Gillian, Part II: Real-World Verification for JavaScript and C

Petar Maksimović[1], Sacha-Élie Ayoun[1],
José Fragoso Santos[2], and Philippa Gardner[1]

[1] Imperial College London, United Kingdom
`{p.maksimovic,s.ayoun17,p.gardner}@imperial.ac.uk`
[2] INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal
`jose.fragoso@tecnico.ulisboa.pt`

Abstract. We introduce verification based on separation logic to Gillian, a multi-language platform for the development of symbolic analysis tools which is parametric on the memory model of the target language. Our work develops a methodology for constructing compositional memory models for Gillian, leading to a unified presentation of the JavaScript and C memory models. We verify the JavaScript and C implementations of the AWS Encryption SDK message header deserialisation module, specifically designing common abstractions used for both verification tasks, and find two bugs in the JavaScript and three bugs in the C implementation.

1 Introduction

Separation logic (SL) [25,40] introduced <u>compositional</u> program verification using Hoare reasoning. Current analysis tools based on ideas from SL include: the automatic tool Infer [8,9] used inside Facebook to find lightweight bugs in Java/C/C++/Obj-C programs; the semi-automatic tool Verifast [26], which provides full verification for fragments of C and Java; the semi-automatic tool JaVerT [21], which provides bug-finding and verification for JavaScript (JS) programs; and the Viper architecture [36,35], which provides a verification backend for multiple programming languages, including Java, Rust, and Python. Our goal is to introduce verification based on SL to Gillian [19], a multi-language platform for symbolic analysis, integrating bug-finding and verification in the spirit of JaVerT and targeting many languages in the spirit of Viper.

Gillian currently supports three types of program analysis: symbolic testing, verification and bi-abduction. In [19], the focus was on symbolic testing, parametrised on complete concrete and symbolic memory models of the target language (TL), and underpinned by a core symbolic execution engine with strong mathematical foundations. Gillian analysis is done on GIL, an intermediate goto language parametric on a set of <u>memory actions</u>, which describe the fundamental ways in which TL programs interact with their memories. To instantiate Gillian to a new TL, a tool developer must: (1) identify the set of the TL memory actions and implement the TL memory models using these actions; and (2) provide a trusted compiler from the TL to GIL, which preserves the TL memory models and the semantics. In [19], Gillian was instantiated to JS and C, and used to find bugs in two real-world data-structure libraries, Buckets.js [43] and Collections-C [41]. Here, we introduce compositional memory models for Gillian, extend Gillian analysis with verification based on separation logic, adapt Gillian-JS and Gillian-C to this compositional setting, and provide verified specifications of the JS and C implementations of the deserialisation module of the AWS Encryption SDK.

The compositional Gillian memory models (§2) are given by the tool developer for each TL instantiation. They are based on partial memories, and formulated using core predicates and the associated consumer and producer actions. Core predicates describe fundamental units of TL memories: e.g., a property of a JS object and a C block cell. Consumers and producers, respectively, frame off and frame on the TL memory resource described by the core predicates. Partiality and frame are familiar concepts from SL [25,40,11]. What is perhaps less familiar is our emphasis on negative resource: i.e., the resource known to be absent from the partial memory. For example, in JS, a new extensible object is known not to contain any property; and, in C, a freed block is known not to be in memory and a cell is known not to exist beyond the block bound. We introduce a methodology for designing Gillian compositional memory models, and apply it to JS and C (§3), resulting in a unexpected similarity between the two models. Our compositional JS memory models follow those given in work on a JS program logic [24] and the JaVerT tool [21], where negative resource was essential for frame preservation, inspired by the use of negative resource to capture stability properties in the CAP concurrent separation logic [14], now used in Iris [27]. Our compositional C memory models are based on the complete CompCert memory model [31]. Despite a large body of work on separation logic for C, we were unable to find a partial C memory model that captures the negative resource in its entirety. The nearest is probably the CH20 formalism [29], which handles freed locations but not block bounds. Negative resource for freed locations has also been used in incorrectness logic [39], and for block bounds in a program logic for WebAssembly [48].

We build Gillian verification on top of our compositional memory models. In particular, using the core predicates, we design an assertion language for writing function specifications in separation logic and, using the consumers and producers, we build a fully parametric spatial entailment engine which enables the use of function specifications in symbolic execution. Gillian also supports user-defined predicates, which allow tool developers to identify the TL language interface familiar to code developers, and code developers to describe and prove properties about the particular data structures in their programs.

We extend Gillian-JS and Gillian-C to enable verification, introducing the JS and C compositional memory models, and using the same trusted compilers as in [19]. With these instantiations, we provide functionally-correct, verified specifications of the message header deserialisation module of the AWS Encryption SDK JS and C implementations (§4, §5). This is stable, critical, industry-grade code (~200loc for JS, ~950loc for C), which uses advanced language features to manipulate complex data structures. To verify this code, we create language-independent predicates to capture the message header, which we then connect without modification to both JS and C memories, giving specifications for the module functions. We also build a library of associated lemmas, used for the verification of both implementations. The verification itself required a substantial improvement of the reasoning capabilities of Gillian, especially when it came to handling arrays of symbolic size. We discovered two bugs in the JS implementation: one a form of prototype poisoning, predicted theoretically in our paper on JaVerT [21]; and another that allowed third parties to potentially alter authenticated, non-secret data. We have also discovered three bugs in the C implementation: one which allowed some malformed headers to be parsed as correct; one over-allocation; and one undefined behaviour. All of these bugs have been fixed.

2 Gillian Verification

We introduce Gillian verification based on separation logic (§2.2), extending the GIL execution engine presented in [19] with compositional memory models (§2.1).

2.1 Compositional Memory Models

GIL is a simple goto intermediate language whose syntax is given below. It is parametric on a set of TL memory actions, $A \ni \alpha$, given per instantiation by the tool developer. GIL values, $v \in \mathcal{V}al$, contain numbers, strings, booleans, uninterpreted symbols (used, e.g., to represent memory locations), simple types (e.g., numbers, strings), function identifiers and lists of values. GIL expressions, $e \in \mathcal{E}xpr$, contain values, program variables, and unary and binary operators (e.g. addition, list concatenation); GIL symbolic expressions, $\hat{e} \in \hat{\mathcal{E}}xpr$, are analogous except that symbolic variables, $\hat{x} \in \hat{\mathcal{X}}$, are used instead of program variables.

GIL Syntax

$$v \in \mathcal{V}al \triangleq i, j, n \in \mathcal{N} \mid s \in \mathcal{S} \mid b \in \mathcal{B} \mid \varsigma \in \mathcal{U} \mid \tau \in \mathcal{T} \mid f \in \mathcal{F} \mid \overline{v} \in List(\mathcal{V}al)$$

$$e \in \mathcal{E}xpr \triangleq v \mid x \in \mathcal{X} \mid \ominus e \mid e_1 \oplus e_2 \qquad \hat{e} \in \hat{\mathcal{E}}xpr \triangleq v \mid \hat{x} \in \hat{\mathcal{X}} \mid \ominus\hat{e} \mid \hat{e}_1 \oplus \hat{e}_2$$

$$\begin{aligned} c \in \mathcal{C}md \triangleq\ & x := e \mid \mathsf{ifgoto}\ e\ \ i \mid x := e(e') \mid x := \alpha(e) \mid & \qquad func \in \mathcal{F}unc \triangleq f(x)\{\overline{c}\} \\ & x := \mathsf{uSym}/\mathsf{iSym}(e) \mid \mathsf{return}\ e \mid \mathsf{fail}\ e \mid \mathsf{vanish} & \qquad \mathtt{p} \in \mathcal{P}rog = \mathcal{P}_!(\mathcal{F}unc) \end{aligned}$$

GIL commands, $c \in \mathcal{C}md$, contain variable assignment, conditional goto, function call, memory actions, allocation of uninterpreted/interpreted symbols, function return, error termination and path cutting. A GIL function, $f(x)\{\overline{c}\}$, comprises an identifier $f \in \mathcal{F}$, a formal parameter x[3], and a body given by a list of commands $\overline{c}$. A GIL program is a set of GIL functions with unique identifiers.

GIL execution is defined in terms of state models, which are parametric on a value set, $\mathsf{V} \supseteq \mathcal{V}al$, and a set of memory actions, A. We distinguish the Boolean value set, $\Pi \subset \mathsf{V}$, and refer to $\pi \in \Pi$ as a context. State models expose an interface consisting of state actions, $A \uplus A_S$, where the actions

[3] The implementation supports multiple parameters.

$$\dfrac{\begin{array}{c}\textsc{Memory Action - Success}\\ \mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e) \\ \sigma.\mathrm{eval}_e\,(-) \rightsquigarrow \mathsf{v} \qquad \sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}')^{\mathcal{S}} \\ \sigma'.\mathrm{setVar}_x\,(\mathsf{v}') \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma'', cs, i{+}1\rangle^{\mathtt{S}}} \qquad \dfrac{\begin{array}{c}\textsc{Memory Action - Error}\\ \mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e) \\ \sigma.\mathrm{eval}_e\,(-) \rightsquigarrow \mathsf{v} \qquad \sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}')^{r} \\ r \neq \mathcal{S} \qquad o = (\text{if } r = \mathcal{E} \text{ then } \mathtt{E} \text{ else } \mathtt{M})\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i\rangle^{o(\mathsf{v}')}}$$

Fig. 1: GIL Execution Semantics: Memory Actions

$A_S = \{\mathrm{setVar}_x\}_{x \in \mathcal{X}} \cup \{\mathrm{setStore}, \mathrm{getStore}\} \cup \{\mathrm{eval}_e\}_{e \in \mathcal{E}xpr} \cup \{\mathrm{assume}, \mathrm{uSym}, \mathrm{iSym}\}$ address store management, expression evaluation, branching, and allocation.

Definition 1 (State Model). *A state model, $S(\mathsf{V}, A) \triangleq \langle |S|, ea\rangle$, comprises: a set of states $\sigma = \langle \mu, \rho, \pi\rangle \in |S|$, containing a memory μ, a variable store ρ, and a (satisfiable) context π[4]; and an action execution function, $ea : (A \uplus A_S) \to |S| \to \mathsf{V} \rightharpoonup \mathcal{P}(|S| \times \mathsf{V} \times \mathcal{R})$, with the result $r \in \mathcal{R} = \{\mathcal{S}, \mathcal{E}, \mathcal{M}\}$ denoting success, non-correctible error, or missing information error, pretty-printed $\sigma.\alpha(\mathsf{v}) \to \{(\sigma_i, \mathsf{v}_i)^{r_i}|_{i \in I}\}$ for all outcomes and $\sigma.\alpha(\mathsf{v}) \rightsquigarrow (\mu_i, \sigma_i)^{r_i}$ for a specific outcome, with countable I. The value set of concrete state models is the set of GIL values, $\mathcal{V}al$[5]; the value set of symbolic state models is the set of symbolic expressions, $\hat{\mathcal{E}}xpr$.*

Definition 2 (GIL Execution Semantics). *Given a state model S, the GIL execution semantics has judgements of the form:*

$$\mathsf{p} \vdash \langle \sigma, cs, i\rangle^{o} \rightsquigarrow_S \langle \sigma', cs', j\rangle^{o'}$$

with: call stacks, $cs \in \mathcal{C}all_S$; command indexes, $i, j \in \mathbb{N}$; and outcomes, $o \in \mathcal{O}$.

The GIL execution semantics is standard for a goto language, except that it is parametrised by the memory actions. Call stacks capture function-related control flow, with $\mathsf{cmd}(\mathsf{p}, cs, i)$ denoting the i-th command of the currently executing function (cf. [33] for details). Outcomes, $o \in \mathcal{O} \triangleq \mathtt{S} \mid \mathtt{N}(\mathsf{v}) \mid \mathtt{E}(\mathsf{v}) \mid \mathtt{M}(\mathsf{v})$, indicate how the execution is to proceed: $\mathtt{S}$ states that it can continue; $\mathtt{N}(\mathsf{v})$ states that it terminated normally with return value v; and $\mathtt{E}(\mathsf{v})$ and $\mathtt{M}(\mathsf{v})$ state that it failed with either a non-correctible or missing information error described by v. We give the rules for memory action execution in Figure 1; all can be found in [33].

Compositional Memory Models. We move from whole-program memory models [19] to compositional memory models by introducing memory core predicates, $\gamma \in \Gamma$, which represent the fundamental units of the TL memory model (e.g., a memory cell). Core predicates take two lists of parameters, in-parameters (or ins), denoted v_i, and out-parameters (or outs), denoted v_o, such that from the ins we can learn the outs. This concept is similar to predicate parameter modes

[4] States also include allocators (cf. [33] for details), elided to limit clutter.
[5] Note that the only satisfiable concrete context is true, meaning that concrete contexts can be elided and concrete states can be viewed as memory-store pairs, $\langle \mu, \rho\rangle$.

of [37] and we use it to implement a parametric spatial entailment engine. An example of a core predicate is the cell assertion, $x \mapsto \mathsf{v}$, which captures a cell in memory at address x having value v. Its in-parameter is x, and its out-parameter is v, because, if we know x, we can find v by looking it up in the memory.

With each core predicate $\gamma \in \Gamma$, we associate a consumer and a producer memory action, denoted by cons_γ and prod_γ respectively, to obtain the set of predicate actions $A_\Gamma = \bigcup_{\gamma\in\Gamma}\{\mathrm{cons}_\gamma, \mathrm{prod}_\gamma\}$, whose meaning is discussed shortly.

Definition 3 (Compositional Memory Model). *Given value set* V *and core predicate set* Γ, *a compositional memory model,* $M(\mathsf{V}, \Gamma) \triangleq \langle |M|, \mathcal{W}f, \underline{ea}_\Gamma \rangle$, *comprises: **(1)** a partial commutative monoid (PCM)*[6], $|M| = (|M|, \bullet, \mathbf{0})$, *where* $\mathbf{0}$ *denotes the (indivisible) empty memory; **(2)** a well-formedness relation,* $\mathcal{W}f \subseteq |M| \times \Pi$, *with* $\mathcal{W}f_\pi(\mu)$ *denoting that memory* μ *is well-formed in (satisfiable) context* π*; and **(3)** a predicate action execution function,* $\underline{ea}_\Gamma : A_\Gamma \times |M| \times \mathsf{V} \times \Pi \rightharpoonup \mathcal{P}(|M| \times \mathsf{V} \times \Pi \times \mathcal{R})$, *pretty-printed* $\mu.\alpha(\mathsf{v})_\pi \to \{(\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}|_{i\in I}\}$ *for all outcomes and* $\mu.\alpha(\mathsf{v})_\pi \rightsquigarrow (\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}$ *for a specific outcome, with countable* I*. The value set of concrete memory models is the set of GIL values,* $\mathcal{V}al$*; the value set of symbolic memory models is the set of symbolic expressions,* $\hat{\mathcal{E}}xpr$*.*

We discuss the most important properties that the components of compositional memory models must satisfy; a full list is available in [33]. The PCM requirement is well-known from separation logic [40,11]. Well-formedness holds only for satisfiable contexts, and describes the separation of symbolic resource and any further TL-specific well-formedness criteria (cf. §3). It must be monotonic with respect to context strengthening, compatible with the PCM composition, and the empty memory must be well-formed in any satisfiable context. The action execution function, $\mu.\alpha(\mathsf{v})_\pi \to \{(\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}|_{i\in I}\}$, denotes that, in a memory μ that is well-formed in the context π, executing action α with parameter v yields a countable number of branches characterised by the non-overlapping[7], satisfiable contexts π_i, each of which implies π and makes the corresponding memory μ_i well-formed, and all of which together cover π (i.e., $\pi \Rightarrow \bigvee_{i\in I} \pi_i$). This last property means that memory actions do not drop paths, which is essential for verification.

The intuition behind consumers and producers is that consumers frame off the core predicate resource (CPR), uniquely determined by the core predicate ins, and the producers frame it on. The following properties capture this intuition. First, we define the CPR of a core predicate $\gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle$ as the memory resulting from its production in $\mathbf{0}$, which must succeed in any satisfiable context:

$$\pi\ \mathtt{SAT} \implies \mathbf{0}.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle, \mathsf{true})^{\mathcal{S}}_\pi \wedge \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle \neq \mathbf{0}.$$

overloading notation for the core predicate and its resource. Moreover, we require that any successful production frames on the CPR:

$$\mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu', \mathsf{true})^{\mathcal{S}}_{\pi'} \implies \mu' = \mu \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle$$

[6] A PCM, $X = \langle X, \bullet, \mathbf{0}\rangle$, comprises a carrier set X (overloaded for simplicity), a partial, associative, and commutative composition operator $\bullet$, and unit element $\mathbf{0}$.

[7] Note that this requirement makes concrete memory actions deterministic.

and also that producers cannot return missing information errors, as they are meant to succeed precisely when the CPR is missing. The consumers, on the other hand, must succeed if and only if the CPR is present in memory:

$$\begin{aligned} \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \pi' \vdash \mu = \mu' \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle \\ \pi \vdash \mu = \mu' \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle \wedge \mathcal{W}f_\pi(\mu) &\implies \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi} \end{aligned}$$

with the resulting context π' having enough information to isolate the CPR[8]. Interestingly, erroneous executions cannot be fully characterised in terms of CPR presence or absence, because of TL-specific error cases: for example, in C, attempting to either get or set the value of a block cell that is beyond the block bound raises an out-of-bounds error (cf. §3). What we require instead is that consumed CPR can always be re-produced, that producers fail in a memory in which consumers succeed, and that producers succeed in a memory in which consumers return a missing information error (and vice versa for the latter):

$$\begin{aligned} \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}'_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'} \\ \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot -)_{\pi} \rightsquigarrow (\mu, \mathsf{false})^{\mathcal{E}}_{\pi'} \\ \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{\mathcal{M}}_{\pi'} &\iff \mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_{\pi} \rightsquigarrow (\mu \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle, \mathsf{true})^{\mathcal{S}}_{\pi'} \end{aligned}$$

The properties given so far allow us, for example, to prove that well-formed memories cannot contain duplicated CPR. The final property below requires that non-missing executions of consumers and erroneous executions of producers must be frame-preserving, with the former formulated as follows:

$$\begin{aligned} &\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{r}_{\pi'} \wedge r \neq \mathcal{M} \wedge (\pi'' \Rightarrow \pi') \wedge \mathcal{W}f_{\pi''}(\mu \bullet \mu_f) \\ &\quad \implies (\mu \bullet \mu_f).\mathrm{cons}_\gamma(\mathsf{v}_i)_{\pi''} \rightsquigarrow (\mu' \bullet \mu_f, \mathsf{v}_o)^{r}_{\pi''} \end{aligned}$$

where π'' effectively maintains well-formedness constraints for μ, adds on further ones required for $\mu \bullet \mu_f$ to be defined and also isolates the consumed CPR. Note that neither missing executions of consumers nor successful executions of producers can be frame preserving, as framing on the appropriate CPR could result in success for the former, and a duplicated resource error for the latter.

Using the consumers and producers, we are able to derive getter and setter actions, $A \triangleq \{\mathrm{get}_\gamma, \mathrm{set}_\gamma : \gamma \in \Gamma\}$, which perform frame-preserving CPR lookup and mutation, as given below. We discuss getters and setters further in §3, in the context of our JS and C instantiations.

Getter: Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} \qquad \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}{\mu.\mathrm{get}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu'', \mathsf{v}_o)^{\mathcal{S}}_{\pi'}}$$

Setter: Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', -)^{\mathcal{S}}_{\pi'} \qquad \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}{\mu.\mathrm{set}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}$$

Getter: Non-Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'} \qquad r \neq \mathcal{S}}{\mu.\mathrm{get}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'}}$$

Setter: Non-Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'} \qquad r \neq \mathcal{S}}{\mu.\mathrm{set}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'}}$$

[8] The $\pi \vdash \ldots$ denotes reasoning under context π. In the concrete case, it can be ignored.

Compositional State Models. Compositional memory models lift to compositional state models, in a similar way to the lifting of the complete memory models illustrated in [19]; see [33] for details. Here, we focus on memory action execution, which is lifted as follows to state action execution, given a memory model $M(\mathsf{V}, \Gamma)$ and $\alpha \in A_\Gamma \uplus A$:

$$ea(\alpha, \langle\mu, \rho, \pi\rangle, \mathsf{v}) \triangleq \{(\langle\mu', \rho, \pi'\rangle, \mathsf{v}')^r \mid \mu.\alpha(\mathsf{v})_\pi \rightsquigarrow (\mu', \mathsf{v}')^r_{\pi'}\}.$$

Observe how the context of the state is passed to the memory execution function, which may then strengthen it before passing it back to the resulting state. We can show that the PCM and well-formedness relation on memories lift to a PCM and well-formedness relation on states, and that state action execution maintains properties analogous to those given for memory models.

2.2 GIL Verification

We give an overview of Gillian verification based on separation logic (SL); see [33] for details. We describe GIL assertions, parameterised by the core predicates of the TL, define assertion satisfiability in a novel, parametric way using the core predicate producers, and provide a mechanism for using verified function specifications in GIL execution.

GIL Assertion Syntax

$p, q \in \mathcal{A} \triangleq \mathsf{emp} \mid p * q \mid \gamma\langle\hat{e}_i \cdot \hat{e}_o\rangle \mid \delta\langle\hat{e}_i \cdot \hat{e}_o\rangle$

$P, Q \in \mathcal{A}srt \triangleq \{p \wedge \pi \mid p \in \mathcal{A}, \pi \in \Pi\}$

$pred \in \mathcal{P}red \triangleq \mathrm{pred}\ \delta\langle\hat{x}_i \cdot \hat{x}_o\rangle := P_1; \ldots; P_n;$

A compositional memory model with core predicates Γ induces an SL-assertion language given on the right. GIL memory assertions, $p, q \in \mathcal{A}$, are formed using the empty assertion, the separating conjunction, the core predicates, and user-defined predicates, whose names come from a dedicated set, $\Delta \ni \delta$. The empty assertion and the separating conjunction are standard. Core predicate assertions are lifted from memory core predicates. User-defined predicates, introduced by example in §3 and §4, are used by tool developers to characterise the interface of the TL, and by code developers to describe the data structures in their programs. They have in- and out-parameters like core predicates, and can have multiple definitions, separated by a semi-colon. Assertions, $P, Q \in \mathcal{A}srt$, extend memory assertions with pure first-order assertions, π, conflated with Boolean symbolic expressions.

Satisfiability. To define assertion satisfiability, we lift memory consumers and producers from core predicates to memory assertions, denoted by $\mu.\mathrm{cons}_\theta(p)$ and $\mu.\mathrm{prod}_\theta(p)$, and then to states and arbitrary assertions, denoted by $\sigma.\mathrm{cons}_\theta(P)$ and $\sigma.\mathrm{prod}_\theta(P)$, using substitutions $\theta : \hat{\mathcal{X}} \mapsto \mathsf{V}$ (extended to symbolic expressions inductively, in the standard way) to map core predicate assertions, with parameters given by symbolic expressions, to the core predicates of the memory model, with parameters given by values. We highlight the successful base case of the memory assertion consumers, where the returned context requires the out-parameters of the assertion to match the ones found in memory:

$$\frac{\mu.\mathrm{cons}_\gamma(\theta(\hat{e}_i))_\pi \rightsquigarrow (\mu', \mathsf{v}'_o)^{\mathcal{S}}_{\pi'} \qquad \pi'' = (\pi' \wedge \mathsf{v}'_o = \theta(\hat{e}_o)) \qquad \pi''\ \mathtt{SAT}}{\mu.\mathrm{cons}_\theta(\gamma\langle\hat{e}_i \cdot \hat{e}_o\rangle)_\pi \rightsquigarrow (\mu', \mathsf{true})^{\mathcal{S}}_{\pi''}}$$

and the successful consumption of an arbitrary assertion $P = p \wedge \pi$:

$$\frac{\mu'.\mathrm{cons}_\theta(p)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi''} \qquad \pi'' \vdash \theta(\pi)}{\langle \mu', \rho, \pi' \rangle.\mathrm{cons}_\theta(p \wedge \pi) \rightsquigarrow (\langle \mu'', \rho, \pi'' \rangle, \mathsf{true})^{\mathcal{S}}}$$

Definition 4 (Satisfiability). *The* <u>*satisfiability relation*</u>*, stating that memory μ' and context π' satisfy assertion $p \wedge \pi$ under substitution θ, is defined by:*

$$\mu', \pi', \theta \models p \wedge \pi \iff \mathbf{0}.\mathit{prod}_\theta(p)_{\mathsf{true}} \rightsquigarrow (\mu_p, \mathsf{true})^{\mathcal{S}}_{\pi_p} \;\wedge\; \pi' \vdash (\mu' = \mu_p \wedge \pi_p \wedge \theta(\pi))$$

and is lifted to states as: $\langle \mu', \rho, \pi' \rangle, \theta \models p \wedge \pi$ if and only if $\mu', \pi', \theta \models p \wedge \pi$.

In Definition 4, the production, when successful, creates the (unique) memory μ_p that corresponds to the resource of the assertion p, with its (unique) well-formedness constraints, π_p. In the concrete case, as the only allowed context is true, the formulation simplifies to the more intuitive $\mathbf{0}.\mathrm{prod}_\theta(p) \rightarrow (\mu', \mathsf{true})^{\mathcal{S}} \wedge \theta(\pi)$.

Specifications. Gillian function specifications have the form $\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}}$, where f is the function identifier, x is the function parameter, $\hat{x}$ is the symbolic variable holding the value of x, P is the pre-condition, Q is the post-condition, and $\hat{e}$ is the return value of the function, with the following, well-known, constraints:

1. program variables do not appear in the pre- or the post-condition, and the function parameter x is accessed using the symbolic variable $\hat{x}$;
2. symbolic variables that appear in a pre-condition are implicitly universally quantified, and can be re-used in the corresponding post-condition; and
3. symbolic variables that appear only in a post-condition are implicitly existentially quantified.

We extend GIL programs with function specifications, accessible via $\mathsf{p}.\mathrm{specs}$, and the GIL execution semantics with rules for folding and unfolding user-defined predicates, as well as with a rule for calling function specifications, the success case of which is given below. Gillian <u>verifies</u> a specification $\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}}$ if, given the identity substitution $\hat{\theta}$ and a symbolic state $\hat{\sigma}$ with store $\{x \mapsto \hat{\theta}(\hat{x})\}$ such that $\hat{\sigma}, \hat{\theta} \models P$, the symbolic execution of f starting from $\hat{\sigma}$ always terminates, for all final symbolic states $\hat{\sigma}_i$ there exists some $\hat{\theta}_i \geq \hat{\theta}$ such that $\hat{\sigma}_i, \hat{\theta}_i \models Q$, and the corresponding return value equals $\hat{\theta}_i(\hat{e})$ under the context of $\hat{\sigma}_i$. We can prove that if Gillian verifies a specification, then its standard SL interpretation holds.

Spec Call - Success

$$\frac{\begin{array}{lr}
\mathsf{cmd}(\mathsf{p}, cs, i) = y := e(e') \text{ with } \theta & \text{function call with substitution } \theta \\
\sigma.\mathrm{eval}_e(-) \rightsquigarrow f \quad \sigma.\mathrm{eval}_{e'}(-) \rightsquigarrow \mathsf{v}' & \text{get function id and parameter value} \\
\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}} \in \mathsf{p}.\mathrm{specs} & \text{get one of the function specifications} \\
\theta' = \theta[\hat{x} \mapsto \mathsf{v}'] & \text{extend substitution with parameter value} \\
\sigma.\mathrm{cons}_{\theta'}(P) \rightarrow \{(\sigma_j, \mathsf{true})^{\mathcal{S}}|_{j \in J}\} & \text{consume pre-condition} \\
j \in J & \text{select a branch} \\
\sigma_j.\mathrm{prod}_{\theta'}(Q) \rightsquigarrow (\sigma_j', \mathsf{true})^{\mathcal{S}} & \text{produce post-condition} \\
\sigma_j'.\mathrm{setVar}_y(\theta'(\hat{e})) \rightsquigarrow \sigma' & \text{assign return value}
\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow \langle \sigma', cs, i{+}1 \rangle}$$

Note that for this rule to succeed, the consumption of P must succeed. The rule is slightly simplified for presentation. First, it assumes to have the substitution upfront; in the implementation, we have a unification algorithm that, starting from the function parameter and using the consumers, learns the substitution. Second, it assumes that the post-condition does not introduce fresh symbolic variables; these are handled using allocators and added to the substitution.

Remark. Due to space constraints, we have not been able to give the full technical details of Gillian verification. These are available in the Gillian technical report [33], where we demonstrate that the overall GIL execution using compositional memory models is frame-preserving (up to the usual renaming of allocated memory locations) and prove a standard verification soundness result.

3 Compositional Memory Models: JavaScript and C

We present the compositional memory models of JS and C, giving the basic actions and core predicates, and some of the user-defined predicates that capture the intuitive interfaces of these languages. The key ideas behind compositional JS memory models were introduced in the JaVerT project [21,20,22]; we transfer them to Gillian. We introduce the compositional C memory models, building on the concrete block-offset memory model of CompCert [31], simplifying the presentation.[9] In doing so, we highlight a striking similarity between the JS and C models that is the result of our emphasis on negative resource.

The JS and C concrete compositional memory models are made up of building blocks that are assigned a unique location (or identifier) from a set of uninterpreted symbols, $\mathcal{L} \subset \mathcal{U}$: for JS, the building blocks are the extensible objects; for C, they are the blocks of linear memory of a given size. Each building block is divided into at least one component. For JS, each object has three components: a property table, $h : \mathcal{S} \rightharpoonup \mathcal{V}al$, partially mapping property names (strings) to values; a domain, $d : \mathcal{P}(\mathcal{S})$, discussed shortly; and metadata, $m : \mathcal{V}al$, which keeps track of internal JS properties for that object [22]. For C, each block has two components: the block contents $k : \mathbb{N} \rightharpoonup \mathcal{V}al$, partially mapping offsets (natural numbers) to values; and a bound, $n : \mathbb{N}$, discussed shortly. Finally, the memory units are, intuitively, the parts of the memory components that cannot be separated further: for JS, these are single object properties, domains, and metadata; for C, these are single block cells and bounds. These memory units directly correspond to the core predicates given in Definitions 6 and 7.

Compositional memory models must keep track of negative resource, which can come from two sources: allocation and deallocation. For JS and C, the negative information originating from allocation has infinite representation: in JS, a freshly created object is known to not have any properties; in C, a freshly allocated block of a given size in C is known not to have offsets beyond that size. This infinite information is captured, for JS, by the object domain whose meaning

[9] We assume that values have the same size in memory and omit permissions. Gillian-C implements the full models, eliding the concurrency-related aspects of permissions.

is that any property not in the domain is absent, and, for C, by the block bound whose meaning is that any accesses beyond that bound result in a buffer overrun error. The negative information originating from deallocation is easier to handle, tracked by a dedicated uninterpreted symbol, $\varnothing \in \mathcal{U}$. In JS, deallocation is at the unit level: only object properties are deleted. This is captured by extending the co-domain of property tables with $\varnothing$: that is, $h : \mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}$. In C, deallocation is at the building-block level: only entire blocks can be deleted. This is captured by extending the co-domain of blocks with $\varnothing$, indicating that a block has been freed.

Due to compositionality, any building block, component or unit can be missing. In the theory, we capture this either implicitly, via absence from the domain of a mapping (e.g., a missing object property for JS or a missing block cell for C), or explicitly, using the symbol $\bot$ (e.g. a missing domain, metadata, or bound).

Definition 5 (Compositional JS and C Memories). *The PCMs of compositional concrete JS and C memories, $|M_{JS}|$ and $|M_C|$, are given by the sets*

$$\begin{aligned} \mu \in |M_{JS}| \ &: \ \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \mathcal{P}(\mathcal{S})_{\bot} \times \mathcal{V}al_{\bot}), \\ \mu \in |M_C| \ &: \ \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\bot})_{\varnothing}, \end{aligned}$$

composition defined as disjoint union, and empty memory $\emptyset$. The PCMs of compositional symbolic JS and C memories, $|\hat{M}_{JS}|$ and $|\hat{M}_C|$, are given by the sets

$$\begin{aligned} \hat{\mu} \in |\hat{M}_{JS}| \ &: \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\bot} \times \hat{\mathcal{E}}xpr_{\bot}), \\ \hat{\mu} \in |\hat{M}_C| \ &: \ \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\bot})_{\varnothing}, \end{aligned}$$

with composition defined as (syntactic) disjoint union, and empty memory $\emptyset$.

In the above definition, symbolic memory models are simple liftings of the concrete ones. In the implementation, we employ heavy optimisation: for example, in Gillian-C, we have developed a complex tree representation of symbolic blocks inspired by [29], enabling tractable reasoning about arrays of symbolic size.

Well-formedness of concrete memories addresses the relationship between positive and negative information, given for JS and C below:

$$\begin{aligned} \mathcal{W}f^{\text{JS}}(\mu) \ &\triangleq \ \forall (h, d, -) \in \mathsf{codom}(\mu).\ d \neq \bot \implies \mathsf{dom}(h) \subseteq d \\ \mathcal{W}f^{\text{C}}(\mu) \ &\triangleq \ \forall (k, n) \in \mathsf{codom}(\mu).\ n \neq \bot \implies \mathsf{dom}(k) \subseteq [0, n) \end{aligned}$$

Well-formedness of symbolic memories additionally has to address separation of locations and separation in any other mappings with symbolic expressions in its domain (e.g. object properties for JS and offsets for C). We give the well-formedness criterion for the symbolic C memory:

$$\hat{\mathcal{W}f}^{\text{C}}_{\pi}(\hat{\mu}) \triangleq \pi \vdash \bigwedge_{\substack{\hat{l}, \hat{l}' \in \mathsf{dom}(\hat{\mu}) \\ \hat{l} \not\equiv \hat{l}'}} \hat{l} \neq \hat{l}' \ \wedge \bigwedge_{\substack{(\hat{k}, -) \in \mathsf{codom}(\hat{\mu}) \\ \hat{o}, \hat{o}' \in \mathsf{dom}(\hat{k}), \hat{o} \not\equiv \hat{o}'}} \hat{o} \neq \hat{o}' \ \wedge \bigwedge_{\substack{(\hat{k}, \hat{n}) \in \mathsf{codom}(\hat{\mu}) \\ \hat{o} \in \mathsf{dom}(\hat{k}), \hat{n} \neq \bot}} \hat{o} < \hat{n}$$

For our JS and C instantiations, the core predicates follow straightforwardly from the units of their memory models.

$$\textsc{CConsCell - Found}\quad \frac{\mu(l) = (k, n) \qquad k(o) = v \qquad k' = k \setminus \{o\} \qquad \mu' = \mu[l \mapsto (k', n)]}{\mu.\mathsf{consCell}([l, o]) \rightsquigarrow (\mu', v)^{\mathcal{S}}}$$

$$\textsc{SConsCell - Use After Free}\quad \frac{\hat{\mu}(\hat{l}') = \varnothing \qquad \pi' = (\hat{l} = \hat{l}') \qquad (\pi \wedge \pi')\ \texttt{SAT}}{\hat{\mu}.\mathsf{consCell}\ ([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}, \mathsf{false})^{\mathcal{E}}_{\pi \wedge \pi'}}$$

$$\textsc{SConsCell - Found}\quad \frac{\begin{array}{c}\hat{\mu}(\hat{l}') = (\hat{k}, \hat{n}) \quad \hat{k}(\hat{o}') = \hat{v} \\ \pi' = ([\hat{l}, \hat{o}] = [\hat{l}', \hat{o}']) \quad (\pi \wedge \pi')\ \texttt{SAT} \\ \hat{k}' = \hat{k} \setminus \{\hat{o}'\} \quad \hat{\mu}' = \hat{\mu}[\hat{l}' \mapsto (\hat{k}', \hat{n})]\end{array}}{\hat{\mu}.\mathsf{consCell}\ ([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}', \hat{v})^{\mathcal{S}}_{\pi \wedge \pi'}}$$

$$\textsc{SConsCell - Missing Cell}\quad \frac{\begin{array}{c}\hat{\mu}(\hat{l}') = (\hat{k}, \hat{n}) \\ \pi_k = (\hat{l} = \hat{l}') \wedge \hat{o} \notin \mathsf{dom}(\hat{k}) \\ \pi_n = (\hat{n} = \bot) \vee (\hat{n} \geq \hat{o}) \quad (\pi \wedge \pi_k \wedge \pi_n)\ \texttt{SAT}\end{array}}{\hat{\mu}.\mathsf{consCell}\ ([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}, \mathsf{false})^{\mathcal{M}}_{\pi \wedge \pi_k \wedge \pi_n}}$$

Fig. 2: Selected rules for the consCell consumer.

Definition 6 (JS Core Predicates). *JS has three core predicates, $\gamma_{JS} \in \Gamma_{JS}$:*

- *the <u>object-property</u> predicate, $(\hat{l}, \hat{p}) \mapsto \hat{v}$, which states that property $\hat{p}$ of object at location $\hat{l}$ contains value $\hat{v}$ (including $\varnothing$ denoting property absence);*
- *the <u>domain</u> predicate, $\mathsf{domain}(\hat{l}, \hat{d})$, which states that object at location $\hat{l}$ has no properties outside the finite set $\hat{d}$;*
- *the <u>metadata</u> predicate, $\mathsf{metadata}(\hat{l}, \hat{m})$, which states that object at location $\hat{l}$ has metadata $\hat{m}$.*

Definition 7 (C Core Predicates). *C has three core predicates, $\gamma_{C} \in \Gamma_{C}$ [10]:*

- *the <u>cell predicate</u>, $(\hat{l}, \hat{o}) \mapsto \hat{v}$, which states that the cell at offset $\hat{o}$ in the block at location $\hat{l}$ contains value $\hat{v}$ (which, this time, does not include $\varnothing$);*
- *the <u>bounds predicate</u>, $\mathsf{bound}(\hat{l}, \hat{n})$, which states that any cell beyond offset $\hat{n}$ in block at location $\hat{l}$ is not there;*
- *the <u>freed predicate</u>, $\hat{l} \mapsto \varnothing$, which states that block at location $\hat{l}$ has been freed.*

We illustrate the C predicate action execution functions, $\underline{\mathsf{ea}}_{\,C}$ and $\underline{\hat{\mathsf{ea}}}_{\,C}$, respectively, with a selection of rules for the C cell-predicate consumer, consCell, given in Figure 2. The remaining rules, as well as the rules for their JS counterparts, $\underline{\mathsf{ea}}_{\,JS}$ and $\underline{\hat{\mathsf{ea}}}_{\,JS}$, can be found in the Gillian technical report [33]. With this information, we can define the compositional concrete and symbolic JS and C memory models.

Definition 8 (JS Memory Models). *The compositional concrete and symbolic JS memory models are defined, respectively, as $M_{JS}(\mathcal{V}al, \Gamma_{JS}) = \langle |M_{JS}|, \mathcal{W}f^{JS}, \underline{\mathsf{ea}}_{\,JS} \rangle$ and $\hat{M}_{JS}(\hat{\mathcal{E}}xpr, \Gamma_{JS}) = \langle |\hat{M}_{JS}|, \hat{\mathcal{W}f}^{JS}, \underline{\hat{\mathsf{ea}}}_{\,JS} \rangle$.*

Definition 9 (C Memory Models). *The compositional concrete and symbolic C memory models are defined, respectively, as $M_{C}(\mathcal{V}al, \Gamma_{JS}) = \langle |M_{C}|, \mathcal{W}f^{C}, \underline{\mathsf{ea}}_{\,C} \rangle$ and $\hat{M}_{C}(\hat{\mathcal{E}}xpr, \Gamma_{JS}) = \langle |\hat{M}_{C}|, \hat{\mathcal{W}f}^{C}, \underline{\hat{\mathsf{ea}}}_{\,C} \rangle$.*

[10] In full C and the Gillian-C implementation, memory values may be of different sizes, and holes may exist between these values due to alignment restrictions. To address this, the implemented cell assertion carries additional information related to, e.g., size and type, similarly to that of [4], and there also exists a `hole` core predicate.

The getters and setters for JS and C are defined using the methodology described in §2. In particular, the JS getters and setters are given by $A_{\text{JS}} = \{\mathsf{getProp}, \mathsf{setProp}, \mathsf{getDomain}, \mathsf{setDomain}, \mathsf{getMetadata}, \mathsf{setMetadata}\}$, and the summary of the execution of the symbolic $\mathsf{getProp}(\hat{l}, \hat{p})$ getter is illustrated below:

$$\begin{tikzcd}
\hat{l} \in \mathsf{dom}(\hat{\mu}) \arrow[r, "\hat{\mu}(\hat{l}) = (\hat{h}, \hat{d}, \hat{m})", "\text{yes}"'] \arrow[d, "\text{no}"] & \hat{p} \in \mathsf{dom}(\hat{h}) \arrow[r, "\text{no}"'] \arrow[d, "\text{yes}"] & \hat{d} = \bot \arrow[r, "\text{yes}"] \arrow[d, "\text{no}"] & \text{✗ } (\hat{l}, \hat{p}) \; \textbf{Missing property} \\
\text{✗ } (\hat{l}, \hat{p}) \; \textbf{Missing object} & \text{✓ } \hat{h}(\hat{p}) & \hat{p} \in \hat{d} \arrow[ur, "\text{yes}"] \arrow[r, "\text{no}"'] & \emptyset
\end{tikzcd}$$

Similarly, the C getters and setters are given by $A_{\text{C}} = \{\mathsf{getCell}, \mathsf{setCell}, \mathsf{getBound}, \mathsf{setBound}, \mathsf{getFreed}, \mathsf{setFreed}\}$ and the summary of the execution of the symbolic $\mathsf{getCell}(\hat{l}, \hat{o})$ getter is illustrated below:

$$\begin{tikzcd}
\hat{l} \in \mathsf{dom}(\hat{\mu}) \arrow[r, "\text{yes}"] \arrow[d, "\text{no}"] & \hat{\mu}(\hat{l}) = \emptyset \arrow[r, "\hat{\mu}(\hat{l}) = (\hat{k}, \hat{n})", "\text{no}"'] \arrow[d, "\text{yes}"] & \hat{o} \in \mathsf{dom}(\hat{k}) \arrow[r, "\text{no}"'] \arrow[d, "\text{yes}"] & \hat{n} = \bot \arrow[r, "\text{yes}"] \arrow[d, "\text{no}"] & \text{✗ } (\hat{l}, \hat{o}) \; \textbf{Missing cell} \\
\text{✗ } (\hat{l}, \hat{o}) \; \textbf{Missing block} & \text{⚡ } \textbf{Use after free} & \text{✓ } \hat{k}(\hat{o}) & \hat{o} \geq \hat{n} \arrow[ur, "\text{no}"] \arrow[r, "\text{yes}"'] & \text{⚡ } \textbf{Buffer overrun}
\end{tikzcd}$$

The similarities in the two diagrams are evident, with the main difference being that JS getters do not throw errors, whereas C getters do.

User-defined JS and C Predicates. Core predicates describe fundamental units of the TL memory model. On top, user-defined predicates build layers of abstraction to describe memory components and building blocks, standard library interfaces, all the way to complex data structures for particular code such as the AWS message header. Using Gillian notation, we present some of the JS and C user-defined predicates; in this notation: $*$ and $\wedge$ are conflated to $*$, with automatic differentiation between spatial and pure assertions[11]; predicate definitions are separated with a semi-colon; and logical variables are prefixed with the # symbol and are implicitly existentially quantified in predicate definitions.

Gillian-JS inherits many user-defined predicates from JaVerT [21], including simple ones for describing JS objects and their properties, as well as advanced ones for specifying scoping, function closures and prototype chains. We focus here on the new `FrozenObject(o, proto, pvs)` predicate, which describes a frozen object[12] `o` with prototype `proto` and property-value pairs `pvs`. We first define the predicate `FrozenObjectProps(o, pvs)` to grab the resource of the object properties:

```
pred FrozenObjectProps(o, pvs) : pvs = [ ];
    pvs = [#p, #v] :: #rpvs * DataPropConst(o, #p, #v) *
    FrozenObjectProps(o, #rpvs);
```

where `DataPropConst(o, #p, #v)` states that the object `o` has a non-writable property `#p` with value `#v`. We then add information about the object prototype and its non-extensibility using the `JSObject(o, proto, ext)` predicate, and also state that the object has no properties other than `pvs` using the domain core predicate:

[11] From the separation logic literature, the pure assertions can be regarded as dotted.
[12] A JS object is frozen if it cannot be extended and all its properties are non-writable.

```
pred FrozenObject(o, proto, pvs) :
    JSObject(o, proto, false) * FrozenObjectProps(o, pvs) *
    FirstProj(pvs, #ps) * ListToSet(#ps, #pss) * domain(o, #pss)
```

where `FirstProj`(pvs, #ps) means that the list #ps is the first projection of the list of pairs pvs, and `ListToSet`(#ps, #pss) means that the elements of the list #ps form the set #pss.

Gillian-C, on the other hand, comes with user-defined predicates capturing, for example, arrays and blocks in memory, as well as automatically-generated predicates describing C structs, with support for nested structs. In particular, the `array`(b, off, c) predicate describes a contiguous fragment of a block b, starting from offset off, with contents described by the mathematical list c:

```
array(b, off, c) : c = [];
                   (b, off) -> #c * array(b, off+1, #d) * c = #c :: #d
```

and the `block`(b, c) predicate captures an entire C block with contents c:

```
block(b, c) : array(b, 0, c) * bound(b, |c|)
```

In the implementation, arrays also exist as core predicates. This allows us to reason about arrays automatically in the symbolic memory (e.g., to split an array into sub-arrays), supported by our tree representation of symbolic blocks, instead of requiring manual application of lemmas.

Finally, we illustrate automatically generated struct-related predicates using the `aws_byte_cursor` structure given below, which contains two fields: an unsigned integer `len`; and a nullable pointer to an array of 8-bit unsigned integers `buf`. This struct is used for traversing the AWS message header (cf. §4), and is intended to capture an array in memory that starts at `buf` and has length `len`.

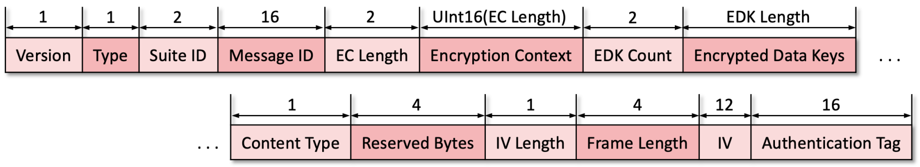
```
struct aws_byte_cursor {  pred struct_aws_byte_cursor(cur, len, buf) :
  size_t len;               (cur == [#b, #o]) * ((#b, #o) -int64-> len) *
  uint8_t *buf;             ((#b, #o +p 8) -int64-> buf) *
}                           is_ptr_or_null(buf)
```

The generated predicate describes the struct's layout in memory and gives basic typing information: it states that an `aws_byte_cursor`, starting from the position given by the pointer `cur`, occupies 16 bytes in memory (8 + 8, given by the type annotation `int64`), with the first 8 bytes taken by `len`, and the second 8 bytes (note the pointer addition +p) taken by `buf`, which is either a pointer or `null`.

4 AWS Encryption SDK Message Header Specification

The encrypted data handled by the AWS Encryption SDK is stored within a structure called a message [3]. The message format has two versions of similar complexity: we verify version 1; version 2 was introduced very recently. Messages consist of a header, a body, and a footer. Here, we describe only the structure of the header, as we are verifying header deserialisation.

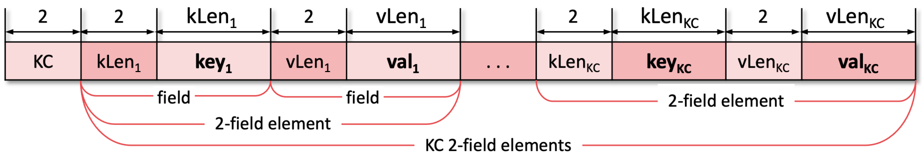
The AWS Encryption SDK message header is a sequence of bytes (buffer) divided into sections, as illustrated below; above each section is its length in bytes.



Our approach is to abstract the header contents into a list and formulate pure predicates that describe its structure in a language-independent way. This allows us to then use the same abstractions as part of further, language-dependent, abstractions for both JS and C. Our design of the abstractions was informed by existing code annotations found in the implementations, which describe simple first-order properties of the code and, in the case of C, can also link to the CBMC [30] bounded model checker. However, these annotations are limited by the expressivity of JS and C, particularly when it comes to reflecting on the memory contents. Our predicates have no such limitations.

We narrow down our exposition to the encryption context, as it illustrates well the language-independent and language-dependent aspects of our specification, and is also the section in which we discovered bugs in both implementations.

Pure Specification of the Encryption Context. The encryption context (EC) is a sequence of bytes that describes a set of key-value pairs. Its structure is given in the diagram below.

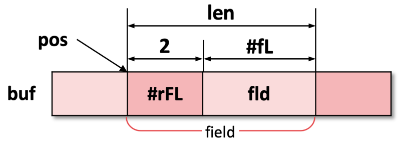


The first two bytes represent the number of key-value pairs, denoted by `KC`, and the rest describe the `KC` key-value pairs themselves. Keys and values are represented by sequences of bytes and, as they are of variable length, are serialised by first having *two bytes* that represent the length, followed by *that many bytes* of the actual key or value; we refer to this pattern as a field, and to a sequence of n fields as an n-element. Then, a key-value pair is serialised as a 2-field element, and all of the key-value pairs form a sequence of `KC` 2-field elements.

We specify the EC by building layers of abstraction, from fields to elements to element sequences to the EC, each of which can either be complete, incomplete (partial, but with correct structure), or malformed (with incorrect structure). In the implementation, these are specified separately and are joined together in appropriate over-arching abstractions. Here, we focus on complete variants only.

The `Field`(`buf`, `pos`, `fld`, `len`) predicate states that the buffer (list of bytes) `buf`, at index `pos`, holds a field with contents `fld` (list of bytes) and total length `len`:

```
pred Field(buf, pos, fld, len) :
 (0 <= pos) * (#rFL = sub(buf, pos, 2)) *
 UInt16(#rFL, #fL) *
 (fld = sub(buf, pos+2, #fL)) *
 (len = 2+#fL) * (pos+len <= |buf|)
```



This predicate uses the GIL operator `sub(l, s, n)`, which returns the sublist of list `l` starting from index `s` and of length `n`, and also the `UInt16(rn, n)` predicate, which states that `n` is a 16-bit big-endian interpretation of the raw 2-byte list `rn`. The `Element(buf, pos, fC, elem, len)` predicate states that buffer `buf` at index `pos` holds a sequence of `fC` fields, with contents `elem` (a list of the appropriate field contents) and total length `len`. It is defined similarly to a standard linked-list predicate, with the 'link' being the fact that the list members are contiguous in memory:

```
pred Element(buf, pos, fC, elem, len) :
  (fC = 0) * (0 <= pos) * (pos <= |buf|) * (elem = [ ]) * (len = 0);
  (0 < fC) * Field(buf, pos, #fld, #fL) * Element(buf, pos+#fL, fC-1, #rFs,
  #rL) * (elem = #fld :: #rFs) * (len = #fL+#rL)
```

Next, analogously to `Element`, we define the `Elements(buf, pos, eC, fC, elems, len)` predicate, which states that the buffer `buf`, at index `pos`, holds a sequence of `eC` elements, each with `fC` fields, with contents `elems` (a list of the appropriate element contents) and of total length `len`. Finally, the `EncryptionContext(buf, KVs)` predicate states that the entire buffer `buf` is an EC with key-value pairs `KVs`, with all keys being unique:

```
pred EncryptionContext(buf, KVs) : (buf = [ ]) * (KVs = [ ]);
     (#rKC = sub(buf, 0, 2)) * UInt16(#rKC, #KC) * (0 < #KC) *
     Elements(buf, 2, #KC, 2, KVs, #len) *
     FirstProj(KVs, #Ks) * Unique(#Ks) * (2+#len = |buf|)
```

Next, we show how this pure specification of the EC contents can be connected without modification to both the JS and C memories.

Encryption Context in JS. In JS, the EC is serialised as an ArrayBuffer, which is a raw binary data buffer in memory, and accessed using a Uint8Array, which is a view on top of that ArrayBuffer starting from a given offset and of a given length, treating the raw data underneath as 8-bit unsigned integers. This Uint8Array view is similar in function to the `aws_byte_cursor` C structure (cf. §3). Abstracting ArrayBuffer contents to lists, we connect these data structures in JS memory to our pure EC specification (cf. Figure 3, top and centre):

```
pred JSSerEC(o, EC, KVs) :
     Uint8Array(o, #aBuf, #off, #len) * ArrayBuffer(#aBuf, #data) *
     (EC = sub(#data, #off, #len)) * EncryptionContext(EC, KVs)
```

In JS, the EC is deserialised into a frozen JS object with prototype `null`, whose properties represent the keys and hold the values. This is done by converting the keys and the values to UTF-8 strings, and is specified as follows:

```
pred JSDeserEC(o, KVs) : toUtf8(KVs, #sKVs) * FrozenObject(o, null, #sKVs)
```

where `toUtf8` converts the list `KVs` point-wise to strings, obtaining `#sKVs`.

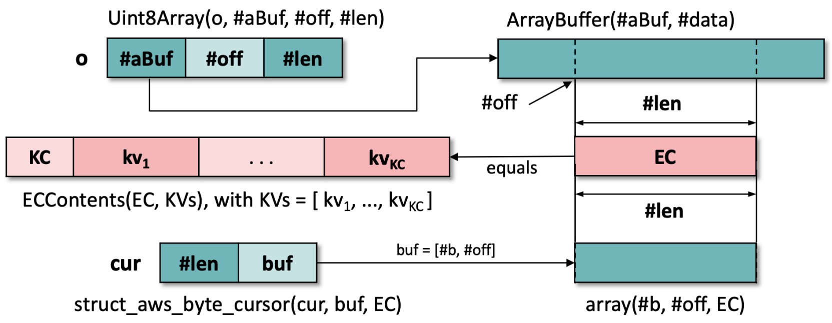

Uint8Array(o, #aBuf, #off, #len)
o
#aBuf
#off
#len
ArrayBuffer(#aBuf, #data)
#off
#len
KC
kv1
. . .
kvKC
equals
EC
ECContents(EC, KVs), with KVs = [kv1, ..., kvKC]
#len
cur
#len
buf
buf = [#b, #off]
struct_aws_byte_cursor(cur, buf, EC)
array(#b, #off, EC)

Fig. 3: Serialised Encryption Context: language-independent pure part (red; middle) and language-specific resource (green; JS above, C below)

Finally, the specification of the `decodeEncryptionContext` function states that the EC deserialisation is performed correctly.

```
{ JSSerEC(eEC, #EC, #KVs)                 }
  function decodeEncryptionContext(eEC)
{ PRE-CONDITION * JSDeserEC(ret, #KVs)   }
```

Encryption Context in C. In C, the EC is serialised as a block in memory, and is traversed using an AWS byte cursor. Using the auto-generated predicate given in §3, we define the `aws_byte_cursor(cur, buf, c)` predicate, stating that `cur` points to a byte cursor which has access to an array starting from `buf`, and holding contents `c`, making the length implicit:

```
  pred aws_byte_cursor(cur, buf, c) :
    struct_aws_byte_cursor(cur, #len, buf) * (buf = [#b, #off]) *
    array(#b, #off, c) * (#len = |c|)
```

A serialised EC can then be described as a valid byte cursor whose contents represent the EC key-value pairs (cf. Figure 3, centre and bottom):

```
pred CSerEC(cur, buf, EC, KVs) :
  aws_byte_cursor(cur, buf, EC) * EncryptionContext(EC, KVs)
```

In C, the EC is deserialised into an AWS hash table, whose keys and values directly correspond to the key/value pairs of the EC, specified as follows, eliding the internal structure of the hash tables due to space constraints:

```
pred CDeserEC(ht, KVs) : valid_hash_table(ht, KVs)
```

The specification of the EC deserialisation function is more complex than for JS. In particular, the byte cursor that originally pointed to the EC ends up shifted to the end of the byte buffer, exposing the array underneath the `CSerEC` predicate.

```
{  empty_hash_table(ec) * CSerEC(cur, #buf, #EC, #KVs)              }
   int aws_cryptosdk_enc_ctx_deserialize(
       struct aws_hash_table *ec, struct aws_byte_cursor *cur)
{  (ret = 0) * CDeserEC(ec, #KVs) * (#buf = [#b, #off]) *
   array(#b, #off, #EC) * aws_byte_cursor(cur, #buf +p |#EC|, [ ])  }
```

5 AWS Encryption SDK Message Header Verification

Using Gillian-JS and Gillian-C, together with the specifications given in §4, we verify full functional correctness of the header deserialisation module of the AWS Encryption SDK JS [2] (~200loc) and C [1] (~950loc) implementations. In particular, we verify that the deserialisation of a complete header is correct, and the deserialisation of an incomplete or a malformed header raises an appropriate error.

Verification Effort and Performance. The JS verification took 3 person-months and the C verification took 2 person-months, with the latter taking less time because a large part of the infrastructure developed for JS could be re-used. We substantially improved the first-order solver of Gillian to reason automatically about complex operations on lists of symbolic length, first used in the modelling of JS ArrayBuffers and then for C dynamic arrays. We created a collection of language-independent predicates and lemmas about their inductive properties (~1.2kloc) that cover the project-specific AWS header, but also re-usable first-order concepts such as list element uniqueness, projections of lists of pairs, conversion from bytes to numbers, and conversion from raw bytes to strings. Similarly, we also had to create language-dependent abstractions and associated lemmas for the JS and C manipulation of the AWS message header (~1.2kloc). Finally, we had to: annotate the code with specifications and loop invariants, with the latter often having more than twenty components; manually apply lemmas to prove numerous complex entailments; and manually unfold user-defined predicates at times (the folding is automated) (~1.1kloc).

On a machine with an Intel Core i7-4980HQ CPU 2.80 GHz, DDR3 RAM 16GB, and a 256GB solid-state hard-drive running macOS, the JS verification takes approximately 45 seconds and the C verification takes approximately six minutes. The C time is longer, in part due to the larger codebase, but mainly due to the complexity of the implementation of the full C memory model, which is able to reason about arrays of symbolic size. This requires frequent satisfiability checks and (for the moment) branching on non-zero array size. These times could both be improved with the implementation of basic merging techniques.

JS Verification: Bugs/Improvements. We discovered two bugs and improved one function implementation to link better with the underlying data structure.

- In the `decodeEncryptionContext` function, the object representing the deserialised EC originally had prototype `Object.prototype` which, in this case, due to the prototype inheritance of JavaScript, meant that if an EC key coincided with a property of `Object.prototype`, an error would be thrown incorrectly. This bug was predicted theoretically in [21], and has since been found in several real-world libraries [42], including `cash` and `jQuery`.
- In the same function, in one of the branches the deserialised EC was returned non-frozen, which constituted a potential vulnerability in that third parties could alter non-secret, but authenticated data.
- The `readElements(eC, fC, buf, pos)` function, which reads `eC` elements with `fC` fields from buffer `buf` at index `pos` into a JS array of arrays, was misaligned

with the underlying data structures. Its parameters were non-intuitive (it received `eC·fC`, `buf`, and `pos`), and used complex array operations to re-form the final return value. We re-implemented this function to construct the returned array of arrays efficiently, simplifying specification and verification, and our implementation was integrated into the codebase.

JS Verification: Caveats. Our JS verification is correct up to the following caveats. First, as the AWS SDK JS implementation is written in TypeScript, we elide types to obtain JS; this could be automated, potentially generating predicates from the types. Next, some ES6 features, such as patterns in function parameters, are not yet supported by Gillian-JS; these we rewrite to ES5 Strict, preserving their meaning. Next, we use axiomatic specifications of the ArrayBuffer, DataView, and UInt8Array ES6 built-in libraries, as well as of the `Object.freeze` and `Array.prototype.map` built-in functions. These would ideally be accompanied with implementations, tested against the official Test262 test suite [16] and verified against their specifications. Finally, as Gillian does not support higher-order reasoning, we axiomatise the `toUtf8` function, passed into the deserialisation module as a parameter, as an injective function from raw bytes to JS strings.

C Verification: Bugs. We discovered three bugs: one logical error; one undefined behaviour; and one over-allocation.

- The deserialisation of the EC mishandled the case when there is not enough data to read it entirely, continuing to read the EDK instead of reporting an error. This allows some malformed headers to be parsed as well-formed.
- The function `aws_byte_cursor_advance`, when called with a `NULL` cursor and a length of 0, resulted in `NULL` + `0` being computed, which is undefined behaviour, although not problematic for most compilers.
- The deserialised EC was stored using `aws_string`, which extends C strings with certain metadata. It is implemented using a structure that includes a flexible array member. We discovered that string creation over-allocated this array by 8 bytes, because our (correct) predicate describing `aws_string`s was not allowing the verification to go through.

C Verification: Caveats. Our C verification is correct up to the following caveats. First, we do not use the `aws_byte_cursor_advance_nospec` function, which advances the byte cursor, but also uses complex computation to protect against the Spectre bug. We instead use `aws_byte_cursor_advance`, which has equivalent behaviour, as our specifications are not expressive enough to capture this distinction. Next, we axiomatise the functions of the AWS hash tables and array list libraries, as their verification is of comparable complexity to the entire deserialisation module. Finally, the AWS allocators of the C implementation, which are passed into some of the functions, contain pointers to memory management functions; this is higher-order in nature. In the verification, we assume those functions are `malloc`, `calloc`, and `realloc`.

6 Related Work

The literature explores many techniques and tools for verifying JS [44,18,22,21] and C [23,26,28,13,7]. We describe: multi-language verification architectures; JS and C verification tools based on separation logic; C memory models related to our models; and other analyses applied to the AWS Encryption SDK.

Multi-Language Verification Architectures. The multi-language verification architectures closest to Gillian are coreStar [6] and Viper [36,35]. Both of these architectures were designed to serve as verification back-ends for TLs and both have at their core a simple intermediate representation with a dedicated symbolic execution engine[13]. However, they work with the TL in different ways.

In coreStar, TL core assertions are modelled as abstract predicates and memory actions as function calls. The function specifications play the role of our consumer and producer actions. The user also has to provide logical axioms, describing properties of the abstract predicates. The Gillian equivalent of these axioms are the implementations of the memory actions using consumers and producers, which can be optimised, but require understanding of the inner workings of Gillian. Like Gillian, coreStar's symbolic execution engine is parametric on the underlying logical theory and can thus be used to reason about any memory model representable using abstract predicates. It is, however, unclear how efficiently this can be done. coreStar has been used inside the tool jStar [15], which has verified implementations of several Java design patterns but was not pushed to more complex Java code. In [21], the authors observed that coreStar was not able to handle tractably even simple JS programs.

Unlike Gillian and coreStar, Viper [35,36] comes with a fixed intermediate language, also called Viper. The user encodes their memory model and corresponding core assertions into the memory model and assertion language of Viper. A key advantage of Viper lies in its expressive permission model, which includes fractional, recursive, and abstract read permissions, as well as in its support for custom mathematical domains, which enable users to extend Viper with their own first-order theories, tailored to the data structures at hand. Viper has mechanisms similar to our consumer and producer actions, called *inhale* and *exhale*. Viper can reason about both sequential and concurrent programs, and has been used to verify programs written in Java, Go, Rust, and Python, but not JS and C. In fact, it is not clear to us how difficult it would be to use Viper to reason about JS objects and the linear memory of C, as neither can be simply expressed using the static objects natively provided by Viper.

Semi-automatic JS and C Verification Tools. There are very few verification tools for JS based on separation logic. For example, JaVerT [21] has been used to verify simple sequential data-structure algorithms. Its successor, JaVerT 2.0 [22], provides whole-program symbolic testing, verification and bi-abductive reasoning [10], unified by a core symbolic execution engine.

[13] Viper includes both a symbolic execution engine and a verification condition generator based on Boogie [5] for its intermediate language.

JaVerT 2.0 verification is more efficient than JaVerT verification, but has still only been applied to simple data-structure algorithms. Gillian [19] builds on JaVerT 2.0, taking the highly non-trivial step of designing the intermediate language, correctness results, and implementation to be parametric on the TL memory models. Despite this generalisation, Gillian substantially outperforms JaVerT 2.0, both for symbolic testing [19] and for verification.

Verifast [26] and the tool in [7] are prominent examples of semi-automatic tools that provide functionally-correct verification of C programs using separation-logic specifications. These tools work with C fragments and simplified memory models. While the tool in [7] has not been applied to real-world code, Verifast has been used to verify, e.g., an implementation of a Policy Enforcement Point (PEP) for Network Admission Control scenarios [38]. One difference between these tools and Gillian is that Gillian specifications can express negative resource, allowing us to differentiate missing resource errors from use-after-free errors. However, Verifast, unlike Gillian, supports reasoning about concurrent programs. There is also much work on using theorem provers to verify both sequential and concurrent C code using separation logic: see, for example, the DeepSpec project [45] and the Iris project [47], which we do not describe here.

Related Formal C Memory Models. Our compositional C memory models were inspired by CompCert [32] and the CH20 formalisation of Krebbers [29]. In particular, our concrete C model is adapted from the complete model of CompCert, which supports reasoning about programs that access in-memory data representations. This feature is used by the AWS deserialisation algorithm, which reads the buffer contents at the byte-granularity.

We present our compositional symbolic C memory model in this paper as a simple lifting of the concrete one. Our implementation is more complex, however, representing blocks as trees holding symbolic values and combining the concepts of memory trees and abstract values from the concrete memory model of the CH2O formalisation. Although not mentioned in [29], CH2O does keep track of some negative resource in that it maintains freed locations, but not block bounds.

Analysis of the AWS Encryption SDK. Amazon has recently directed considerable effort towards the formal analyses of their codebase, with a number of tools incorporated into their CI pipeline. For example, the main cryptographic algorithms of the AWS Encryption SDK have certified implementations in the specification language Cryptol [17], underpinned by SAW [12]. These implementations, however, have not yet been proven equivalent to the corresponding C implementation. In addition, the C implementation of the AWS Encryption SDK includes a symbolic test suite run using CBMC [30]. This implementation makes heavy use of the aws-c-common data-structure library, which is annotated with first-order assertions checked by CBMC. CBMC is a mature, industrial-strength tool, likely to outperform and have broader coverage than the symbolic testing of Gillian-C, with substantially fewer annotations than Gillian verification. However, as CBMC is a bounded model checker, it provides weaker correctness guarantees and is not compositional. Its expressivity is also somewhat constrained by the expressivity of the C runtime. For example, it does not allow reasoning about the size of allocated memory. Gillian specifications have this expressivity, as highlighted by the discovered over-allocation bug. The subtle logical bug found by Gillian also demonstrates the importance of being able to express full, functionally-correct specifications. We believe there has been no previous analysis of the JS implementation of AWS Encryption SDK.

7 Conclusions

We have introduced compositional verification to the Gillian platform. Our work includes a methodology for designing compositional TL memory models, distinguishing negative resource from missing resource and using the JS and C memory models as demonstrator examples. It also includes a novel, parametric approach to assertion interpretation, independent of the TL, enabling compositional use of function specifications in verification. We have been able to push the Gillian verification to self-contained, critical, real-world AWS JS and C code. The bugs and suggestions for code improvements that arose during this verification process have all been accepted by the developers and incorporated into the codebase. To our knowledge, this is the first time that industry-grade JS code has been fully verified and the first time that, in one verification platform, the same abstractions were used to verify industry code from languages as different as JS and C. The artifact accompanying this paper can be found at [34], and the entire Gillian development at [46]. In future, we will publish correctness results for Gillian verification [33], as part of an in-depth theoretical study of program correctness and incorrectness for symbolic testing, verification and bi-abductive reasoning being developed in Gillian.

Acknowledgments. We thank AWS engineers and Mike Dodds from Galois for several inspiring meetings which led to our focus on verifying the JS implementation of the AWS Encryption SDK message header deserialisation module. We would especially like to thank Ryan Emery for many detailed discussions about his JS code. We thank the reviewers, whose comments have improved the overall quality of the paper. Gardner and Maksimović were supported by the EPSRC Fellowship ‘VetSpec: Verified Trustworthy Software Specification’ (EP/R034567/1). Fragoso Santos was supported by national funds through Fundação para a Ciência e a Tecnologia (UIDB/50021/2020, INESC-ID multi-annual funding) and project INFOCOS (PTDC/CCI-COM/32378/2017). Ayoun was supported by a Department of Computing PhD Scholarship from Imperial.

References

1. Amazon Web Services: AWS Encryption SDK: C Implementation. https://github.com/aws/aws-encryption-sdk-c (2020)
2. Amazon Web Services: AWS Encryption SDK: JS Implementation. https://github.com/aws/aws-encryption-sdk-javascript (2020)

3. Amazon Web Services: AWS Encryption SDK: Message Format. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/message-format.html (2020)
4. Appel, A.W., Blazy, S.: Separation Logic for Small-Step Cminor. In: TPHOL (2007)
5. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO (2005)
6. Botinčan, M., Distefano, D., Dodds, M., Grigore, R., Naudžiūnienė, D., Parkinson, M.J.: coreStar: The core of jstar. In: Boogie (2011)
7. Botinčan, M., Parkinson, M.J., Schulte, W.: Separation Logic Verification of C Programs with an SMT Solver. Electr. Notes Theor. Comput. Sci. **254**, 5–23 (2009)
8. Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: NFM (2011)
9. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving Fast with Software Verification. In: NFM (2015)
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. JACM **58**, 26:1–26:66 (2011)
11. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS (2007)
12. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous Formal Verification of Amazon s2n. In: CAV (2018)
13. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOL (2009)
14. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP (2010)
15. Distefano, D., Parkinson, M.: jStar: Towards practical verification for Java. In: OOPSLA (2008)
16. ECMA TC39: Test262 Test Suite. https://github.com/tc39/test262 (2017)
17. Erkök, L., Matthews, J.: High Assurance Programming in Cryptol. In: CSIIRW (2009)
18. Fournet, C., Swamy, N., Chen, J., Dagand, P.E., Strub, P.Y., Livshits, B.: Fully Abstract Compilation to JavaScript. In: POPL (2013)
19. Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, Part I: A Multi-language Platform for Symbolic Execution. In: PLDI (2020)
20. Fragoso Santos, J., Maksimović, P., Grohens, T., Dolby, J., Gardner, P.: Symbolic Execution for JavaScript. In: PPDP (2018)
21. Fragoso Santos, J., Maksimović, P., Naudžiūnienė, D., Wood, T., Gardner, P.: JaVerT: JavaScript Verification Toolchain. PACMPL **2**(POPL), 50:1–50:33 (2018)
22. Fragoso Santos, J., Maksimović, P., Sampaio, G., Gardner, P.: JaVerT 2.0: Compositional Symbolic Execution for JavaScript. PACMPL **3**(POPL), 66:1–66:31 (2019)
23. Frumin, D., Gondelman, L., Krebbers, R.: Semi-automated Reasoning About Non-determinism in C Expressions. In: PLS (2019)
24. Gardner, P., Maffeis, S., Smith, G.D.: Towards a Program Logic for JavaScript. In: POPL (2012)
25. Ishtiaq, S.S., O'Hearn, P.W.: BI as an Assertion Language for Mutable Data Structures. In: Hankin, C., Schmidt, D. (eds.) POPL (2001)

26. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: NFM (2011)
27. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28** (2018)
28. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015)
29. Krebbers, R.: A Formal C Memory Model for Separation Logic. Journal of Automated Reasoning **57**(4), 319–387 (2016)
30. Kroening, D., Tautschnig, M.: CBMC: C bounded model checker. In: TACAS (2014)
31. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (2012)
32. Leroy, X.: Formal Verification of a Realistic Compiler. Commun. ACM **52**(7), 107–115 (2009)
33. Maksimović, P., Santos, J.F., Ayoun, S.E., Gardner, P.: Gillian: A Multi-Language Platform for Unified Symbolic Analysis (2021), `http://arxiv.org/abs/2105.14769`
34. Maksimović, P., Ayoun, S.E., Fragoso Santos, J., Gardner, P.: Artifact: Gillian, Part II: Real-World Verification for JavaScript and C (2021), `https://doi.org/10.5281/zenodo.4838116`
35. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: VMCAI (2016)
36. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Dependable Software Systems Engineering (2017)
37. Nguyen, H.H., Kuncak, V., Chin, W.N.: Runtime checking for separation logic. In: VMCAI. pp. 203–217 (2008)
38. Philippaerts, P., Mülberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software Verification with VeriFast: Industrial Case Studies. Science of Computer Programming **82**, 77–97 (2014)
39. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In: CAV (2020)
40. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS (2002)
41. S. Panić: Collections-C: A Library of Generic Data Structures. `https://github.com/srdja/Collections-C` (2014)
42. Sampaio, G., Santos, J.F., Maksimović, P., Gardner, P.: A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications. In: ECOOP (2020)
43. Santos, M.: Buckets-js: A javascript data structure library. `https://github.com/mauriciosantos/Buckets-JS` (2016)
44. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying Higher-order Programs with the Dijkstra Monad. In: PLDI (2013)
45. The DeepSpec Team: The DeepSpec Project. `https://deepspec.org/main` (2021)
46. The Gillian Team: Gillian. `https://gillianplatform.github.io` (2020)
47. The Iris Team: The Iris Project. `https://iris-project.org` (2021)
48. Watt, C., Maksimović, P., Krishnaswami, N.R., Gardner, P.: A Program Logic for First-Order Encapsulated WebAssembly. In: ECOOP (2019)