of the program α(L); P is now distinct from L; P so the value of using Security Annotations as a testing mechanism is reduced (since behavior in a production environment may differ). However, it is still possible that the Security Annotations could correctly specify the modified L, meaning that there may be security guarantees for α(L); P, it would just be impossible to transfer these guarantees to the program L; P.

P **affecting** L**.** This limitation describes the case when a program directly, or indirectly, modifies the library it calls. For example, suppose L contains the method:

```
function len(str) {
  return str.length;
}.
```

If P modifies the definition of `String.length` at runtime, then `len` will not behave as expected. This would therefore undermine the annotation specification described in α(L).

P affecting α. This case is similar, but more subtle, than the P affecting L. We have assumed P does not manipulate Security Annotations through `as`, `drop` and `cpAnn`. Since this is the only way that Security Annotations can be manipulated in a $S5_{SA}$ program, P cannot directly modify which annotations are manipulated by the shim α. However, in the same way L can be modified by P, so too could α if the shim adds additional logic to decide, for example, which type of key annotation should be added (e.g., similar to the conditional logic in Listing 5.3). This modification could cause incorrect annotations to be added to values, causing executions of P to pass annotation checks when they should fail (or vice versa). In our native JavaScript implementation (Chapter 7), it is further possible that a non-malicious yet pathological P may cause effects equivalent to the expressions which manipulate annotations. This would cause executions of P to pass annotation checks when they should fail.

Addressing the effect of P on α and L. Both of the two above limitations are concerned with the effect of our arbitrary program P with either α or L. We consider the problem out of scope for this thesis; the problem of dealing with interaction between trusted and untrusted code is treated by systems such as DJS [11] which can offer guarantees of noninterference in such a scenario. However, such guarantees come at a significant cost in restricting the languages both L and α can be written in. Since the approach presented in this thesis is designed to allow the annotation and testing of the use of arbitrary libraries, this restriction would severely limit the applicability of the approach.

5.9 Related Work

A variety of formalizations of the JavaScript language exist [88, 77, 43, 81]; these are discussed in Section 2.2. Since we do not intend the checking of Security Annotations to be performed directly in a formal intermediate representation, we prioritize a model amenable to clear concise representations of Security Annotations, where security proofs are tractable. Security Annotations allow developers to analyze their program for security bugs without the need for manual intervention; this contrasts with the high level of developer input required to achieve verification with the formal framework JaVerT [88]. Since S5 remains closer to the minimal lambda calculus described Chapter 4, this allows for a more natural translation.

Our work is complementary to work on cryptographic API usage in Android applications [54, 32]. *CrySL* [53] is a DSL for the specifying correct usage of the Java Cryptography Architecture. The motivation of this work is similar to this thesis, intending cryptographic experts write specifications which are then statically checked without modification of the application. Our approach of encoding pre- and postconditions via Security Annotations on values and objects embraces the dynamicity of JavaScript, for which static analysis has historically proved difficult.

The use of statically typed JavaScript dialects to ensure safety is common [27, 108, 25, 83]. The effective use of such static typing approaches would require the modification of APIs and language semantics, e.g., directly prohibiting byte array indexing of Key types. Further, we allow developers to test existing implementations dynamically without rewriting into such a dialect. Similarly, existing DSLs for cryptographic code in JavaScript [12, 52, 10] are not amenable for use on existing applications: these languages are small subsets of JavaScript without many of the common idioms and advantages of the language.

Refinement type systems for security property checking [7, 13, 14, 98]

demonstrate the viability of verification of security properties in mainstream implementations. Through Security Annotations, we offer idiomatic and concise descriptions of security properties, designing an alternative system which avoids the need for developers to directly annotate program terms to achieve results. Since we check Security Annotations at runtime, we do not offer formal guarantees in the manner of this work.

The work of Taly *et al.* details the process of constructing automated analysis for security-critical JavaScript APIs [99]. The work focuses on restricting security critical API usage by confinement: the trusted code hides all security-critical resources behind an API which provides methods to access these resources from untrusted code securely. Of course, it is necessary to ensure that such a mechanism is secure against arbitrary untrusted code, i.e. that an attacker cannot gain access to confined resources without using these APIs. This work differs substantially from the aims of this chapter (and, indeed, this thesis): we explicitly assume that the APIs themselves are secure and correct.

Sound Regular Expression Semantics for the Dynamic Symbolic Execution of JavaScript

6

In this chapter, we discuss an extension of the DSE engine ExpoSE (described in Section 2.4.2) to support the modeling of JavaScript's flavor of extended regular expressions, which encode non-regular languages. Many programs making use of cryptographic functionality naturally involve the manipulation of strings, e.g., processing encrypted messages received across a network. A lack of comprehensive support for string manipulation in a DSE engine would therefore be detrimental to any program analysis focusing on finding cryptographic errors. We describe a logical model for analyzing these extended regular expressions in terms of regular languages and string constraints which allows for the analysis of such operations.

Although modern constraint solvers support regular expressions in the language-theoretic sense, support for regex (see Section 1.4), which can represet non-regular langauges [3], is limited. In tools reasoning about string-manipulating programs, these extensions to the machinery of regular expressions are usually ignored or imprecisely approximated. This can, within the context of dynamic symbolic execution (DSE) for test generation, lead to missed bugs where constraints would have to include membership in non-regular languages.

To date, there has been only limited progress on this problem, mostly addressing immediate needs of implementations with approximate solutions, e.g., for capture groups [90] and backreferences [91]. We extend these to a comprehensive model for both these features, lookaheads and matching precedence.

In this chapter, we propose a novel, sound, scheme for supporting ECMAScript regex in dynamic symbolic execution. We rely on the specification of regexes and their associated methods in ECMAScript 2015 (ES6) [31]. However, our methods are easily transferable to most other existing implementations. In particular, we describe the following:

- We fully model ES6 regex in terms of classical regular languages and string constraints. We introduce the notion of a *capturing language* to make the problem of matching and capture group assignment self-contained.
- We introduce a counterexample-guided abstraction refinement (CEGAR) scheme to address the effect of greediness on capture groups, which allows us to deploy our model in DSE without sacrificing soundness for under-approximation.

In the remainder of the chapter we review ES6 regexes (Section 6.1). We then present an overview of our approach by example (Section 6.2). We detail our regex model using a novel formulation (Section 6.3), and we propose a CEGAR scheme to address matching precedence (Section 6.4). We describe how this can be used to provide a model for JavaScript's regex API (Section 6.5) and some consequences of this model (Section 6.6). Finally, we discuss related work (Section 6.7)

This chapter is drawn from work carried out in collaboration with my colleague Blake Loring and previously published in both *SPIN 2017* [60] and *PLDI 2019* [61]. Here, we present the formal logical model for extended regular expressions based on the notion of Capturing Languages

(Section 6.3), which was the core contribution by the author of this thesis. The author was also involved in the design of the CEGAR algorithm described in Section 6.4. For a full evaluation of this work, the reader is referred to the full paper [61] which contains experiments not carried out by the author of this thesis.

6.1 ECMAScript Regex

We review the ES6 regex specification, focusing on differences to classical regular expressions. We begin with the regex API and then explain capture groups, backreferences and operator precedence. ES6 regexes are comparable to those of other languages but lack Perl's recursion and lookbehind and do not require POSIX-like longest matches.

Methods and Anchors. ES6 regexes are `RegExp` objects, created from literals or the `RegExp` constructor, with two methods, `test` and `exec`, which expect a string argument. String objects offer the `match`, `split`, `search` and `replace` methods that expect a `RegExp` argument. A regex accepts a string if any portion of the string matches the expression, i.e., it is implicitly surrounded by wildcards; relative position in the string can be controlled with *anchors*, with ^ and $ matching the start and end, respectively.

Flags. Regexes can contain *flags* which modify the behavior of matching operations. The *ignore case* flag `i` ignores character cases when matching. The *multiline* flag `m` redefines anchor characters to match either the start and end of input or newline characters. The *unicode* flag `u` changes how unicode literals are escaped within an expression. The *sticky* flag `y` forces matching to start at `RegExp.lastIndex`, which is updated with the index of the previous match. Therefore, `RegExp` objects become stateful as seen

in the following example:

```
r = /goo+d/y;
r.test("goood"); // true; r.lastIndex = 6
r.test("goood"); // false; r.lastIndex = 0
```

The meaning of the *global* flag `g` varies. It extends the effects of `match` and `replace` to include all matches on the string (but not the matches of capture groups) and it is equivalent to the sticky flag for the `test` and `exec` methods of `RegExp`.

Capture Groups Parentheses in regexes not only change operator precedence (e.g., `(ab)*` matches any number of repetitions of the string `"ab"` while `ab*` matches the character `"a"` followed by any number of repetitions of the character `"b"`) but also create *capture groups*. Capture groups are implicitly numbered from left to right by order of the opening parenthesis. For example, `/a|((b)*c)*d/` is numbered as `/a|(`^{1}`(`^{2}`b)*c)*d/`. Where only bracketing is required, a non-capturing group can be created by using the syntax `(?:...)`. For regexes, capture groups are important because the regex engine will record the *most recent* substring matched against each capture group. Capture groups can be referred to from within the expression using backreferences. The last matched substring for each capture group is also returned by some of the API methods. In JavaScript, the return values of `match` and `exec` are arrays, with the whole match at index 0 (the implicit capture group 0), and the last matched instance of the i^{th} capture group at index i. In the example above, `"bbbbcbcd".match(/a|((b)*c)*d/)` will evaluate to the array `["bbbbcbcd", "bc", "b"]`. The interpretation of the matched results can be subtle: if the contents of a capture group did not match, the corresponding entry will be **`undefined`**. For example, `"a".match(/a(b)?/)` evaluates to `["a", `**`undefined`**`]`. If the capture group matches the empty string, the entry is the empty string.

For example, `"a".match(/a(b?)/)`, evaluates to `["a", ""]`.

Backreferences A *backreference* in a regex refers to a numbered capture group and will match the most recent match of the capture group. In general, the addition of backreferences to regexes makes the accepted languages non-regular [3]. Inside quantifiers (Kleene star, Kleene plus, and other repetition operators), the string matched by the backreference can change across multiple matches. For example, the regex `/((a|b)\2)+/` can match the string `"aabb"`, with the backreference `\2` being matched twice: the first time, the capture group contains `"a"`, the second time it contains `"b"`. This logic applies recursively, and it is possible for backreferences to in turn be part of other capture groups.

Operator Evaluation Table 6.1 lists the regular expression operators of interest. Some operators can be rewritten into semantically equivalent expressions to reduce the number of cases to handle (shown in the **Rewriting** column). Regexes distinguish between *greedy* and *lazy* evaluation. Greedy operators consume as many characters as possible such that the entire regular expression still matches; lazy operators consume as few characters as possible. This distinction—called *matching precedence*—is unnecessary for classical regular languages, but does affect the assignment of capture groups and therefore backreferences. *Zero-length assertions* or *lookarounds* do not consume any characters but still restrict the accepted word, enforcing a language intersection, available through *lookahead* and *lookbehind*. Positive or negative lookaheads can contain any regex, including capture groups and backreferences. For example, the regex `/a(?=b)/` matches `"a"` in the string `"ab"` but not in the string `"ac"` since the `"a"` must be followed by a `"b"`, but this second character is not consumed. In ES6, *lookbehind* is only available through `\b` (word boundary), and `\B` (non-word boundary), which are commonly used to only (or never) match whole words in

Table 6.1: Regular expression operators, separated by classes of precedence.

Operator	Name	Rewriting
`(r)`	Capturing parentheses	
`\n`	Backreference	
`(?:r)`	Non-capturing parentheses	
`(?=r)`	Positive lookahead	
`(?!r)`	Negative lookahead	
`\b`	Word boundary	
`\B`	Non-word boundary	
$r*$	Kleene star	
$r*?$	Lazy Kleene star	
$r+$	Kleene plus	$r*r$
$r+?$	Lazy Kleene plus	$r*?r$
$r\{m,n\}$	Repetition	$r^n \mid \ldots \mid r^m$
$r\{m,n\}?$	Lazy repetition	$r^m \mid \ldots \mid r^n$
$r?$	Optional	$r \mid \epsilon$
$r??$	Lazy optional	$\epsilon \mid r$
$r_1 r_2$	Concatenation	
$r_1 \mid r_2$	Alternation	

a string. For example, the regex `/\bexample\b/` matches the whole word `"example"` in the string `"my example string"`, but not in the string `"my examples are good"` since the substring `"example"` is followed by an `"s"` rather than a word boundary character.

6.2 Approach

We describe the approach; first, we define the word problem for regex and how it arises in DSE. We introduce our model for regex by example and explain how to eliminate spurious solutions by refinement.

The word problem and capturing languages. For any given classical regular expression r, $w \in \mathcal{L}(r)$ means w is a word within the (regular) language generated by r. For a regex R, we also need to record values of capture groups within the regex. To this end, we introduce the definition:

Definition 6.1 (Capturing Language)**.** The *capturing language* of a regex R, denoted $\mathcal{L}_c(R)$, is the set of tuples $(w, C_0, \ldots, C_n)$ such that w is a word of the language of R and each $C_0, \ldots, C_n$ is the substring of w matched by the corresponding numbered capture group in R.

A word w is therefore matched by a regex R if and only if $\exists C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \in \mathcal{L}_c(R)$. It is not matched if and only if $\forall C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R)$. For readability, we will usually omit quantifiers for capture variables where they are clear from the context.

Regex in DSE. The code in Listing 6.1 parses numeric arguments between XML tags from its input variable `args`, an array of strings. The regex in line 4 breaks each argument into two capture groups, the tag and the numeric value (`parts[0]` is the entire match). When the tag is `"timeout"`, it sets the `timeout` value accordingly (line 7). On line 11, a runtime assertion checks that the timeout value is truly numeric after the arguments have been processed. The assertion can fail because the program contains a bug: the regex (line 4) uses a Kleene star and therefore also admits the empty string as the number to set, and JavaScript's dynamic type system will allow setting `timeout` to `""`.

DSE finds such bugs by systematically enumerating paths, including the failure branches of assertions [41]. Starting from a concrete run with input, say, `args[0] = "foo"`, the DSE engine will attempt to build a *path condition* that encodes the branching decisions in terms of the input values. It then attempts to systematically flip clauses in the path condition and query an SMT solver to obtain input assignments covering different paths.

```
1  let timeout = '500';
2  for (let i = 0; i < args.length; i++) {
3    let arg = args[i];
4    let parts = /<(\w+)>([0-9]*)<\/\1>/.exec(arg);
5    if (parts) {
6      if (parts[1] === "timeout") {
7        timeout = parts[2];
8      }
9    }
10 }
11 assert(/^[0-9]+$/.test(timeout) == true);
```

Listing 6.1: Using complex regex features to match an XML tag.

This process repeats forever or until all paths are covered (this program has an unbounded number of paths as loops over an input string).

Without support for regex, the DSE engine will *concretize* `arg` on the call to `exec`, assigning the concrete result to `parts`. With all subsequent decisions therefore concrete, the path condition becomes $pc =$ **`true`** and the engine will be unable to cover more paths and find the bug.

Implementing regex support ensures that `parts` is *symbolic*, i.e., its elements are represented as formulas during symbolic execution. The path condition for the initial path thus becomes $pc = (\texttt{args[0]}, \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2) \notin \mathcal{L}_c(\mathrm{R})$ where $\mathrm{R} = \texttt{<(\w+)>([0-9]*)<\/\1>}$. Negating the clause and solving yields, e.g., `args[0]` = `"<a>0</a>"`. DSE then uses this input assignment to cover a second path with $pc = (\texttt{args[0]}, \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2) \in \mathcal{L}_c(\mathrm{R}) \wedge \mathcal{C}_1 \neq$ `"timeout"`. Negating the last clause yields, e.g., "`<timeout>0</timeout>`", entering line 7 and making `timeout` and therefore the assertion symbolic. This leads to $\mathrm{pc} = (\texttt{args[0]}, \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2) \in \mathcal{L}_c(\mathrm{R}) \wedge \mathcal{C}_1 =$ `"timeout"` $\wedge$ $(\mathcal{C}_2, \mathcal{C}_0') \in \mathcal{L}_c(\texttt{^[0-9]+\$})$, which, after negating the last clause, triggers the bug with the input "`<timeout></timeout>`".

Modeling capturing language membership. We model capturing language membership constraints in the path condition in terms of classical regular language membership and string constraints since they cannot be directly expressed in SMT. For a given ES6 regex R, we rewrite R according to Table 6.1. For consistency with the JavaScript API, we also introduce an outer capture group $\mathcal{C}_0$. Consider the regex R = `(?:a|(b))\1`. After preprocessing, the capturing language membership problem becomes

$$(w, \mathcal{C}_0, \mathcal{C}_1) \in \mathcal{L}_c(\texttt{(?:.|\n)*?((?:a|(b))\1)(?:.|\n)*?}),$$

a generic rewriting that allows for characters to precede and follow the match in the absence of anchors (the reason for the particular form of rewriting is described in Section 6.3). We recursively reduce capturing language membership to regular membership. To begin, we translate the purely regular Kleene stars and the outer capture group to obtain

$$\begin{aligned}(w, \mathcal{C}_0, \mathcal{C}_1) \in \mathcal{L}_c(\mathrm{R}) \implies\ & w = w_1 +\!+ w_2 +\!+ w_3 \wedge w_1 \in \mathcal{L}(\texttt{(:?.|\n)*?}) \\ & \wedge (w_2, \mathcal{C}_1) \in \mathcal{L}_c(\texttt{(?:a|(b))\1}) \wedge\ \mathcal{C}_0 = w_2 \\ & \wedge w_3 \in \mathcal{L}(\texttt{(:?.|\n)*?}),\end{aligned}$$

where $+\!+$ is string concatenation. We continue by decomposing the regex until there are only purely regular terms or standard string constraints. Next, we translate the nested capturing language constraint

$$\begin{aligned}(w_2, \mathrm{C}_1) \in \mathcal{L}_c(\texttt{(?:a|(b))\1}) \implies & \\ & w_2 = w_1' +\!+ w_2' \wedge (w_1', \mathcal{C}_1) \in \mathcal{L}_c(\texttt{a|(b)}) \wedge (w_2') \in \mathcal{L}_c(\texttt{\1}).\end{aligned}$$

To treat alternation, either the left is satisfied and the capture is undefined (denoted $\varnothing$), or the right is satisfied and the capture is locked to the match:

$$(w_1' \in \mathcal{L}(\mathtt{a}) \wedge \mathcal{C}_1 = \varnothing) \vee (w_1' \in \mathcal{L}(\mathtt{b}) \wedge \mathcal{C}_1 = w_1').$$

Finally we model the backreference, which is case dependent on whether the capture group it refers to is defined or not:

$$(\mathcal{C}_1 = \varnothing \implies w_2' = \epsilon) \wedge (\mathcal{C}_1 \neq \varnothing \implies w_2' = \mathcal{C}_1).$$

Putting this together, we obtain a model for R:

$$\begin{aligned}(w, \mathcal{C}_0, \mathcal{C}_1) \in \mathcal{L}_c(\mathtt{R}) \implies\ & w = w_1 \mathbin{+\!\!+} w_1' \mathbin{+\!\!+} w_2' \mathbin{+\!\!+} w_3 \wedge \mathcal{C}_0 = w_1' \mathbin{+\!\!+} w_2' \\ & \wedge \big((w_1' \in \mathcal{L}(\mathtt{a}) \wedge \mathcal{C}_1 = \varnothing) \vee (w_1' \in \mathcal{L}(\mathtt{b}) \wedge \mathcal{C}_1 = w_1')\big) \\ & \wedge (\mathcal{C}_1 = \varnothing \implies w_2' = \epsilon) \wedge (\mathcal{C}_1 \neq \varnothing \implies w_2' = \mathcal{C}_1) \\ & \wedge w_1 \in \mathcal{L}(\texttt{(:?.|\n)*?}) \wedge w_3 \in \mathcal{L}(\texttt{(:?.|\n)*?}).\end{aligned}$$

Refinement. Because of matching precedence (greediness), these models permit assignments to capture groups that are impossible in real executions. For example, we model `/^a*(a)?$/` as

$$\begin{aligned}(w, \mathcal{C}_0, \mathcal{C}_1) \in \mathcal{L}_c(\texttt{/\^a*(a)?\$/}) \implies\ & w = w_1 \mathbin{+\!\!+} w_2 \\ & \wedge w_1 \in \mathcal{L}(\texttt{a*}) \wedge w_2 \in \mathcal{L}(\texttt{a}|\epsilon) \wedge \mathcal{C}_0 = w \wedge \mathcal{C}_1 = w_2.\end{aligned}$$

This allows $\mathcal{C}_1$ to be either `a` or the empty string ϵ, i.e., we permit spurious members of the capturing language under our model, such as the tuple `("aa", "aa", "a")`. Because `a*` is *greedy*, it will always consume both characters in the string `"aa"`; therefore, `(a)?` can only match ϵ. This problem posed by *greedy* and *lazy* operator semantics remains unaddressed by previous work [90, 104, 91]. To address this, we use a counterexample-guided

abstraction refinement scheme that validates candidate assignments with an ES6-compliant matcher. Continuing the example, the candidate element (`"aa"`, `"aa"`, `"a"`) is validated by running a concrete matcher on the string `"aa"`, which contradicts the candidate captures with $\mathcal{C}_0 =$ `"aa"` and $\mathcal{C}_1 = \epsilon$. The model is refined with the counter-example to the following:

$$w = w_1 +\!+ w_2 \wedge w_1 \in \mathcal{L}(\texttt{a}\star) \wedge w_2 \in \mathcal{L}(\texttt{a}|\epsilon) \wedge \mathcal{C}_0 = w \wedge \mathcal{C}_1 = w_2$$
$$\wedge \big(w = \texttt{"aa"} \implies (\mathcal{C}_0 = \texttt{"aa"} \wedge \mathcal{C}_1 = \epsilon)\big).$$

We then generate and validate a new candidate $(w, \mathcal{C}_0, \mathcal{C}_1)$ and repeat the refinement until a satisfying assignment passes the concrete matcher.

6.3 Modeling ES6 Regex

We detail the modeling of the capturing language of a given regex R. First, we preprocess R into an equivalent regex R′ (Section 6.3.1). Next, we model constraints $(w, \mathcal{C}_0, \ldots, \mathcal{C}_n) \in \mathcal{L}_c(\mathtt{R}')$ by recursively translating terms in the abstract syntax tree (AST) of R′ to regular language membership and string constraints (Section 6.3.2-6.3.3). Finally, we model negated constraints $(w, \mathcal{C}_0, \ldots, \mathcal{C}_n) \notin \mathcal{L}_c(\mathtt{R}')$ (Section 6.3.4).

6.3.1 Preprocessing

Rewritable operators. We make the concatenation of $\mathtt{R}_1$ and $\mathtt{R}_2$ explicit as the binary operator $\mathtt{R}_1 \cdot \mathtt{R}_2$. Any regex can then be split into combinations of atomic elements, capture groups and backreferences (referred to collectively as *terms*, per ES6's specification [31]), joined by explicit operators. Using the rules in Table 6.1, we rewrite any R to an equivalent regex R′ containing only alternation, concatenation, Kleene star, capture groups, non-capturing parentheses, lookarounds, and backreferences. We rewrite

lazy quantifiers to their greedy equivalents, as our models are agnostic to matching precedence (this is dealt with in refinement).

Renumbering capture groups. Rewrite rules for Kleene plus and repetition duplicate capture groups, e.g., rewriting `/(a){1,2}/` to `/(a)(a)|(a)/` adds two capture groups. We therefore explicitly relate capture groups between the original and rewritten expressions. The rewriting of a Kleene plus expression, $S+$ containing K capture groups ($S{*}S$), has $2K$ capture groups. For a constraint of the form $(\mathcal{C}_1, \ldots, \mathcal{C}_K) \in \mathcal{L}_c(S+)$, the rewriting yields $(\mathcal{C}_0, \mathcal{C}_{1,1}, \ldots, \mathcal{C}_{K,1}, \mathcal{C}_{1,2}, \ldots, \mathcal{C}_{K,2}) \in \mathcal{L}_c(S{*}S)$. As $S{*}S$ contains two copies of S, $\mathcal{C}_{i,j}$ corresponds to the i^{th} capture in the j^{th} copy of S in $S{*}S$. We express this correspondence between captures as

$$(w, \mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_K) \in \mathcal{L}_c(S+) \iff$$
$$(w, \mathcal{C}_0, \mathcal{C}_{1,1}, \ldots, \mathcal{C}_{K,1}, \mathcal{C}_{1,2}, \ldots, \mathcal{C}_{K,2}) \in \mathcal{L}_c(S{*}S) \wedge \forall i \in \{1, \ldots, K\},\ \mathcal{C}_i = \mathcal{C}_{i,2}.$$

If $S\{m, n\}$ has K capture groups, then $S' = S^n \mid \ldots \mid S^m$ has $\frac{K}{2}(n+m)(n-m+1)$ captures. In S', suppose we index our captures as $\mathcal{C}_{i,j,k}$ where $i \in \{1, \ldots, K\}$ is the index of the capture group in S, $j \in \{0, \ldots, n-m\}$ denotes which alternate the capture group is in (0 being the rightmost), and $k \in \{0, \ldots, m+j-1\}$ indexes the copies of S within each alternate. Intuitively, we pick a single $x \in \{0, \ldots, n-m\}$ that corresponds to the first satisfied alternate. Comparing the assignment of captures in $S\{m, n\}$ to S', we know that the value of the capture is the last possible match, so $\mathcal{C}_i = \mathcal{C}_{i,x,m+x-1}$ for all $i \in \{1, \ldots, K\}$. Formally, this direct correspondence can be expressed

as

$$
\begin{aligned}
&(w,\mathcal{C}_0,\mathcal{C}_1,\ldots,\mathcal{C}_K) \in \mathcal{L}_c(S\{m,n\}) \iff \\
&\quad (w,\mathcal{C}_0,\mathcal{C}_{1,0,0},\ldots,\mathcal{C}_{K,n-m,n}) \in \mathcal{L}_c(S^n \mid \ldots \mid S^m) \\
&\quad \wedge \exists x \in \{0,\ldots,n-m\} : \big((w,\mathcal{C}_0,\mathcal{C}_{1,x,0},\ldots,\mathcal{C}_{K,x,m+x-1}) \in \mathcal{L}_c(S^{m+x}) \\
&\quad\ \wedge \forall x' > x,\ (w,\mathcal{C}_0,\mathcal{C}_{1,x',0},\ldots,\mathcal{C}_{K,x',m+x'-1}) \notin \mathcal{L}_c(S^{m+x'}) \\
&\quad\ \wedge \forall i \in \{1,\ldots,K\},\ \mathcal{C}_i = \mathcal{C}_{i,x,m+x-1}\big).
\end{aligned}
$$

6.3.2 Operators and Capture Groups

Let t be the next term to process in the AST of R'. If t is capture-free and purely regular, there is nothing to do in this step. If t is non-regular, it contains capture groups numbered i through $i+k$. At each recursive step, we express membership of the capturing language $(w,\mathcal{C}_i,...,\mathcal{C}_{i+k}) \in \mathcal{L}_c(t)$ through a model consisting of string and regular language membership constraints, and a set of remaining capturing language membership constraints for subterms of t. We record the locations of capture groups within the regex during preprocessing. When splitting t into subterms t_1 and t_2, capture groups $\mathcal{C}_i,\ldots,\mathcal{C}_{i+j}$ are contained in t_1 and $\mathcal{C}_{i+j+1},\ldots,\mathcal{C}_{i+k}$ are contained in t_2 for some j. Models for individual operations are given in Table 6.2.

When matching an alternation $|$, capture groups on the non-matching side will be undefined, denoted by $\varnothing$, which is distinct from the empty string ϵ. When modeling quantification $t = t_1*$, we assume t_1 does not contain backreferences: we model t via the expression $\widehat{t_1}*t_1 \mid \epsilon$, where $\widehat{t_1}$ is a regular expression corresponding to t_1, except each set of capturing parentheses is rewritten as a set of non-capturing parentheses. In this way, $\widehat{t_1}$ is regular (it is backreference-free by assumption). However, $\widehat{t_1}*t_1 \mid \epsilon$ is not semantically equivalent to t: if possible, capturing groups must be

Table 6.2: Models for regex operators.

Operation	t	**Overapproximate Model for** $(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t)$
Alternation	$t_1 \mid t_2$	$\big((w, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1) \wedge \mathcal{C}_{i+j+1} = ... = \mathcal{C}_{i+k} = \varnothing\big)$ $\vee\big((w, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2) \wedge \mathcal{C}_i = ... = \mathcal{C}_{i+j} = \varnothing\big)$
Concatenation	$t_1 \cdot t_2$	$w = w_1 +\!+ w_2 \wedge (w_1, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1)$ $\wedge\, (w_2, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$
Backreference-free Quantification	t_1*	$w = w_1 +\!+ w_2$ $\wedge\, w_1 \in \mathcal{L}(\widehat{t_1*}) \wedge (w_2, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1 \mid \epsilon)$ $\wedge\, \big(w_2 = \epsilon \implies (w_1 = \epsilon \wedge \mathcal{C}_i = \ldots = \mathcal{C}_{i+k} = \varnothing)\big)$
Positive Lookahead	$(?{=}t_1)\, t_2$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1 .*)$ $\wedge (w, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$
Negative Lookahead	$(!{=}t_1)\, t_2$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \notin \mathcal{L}_c(t_1 .*)$ $\wedge (w, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$
Input Start	$t_1\hat{}$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(.* \langle)$
Input Start (Multiline)	$t_1\hat{}$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(.* \langle \mid \backslash n)$
Input End	$\$t_1$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(\rangle .*)$
Input End (Multiline)	$\$t_1$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}(\rangle \mid \backslash n .*)$
Word Boundary	$t_1 \backslash b\ t_2$	$w = w_1 +\!+ w_2 \wedge (w_1, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1)$ $\wedge\, (w_2, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$ $\wedge\Big(\big((w_1 \in \mathcal{L}(.*\backslash W) \vee w_1 = \epsilon) \wedge w_2 \in \mathcal{L}(\backslash w .*)\big)$ $\vee\big(w_1 \in \mathcal{L}(.*\backslash w) \wedge (w_2 \in \mathcal{L}(\backslash W .*) \vee w_2 = \epsilon)\big)\Big)$
Non-Word Boundary	$t_1 \backslash B\ t_2$	$w = w_1 +\!+ w_2 \wedge (w_1, \mathcal{C}_i, ..., \mathcal{C}_{i+j}) \in \mathcal{L}_c(t_1)$ $\wedge\, (w_2, \mathcal{C}_{i+j+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_2)$ $\wedge\big((w_1 \notin \mathcal{L}(.*\backslash W) \wedge w_1 \neq \epsilon) \vee w_2 \notin \mathcal{L}(\backslash w .*)\big)$ $\wedge\big(w_1 \notin \mathcal{L}(.*\backslash w) \vee (w_2 \notin \mathcal{L}(\backslash W .*) \wedge w_2 \neq \epsilon)\big)$
Capture Group	(t_1)	$(w, \mathcal{C}_{i+1}, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1) \wedge \mathcal{C}_i = w$
Non-Capturing Group	$(?{:}t_1)$	$(w, \mathcal{C}_i, ..., \mathcal{C}_{i+k}) \in \mathcal{L}_c(t_1)$
Base Case	t regular	$w \in \mathcal{L}(t)$

satisfied, so $\hat{t_1}*$ cannot consume all matches of the expression. We encode this constraint with the implication that $\hat{t_1}*$ must match the empty string whenever $t_1 \mid \epsilon$ does.

Lookahead constrains the word to be a member of the languages of both the assertion expression and t_2. The word boundary `\b` is effectively a single-character lookaround for word and non-word characters. Since the boundary can occur both ways, disjunction allows the end of w_1 and the start of w_2 to be word and non-word, or non-word and word characters, respectively. The non-word boundary `\B` is defined as the dual of `\b`.

For capture groups, we bind the next capture variable $\mathcal{C}_i$ to the string matched by t_1. The i^{th} capture group must be the outer capture and the remaining captures $\mathcal{C}_{i+1}, \ldots, \mathcal{C}_{i+k}$ must therefore be contained within t_1. There is nothing to be done for non-capturing groups and recursion continues on the contained subexpression.

Anchors assert the start (`^`) and end (`$`) of input; we represent the beginning and end of a word via the meta-characters $\langle$ and $\rangle$, respectively. In most instances when handling these operations, t_1 will be ϵ; this is because it is rare to have regex operators prior to those marking the start of input (or after marking the end of input, respectively). In both these cases, we assert that the language defines the start or end of input—and that as a result of this, the language of t_1 must be an empty word, though the capture groups may be defined (say through t_1 containing assertions with nested captures). We give separate rules for matching a regular expression with the multiline flag set, which modify the behavior of anchors to accept either our meta-characters or a line break.

6.3.3 Backreferences

Table 6.3 describes the different cases of backreferences in the AST of regex R; `\k` is a backreference to the k^{th} capture group of R. Intuitively, each in-

Table 6.3: Modeling backreferences.

Type of $\backslash k$	**Capturing Language**	**Approximation**	**Model**
Empty	$(w) \in \mathcal{L}_c(\backslash k)$	Exact	$w = \epsilon$
Immutable	$(w) \in \mathcal{L}_c(\backslash k)$	Overapproximate	$(\mathcal{C}_k = \varnothing \implies w = \epsilon)$ $\wedge (\mathcal{C}_k \neq \varnothing \implies w = \mathcal{C}_k)$
Immutable	$(w) \in \mathcal{L}_c(\backslash k\star)$	Overapproximate	$(\mathcal{C}_k = \varnothing \implies w = \epsilon) \wedge (\mathcal{C}_k \neq \varnothing \implies \exists m \geq 0 : w = {+\!+}_{i=0}^{m} \mathcal{C}_k)$
Mutable	$(w,\mathcal{C}_k) \in \mathcal{L}_c(\texttt{(?:}(t_1)\backslash k\texttt{)}\star)$ t_1 is capture group-free	Overapproximate	$(w = \epsilon \wedge \mathcal{C}_k = \varnothing)$ $\vee (\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge\, \mathcal{C}_k = \mathcal{C}_{k,m} \wedge \forall i > 1,$ $((\sigma_{i,1}, \mathcal{C}_{k,i}) \in \mathcal{L}_c(t_1) \wedge \sigma_{i,2} = \mathcal{C}_{k,i}))$
Mutable	$(w,\mathcal{C}_k) \in \mathcal{L}_c(\texttt{(?:}(t_1)\backslash k\texttt{)}\star)$ t_1 is capture group-free	Unsound	$(w = \epsilon \wedge \mathcal{C}_k = \varnothing)$ $\vee (\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge\, (\sigma_{i,1}, \mathcal{C}_k) \in \mathcal{L}_c(t_1)$ $\wedge \forall i \geq 1, (\sigma_{i,1} = \sigma_{1,1} \wedge \sigma_{i,2} = \sigma_{1,1}))$

stance of a backreference is a variable that refers to a capture group and has a type that depends on the structure of R. We call a backreference *immutable* if it can only evaluate to a single value when matching; it is *mutable* if it can take on multiple values, which is a rare but particularly tricky case. For example, consider `/((a|b)\2)+\1\2/`. Here, the backreference `\1` and the second instance of `\2` are immutable. However, the first instance of `\2` is mutable: each repetition of the outer capture group under the Kleene plus can change the value of the second (inner) capture group, in turn changing the value of the backreference inside this quantification. For example, the string `"aabbaabbbbb"` satisfies this regex, but `"aabaaabaa"` does not. To fully characterize these distinctions, we introduce the following definition:

Definition 6.2 (Backreference Type)**.** Let t be the k^{th} capture group of a regex R; then

(i) \k is *empty* if either k is greater than the number of capture groups in R, or \k is encountered before t in a post-order traversal of the AST of R;

(ii) \k is *mutable* if \k is not empty, and both t and \k are subterms of some quantified term Q in R;

(iii) otherwise, \k is *immutable*.

Empty backreferences are modeled as ϵ: they refer either to a capture group which is a superterm, e.g., `/(a\1)/`, or appears later, e.g., `/\1(a)/`.

There are two cases for immutable backreferences. In the first, the backreference is not quantified: $\mathcal{C}_k$ has already been modeled with an equality constraint, so we bind the backreference to it. In the second, the backreference occurs within quantification; the matched word is a finite concatenation of identical copies of the capture group. Both models cover the corner case where the capture group is $\varnothing$ due to alternation or an empty Kleene star; following the standard, the backreference evaluates to ϵ.

Mutable backreferences appear as $(...t_1...\backslash k...)*$ where t_1 is the k^{th} capture group. ES6 does not support forward referencing of backreferences: in $(...\backslash k...t_1...)*$, \k is empty. Table 6.3 describes the simplest case, other patterns are straightforward generalizations. We assume t_1 is the k^{th} capture group but is otherwise capture group-free. We treat the entire term at once: words in the language are either ϵ, or for some number of iterations, we have the concatenation of a word in the language of t_1 followed by a copy of it. We introduce variables $\mathcal{C}_{k,i}$ referring to the values of the capture group in each iteration, encoding the repeated matching on the string until the final value for $\mathcal{C}_k$. We need not deal with the possibility that any $\mathcal{C}_{k,i}$ is $\varnothing$, since the quantification ends as soon as t_1 does not match.

Unfortunately, constraints for mutable references generated from this model are hard to solve and infeasible for current SMT solvers: they re-

quire "guessing" a partition of the matched string variable into individual and varying components. To make solving such queries practical, the last row of Table 6.3 describes an unsound alternative to the previous rule, where we treat quantified backreferences as immutable. For example, returning to `/((a|b)\2)+\1\2/`, we accept (`"aaaaaaaaa"`, `"aaaaaaaaa"`, `"aa"`, `"a"`), but not the tuple (`"aabbaabbbbb"`, `"aabbaabbbbb"`, `"bb"`, `"b"`).

6.3.4 Modeling Non-membership

We define an analogous model for non-membership of the form $\forall \mathcal{C}_0, \ldots, \mathcal{C}_n : (w, \mathcal{C}_0, \ldots, \mathcal{C}_n) \notin \mathcal{L}_c(R)$. Intuitively, non-membership models assert that for all capture group assignments there exists some partition of the word such that one of the individual constraints is violated. Most models are simply negated. In concatenation and quantification, only language and emptiness constraints are negated, so the models take the form

$$w = w_1 +\!+ w_2 \wedge \big(\ldots \notin \mathcal{L}_c(\ldots) \vee \ldots \notin \mathcal{L}_c(\ldots) \vee (w_2 = \epsilon \wedge \neg(w_1 = \epsilon \ldots)) \big).$$

In the same manner, the model for capture groups is

$$(w, \mathcal{C}_{i+1}, ..., \mathcal{C}_{i+k}) \notin \mathcal{L}_c(t_1) \wedge \mathcal{C}_i = w.$$

Returning to our example in Section 6.2, the negated model for $\forall \mathcal{C}_0, \mathcal{C}_1 : (w, \mathcal{C}_0, \mathcal{C}_1) \notin \mathcal{L}_c(\texttt{(?:a|(b))\1})$ becomes

$$\begin{aligned} &\forall \mathcal{C}_0, \mathcal{C}_1 : w = w_1 +\!+ w_1' +\!+ w_2' +\!+ w_3 \wedge \mathcal{C}_0 = w_1' +\!+ w_2' \\ &\wedge \big(\neg \big((w_1' \in \mathcal{L}(a) \wedge \mathcal{C}_1 = \varnothing) \vee (w_1' \in \mathcal{L}(b) \wedge \mathcal{C}_1 = w_1') \big) \\ &\quad \vee \neg(\mathcal{C}_1 = \varnothing \implies w_2' = \epsilon) \vee \neg(\mathcal{C}_1 \neq \varnothing \implies w_2' = \mathcal{C}_1) \\ &\quad \vee w_1 \notin \mathcal{L}(\texttt{(:?.|\n)*?}) \vee w_3 \notin \mathcal{L}(\texttt{(:?.|\n)*?}) \big). \end{aligned}$$

6.4 Matching Precedence Refinement

We explain the matching precedence problem (Section 6.4.1) and address it through a counterexample-guided abstraction refinement (CEGAR) scheme (Section 6.4.1). We discuss termination of the scheme (Section 6.4.2) and the overall soundness of our approach (Section 6.4.3).

6.4.1 Matching Precedence

The model in Tables 6.2 and 6.3 does not account for matching precedence. A standards-compliant ES6 regex matcher will derive a unique set of capture group assignments when matching a string w, because matching precedence dictates that greedy (non-greedy) expressions match as many (as few) characters as possible before moving on to the next [31]. These requirements are not part of our model, as encoding them directly into SMT would require nesting of quantifiers for each operator, making even the simplest examples infeasible for automated solving. Since we seek to model ES6 regex in way amenable to automated solving, even at the cost of soundness in the case of mutable backreferences, we use an external refinement scheme to solve the problem of matching precedence rather than bake this into our model.

CEGAR for ES6 regex models We eliminate infeasible elements of the capturing language admitted by our model through a scheme of counter example-guided abstraction refinement (CEGAR). Algorithm 2 is a CEGAR-based satisfiability checker for constraints modeled from ES6 regexes, which relies on an external SMT solver with classical regular expression and string support and an ES6-compliant regex matcher. The algorithm takes an SMT problem P (derived from the DSE path condition) as a conjunction of constraints, some of which model the $m \geq 0$ original capturing lan-

Algorithm 2: Counterexample-guided abstraction refinement scheme for matching precedence.

Input : Constraint problem P including models for $\mathfrak{m}$ constraints $(w_j, \mathcal{C}_{0,j}, \ldots, \mathcal{C}_{n_j,j}) \boxdot_j \mathcal{L}_c(\mathsf{R}_j)$.

Output: `null` if P is unsatisfiable, or a satisfying assignment for P otherwise

1 $M :=$ **`null`**;
2 *Failed* := **`false`**;
3 **do**
4 $\quad M :=$ `Solve`(P);
5 $\quad$ **if** $M =$ ***`null`*** **then**
6 $\quad\quad$ **return** ***`null`***;
7 $\quad$ *Failed* := **`false`**;
8 $\quad$ **for** $j := 0$ **to** $\mathfrak{m} - 1$ **do**
9 $\quad\quad (C^{\natural}_{0,j}, \ldots, C^{\natural}_{n_j,j}) :=$ `ConcreteMatch`$(M[w_j], \mathsf{R}_j)$;
10 $\quad\quad$ **if** $(C^{\natural}_{0,j}, \ldots, C^{\natural}_{n_j,j})$ **then**
11 $\quad\quad\quad$ **if** $\boxdot_j = \in$ **then**
12 $\quad\quad\quad\quad$ **for** $i := 0$ **to** n_j **do**
13 $\quad\quad\quad\quad\quad$ **if** $C^{\natural}_{i,j} \neq M[\mathcal{C}_{i,j}]$ **then**
14 $\quad\quad\quad\quad\quad\quad$ *Failed* := **`true`**;
15 $\quad\quad\quad\quad\quad\quad \mathsf{P} := \mathsf{P} \wedge (w_j = M[w_j] \implies \bigwedge_{0 \leq i \leq n_j} \mathcal{C}_{i,j} = C^{\natural}_{i,j})$;
16 $\quad\quad\quad\quad\quad$ **else** `// Non-membership query`
17 $\quad\quad\quad\quad\quad\quad$ *Failed* := **`true`**;
18 $\quad\quad\quad\quad\quad\quad \mathsf{P} := \mathsf{P} \wedge (w_j \neq M[w_j])$;
19 $\quad\quad\quad$ **else** `// No concrete match`
20 $\quad\quad\quad\quad$ **if** $\boxdot_j = \in$ **then**
21 $\quad\quad\quad\quad\quad$ *Failed* := **`true`**;
22 $\quad\quad\quad\quad\quad \mathsf{P} := \mathsf{P} \wedge (w_j \neq M[w_j])$;
23 **while** *Failed*;
24 **return** M;

guage membership constraints. We number the problem's original capturing language constraints $0 \leq j < m$ so that we can refer to them as $(w_j, \mathcal{C}_{0,j}, \ldots, \mathcal{C}_{n_j,j}) \boxdot_j \mathcal{L}_c(R_j)$, where $\boxdot \in \{\in, \notin\}$. The algorithm returns `null` if P is unsatisfiable, or a satisfying assignment with correct matching precedence.

In a loop, we first pass the problem P to an external SMT solver. The solver returns a satisfying assignment M or `null` if the problem is unsatisfiable, in which case we are done (lines 4–6). If M is not null, the algorithm uses a concrete regular expression matcher (e.g., Node.js's built-in matcher) to populate concrete capture variables $\mathcal{C}^{\natural}_{i,j}$ corresponding to the words w_j in M.

Lines 8–22 describe how the assignments of capture groups are checked for each regular expression R_j in the original problem P. We first check whether the concrete matcher returned a list of valid capture group assignments, i.e., whether the word $M[w_j]$ from the satisfying assignment matches concretely. If it did, then w_j is a member of the language generated by R_j. If $\boxdot_j = \in$, i.e., the membership constraint was positive, then we must check if the capture group assignments are consistent with those from M (line 13). If they are, we move on to the next regex, otherwise we refine the constraint problem by fixing capture group assignments to their concrete values for the matched word (line 15). Dually, if a modeled non-membership constraint was satisfiable but the word from the current satisfying assignment $M[w_j]$ did match concretely, we refine the problem by asserting that w must not equal that word (line 18). We do the same if $M[w_j]$ did not match concretely but came from a satisfied positive membership constraint (line 22).

If no refinement was necessary we have confirmed the overall assignment satisfies P and return M (line 24). Otherwise, the loop continues with solving the refined problem.

6.4.2 Termination of the Scheme

Unsurprisingly, CEGAR may require arbitrarily many refinements on pathological formulas and never terminate. This is unavoidable due to undecidability [17]. In practice, we therefore impose a limit on the number of refinements, leading to *unknown* as a possible third result. SMT solvers already may timeout or report *unknown* for complex string formulas, so this does not lead to additional problems in practice.

6.4.3 Soundness of the Model

When constructing the models in Tables 6.2 and 6.3, we followed the regex semantics laid out in the ES6 standards document [31]. The ES6 standard is written in a semi-formal fashion, so we are confident that our translation into logic is accurate, but cannot have formal proof. Existing attempts to encode ECMAScript semantics into logic such as JSIL [20] or KJS [77] do not include regexes.

With the exception of the optimized rule for mutable backreferences, our models are overapproximate, because they ignore matching precedence. When the CEGAR loop terminates, any spurious solutions from overapproximation are eliminated. As a result, we have an *exact* procedure to decide (non)-membership for capturing languages of ES6 regexes without quantified backreferences. In the presence of quantified backreferences, the model after CEGAR termination becomes *underapproximate*. Since DSE itself is an underapproximate program analysis (due to concretization, solver timeouts, and partial exploration), our model and refinement strategy are *sound for DSE*.

6.5 Modeling the ES6 Regex API

The ES6 standard specifies several methods that evaluate regexes [31]. We follow its specified pseudocode for `RegExp.exec(s)` to implement matching and capture group assignment in terms of capturing language membership in Algorithm 3. Notably, our algorithm implements support for all flags and operators specified for ES6. `RegExp.test(s)` is precisely equivalent to the expression `RegExp.exec(s) !==` **`undefined`**. In the same manner, one can construct models for other regex functions defined for ES6. Our implementation includes partial models for the remaining functions that allow effective test generation in practice but are not semantically complete.

Algorithm 3: `RegExp.exec(`*input*`)`

```
1  input' := '⟨' + input + '⟩';
2  if sticky or global then
3  |   offset := lastIndex > 0 ? lastIndex + 1 : 0;
4  |   input' := input'.substring(offset);
5  source' := '(:?.|\n)*?(' + source + ')(:?.|\n)*?';
6  if caseIgnore then
7  |   source' := rewriteForIgnoreCase(source');
8  if (input', C_0, ..., C_n) ∈ L_c(source') then
9  |   Remove ⟨ and ⟩ from (input', C_0, ..., C_n);
10 |   lastIndex := lastIndex + C_0.startIndex + C_0.length;
11 |   result := [C_0, ..., C_n];
12 |   result.input := input;
13 |   result.index := C_0.startIndex;
14 |   return result;
15 else
16 |   lastIndex := 0;
17 |   return undefined;
```

Algorithm 3 first processes flags to begin from the end of the previous

match for sticky or global flags, and it rewrites the regex to accept lower and upper case variants of characters for the ignore case flag. We introduce the ⟨ and ⟩ meta-characters to *input* which act as markers for the start and end of a string during matching. Next, if the sticky or global flags are set we slice *input* at `lastIndex` so that the new match begins from the end of the previous. Due to the introduction of our meta-characters the `lastIndex` needs to be offset by 1 if it is greater than zero. We then rewrite the regex source to allow for characters to precede and succeed the match. Note that we use `(?:.|n)*?` rather than `.*?` because the wildcard `.` consumes all characters except line breaks in ECMAScript regexes. To avoid adding these characters to the final match we place the original regex source inside a capture group. This forms $\mathcal{C}_0$, which is defined to be the whole matched string [31]. Once preprocessing is complete we test whether the input string and fresh string for each capture group are within the capturing language for the expression. If they are then a results object is created which returns the correctly mapped capture groups, the input string, and the start of the match in the string with the meta-characters removed. Otherwise `lastIndex` is reset and **`undefined`** is returned.

6.6 Consequences of this Model

A detailed empirical evaluation of the model described in Sections 6.3-6.5 is contained in the full paper detailing this work [61]. We report a brief summary of the results, demonstrating that the model allows deeper analysis of string-manipulating programs.

The evaluation of this model is performed within the context of the DSE engine ExpoSE (Section 2.4.2). The model was evaluated on a sample of 1,131 packages from Node.js' NPM package repository which manipulated strings through regexes. For each of these, ExpoSE automatically

Table 6.4: Contribution of different components of the model to testing 1,131 NPM packages, showing number (#) and fraction (%) of packages with line coverage improvements.

Regex Support Level	Improved #	Improved %
Concrete Regular Expressions	-	-
+ Modeling RegEx	528	46.68%
+ Captures & Backreferences	194	17.15%
+ Refinement	63	5.57%
All Features vs. Concrete	617	54.55%

generates a meaningful test harness by executing all exported methods of the library with symbolic arguments.

Each package was executed for one hour across four distinct levels of regex support, and the result is that supporting these complex features of regexes improves the line coverage of the analysis. These results are summarized in Table 6.4. As baseline, all regex methods are executed concretely, concretizing the arguments and results. Second, our model for ES6 regex and their methods is added without support for capture groups or backreferences. Third, full support for capture groups and backreferences is enabled. Fourth, the refinement scheme to address matching precedence is added. Explicitly, these results mean that after deploying our model we are able to solve path constraints involving strings which were previously unsolvable, allowing for deeper exploration of these programs.

6.7 Related Work

There have been several approaches for symbolic execution of JavaScript; most include some limited support for classical regular expressions. In theory, regex engines can be symbolically executed themselves through

the interpreter [21]. While this removes the need for modeling, in practice the symbolic execution of the entire interpreter and regex engine quickly becomes infeasible due to path explosion. Li *et al.* [57] presented an automated test generation scheme for programs with regular expressions by on-line generation of a matching function for each regular expression encountered, exacerbating path explosion. Saxena *et al.* [90] proposed the first scheme to encode capture groups through string constraints. Li *et al.* [55] describe a custom browser and symbolic execution engine for JavaScript and the browser DOM, and a string constraint solver *PASS* [56] with support for most JavaScript string operations. Although all of these approaches feature some support for ECMAScript regex (e.g., limited support for capture groups), they ignore matching precedence and do not support backreferences or lookaheads.

Thomé *et al.* [103] propose a heuristic approach for solving constraints involving unsupported string operations. We choose to model operations unsupported by the solver and employ a CEGAR scheme to ensure correctness. The use of a refinement scheme to solve complex constraint problems, including support for context-free languages, has been proposed in previous work [2]. The language of regular expressions with backreferences is not context-free [24] and, as such, their scheme does not suffice for encoding all regexes; however, their approach could serve as richer base theory than classic regular expressions. Scott *et al.* [91] suggest backreferences can be eliminated via concatenation constraints, however they do not present a method for doing so.

Further innovations from the string solving community, such as work on the decidability of string constraints involving complex functions [26, 47] or support for recursive string operations [105, 106], are likely to improve the performance of the approach described in this chapter in future. We incorporate our techniques at the level of the DSE engine rather than the constraint solver, which allows our tool to leverage advances in string

solving techniques; at the same time, we avoid integrating language-specific details for regular expressions into a generic solver.

7 Security Annotations for JavaScript

In this chapter we describe the design and implementation of Security Annotations in full JavaScript. Our implementation is based on source code instrumentation [92] and built on top of the dynamic symbolic execution engine ExpoSE [60, 61]. We first describe the implementation of Security Annotations, ExpoSE$_{SA}$ (Section 7.1). This implementation is guided by S5$_{SA}$, and, in Section 7.2 we discuss how we ensure faithfulness between this formal model and ExpoSE$_{SA}$. We then describe a strategy for using ExpoSE$_{SA}$ to test real JavaScript applications which use cryptography (Section 7.3). Section 7.4 details the expressiveness of ExpoSE$_{SA}$ through extending the case study of Section 5.7 to full JavaScript. Finally, we discuss the analysis of two real-world applications (Section 7.5).

7.1 Implementation

In this section we describe the implementation of Security Annotations on top of ExpoSE[1]. First, we explain the structure of Security Annotations themselves (Section 7.1.1). We discuss how Security Annotations are attached to values (Section 7.1.2); complications involving objects and their

[1]This implementation is available at: `https://github.com/ExpoSEJS/ExpoSE/tree/features/annotations`.

references (Section 7.1.3) and finally how Security Annotations are manipulated (Section 7.1.4).

7.1.1 Security Annotations

Within ExpoSE$_{SA}$, Security Annotations are implemented as JavaScript classes; each specific annotation extends a master `Annotation` class. Annotation hierarchies are induced by class inheritance: one class extends another if the annotation it represents is a subannotation of the other, i.e., if A_1 and A_2 are atomic annotations such that $A_1 \prec: A_2$, then the class representing A_1 extends the class representing A_2.

Each annotation class comes equipped with associated helper methods defined in the master `Annotation` class. These methods encode a procedure for deciding whether an atomic annotation, A_1, is a subannotation of an atomic annotation, A_2, through JavaScript's built-in `instanceof` operator. These methods also describe the construction of new annotations from old (i.e., by composition or `cut`). Finally, each annotation also possesses a specific property naming the annotation.

In order to use a Security Annotation in a program, one must construct a concrete instance of the class. This means there can be multiple instances of a single annotation, however, this does not present a problem. Annotation hierarchies are defined through the classes themselves rather than on any specific instances: in practice, two instances of the same annotation behave identically. Second, since annotations are only declared and manipulated within trusted APIs, we do not need to worry about developers constructing new instances of annotations.

In S5$_{SA}$, Security Annotations are directly added to an individual store in memory and can be recalled directly. Since ExpoSE$_{SA}$ is built through source code instrumentation, we do not directly control the memory of the application. Therefore, to use an annotation, we need a reference to

our concrete instance of it. In practice, this means that we must bind our concrete instances of annotations to variables. In our WebCrypto shim, names of these variables are defined as the capitalized annotation name, preceded by an underscore—e.g., `PrivKey` is represented by the variable `_PRIVKEY`. Within the case studies discussed in this chapter, we did not encounter problems with this naming scheme resulting in variable reuse.

In practice, this approach makes the construction of Security Annotations a little more complex than simply declaring **`SecAnn`** A in the manner of $S5_{SA}$. Instead, declaring two Security Annotations, A and B with $A \prec: B$, is achieved through:

```
var _A = SecAnn("A");
var _B = SecAnn("B", _A);
_A = new _A([]);
_B = new _B([]);
```

Note here that we must define the subannotation relationship prior to constructing the concrete instances of the annotations. Additionally, the **`SecAnn Extends`** syntax is replaced by simply providing an additional argument to **`SecAnn`**.

7.1.2 Attaching Annotations

Recall that in $S5_{SA}$ values are comprised of prevalues, equivalent to JavaScript values (e.g., numbers, strings and booleans) and Security Annotations. In our $ExpoSE_{SA}$, we recognize that the vast majority of values in a program will not possess security properties; therefore, insisting all values carry a corresponding annotation is overly expensive.

In $ExpoSE_{SA}$, we distinguish between standard JavaScript values and those with associated security properties. When a security property is associated to a value it is *wrapped*—constructing a pair of value and Security Annotation. Specifically, the first time a value is coerced to attach (or dis-

```
1 setAnnotations = function(v, ann) {
2   if (!state.isWrapped(v)) {
3     v = new WrappedValue(v);
4   }
5   v.annotations = ann;
6   return v;
7 }
```

Listing 7.1: Annotating a JavaScript Value.

card) Security Annotations, we construct a wrapped value with the same base value and the required annotation, and return this wrapped value. This approach follows the essential structure of the evaluation rules for coercion by returning the value with modified annotations (recall, e.g., [E-AsW] in Figure 5.4).

The primary internal function involved in setting the annotation of a value is given in Listing 7.1. This function does not deal with the specific semantics of, for example, **as** or **drop**. Instead, it is called in the logic of these coercions to set the resulting annotation. In particular, it checks if the value is already wrapped, and, if not, constructs a new wrapped value with the same base value. It then overwrites any previous annotation with the desired Security Annotation `ann`.

7.1.3 Complications: Annotating Objects.

The problem. Consider the example in Listing 7.2: the program constructs an array, fills it, and then checks the resulting array meets the annotation guard. In particular, the function `fillArray` fills an array and then annotates the array with `_A`. The method `annotate` on line 7 is an analogue for **as**, and is described in Section 7.1.4). The `enforce` method

```
1  var _A = new (SecAnn("A"))([]);
2
3  var fillArray = function(arr) {
4    for (var i = 0; i < arr.length; i++) {
5      arr[i] = 1;
6    }
7    annotate(arr, _A);
8  };
9
10 var x = new Uint8Array(16);
11 fillArray(x);
12 enforce(x, _A);
```

Listing 7.2: Annotating objects vs. their references.

(line 12) asserts that the annotation of the first argument is at least the second argument (see Section 7.1.4). Line 10 constructs an array and saves a reference to it in the variable `x`; we call this reference r. This reference r is then passed to the method `fillArray` and, on line 7, the annotation of r is upcast to `_A`. This is achieved via the method described in Listing 7.1, which acts only on JavaScript values, and so acts on references—not the object r points to.

The reference stored in `x` has not been annotated, so a new `WrappedValue` is constructed from r; this new reference, r', points to the same object. Since it is wrapped, r' can then be annotated with `_A`, and this is then returned to the method `fillArray`. However, since the result of `annotate` is not saved to a variable, the annotated reference r' is lost. When control returns from the function to the main body of the program, only the reference r, stored in `x`, remains. The enforcement on line 12 therefore fails when we would expect it to succeed.

Unfortunately, this problem manifests itself in cryptographic APIs. Re-

call the shim for the `getRandomValues` method of the WebCrypto API given on lines 8-10 Listing 5.3. This method takes as argument a reference to an array, and fills this array with random values. The array itself is not returned. Our shim annotates the argument after modification of the array with `CSRV`. The structure of this method is precisely the structure of `fillArray` in Listing 7.2.

The solution in S5$_{SA}$. In S5$_{SA}$, we had full control over the memory of the application. This means that we had full control over the object store, Θ, variable store, σ, and the annotation store, $\mathcal{A}$. Θ is a mapping from references, r, to objects, θ<S>, which form the image of Θ. Manipulation of object annotations is guided by evaluation rules such as [E-ASR-STORE] in Figure 5.10, which describes the evaluation of r **as** S, under the assumption that r is a reference to the object θ<R> under Θ. To evaluate this expression we replace the mapping Θ directly with a new mapping Θ' and evaluate to r. Θ' is identical up to any instances of θ<R> in the image of Θ, which in the image of Θ' are replaced by θ<R $*$ S>. Any reference to θ<R> under Θ will, under Θ', therefore map to our object with the updated annotation as desired. In this way, S5$_{SA}$ allows for the updating of the annotation associated to an object in the program globally, without the need to update the annotation for each reference.

The S5$_{SA}$ solution cannot be reused in ExpoSE$_{SA}$. The solution for our reference implementation relies on the fact that we have complete control of the memory at runtime and can, when required, alter it to update object annotations. Since ExpoSE$_{SA}$ is based on source code instrumentation, we do not control the memory, and as such cannot simply use this solution. In particular, we are only able to wrap (and therefore annotate) references to the object, directly contrasting the model described in Chapter 5.

A solution for WebCrypto in ExpoSE$_{SA}$. In this thesis we offer a limited solution, which suffices for the modeling of security properties in the WebCrypto API. In particular, the problem occurs when the setting of annotations is required on an unwrapped value. The new reference is created, and this new reference is annotated, but may subsequently be lost unless the old reference is not specifically overwritten, which we cannot guarantee. This problem does not occur if the annotation is set on a wrapped value: in this case, the `annotations` property is simply updated.

A natural solution is to automatically wrap any new reference to an object; in general, this is not possible. Instead, we note that within WebCrypto, the only JavaScript objects which require annotation are instances of `TypedArray` (e.g., `Uint8Array`). These must be declared with an explicit constructor. Through ExpoSE, we hook these constructors in order to automatically wrap references to `TypedArray` objects. In Listing 7.2, this means that the call to `annotate` does not create a new reference, but instead updates the `annotations` property of the wrapped reference stored in `x` (per Listing 7.1). The enforcement of `A` on line 12 therefore succeeds as desired.

This automatic wrapping of references is only for those instances of `TypedArray` which could be annotated as a result of using the WebCrypto API; this is therefore not a generic solution. However, it ensures that annotations are properly attached to those objects used in cryptographic operations.

7.1.4 Manipulating Annotations

We extend ExpoSE's built-in `S$` library with Security Annotation manipulation methods. The `S$` library enables the construction of symbolic values and assumptions or assertions about such values. In essence, this library provides a means for developers to enable testing of their programs us-

ing ExpoSE. We therefore extend `S$` with our methods for Security Annotations since they naturally fit in the same space, in that these methods enable property testing.

We provide a method for declaring Security Annotations: `S$.`**`SecAnn`**`(name, parent)`, described in detail in Section 7.1.1. The first argument, `name`, is a string defining the name of the annotation; `parent` is an optional argument declaring the immediate parent annotation in the lattice. If no annotation is provided, then this is considered to be `Top`.

The method `S$.annotate(val, ann)` is an analogue for the expression `val` **`as`** `ann`. The argument `val` is any value and `ann` a concrete instance of an annotation. The return value of this method is a wrapped value with base value `val`. The wrapped value has annotation at least `ann`. If the value was previously wrapped, it is the composition of `ann` and the previously valid annotation; otherwise, the resulting annotation is simply `ann`. The `S$.`**`drop`**`(val, ann)` and `S$.`**`cpAnn`**`(val1, val2)` methods are natural analogues of the **`drop`** and **`cpAnn`** expressions, with the same divergences as `S$.annotate`.

Finally, we introduce a method `S$.enforce(val, ann)`. This method is essentially the same as the `assert` macro from Section 5.6, and is equivalent to applying the rule [E-APP] from Figure 5.11 to the function **`function`**`(x : ann){})(val)`. In particular, we first check if the annotation enforced, `ann`, is `Top`. If so, the check passes and we are done. Next, we check if `val` corresponds to a prevalue w' described in Chapter 5. If so, we throw `FailedSecurityCheck`, since the value is not enforced against `Top`. This preserves the intuition that **`null`** and **`undefined`** cannot possess security properties. Finally, we check that the annotation of `val` is a subannotation of `ann`; if not, `FailedSecurityCheck` is thrown.

We do not reintroduce the function syntax **`function`**`(x : ann)`, or explicit expressions such as `val` **`as`** `ann` for manipulation of Security Annotations. Instead, we simply expose the methods described above through

the `S$` library. This avoids the need to introduce syntax transformation as a pre-processing step on the JavaScript code to be analyzed, which may not be faithful. In addition, this means we do not need to modify the JavaScript engine.

7.2 Establishing Faithfulness

It is important to have confidence that ExpoSE$_{SA}$ respects the mechanisms of Security Annotations built up throughout this thesis. These language in Chapter 5 is accompanied by an implementation which is thoroughly tested via a comprehensive test suite, ensuring the implementation precisely matches the formal model. This implementation is well-behaved, and we leveraged the underlying model to prove properties related to Security Annotations (Section 5.8). It is therefore natural to use S5$_{SA}$ as a reference implementation to guide ExpoSE$_{SA}$ and to check the two implementations agree on a conformance test suite. Any formal proof of correctness of this implementation is beyond reach; however, we can offer assurance that ExpoSE$_{SA}$ is well-behaved at least for some subset of the language.

This test suite provides a natural opportunity for checking the faithfulness of ExpoSE$_{SA}$. We translate this test suite into a form compatible with `S$`[2]. The resulting test suite contains 58 distinct tests which exercise the judgments described in Chapter 5; in addition, the combination of judgments are also tested to ensure that Security Annotation manipulations interact as desired. Further, the test suite checks that the subannotation relation is respected. Each test is passed by S5$_{SA}$: failed tests indicate a divergence between our implementations.

Adhering to the test suite in its entirety would not guarantee a correct

[2]This test suite is available alongside the implementation of ExpoSE$_{SA}$.

implementation. Recall that S5 itself is not a complete description of JavaScript itself; rather, it models the essential features of the language. We therefore have no mechanism for testing features of JavaScript not covered by S5. However, our model still allows us to gain a degree of confidence based on "observational equivalence": it should be impossible to distinguish between the two implementations by observing test outputs.

In some cases, differences in implementation do not cause divergences between the implementations. As described in Section 7.1.2, values only possess Security Annotations if they are wrapped as a result of a Security Annotation manipulation. One might expect this to create divergences in the behavior of enforcement: since an unwrapped value does not possess annotations, it should fail any annotation check. Recall that, in our reference implementation, the Security Annotation of each value is considered at least `Top`; in ExpoSE$_{SA}$ we mirror this behavior by passing enforcement checks if `Top` is enforced. This means that although many values in a program aren't wrapped, ExpoSE$_{SA}$ still considers them to possess `Top`.

The principal divergence relates to the annotation of objects in Section 7.1.2. In particular, since annotations are attached to references rather than objects, annotations may be lost if references are not passed carefully, e.g., modifying the annotation of an object inside a function and not returning this reference from the function. This is mitigated in the specific case of typed arrays, as previously discussed; however, for generic arrays, this results in divergence from the reference implementation.

This divergence mean that we can distinguish between our reference implementation and ExpoSE$_{SA}$. However, the divergence is known, and are mitigated for the purposes of analyzing the desired APIs. At the point of enforcing Security Annotations for the WebCrypto API, the two implementations are observationally equivalent. This closeness therefore gives us confidence in the validity of analyses based on Security Annotations, even if we cannot leverage the formal guarantees discussed in Section 5.8.

7.3 A Testing Strategy for the Detection of Cryptographic Errors

In this section, we describe the overall testing strategy for real-world applications. Motivated by the security guarantees offered by the enumeration of program paths in Section 5.8, we use dynamic symbolic execution, through ExpoSE, to construct test cases enumerating the distinct control flow paths. We do this by marking input values to the program as symbolic. For example, consider a node.js application, for which the input is represented as an array with name `process.argv`. To test such an application, we make the entire array symbolic; this will explore all paths based on the input to the program. We combine this DSE with Security Annotations by adding in our drop-in replacement for the WebCrypto API, which contains a Security Annotation specification. When a path results in a `FailedSecurityCheck` error being thrown, we have found a path resulting in a possible security property violation, along with a test case which recreates this error. DSE in general will not terminate, so the absence of security property violations cannot be proved. However, the systematic exploration of program paths allows us to find possible security bugs.

Limitations. We review the limitations of our approach in the context of native JavaScript. In particular, recall that we cannot guarantee that the program under test does not affect either the library itself or the annotated shim of the library. In the setting of native JavaScript this is exacerbated, since Security Annotations themselves are implemented in pure JavaScript. Although the programs under test themselves should not use the `S$` library which exposes methods which manipulate Security Annotations, it is possible for a program to cause equivalent effects.

As discussed in in Section 5.8.4, we comment briefly on the effect of the

program under test on libraries and the system of annotations. Similarly, we note that, unlike the implementation of $S5_{SA}$, the implementation in native JavaScript is susceptible to program under test directly accessing and manipulating annotations, even if the program in question is not intentionally malicious. In these cases, we cannot guarantee that a program path which executes safely is free from possible security violations. To prevent this, one could insist the library and system of annotations are written within a system such as DJS [11]. This system could then be trusted to run safely in the untrusted environment of the program under test. However we consider this out of scope for this thesis, since this would limit the applicability of the approach to only libraries written in such restrictive language subsets. In particular, the testing of applications using WebCrypto would not be possible, since they are not written within such a restrictive subset.

7.4 From $S5_{SA}$ to $ExpoSE_{SA}$

We return to the case study described in Section 5.7, and use it to demonstrate the scope and expressiveness of $ExpoSE_{SA}$. We discuss the limitations of $S5_{SA}$ and the modifications of the application required to execute within this reference interpreter. We describe how each limitation of the reference interpreter is solved in $ExpoSE_{SA}$, allowing for an execution of the case study; we discuss minor library mocks which allow for ease of testing, but do not require modifications to the developer's application[3].

Execution in $S5_{SA}$. To execute this case study in $S5_{SA}$, we made significant alterations to the application since $S5_{SA}$ cannot execute asynchronous

[3]The full source for this application and testing environment are available at: `https://github.com/duncan-mitchell/secAnn-caseStudies`.

code. This required two substantial changes: first, extensive library mocks were necessary. Second, changes had to be made to the application in order to remove the handling of asynchronous library calls. The `net` library is reliant on listeners which are triggered by network activity; we therefore constructed a purely synchronous mock of it. Second, we constructed a synchronous mock of `WebCrypto`. The case study was adapted for these libraries in the following ways: first, since the mock for WebCrypto was synchronous, all code handling this asynchronicity was removed. Our synchronous mock does not perform cryptographic operations, the reasons for this are discussed below. Second, in order to use our mock of `net`, we explicitly called the listener's callback function each time we expected the event to fire.

A mock for the `net` API in ExpoSE$_{SA}$. We must still provide a mock of the `net` API in order to execute this case study in ExpoSE$_{SA}$. The `net` API module enables the construction of servers and clients over stream-based TCP and IPC. ExpoSE$_{SA}$ supports the listener-based architecture of the networking in this module; however, it does not support the actual networking itself. Our mock closely follows the behavior of the real API, by basing our mock on the node.js module `events` [71]. Addresses and port numbers are therefore fixed constants, and calls to construct servers and sockets define objects which will return the required information without actually manipulating the relevant ports. Calls to a socket or server's `write` method are mocked to emit an event triggering the relevant listener. Since addresses and port numbers are fixed, this mock only supports a single server and socket.

A mock for the WebCrypto API in ExpoSE$_{SA}$. There is no strict requirement to mock WebCrypto in order to execute applications using this API in ExpoSE$_{SA}$. However, in the interests of more efficient testing, we pro-

```
1  decrypt: function(alg, key, data) {
2    if(alg.name === 'AES-CBC' ||
3       alg.name === 'AES-GCM' ||
4       alg.name === 'AES-CTR' ||
5       alg.name === 'RSA-OAEP') {
6      return new Promise(function(resolve, reject) {
7       setTimeout(function() {
8         resolve(data);
9       }, 300);
10     });
11   } else { throw 'InvalidAccessError' }
12 }
```

Listing 7.3: A mock for WebCrypto's `decrypt` method.

vide a simple mock for the API. In particular, our mock is faithful to the specification for WebCrypto, except that it does not perform the cryptographic primitives (in line with the mock for $S5_{SA}$). For example, consider the mock for the `decrypt` API given in Listing 7.3. Note that the `data` argument supplied containing the ciphertext, is an `ArrayBuffer`. All WebCrypto APIs, with the exception of `getRandomValues`, return a JavaScript promise (see Section 2.3) to the return value. In the case of `decrypt`, this is an `ArrayBuffer` containing the plaintext of the provided ciphertext. In this mock, we do not perform the cryptographic operation and return an actual decryption inside the promise: instead, we provide the supplied ciphertext inside a promise, since this provides a value of the correct return type. In order to ensure this mock is asynchronous, we use the `setTimeout` method to delay the execution of the function by 300 milliseconds. Although this means that when testing the application no actual cryptographic operations will be performed, this avoids the path explosion problem in DSE being exacerbated. In particular, crypto-

```
1  var decShim = function(alg, key, data) {
2    if (/AES/.test(alg.name)) {
3      S$.enforce(key, _SYMKEY);
4    } else if (/RSA/.test(alg.name)) {
5      S$.enforce(key, _PRIVKEY);
6    } else { throw FailedSecurityCheck; }
7    var res = wc.decrypt(alg, key, data);
8    res = res.then(function(val) {
9      return S$.annotate(S$.drop(S$.cpAnn(data, val),
         _CIPHERTEXT), _PLAINTEXT);
10   });
11   return res;
12 };
```

Listing 7.4: A shim for WebCrypto's `decrypt` method.

graphic operations often feature many different internal branching conditions, and passing symbolic arguments may cause the path space for DSE to explore to grow enormously, preventing exploration of the behavior we care about. $ExpoSE_{SA}$ is designed for testing and not production use, and we focus on testing use of the WebCrypto API; we trust that WebCrypto correctly implements the underlying primitives correctly. There is therefore little value in executing these operations. Further, since the `encrypt` mock does not perform the encryption operation, it also returns a promise to the provided plaintext argument, so we maintain the invariant that the decryption of an encryption is still the original data.

Modifying the WebCrypto shim. The shim in Listing 5.3 does not account for asynchronicity; in Listing 7.4 we demonstrate the necessary modifications to enable analysis in $ExpoSE_{SA}$ for one API method. First, we replace uses of Security Annotation manipulation and enforcement expressions with equivalent calls to the `S$` library (e.g., line 9). Second, we ensure that

```
1 var S$ = require('S$');
2 var window = require('./window-mock');
3 var net = require('./net-mock');
4 var shim = require('./wcShim');
5
6 shim(window);
7 process.argv = S$.symbol('Args', ['']);
```

Listing 7.5: Modifications to test an application in ExpoSE$_{SA}$.

we annotate the result of the cryptographic operation and not a promise to it by calling the **then** method of the promise (lines 8-10). After annotating, a promise to the annotated result is then returned from the shim (line 11).

To use this shim in applications, we define it inside a function and export this function as part of a node.js module. This module can then be included in the application under test and the function called.

Developer modifications for DSE. Listing 7.5 gives the inserted code in order to enable the testing of the application under DSE. No other changes to the code by a developer are necessary (besides, in this example, rewriting cryptographic API calls into asynchronous code blocks to match the faithful WebCrypto mock). Line 1 defines the helper library for ExpoSE$_{SA}$ to allow creation of symbolic values. Lines 2-3 include the API mocks discussed previously. Lines 4-6 include the shim and then call it to redefine the WebCrypto API, as described previously. Finally, line 7 uses the `S$.symbol` method to create a new symbol to simulate program input. This symbolic object is stored in `process.argv`, which stores the arguments with which the node.js process was called. This symbol is given the name `'Args'` and initialized to an array containing a single element, the empty string; this is equivalent to the case where no input is provided.

```
1 function getMsg () {
2   var args = process.argv;
3   if (args.length != 3 || args[2] === "") {
4     throw ("cryptoApp takes a single argument");
5   } else { return args[2]; }
6 }
```

Listing 7.6: Processing the message to be sent.

Testing the application. Armed with an application with the necessary modifications made to enable DSE, we can now test the application with $ExpoSE_{SA}$. The discussion in Section 5.8.3 describes two distinct program paths: one where an argument is not provided (in which case an error is thrown) and one where some argument is provided, in which case a message is encrypted and sent across the network. In our native JavaScript implementation, we modify the function which checks the validity of the argument to insist the provided argument is not the empty string. This function is given in Listing 7.6. The stipulation that `process.argv` is of length 3 comes from the fact that, in a native execution, `process.argv[0]` corresponds to the interpreter (usually node), `process.argv[1]` to the name of the application.

This application therefore has three control-flow paths, rather than two. First: the array provided may not be of length 3, and $ExpoSE_{SA}$ generates a test case for this where `process.argv` is the empty array. Second: the argument provided as the third element of the array is the empty string. $ExpoSE_{SA}$ generates a test case for this where `process.argv` is `["", " ", ""]`. In both these cases, an error is thrown because no argument is provided as a message to send to the server. The final path corresponds to the case that an argument is provided ($ExpoSE_{SA}$ provides the concrete test case `["!0!","!0!","!0!"]`). In this case, in line with Section 5.7,

`FailedSecurityCheck` is thrown because the IV is modified prior to encryption. When this bug is removed, this test case executes successfully. At this point, the DSE terminates since the all control flow paths of the program have been explored, and concrete test cases provided for each[4].

7.5 Case Studies

We discuss the testing of two case studies of JavaScript applications which make use of WebCrypto[5]. The first, secret-notes [100], is an application supporting a tutorial explaining correct usage of the WebCrypto API (Section 7.5.1). The second, hat.sh [93], is a web application offering a file encryption and decryption service (Section 7.5.2).

7.5.1 secret-notes

We describe the analysis of a prominent tutorial for the WebCrypto API [100], which comes equipped with the associated source code[6]. We first give a brief overview of the application, describe a testing framework and extensions to the shim for WebCrypto. Finally, we discuss the results of analysis.

Overview of the application. secret-notes is a web application designed as a tutorial for the WebCrypto API. The application allows a user to write a note, which is saved and then can later be restored. It contains four distinct JavaScript modules—teaching different elements of the API—each

[4]Since the entire array for `process.argv` is symbolic, the test cases to exercise these correspond to an explicit declaration of `process.argv` in a testing framework. To replay these tests using node.js, the first corresponds to not supplying an argument, the second to supplying the empty string, and the final test case corresponds to supplying the string `"!0!"` as argument.

[5]The source code of these applications and test harnesses are available at `https://github.com/duncan-mitchell/secAnn-caseStudies`.

[6]`https://github.com/ttaubert/secret-notes`

providing a different type of security. In the first, no security is offered in saving and loading notes; in the second, data integrity is ensured by use of the `digest` method. In the third, authenticity is provided: the user provides a password, for which a signing key is derived using the PBKDF2 algorithm (password-based key derivation function) of the `deriveKey` API. The resulting key is then used to generate a HMAC (hash-based message authentication code) through the `sign` method. Finally, the fourth provides secrecy through AES in Galois/Counter Mode (GCM) and the `encrypt` method. In order to derive a key for this encryption, a password is again supplied. In total, the cryptographic portion of this codebase covers approximately 300 lines of code.

Testing the application. We are interested in testing the cryptographic portion of functionality in this application. Each tutorial file follows the same structure: it exposes two functions, `save` and `load`, to the rest of the application. Enabling the testing of these functions is our core aim since they comprise the cryptographic logic of the application. These functions take input (the notes to be stored, and, where necessary, a password) and either save into local storage or retrieve these notes from local storage. In the web application, save and load trigger the corresponding functions. Alongside cryptographic operations, the application utilizes the `localforage` library for web applications to provide an offline data store [67].

ExpoSE$_{SA}$ is desgned to test node.js applications; to replicate the web application interface, we construct a simple test harness to exercise each pair of `save` and `load` functions. The test harness for the tutorial enforcing integrity is given in Listing 7.7; `'./versions/integrity'` simply refers to the relevant cryptographic tutorial and `getNote` is a function similar to the `getMsg` function in Listing 7.6. To enable standalone testing, we mock the web-only `localforage` library by adding a simple data store in

```
1 var integrity = require('./versions/integrity');
2 var v2 = integrity.save(getNote()).then(function() {
3   integrity.load().then(function(res) {
4     console.log('integrity: ' + res);
5   })
6 });
```

Listing 7.7: A testing harness for the integrity tutorial in secret-notes.

a JavaScript object. We provide both notes to be saved and passwords to secure these notes through arguments to the node.js harness. Within the tutorial files, no modifications are necessary besides the addition of our library mocks. Similar to Section 7.4, we mark the input to the program as symbolic to examine control flow paths through the `save` and `load` cryptographic functions. This enables us to test the core of the application in standalone JavaScript.

Extending the WebCrypto shim. The shim of WebCrypto given in Listing 5.3 requires extension to cover additional WebCrypto API methods used in this application. Method shims for `digest` (which computes a hash of data for integrity purposes) and `verify` (which verifies the provided signature) are given in Listing 7.8. Note that the shim for `digest` uses a new annotation `_HASH`, which we distinguish from `_SIGNATURE` to ensure that the result of a `digest` method is not used to check for authenticity, only integrity. Listing 7.9 gives an extended shim for `deriveKey`, incorporating valid uses of the PBKDF2 algorithm to derive a cryptographic key from a password. In particular, this specification is similar to the case where we derive a symmetric key using ECDH, however there are distinct security requirements. First, there is no security requirement on the `masterKey` argument, since it is a user-provided password we expect to

```
1  var verifyShim = function(alg, key, sig, data) {
2    S$.enforce(sig, _SIGNATURE);
3    if (/HMAC/.test(alg.name)) {
4      S$.enforce(key, _SYMKEY);
5    } else if (/RSA|ECDSA/.test(alg.name)) {
6      S$.enforce(key, _PRIVKEY);
7    } else { throw "FailedSecurityCheck"; }
8    var res = wc.verify(alg, key, sig, data);
9    res = res.then(function(val) {
10     return val;
11   });
12   return res;
13 }
14
15 var digestShim = function(alg, data) {
16   if (/SHA-(?:256|384|512)/.test(alg.name)) {
17     var res = wc.digest(alg, data);
18     res = res.then(function(sig) {
19       return S$.annotate(sig, _HASH);
20     });
21     return res;
22   } else { throw "FailedSecurityCheck"; }
23 }
```

Listing 7.8: Shims for `digest` and `verify`.

```
1  var dkShim = function(alg, masterKey, derivedKeyAlg,
       extractable, keyUsages) {
2    if (alg.name === 'ECDH') {
3      S$.enforce(alg['public'], _PUBKEY);
4      S$.enforce(masterKey, _PRIVKEY);
5    } else if (/HKDF|PBKDF2/.test(alg.name)) {
6      if (alg.hash === 'SHA-1') {
7        throw 'FailedSecurityCheck';
8      }
9      S$.enforce(alg['salt'], _CSRV);
10   } else { throw FailedSecurityCheck; }
11   let res = wc.deriveKey(alg, masterKey,
       derivedKeyAlg, extractable, keyUsages);
12   res = res.then(function(key) {
13     return S$.annotate(key, _SYMKEY);
14   });
15   return res;
16 };
```

Listing 7.9: Extending the shim for `deriveKey`.

be of low entropy. Second, we ensure that the underlying primitive used by PBKDF2 is not the insecure SHA-1 hashing algorithm; in that case, we throw a security error (line 7). Third, we ensure that the provided salt is a CSRV, otherwise, the resulting key may be predictable (line 9).

Results. In analyzing this application, we do not find any control flow paths which violate the underlying cryptographic contracts described in our specification by Security Annotations. We do find a possible insecurity in the tutorial demonstrating authenticity: in this case, the underlying primitive used for deriving the key from the password is SHA-1; as discussed, this is insufficient for future use cases. This security flaw was known to the authors: a comment in the source code notes the chosen algorithm may be insecure and should be updated. However, at the time of the application being written (2014), this was the only supported primitive for use with PBKDF2.

7.5.2 hat.sh: A File Encryption Serivce

The open source[7] application, hat.sh [93], is a browser-based file encryption service underpinned by WebCrypto. We first describe the application, and a methodology for testing it. We then describe key functions within the application and finally discuss the results of the analysis.

Overview of the application. hat.sh is a popular web application designed to allow users to encrypt and decrypt files without the need to upload them to a server. Instead, the encryption and decryption of files is performed through WebCrypto within the browser itself. First, the user uploads a file to be encrypted (or, respectively, decrypted). Second, the

[7] `https://github.com/sh-dv/hat.sh`

user can choose between optionally providing their own password, or allowing the application to generate one. This password is then used to generate a key for encryption (or decryption) through PBKDF2 (and the `deriveKey` API). After this key is derived, the file is encrypted (or decrypted) and the result downloaded by the user. The core logic of the application is approximately 350 lines of code.

Testing the application. As discussed in Section 7.5.1, ExpoSE$_{SA}$ is designed to test node.js applications; in order to test the full logic of this application we must mock certain web-specific method calls and libraries. In particular, we provide mocks for methods that manipulate the DOM; these do not affect the overall logic of the application, but simply ensure that we do not need to alter the core application. Second, we mock button listeners, and then trigger them within a test harness in sequence replicating user input. We also construct simple mocks of certain built-in APIs (e.g., `FileReader` and `Blob`) which are not present in node.js. Finally, we make minor alterations to some application methods to avoid use of the `async` and `await` keywords, which are unsupported by ExpoSE$_{SA}$. In these applications, we replace these keywords with idiomatic translations to the traditional `.then` paradigms for asynchronous code used throughout this thesis.

We then construct a test harness mimicking the UI: we simulate a chosen password and file to upload through command-line arguments. As in our previous case studies, this array of command-line arguments is left symbolic. We then use our test harness to simulate button clicks within the UI: first, if a password is provided as an argument, we click the button to process this password and proceed to encrypt the provided file. Otherwise, we use the application to generate a password and proceed to encrypt the provided file with this password. If the encryption succeeded (and a password was provided), we then decrypt the file. In the case that

```
1  async function deriveSecretKey() {
2    let getSecretKey = await importSecretKey();
3    let rawPassword = str2ab(password.value);
4    return window.crypto.subtle.deriveKey({
5       name: DEC.algoName1,
6       salt: rawPassword,
7       iterations: DEC.itr,
8       hash: { name: DEC.hash },
9     },
10    getSecretKey,
11    { length: DEC.algoLength, },
12    false,
13    DEC.perms2
14   )
15 }
```

Listing 7.10: Deriving a key from a password in hat.sh [93].

the password was generated, we do not perform decryption as our mock does not extract the generated password. In total, the test harness itself which drives the application is approximately 50 lines of code.

A closer look at key derivation. The function which derives a symmetric key for encryption and decryption is given in Listing 7.10. First, a password is asynchronously imported using the application's `importSecretKey` function, which reads `password.value` (line 2). On line 3, the raw value of the password is converted into an `ArrayBuffer` using the method `str2ab`, based on an example from the WebCrypto documentation. Finally, the method returns the result of a call to the `deriveKey` API. The specified algorithm (stored in an object `DEC`) is PBKDF2, and, per the specification, the cryptographic salt should be at least a 128-bit random value [110]. Note that the user-supplied password is passed as both the password to be used

```
1 function generateKey() {
2   const usedChars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ
       abcdefghijklmnopqrstuvwxyz0123456789@#_+=';
3   let keyArray = new Uint8Array(16);
4   window.crypto.getRandomValues(keyArray);
5   keyArray = keyArray.map(x =>
       usedChars.charCodeAt(x % usedChars.length));
6   const randomizedKey =
       String.fromCharCode.apply(null, keyArray);
7   password.value = randomizedKey;
8 }
```

Listing 7.11: Automatically generating a password in hat.sh [93].

as the base for the derivation (line 10) and as the salt itself (line 6).

Automatic password generation. Listing 7.11 contains the automatic password generation function for the case where a user elects to not supply their own password for the encryption. Here, the developer safely generates 128 bits of random data using `getRandomValues` (line 4). They then format this data as a string of human-readable characters from the string `usedChars` (line 2). In doing so, they replace each 8-bit integer `x` within the array with the character code for the value of `x % 67` to ensure the characters remain within this human-readable range (line 5). We discuss the consequences of the developer's decision below.

Results. Using ExpoSE$_{SA}$, we find two possible security flaws within this application along the two distinct control flows of the application: one when a password is provided, and the other when it is automatically generated. Both relate to the deployment of the PBKDF2 algorithm to derive a secure encryption key from the password given in Listing 7.10. Both

possible security flaws trigger `FailedSecurityCheck` errors due to the provided salt not meeting the annotation guard of `_CSRV` in `deriveKey` (line 9 of Listing 7.9).

When a user provides a password, since this password is by nature not randomly generated—but user-supplied—it is not annotated with `_CSRV`. The subsequent use of this password to derive the key from is not problematic: PBKDF2 expects a password of little to no entropy. As such, our WebCrypto shim does not have an annotation guard for the `masterKey` argument (Listing 7.9). However, as discussed previously, the developer reuses the password as the salt; since this is not a randomly generated value, the security of the resulting key cannot be guaranteed. A fix to this would be to generate a random salt: since this value can be public, providing it as part of the preamble to the encrypted file when downloaded (alongside the IV used for encryption), would allow the re-computation of the secret key at a later point.

The second possible security flaw involves the case where a user does not provide a password. Here, there is no problem with the password generation (Listing 7.11) for the purposes of using it as a user passphrase (since, as mentioned above, PBKDF2 expects a low entropy password). However, note that the array of random values is overwritten to correspond to an integer between 0 and 66 rather than 0 and 256. In total, this reduces the entropy of this array by approximately 32 bits[8]). The modification of this array leads to the discarding of `_CSRV`, and so, on entry to `deriveKey` in Listing 7.10, `FailedSecurityCheck` is thrown due to the insufficient salt. This can, as before, be solved by simply generating the salt at the necessary time rather than reusing the password within the call to

[8]This approximation assumes `usedChars` has length 64 rather than 67. In fact, there is also a bias towards the first 55 characters of the string: for each of these, 4 possible 8-bit integers map to these characters, whereas for the last 12 characters, only 3 possible integers do. This bias also makes brute force attacks to guess the password easier.

`deriveKey`. Both of these vulnerabilities were reported to the developer, who acknowledged them and patched the application to fix these issues.

8 Conclusions

This thesis has presented a novel mechanism for checking meta-properties of programs. In particular, we described a theoretical grounding for Security Annotations, which facilitate both the specification of cryptographic APIs in JavaScript and the checking of their usage. We implemented this system within formal languages and on top of the dynamic symbolic execution engine ExpoSE to enable the analysis of real-world applications. We provided security guarantees in these formal settings and demonstrated the applicability of the approach through case studies in full JavaScript.

Historically, JavaScript has proved notoriously difficult to analyze. With many unconventional features and a purely dynamic type system, the prevailing attitude has been that JavaScript is ill-suited to cryptographic implementations. The standardization of cryptographic APIs, however, has meant that discouraging its use is no longer an option. Previous work on this topic has focused on verifying bespoke implementations with the help of manual work by cryptographic experts. The checking of mainstream implementations has been restricted to languages amenable to static checking; we extend the status quo to the automatic runtime analysis of JavaScript programs using standardized APIs.

In this thesis, we have introduced Security Annotations, a mechanism allowing values to carry meta-properties as type tags. Throughout this thesis, we have built a formal grounding for Security Annotations in stages

in order to ensure correctness and usefulness of the mechanism. We first described the mechanisms of Security Annotations within a small functional language, and proved safety properties about such a language. This necessary first step ensured the correctness of Security Annotations as a language mechanism. Next, we migrated Security Annotations to a formal semantics for JavaScript and proved security guarantees within this setting, demonstrating the use case for this mechanism. We further demonstrated the applicability of this beyond the theoretical through case study, using Security Annotations to prove security properties about the usage of cryptographic APIs. Finally, we used this formal semantics to test an implementation of Security Annotations within full JavaScript. We built this implementation on top of the DSE engine ExpoSE, and used this to enable the automatic testing of real applications. We proved the viability of using Security Annotations as a mechanism to check API usage through the discovery of a security flaw in a file encryption service using the WebCrypto API.

A core challenge in the analysis of applications using cryptography we encountered was the lack of support for systematic testing of programs with string manipulation. In particular, JavaScript's particularly expressive flavor of regular expressions has historically thwarted the analysis of such programs. When analyzing such a program under dynamic symbolic execution, these features could not be reasoned about, causing a loss of coverage in programs using regular expressions. In this thesis, we presented the first comprehensive model for EcmaScript 6 regular expressions. Our key insight was that we could model the path condition constraints involving these regexes in terms of formal regular languages and string concatenation. This model enables the constraint solver to reason about such path condition constraints and determine path feasibility. As such, these complex features no longer limit the efficacy of the use of DSE as a testing tool for JavaScript programs.

This thesis has proposed and demonstrated the viability of runtime enforcement of cryptographic properties in JavaScript, a language that is traditionally notoriously difficult to analyze. We presented the design of Security Annotations from a language-agnostic basis, through smaller languages through to the analysis of real-world JavaScript code. We designed a specification for the standardized WebCrypto API through Security Annotations. Such a specification enables the automatic testing of implementations, removing the need for intervention by cryptographic experts.

Future Work There are several plausible avenues of future work which would extend the applicability of the approach described in this thesis. Web applications are a natural fit for developers desiring security beyond TLS/SSL. The next step in enabling the testing of a broader range of applications is the extension of Security Annotations to the browser. Our current system allows the checking of the core logic of web applications (e.g., the hat.sh case study in Section 7.5.2). Extending support to web applications would make the process of testing more viable for mainstream developers, and allow the testing of a broader range of targets.

Within this thesis, annotation specifications—both for cryptographic APIs and for library functions which may manipulate cryptographic objects—are designed by hand. This thesis presented only a specification for the WebCrypto API: extending this to alternative cryptographic APIs could improve the applicability and practicality of the approach. Further, specifications for other library calls are currently ad-hoc. In order for the approach described in this thesis to be usable by developers without expert intervention, a systematic approach to the annotation of these methods is required. Such an approach could include automatic inference of specifications for library methods, which would ensure Security Annotations could be used by developers as additional functionality is added to the JavaScript standard, or new libraries are used by developers. There ex-

ists a strong body of work on specifications for JavaScript libraries; one approach could be to infer Security Annotation specifications from type specifications designed for typed languages such as TypeScript.

Two other extensions to the work proposed in this thesis could broaden its scope. First, the mechanism of Security Annotations itself is language-agnostic; one could apply this mechanism to other dynamically typed languages, e.g., Python, which are also used to implement cryptographic solutions, particularly in server-side code. Second, Security Annotations themselves are fairly generic: although designed to express cryptographic meta-properties, they can be used to express other security properties. This would enable the checking of broader program properties within JavaScript, for example, checking access control in web applications.

Bibliography

[1] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An SMT solver for string constraints. In *Computer Aided Verification (CAV)*, 2015.

[2] P. A. Abdulla, M. F. Atig, Y.-F. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Flatten and conquer: A framework for efficient analysis of string constraints. In *ACM SIGPLAN Conf. Programming Language Design and Implementation*, PLDI, 2017.

[3] A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. A)*, pages 255–300. MIT Press, 1990.

[4] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *IEEE Symposium on Security and Privacy (S&P)*, Washington, DC, USA, 2013. IEEE Computer Society.

[5] T. Arcieri. What's wrong with in-browser cryptography? `https://tonyarcieri.com/whats-wrong-with-webcrypto` (Last accessed: 17 June 2019), Dec. 2013.

[6] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII*. 2014.

[7] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Prog. Lang. Syst.*, 33(2):8:1–8:45, 2011.

[8] D. J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[9] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symp. Security and Privacy (S&P 2015)*, pages 535–552, 2015.

[10] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symp. on Security and Privacy (S&P)*, 2017.

[11] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *Proceedings of the 22Nd USENIX Conference on Security*, 2013.

[12] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Defensive JavaScript – building and verifying secure web components. In *Found. of Security Analysis and Design (FOSAD)*, 2014.

[13] K. Bhargavan, C. Fournet, and N. Guts. Typechecking higher-order security libraries. In *Asian Symp. on Programming Languages and Systems (APLAS)*, 2010.

[14] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symp. on Security and Privacy (S&P)*, 2013.

[15] K. Bhargavan and G. Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on http over tls and openvpn. In

ACM SIGSAC Conf. on Computer and Communications Security (CCS), New York, NY, USA, 2016. ACM.

[16] N. Bjørner, V. Ganesh, R. Michel, and M. Veanes. SMT-LIB sequences and regular expressions. In *Int. Workshop on Satisfiability Modulo Theories (SMT)*, 2012.

[17] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[18] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, Oct.–Dec. 2008.

[19] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, Feb.–Mar. 2008.

[20] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *ACM SIGPLAN-SAGACT Symposium on Principles of Programming Languages*, 2014.

[21] S. Bucur, J. Kinder, and G. Candea. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[22] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2008.

[23] N. Campbell. bcrypt. `https://www.npmjs.com/package/bcrypt` (Last accessed: 21 July 2019), 2010.

[24] C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. J. Foundations of Computer Science*, 14(06), 2003.

[25] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi. Fast and precise type checking for JavaScript. *Proc. ACM Prog. Lang.*, 1(OOPSLA):48:1–48:30, 2017.

[26] T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu. What is decidable about string constraints with the replaceall function. *PACMPL*, 2(POPL):3:1–3:29, 2018.

[27] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Sys., Lang., and App. (OOPSLA)*, 2012.

[28] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[29] Digital Bazaar, Inc. forge. `https://github.com/digitalbazaar/forge` (Last accessed: 17 June 2019), 2010.

[30] T. Disney. contracts.js. `https://github.com/disnet/contracts.js` (Last accessed: 17 June 2019), 2017.

[31] ECMA International. *ECMAScript 2015 Language Specification*, 2015.

[32] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM SIGSAC Conf. on Comp. and Comm. Security (CCS)*, 2013.

[33] C. Flanagan. Hybrid type checking. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 245–256, 2006.

[34] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, 1999.

[35] C. Fournet, K. Bhargavan, and A. D. Gordon. Foundations of security analysis and design vi. chapter Cryptographic Verification by Typing for a Sample Protocol Implementation, pages 66–100. Springer-Verlag, Berlin, Heidelberg, 2011.

[36] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner. Javert 2.0: Compositional symbolic execution for javascript. volume 3, 2019.

[37] T. Freeman and F. Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Prog. Lang. Design and Implementation*, PLDI '91, 1991.

[38] X. Fu, M. C. Powell, M. Bantegui, and C. Li. Simple linear string constraints. *Formal Asp. Comput.*, 25(6), 2013.

[39] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang. (POPL)*, 2012.

[40] GitHub Inc. Electron. `https://electronjs.org` (Last accessed: 17 June 2019), 2019.

[41] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. ACM SIGPLAN 2005 Conf. Prog. Lang. Design and Implementation (PLDI 2005)*, 2005.

[42] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp. (NDSS)*, 2008.

[43] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Eur. Conf. on Object-Oriented Prog. (ECOOP)*, 2010.

[44] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. Jsflow: Tracking information flow in JavaScript and its apis. In *29th Annual ACM Symposium on Applied Computing*, 2014.

[45] D. Hedin, A. Sjösten, F. Piessens, and A. Sabelfeld. A principled approach to tracking information flow in the presence of libraries. In *Principles of Security and Trust (POST)*, 2017.

[46] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Principles of Security and Trust (POST)*, 2015.

[47] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL):4:1–4:32, 2018.

[48] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symp. on Security and Privacy (S&P)*, 2006.

[49] B. Kaliski. Pkcs #5: Password-based cryptography specification version 2.0, 2000.

[50] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[51] M. Keil and P. Thiemann. Treatjs: Higher-order contracts for javascripts. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2015.

[52] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symp. on Security and Privacy (EuroS&P)*, 2017.

[53] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. CrySL: Validating correct usage of cryptographic APIs. In *Eur. Conf. on Object Oriented Prog. (ECOOP)*, 2018.

[54] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Asia-Pacific Workshop on Systems*, 2014.

[55] G. Li, E. Andreasen, and I. Ghosh. Symjs: automatic symbolic testing of JavaScript web applications. In *Int. Symp. Foundations of Software Engineering (FSE)*, 2014.

[56] G. Li and I. Ghosh. Pass: String solving with parameterized array and interval automaton. In *Haifa Verification Conf. (HVC)*, 2013.

[57] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering (ASE)*, 2009.

[58] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Int. Conf. Computer Aided Verification (CAV)*, 2014.

[59] T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, and C. Barrett. A decision procedure for regular membership and length constraints

over unbounded strings. In *Int. Symp. on Frontiers of Combining Systems*, FroCoS, 2015.

[60] B. Loring, D. Mitchell, and J. Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *24th International SPIN Symposium on Model Checking of Software*, SPIN '17, 2017.

[61] B. Loring, D. Mitchell, and J. Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2019.

[62] M. Madsen, O. Lhoták, and F. Tip. A model for reasoning about JavaScript promises. *Proc. ACM Prog. Lang.*, 1(OOPSLA):86:1–86:24, 2017.

[63] M. Matsui. Linear cryptanalysis method for des cipher. In *Advances in Cryptology — EUROCRYPT '93*, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[64] D. Mitchell and J. Kinder. A formal model for checking cryptographic api usage in javascript. In *European Symp. on Research in Comp. Security (ESORICS)*, 2019.

[65] D. Mitchell, L. T. van Binsbergen, B. Loring, and J. Kinder. Checking cryptographic API usage with composable annotations. In *ACM SIGPLAN Workshop on Partial Evaluation and Prog. Manipulation (PEPM)*, 2018.

[66] J. Mott. crypto-js. `https://code.google.com/archive/p/crypto-js/` (Last accessed: 17 June 2019), 2013.

[67] Mozilla. localforage. `https://localforage.github.io/localForage/` (Last accessed: 28 August 2019), 2013.

[68] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping through hoops: why do java developers struggle with cryptography APIs? In *Int. Conf. on Software Eng. (ICSE)*, 2016.

[69] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[70] Node.js Foundation. Node.js crypto API. `https://nodejs.org/api/crypto.html` (Last accessed: 17 June 2019), 2010.

[71] Node.js Foundation. Node.js events API. `https://nodejs.org/api/net.html` (Last accessed: 23 August 2019), 2010.

[72] Node.js Foundation. Node.js net API. `https://nodejs.org/api/net.html` (Last accessed: 17 June 2019), 2010.

[73] Node.js Foundation. Node.js. `https://nodejs.org` (Last accessed: 17 June 2019), 2019.

[74] U. S. N. I. of Standards and T. (NIST). Announcing the advanced encryption standard (AES). Technical report, 2001.

[75] D. Olson. Client-side encryption. `https://www.braintreepayments.com/blog/client-side-encryption/` (Last accessed: 17 June 2019), May 2011.

[76] Oracle. Java cryptography architecture (jca) reference guide. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html/` (Last accessed: 25 July 2019), 2019.

[77] D. Park, A. Stefănescu, and G. Roşu. Kjs: A complete formal semantics of JavaScript. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2015.

[78] C. S. Pasareanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Int. Conf. on Automated Software Eng. (ASE)*, pages 179–180, 2010.

[79] A. Petcher and G. Morrisett. The foundational cryptography framework. In *Principles of Security and Trust*. Springer, 2015.

[80] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[81] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Symp. on Dynamic Languages (DLS)*, 2012.

[82] T. Ptacek. JavaScript cryptography considered harmful. `https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2011/august/javascript-cryptography-considered-harmful/` (Last accessed: 17 June 2019), Aug. 2011.

[83] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang. (POPL)*, 2015.

[84] E. Rescorla. The transport layer security (TLS) protocol version 1.3. Federal information processing standards publication 197, IETF, 2018.

[85] G. Rosu and T. Serbanuta. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[86] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[87] J. F. Santos, P. Gardner, P. Maksimovic, and D. Naudziuniene. Towards logic-based verification of javascript programs. In *Int. Conf. on Automated Deduction (CADE)*, 2017.

[88] J. F. Santos, P. Maksimovic, D. Naudziuniene, T. Wood, and P. Gardner. JaVerT: JavaScript verification toolchain. *Proc. ACM Prog. Lang.*, 2(POPL):50:1–50:33, 2018.

[89] J. F. Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of javascript. In *ICT Systems Security and Privacy Protection*, 2014.

[90] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symp. Sec. and Privacy (S&P)*, 2010.

[91] J. D. Scott, P. Flener, and J. Pearson. Constraint solving on bounded string variables. In *Integration of AI and OR Tech. in Constraint Prog. (CPAIOR)*, 2015.

[92] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering, (ESEC/FSE)*, 2013.

[93] sh-dv. Hat.sh. `https://hat.sh` (Last accessed: 30 August 2019), 2019.

[94] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in JavaScript. In *Annual Computer Security Applications Conference (ASAC)*, 2009.

[95] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining javascript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[96] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. In *Advances in Cryptology – CRYPTO 2017*. Springer International Publishing, 2017.

[97] N. Sullivan. How we built origin CA: Web crypto. `https://blog.cloudflare.com/how-we-built-origin-ca-web-crypto/` (Last accessed: 17 June 2019), May 2016.

[98] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ACM SIGPLAN Int. Conf. on Func. Prog. (ICFP)*, 2011.

[99] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Symp. on Security and Privacy (S&P)*, 2011.

[100] T. Taubert. Keeping secrets with javascript. `https://timtaubert.de/talks/keeping-secrets-with-javascript/` (Last accessed: 28 August 2019), 2014.

[101] The Guardian. The NSA files. `https://www.theguardian.com/us-news/the-nsa-files` (Last accessed: 17 June 2019).

[102] S. Thomas. Decryptocat. `https://tobtu.com/decryptocat.php` (Last accessed: 19 June 2019), 2013.

[103] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand. Search-driven

string constraint solving for vulnerability detection. In *Int. Conf. Software Engineering (ICSE)*, 2017.

[104] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Conf. Computer and Commun. Sec. (CCS)*, 2014.

[105] M. Trinh, D. Chu, and J. Jaffar. Progressive reasoning over recursively-defined strings. In *Computer Aided Verification (CAV)*, 2016.

[106] M. Trinh, D. Chu, and J. Jaffar. Model counting for recursively-defined strings. In *Computer Aided Verification (CAV)*, 2017.

[107] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST)*, 2010.

[108] P. Vekris, B. Cosman, and R. Jhala. Refinement types for TypeScript. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2016.

[109] X. Wang and H. Yu. How to break md5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[110] M. Watson. Web cryptography API. W3C recommendation, W3C, Jan. 2017.

[111] J. G. Wright and S. D. Wolthusen. Stealthy injection attacks against iec61850's goose messaging service. *IEEE PES Innovative Smart Grid Technologies Conf. Europe (ISGT-Europe)*, 2018.

[112] C. Zapponi. GitHut: A small place to discover languages in GitHub. `https://githut.info` (Last accessed: 17 June 2019), 2019.

[113] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3), 2017.

[114] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Computer Aided Verification (CAV)*, 2015.

[115] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Foundations of Software Engineering (FSE)*, 2013.

[116] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. In *ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, New York, NY, USA, 2017. ACM.