Formal Methods Assessment Report

Agoric Swingset Kernel and Userspace, Phase 1: Source Code Inspection and Protocol Modelling

09.11.2021

Initial revision: 08.07.2021

Authors: Andrey Kuprianov

Contents

Assessment overview 4
- The project 4
- The Agoric SwingSet Platform 4
- Scope of this report 5
- Conducted work 5
- Timeline 6
- Conclusions 6

Assessment Dashboard 7

Assessment Scope 8

TLA+ SwingSet Kernel Model 9
- Kernel types 9
- Kernel variables and actions 9
- Kernel execution 11
- Kernel tests 12
- Future model evolution 13

Findings 14

IF-AGORIC1-01: Mismatched number of arguments 15
- Involved artifacts 15
- Description 15
- Recommendation 16

IF-AGORIC1-02: Incomplete type-tracking in the kernel 17
- Involved artifacts 17
- Description 17
- Recommendation 17

IF-AGORIC1-03: Undocumented function format change 18
- Involved artifacts 18
- Description 18
- Recommendation 19

IF-AGORIC1-04: Problematic pattern in processQueueMessage 20
- Involved artifacts 20
- Description 20
- Recommendation 20

IF-AGORIC1-05: Unrestricted number of vat slots per vat 21
- Involved artifacts 21
- Description 21
- Recommendation: 22

IF-AGORIC1-06: Unrestricted length of vat slot IDs 23
- Involved artifacts 23
- Description 23
- Recommendation 23

IF-AGORIC1-07: Scattered updates to kernel tables **24**
Involved artifacts 24
Description 24
Recommendation 24

IF-AGORIC1-08: Possible cycles in promise resolutions **25**
Involved artifacts 25
Description 25
Recommendation 25

IF-AGORIC1-09: Possible loss of vat termination events **26**
Involved artifacts 26
Description 26
Recommendation 26

IF-AGORIC1-10: Inconsistencies in vat bookkeeping **27**
Involved artifacts 27
Description 27
Recommendation 28

Assessment overview

The project

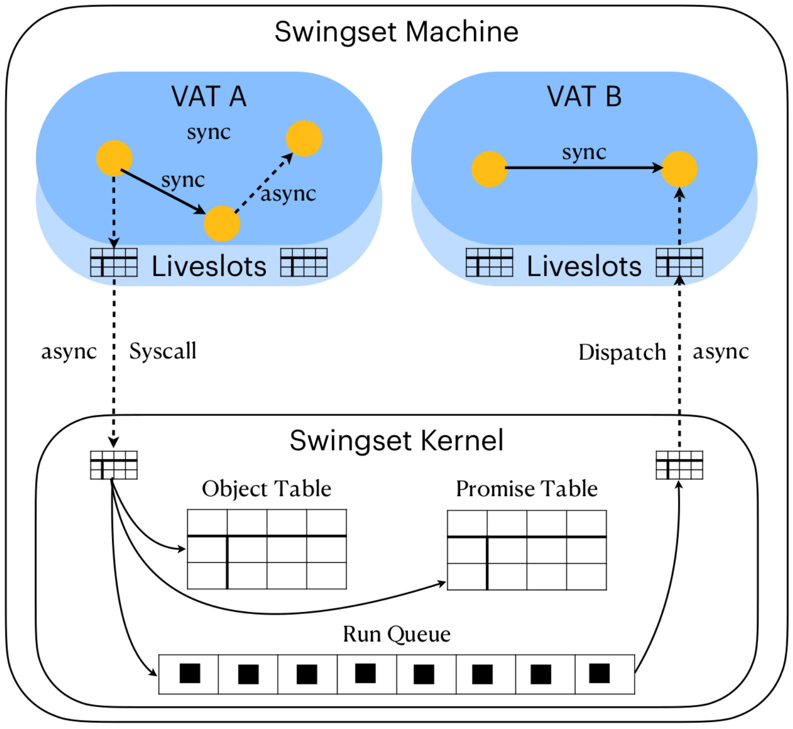
In April 2021, Agoric engaged Informal Systems to conduct a formal methods assessment over the documentation and the current state of the implementation of *Agoric SwingSet Kernel*: the core component of the Agoric smart contracts platform.

The Agoric SwingSet Platform

The Agoric SwingSet is the basis for development with the Agoric SDK, a development kit for writing smart contracts in a secure, hardened version of JavaScript. Smart contracts built with the Agoric SDK have mediated asynchronous communication, and messages can only be sent along references according to the rules of the Object Capabilities model that the SwingSet implements.

The Object Capabilities (OCAP) model is a model for reasoning about communication. An Object-Capability is a transferrable, unforgeable authorization to use a designated object. The SwingSet machine allows JavaScript code to communicate according to the model, and executes code in a userspace similar to that offered by a Unix operating system. The SwingSet kernel component operates analogously to a Unix kernel, and vats correspond to Unix userspace processes. The kernel provides services for isolation, composition, and communication between vats: it enforces the OCAP properties.

The figure below shows a diagram of the architecture of the SwingSet kernel and two interacting vats. Each vat is a unit of synchrony and synchronous communication only occurs inside a single vat. The liveslots layers mediate access to the outside world from userspace code. Every remote object access is implemented via translation tables in the liveslots layers and the kernel, which operates by pulling work off from a run queue.



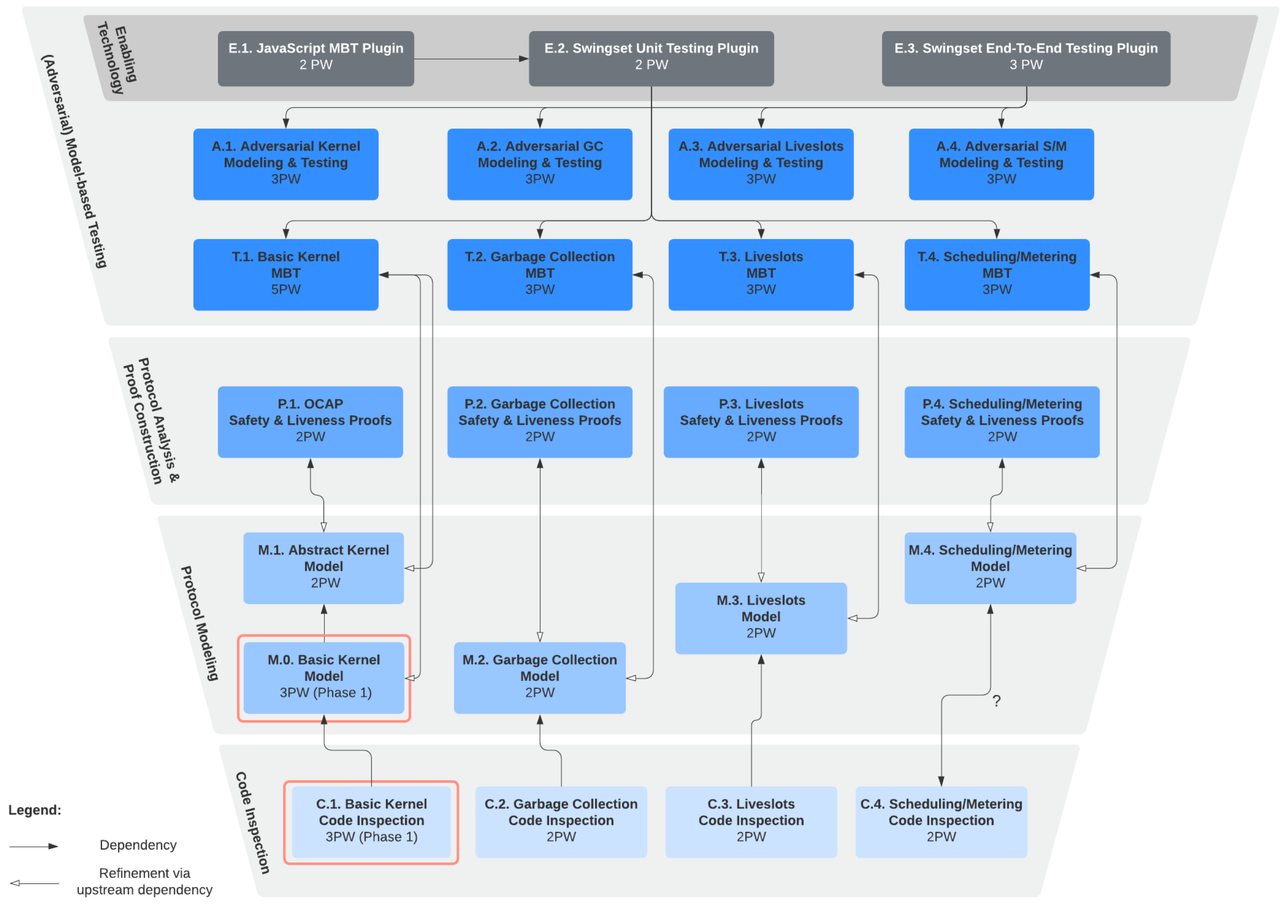
Scope of this report

The agreed-upon workplan consisted of the following tasks:

- Task C.1. Examine the documentation and source code of SwingSet kernel, excluding the liveslots.
  - a rolling review over the assessment period, starting from commit 23ed67c0
- Task M.0. Construct the TLA+ model of the SwingSet kernel, modeling key interactions in the kernel:
  - interactions with vats via the syscall interface;
  - lifetime and evolution of kernel objects and promises;
- Task P.1. Perform bounded verification of the TLA+ model focusing on OCAP model requirements.

Taking into account the large size of the code base, it was decided that Informal Systems would only perform the source code and documentation review limited to the scope necessary to construct the TLA+ kernel model. Moreover, due to the inherent complexity of formal protocol modelling, it was decided that the scope of the assessment be determined by the time span allocated for it with some tasks continuing if necessary during the follow-up phases.

This report covers Task C.1 and Task M.0 that were conducted May 6 through June 29, 2021 by Andrey Kuprianov, Senior Research Engineer at Informal Systems; Task P.1 turned out to be infeasible due to the timing constraints, and will be conducted in a follow-up phase.


Enabling Technology
(Adversarial) Model-based Testing
E.1. JavaScript MBT Plugin 2 PW
E.2. Swingset Unit Testing Plugin 2 PW
E.3. Swingset End-To-End Testing Plugin 3 PW
A.1. Adversarial Kernel Modeling & Testing 3PW
A.2. Adversarial GC Modeling & Testing 3PW
A.3. Adversarial Liveslots Modeling & Testing 3PW
A.4. Adversarial S/M Modeling & Testing 3PW
T.1. Basic Kernel MBT 5PW
T.2. Garbage Collection MBT 3PW
T.3. Liveslots MBT 3PW
T.4. Scheduling/Metering MBT 3PW
Protocol Analysis & Proof Construction
P.1. OCAP Safety & Liveness Proofs 2PW
P.2. Garbage Collection Safety & Liveness Proofs 2PW
P.3. Liveslots Safety & Liveness Proofs 2PW
P.4. Scheduling/Metering Safety & Liveness Proofs 2PW
Protocol Modeling
M.1. Abstract Kernel Model 2PW
M.0. Basic Kernel Model 3PW (Phase 1)
M.2. Garbage Collection Model 2PW
M.3. Liveslots Model 2PW
M.4. Scheduling/Metering Model 2PW
?
Code Inspection
C.1. Basic Kernel Code Inspection 3PW (Phase 1)
C.2. Garbage Collection Code Inspection 2PW
C.3. Liveslots Code Inspection 2PW
C.4. Scheduling/Metering Code Inspection 2PW
Legend:
Dependency
Refinement via upstream dependency

Conducted work

Starting May 6, Informal Systems conducted an assessment of the existing documentation and the code. Agoric gave a one-hour presentation with an overview over the protocol and a code walk-through with focus on the scope of the assessment. Our team started with reviewing SwingSet kernel documentation, to get an overview of the kernel design principles, and with the review of some critical components of Hardened JavaScript, which are foundational to platform security. We then continued with the review of the SwingSet kernel source code; the code inspection was limited to the scope necessary to construct the TLA+ kernel model. After gaining a general understanding of kernel protocols and interactions, we continued with the modelling effort, and accurately captured interactions within the kernel in the TLA+ Swingset kernel model.

Over the Keybase channel between Agoric and Informal we shared documents with preliminary findings, which we discussed during online meetings. In this document, we distilled the central findings into numbered findings, as well as briefly described the characteristics of the constructed TLA+ kernel model. Moreover, we opened issues for the findings in the Agoric-SDK repository.

Timeline

- 06.05.2021: Start of assessment
- 06.05.2021: Kickoff meeting with a code walk-through (1 hour)
- 21.05.2021: Meeting Agoric/Informal with discussion of the first set of issues
- 14.06.2021: Meeting Agoric/Informal with discussion of the second set of issues, and the first version of the TLA+ kernel model
- 29.06.2021: Meeting Agoric/Informal with discussion of the final version of the TLA+ kernel model
- 29.06.2021: End of assessment
- 08.07.2021: Submission of the first draft of this report
- 02.11.2021: Correction suggestions received from Agoric
- 09.11.2021: Submission of the final version of this report

Conclusions

Overall we found the code to be well organized, well documented, and faithful to the specification. Despite the general high quality of the implementation work, we found several significant issues regarding code quality, data representation, code organization, and divergence from the specification. These are detailed in the relevant findings. The main source of issues seems to be the choice of JavaScript as the kernel implementation language with its weak and dynamic typing. While this choice is convenient in some cases and makes the implementation more concise, introducing strict type tracking into the kernel seems to be able to bring substantial benefits, and to eliminate many sources of potential bugs. As the first step in that direction, we have provided within the scope this assessment a typed TLA+ specification of the core kernel functionality. Already the process of the model construction has helped to identify some protocol and implementation issues; continuing this thread with model checking and model-based testing should bring a high level of assurance in the kernel correctness.

We have also identified two resource exhaustion attacks that have not yet been dealt with in the current implementation, but may be resolved through the upcoming addition of metering to the kernel.

Assessment Dashboard

Target Summary

- **Name**: Agoric SwingSet Kernel
- **Specification & Code Version**: commits 23ed67c0 through 5e1024e4
- **Type**: Specification and Implementation
- **Platform**: JavaScript

Engagement Summary

- **Dates**: 10.05.2021 – 29.06.2021
- **Method**: Manual review & formal protocol modelling
- **Employees Engaged**: 1
- **Time Spent**: 6 person-weeks

Severity Summary

Finding Severity	#	Findings
High-Severity Issues	0	
Medium-Severity Issues	4	05, 08, 09, 10
Low-Severity Issues	5	01, 02, 03, 04, 06
Informational-Severity Issues	1	07
Total	**10**	

Finding Severities

- *Informational*: The issue does not pose an immediate risk (it is subjective in nature). Findings with informational severity are typically suggestions around best practices or readability.
- *Low*: The issue is objective in nature, but the security risk is relatively small or does not represent a security vulnerability.
- *Medium*: The issue is a security vulnerability that may not be directly exploitable or may require certain complex conditions in order to be exploited.
- *High*: The issue is an exploitable security vulnerability.

Category Breakdown

Finding Category	#	Findings
Protocol	0	
Implementation	4	05, 06, 08, 09,
Code Structure	6	01, 02, 03, 04, 07, 10
Total	**10**	

Finding Categories

- *Protocol*: A flaw or problem in an abstract protocol or algorithm.
- *Implementation*: A problem with the source code. For example a bug, a divergence from a specification, or a poor choice of data structure.
- *Code structure*: A problem that impacts the extent to which the project is maintainable and understandable by developers in the long term.
- *Documentation*: A lack of documentation, or insufficient clarity, accuracy, understandability or readability of existing documentation.

Assessment Scope

The scope of the assessment has been defined as the Agoric SDK SwingSet kernel, with minimal dependencies necessary to construct the formal kernel model.

During the assessment, we have inspected the following source repositories:

- `endojs/endo/packages/ses`:
  - commit f7dcf050
- `Agoric/agoric-sdk/packages/SwingSet/kernel` (a rolling review over the assessment period):
  - 10.05 - 19.05: commit 23ed67c0
  - 20.05 - 26.05: commit 0cae5d77
  - 27.05 - 01.06: commit 069201d6
  - 02.06 - 06.06: commit 9feea167
  - 07.06 - 10.06: commit ae2ac52fc
  - 11.06 - 29.06: commit 5e1024e4

TLA+ SwingSet Kernel Model

One of the main goals of the formal methods assessment was constructing a formal TLA+ model of the SwingSet kernel that faithfully represents interactions with vats via the syscall interface as well as the lifetime and evolution of kernel objects and promises.

Kernel types

The file kernel_typedef.tla describes kernel-specific types and typed constants. Here is the excerpt with the most important kernel types:

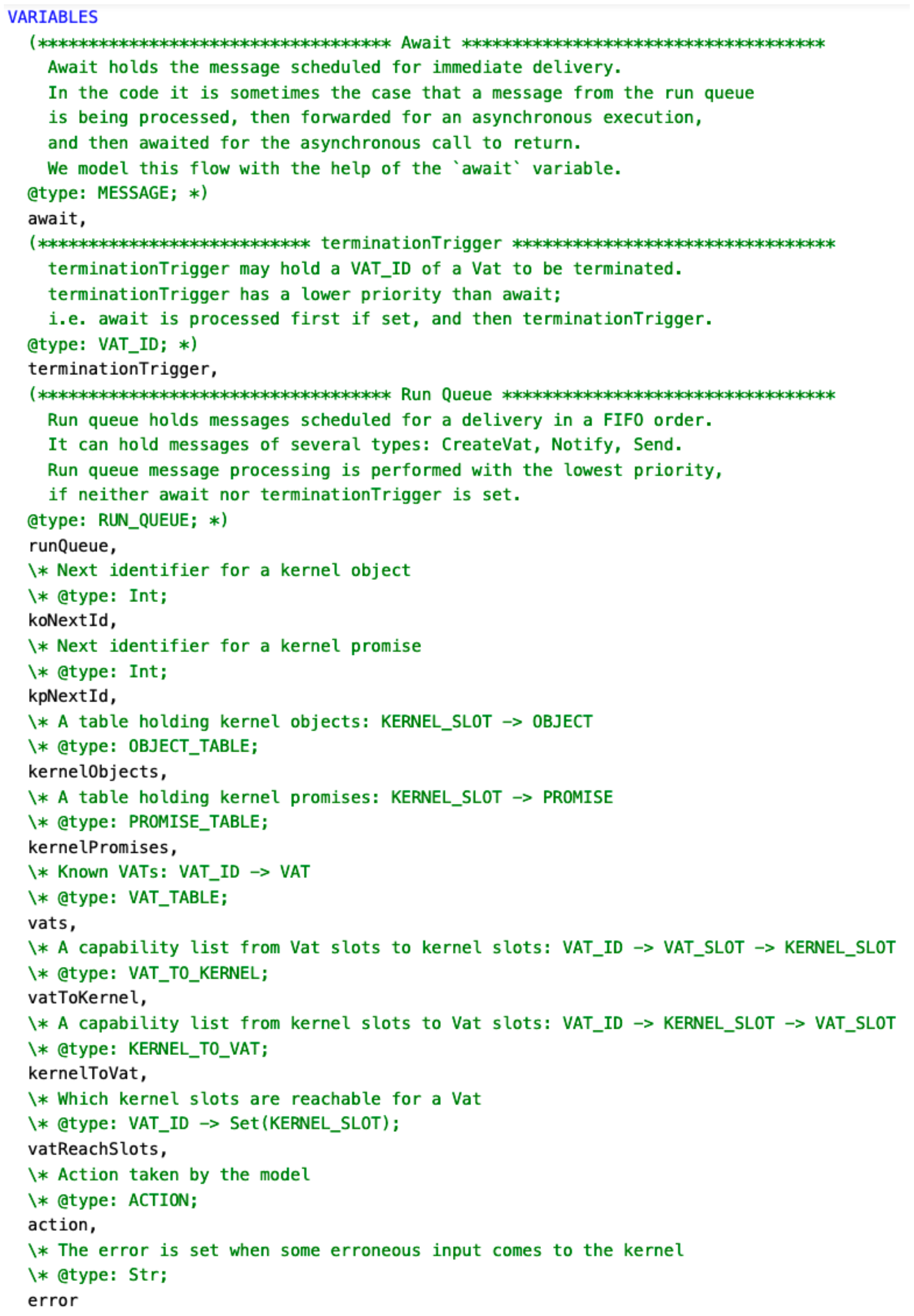
```
(*
    @typeAlias: VAT_SLOT = [ type: VAT_SLOT_T, id: Int, exported: Bool ];
    @typeAlias: VAT = [ oNextId: Int, pNextId: Int, enablePipelining: Bool ];

    @typeAlias: KERNEL_SLOT = [ type: KERNEL_SLOT_T, id: Int ];

    @typeAlias: VAT_MESSAGE = [ result: KERNEL_SLOT, slots: Set(KERNEL_SLOT)];
    @typeAlias: MESSAGE = [ type: MESSAGE_T, vatID: VAT_ID, enablePipelining: Bool,
                            kpid: KERNEL_SLOT, target: KERNEL_SLOT, vatMsg: VAT_MESSAGE];

    @typeAlias: RUN_QUEUE = Seq(MESSAGE);

    @typeAlias: RESOLUTION = [ kpid: KERNEL_SLOT, rejected: Bool, slots: Seq(KERNEL_SLOT) ];

    @typeAlias: OBJECT = [ owner: VAT_ID ];
    @typeAlias: PROMISE = [state: PROMISE_T, decider: VAT_ID, subscribers: Set(VAT_ID),
                           queue: RUN_QUEUE, slots: Set(KERNEL_SLOT)];

    @typeAlias: VAT_TABLE = VAT_ID -> VAT;
    @typeAlias: VAT_TO_KERNEL = VAT_ID -> VAT_SLOT -> KERNEL_SLOT;
    @typeAlias: KERNEL_TO_VAT = VAT_ID -> KERNEL_SLOT -> VAT_SLOT;
    @typeAlias: OBJECT_TABLE = KERNEL_SLOT -> OBJECT;
    @typeAlias: PROMISE_TABLE = KERNEL_SLOT -> PROMISE;
*)
```

Kernel variables and actions

The main model file kernel.tla contains:

- variables (representing the kernel state)
- parameterized actions (which modify the state)
- action preconditions (when are the action parameters considered valid)
- variables that are changed / unchanged by each action

The kernel state in this version of the model is described by the following 13 state variables:



```
VARIABLES
  (*********************************** Await ************************************
    Await holds the message scheduled for immediate delivery.
    In the code it is sometimes the case that a message from the run queue
    is being processed, then forwarded for an asynchronous execution,
    and then awaited for the asynchronous call to return.
    We model this flow with the help of the `await` variable.
  @type: MESSAGE; *)
  await,
  (*************************** terminationTrigger ********************************
    terminationTrigger may hold a VAT_ID of a Vat to be terminated.
    terminationTrigger has a lower priority than await;
    i.e. await is processed first if set, and then terminationTrigger.
  @type: VAT_ID; *)
  terminationTrigger,
  (*********************************** Run Queue *********************************
    Run queue holds messages scheduled for a delivery in a FIFO order.
    It can hold messages of several types: CreateVat, Notify, Send.
    Run queue message processing is performed with the lowest priority,
    if neither await nor terminationTrigger is set.
  @type: RUN_QUEUE; *)
  runQueue,
  \* Next identifier for a kernel object
  \* @type: Int;
  koNextId,
  \* Next identifier for a kernel promise
  \* @type: Int;
  kpNextId,
  \* A table holding kernel objects: KERNEL_SLOT -> OBJECT
  \* @type: OBJECT_TABLE;
  kernelObjects,
  \* A table holding kernel promises: KERNEL_SLOT -> PROMISE
  \* @type: PROMISE_TABLE;
  kernelPromises,
  \* Known VATs: VAT_ID -> VAT
  \* @type: VAT_TABLE;
  vats,
  \* A capability list from Vat slots to kernel slots: VAT_ID -> VAT_SLOT -> KERNEL_SLOT
  \* @type: VAT_TO_KERNEL;
  vatToKernel,
  \* A capability list from kernel slots to Vat slots: VAT_ID -> KERNEL_SLOT -> VAT_SLOT
  \* @type: KERNEL_TO_VAT;
  kernelToVat,
  \* Which kernel slots are reachable for a Vat
  \* @type: VAT_ID -> Set(KERNEL_SLOT);
  vatReachSlots,
  \* Action taken by the model
  \* @type: ACTION;
  action,
  \* The error is set when some erroneous input comes to the kernel
  \* @type: Str;
  error
```

Each parameterized action is modelled after the source code, by reproducing at the abstract level what happens in the implementation. Here we show only one such parameterized action, `CreateVat`:

```
(*********************************** Create Vat ******************************
  CreateVat is triggered when a "create-vat" run queue message is processed.
******************************************************************************)

\* kernel.js processCreateVat(): https://git.io/JnvdR
\* type: (VAT_ID, Bool) => Bool
CreateVatPre(vatID, enablePipelining) ==
  UnknownVat(vatID)

CreateVatChanges == <<vats, vatToKernel, kernelToVat, vatReachSlots>>
CreateVatNotChanges == <<await, terminationTrigger, runQueue, koNextId,
                         kpNextId, kernelObjects, kernelPromises>>

\* kernel.js processCreateVat(): https://git.io/JnvdR
\* @type: (VAT_ID, Bool) => Bool;
CreateVat(vatID, enablePipelining) ==
  /\ vats' = [k \in DOMAIN vats \union {vatID} |->
      IF k = vatID THEN NEW_VAT(enablePipelining) ELSE vats[k] ]
  /\ vatToKernel' = [k \in DOMAIN vatToKernel \union {vatID} |->
      IF k = vatID THEN EMPTY_VAT_TO_KERNEL_ENTRY ELSE vatToKernel[k] ]
  /\ kernelToVat' = [k \in DOMAIN kernelToVat \union {vatID} |->
      IF k = vatID THEN EMPTY_KERNEL_TO_VAT_ENTRY ELSE kernelToVat[k] ]
  /\ vatReachSlots' = [k \in DOMAIN vatReachSlots \union {vatID} |->
      IF k = vatID THEN {} ELSE vatReachSlots[k] ]
```

Kernel execution

The file kernel_exec.tla defines execution semantics for the kernel:

- constants (representing the model search space)
- initialization of state variables and constants
- externally observable actions
- scheduling of actions

The model concisely describes the whole kernel state (E stands for Exported, I for Imported, V for Vat, K for Kernel, O for Object, P for Promise; thus e.g. EVO stands for Exported Vat Object):

```
\* An initial kernel state with some default contents in tables
DefaultInit ==
  InitDefaultsExcept([
    koNextId |-> 3,
    kpNextId |-> 3,
    kernelObjects |-> KO(1) :> NEW_OBJECT("a") @@ KO(2) :> NEW_OBJECT("b"),
    kernelPromises |-> KP(1) :> NEW_PROMISE("a") @@ KP(2) :> NEW_PROMISE("b"),
    vats |-> "a" :> NEW_VAT(FALSE) @@ "b" :> NEW_VAT(FALSE),
    vatToKernel   |-> "a" :> (EVO(1) :> KO(1) @@ IVP(1) :> KP(2)) @@
                      "b" :> (EVO(1) :> KO(2) @@ IVP(1) :> KP(1)),
    kernelToVat   |-> "a" :> (KO(1) :> EVO(1) @@ KP(2) :> IVP(1)) @@
                      "b" :> (KO(2) :> EVO(1) @@ KP(1) :> IVP(1)),
    vatReachSlots |-> "a" :> { KO(1), KP(2)} @@  "b" :> { KO(2), KP(1)}
  ])
```

This file also defines the externally observable actions and steps based on the parameterized actions from kernel.tla Actions are defined as follows:

- Do not change unchanged variables (this allows to search over all possible inputs)
- Save the action being executed in the "action" variable

- If the action precondition is satisfied, perform an update
- Otherwise do not perform an update, but set the “error” variable instead

Steps differ from actions in that they quantify existentially on action parameters, using model constants

```
CreateVatAction(vatID, enablePipelining) ==
  UNCHANGED CreateVatNotChanges /\
    action' = [ type |-> "CreateVat", vatID |-> vatID,
                enablePipelining |-> enablePipelining ] /\
    IF CreateVatPre(vatID, enablePipelining) THEN
       /\ CreateVat(vatID, enablePipelining)
       /\ error' = NULL
    ELSE
       /\ UNCHANGED CreateVatChanges
       /\ error' = "CreateVat"

CreateVatStep ==
  UNCHANGED <<await, terminationTrigger>> /\
  \E vatID \in VAT_IDS:
  \E enablePipelining \in {TRUE, FALSE}:
    CreateVatAction(vatID, enablePipelining)
```

The execution semantics is defined in terms of prioritized processing of enabled steps:

```
\* Kernel scheduling happens in the priority order:
\* - awaited step
\* - termination step
\* - any other step
DefaultNext ==
  IF await /= NoMessage THEN
    AwaitStep
  ELSE IF terminationTrigger /= NULL_VAT_ID THEN
    TerminationStep
  ELSE
    \/ ProcessQueueMessageStep
    \/ CreateVatStep
    \/ ExportStep
    \/ ImportStep
    \/ SendStep
    \/ SubscribeStep
    \/ ResolveStep
    \/ ExitStep
```

Kernel tests

Finally, the file kernel_test.tla defines unit tests for the kernel model; those tests serve as simple soundness checks for the model, ensuring e.g. that all variables are updated correctly on all logical branches. Below are a couple of tests for the `CreateVat` action which make sure that the creation of (un)known vat is processed soundly.

```
TestCreateUnknownVatCInit == CInit
TestCreateUnknownVatBefore ==
  InitDefaultsExcept([
    vats |-> [ id \in {"a", "b"} |-> NEW_VAT(FALSE)]
  ])
TestCreateUnknownVatAction == CreateVatAction("c", TRUE)
TestCreateUnknownVatAssert ==
  /\ "c" \in DOMAIN vats
  /\ "c" \in DOMAIN vatToKernel
  /\ "c" \in DOMAIN kernelToVat


TestCreateKnownVatCInit == CInit
TestCreateKnownVatBefore ==
  InitDefaultsExcept([
    vats |-> [ id \in {"a", "b"} |-> NEW_VAT(FALSE)]
  ])
TestCreateKnownVatAction == CreateVatAction("a", TRUE)
TestCreateKnownVatAssert ==
  /\ error = "CreateVat"
```

Future model evolution

The current version of the SwingSet kernel model accurately represents the interactions between kernel and vats, and the life cycle of kernel objects and promises. Unfortunately, the precise model is too heavy in terms of the state space for the invariants to be efficiently model checked. We can identify the following future steps:

- Formulate OCAP model properties (such as "connectivity begets connectivity") as model invariants;
- Construct specialized, abstracted versions of the main kernel model for:
  - OCAP invariants;
  - Kernel garbage collection protocol and its invariants;
  - Kernel scheduling/metering and its invariants;
- Prove refinement relations between the main model and its abstracted variants;
- Perform bounded verification of the above invariants on the abstracted variants of the kernel model;
- Construct and execute model-based tests, thus providing security assurance via checking implementation conformance to the formal specification.

Findings

ID	Title	Severity	Category	Issue
IF-AGORIC1-01	Mismatched number of arguments	Low	Structure	sdk#3172
IF-AGORIC1-02	Incomplete kernel type-tracking	Low	Structure	sdk#3173
IF-AGORIC1-03	Undocumented function format change	Low	Structure	sdk#3174
IF-AGORIC1-04	Problematic pattern in processQueueMessage	Low	Structure	
IF-AGORIC1-05	Unrestricted number of slots per vat	Medium	Impl.	sdk#3243
IF-AGORIC1-06	Unrestricted length of vat slot IDs	Low	Impl.	sdk#3242
IF-AGORIC1-07	Scattered updates to kernel tables	Info	Structure	sdk#3312
IF-AGORIC1-08	Possible cycles in promise resolutions	Medium	Impl.	sdk#3313
IF-AGORIC1-09	Possible loss of vat termination events	Medium	Impl.	sdk#3315
IF-AGORIC1-10	Inconsistencies in vat bookkeeping	Medium	Structure	sdk#3316

Finding Severities

- *Informational*: The issue does not pose an immediate risk (it is subjective in nature). Findings with informational severity are typically suggestions around best practices or readability.
- *Low*: The issue is objective in nature, but the security risk is relatively small or does not represent a security vulnerability.
- *Medium*: The issue is a security vulnerability that may not be directly exploitable or may require certain complex conditions in order to be exploited.
- *High*: The issue is an exploitable security vulnerability.

Finding Categories

- *Protocol*: A flaw or problem in an abstract protocol or algorithm.
- *Implementation*: A problem with the source code. For example a bug, a divergence from a specification, or a poor choice of data structure.
- *Code structure*: A problem that impacts the extent to which the project is maintainable and understandable by developers in the long term.
- *Documentation*: A lack of documentation, or insufficient clarity, accuracy, understandability or readability of existing documentation.

In the effort to improve the usefulness of this report, we do not include here descriptions for issues in the *Documentation* category; all issues can be found in the Agoric-SDK repository.

IF-AGORIC1-01: Mismatched number of arguments

Category	Code Structure
Severity	Low
Issue	agoric-sdk#3172

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/kernel.js

Description

In multiple places in `kernel.js` function resolveToError is called with 2 arguments instead of 3. At the same time, in 2 places it is actually called with 3 arguments. Here is the function:

```
  function resolveToError(kpid, errorData, expectedDecider) {
    doResolve(expectedDecider, [[kpid, true, errorData]]);
  }
```

It calls doResolve:

```
function doResolve(vatID, resolutions) {
    if (vatID) {
      insistVatID(vatID);
    }
    for (const resolution of resolutions) {
      const [kpid, rejected, data] = resolution;
      insistKernelType('promise', kpid);
      insistCapData(data);
      const p = kernelKeeper.getResolveablePromise(kpid, vatID);
      const { subscribers, queue } = p;
      let idx = 0;
      for (const dataSlot of data.slots) {
        kernelKeeper.incrementRefCount(dataSlot, `resolve|s${idx}`);
        idx += 1;
      }
      kernelKeeper.resolveKernelPromise(kpid, rejected, data);
      notifySubscribersAndQueue(kpid, vatID, subscribers, queue);
      const tag = rejected ? 'rejected' : 'fulfilled';
      if (p.policy === 'logAlways' || (rejected && p.policy === 'logFailure')) {
        console.log(
          `${kpid}.policy ${p.policy}: ${tag} ${JSON.stringify(data)}`,
        );
      } else if (rejected && p.policy === 'panic') {
        panic(`${kpid}.policy panic: ${tag} ${JSON.stringify(data)}`);
      }
    }
  }
```

This function in turn passes the (possibly undefined) `vatID` to `getResolveablePromise` and `notifySubscribersAndQueue`.

```
function getResolveablePromise(kpid, expectedDecider) {
  insistKernelType('promise', kpid);
  if (expectedDecider) {
    insistVatID(expectedDecider);
  }
  const p = getKernelPromise(kpid);
  assert(p.state === 'unresolved', X`${kpid} was already resolved`);
  if (expectedDecider) {
    assert(
      p.decider === expectedDecider,
      X`${kpid} is decided by ${p.decider}, not ${expectedDecider}`,
    );
  } else {
    assert(!p.decider, X`${kpid} is decided by ${p.decider}, not the kernel`);
  }
  return p;
}
```

This function seems to expect that `expectedDecider` may be undefined. On the other hand,

```
function notifySubscribersAndQueue(kpid, resolvingVatID, subscribers, queue) {
  insistKernelType('promise', kpid);
  for (const vatID of subscribers) {
    if (vatID !== resolvingVatID) {
      notify(vatID, kpid);
    }
  }
  // omitted for brevity
}
```

doesn't seem to care about `resolvingVatID` being possibly undefined.

The issue was uncovered by renaming `kernel.js` to `kernel.ts` and running `tsc`; see IF-AGORIC-06 for more details.

```
src/kernel/kernel.ts:405:7 - error TS2554: Expected 3 arguments, but got 2.

405       resolveToError(msg.result, VAT_TERMINATION_ERROR);
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  src/kernel/kernel.ts:339:44
    339   function resolveToError(kpid, errorData, expectedDecider) {
                                                   ~~~~~~~~~~~~~~~
    An argument for 'expectedDecider' was not provided.
```

Recommendation

While the current code may work fine with the mismatching argument lists, this may lead to errors in the future, as it introduces implicit assumptions down the whole call chain. We recommend refactoring all relevant functions into two variants: one with the `VatID` argument present, and another, without it.

IF-AGORIC1-02: Incomplete type-tracking in the kernel

Category	Code Structure
Severity	Low
Issue	agoric-sdk#3173

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/

Description

The SwingSet kernel is written in JavaScript, and thus uses weak and dynamic typing. While some typing information is tracked in the comments, this tracking is very incomplete. This may lead to a lot of subtle and undetectable bugs, that would be easy to detect and prevent if a language with strict typing is used. Luckily, there is an easy JavaScript -> TypeScript migration path, that brings a lot of benefits early. Even a very quick experiment with renaming all SwingSet `.js` files into `.ts` files, and running the TypeScript compiler, allowed to find these issues (among others):

```
> yarn tsc

src/kernel/initializeKernel.ts(23,24): error TS2554: Expected 2 arguments, but got 1.
src/kernel/initializeKernel.ts(120,60): error TS2554: Expected 1 arguments, but got 2.
src/kernel/kernel.ts(334,9): error TS2554: Expected 2 arguments, but got 1.
src/kernel/kernel.ts(405,7): error TS2554: Expected 3 arguments, but got 2.
src/kernel/kernel.ts(440,9): error TS2554: Expected 3 arguments, but got 2.
src/kernel/kernel.ts(456,17): error TS2554: Expected 3 arguments, but got 2.
src/kernel/kernel.ts(462,17): error TS2554: Expected 3 arguments, but got 2.
src/kernel/kernel.ts(477,13): error TS2554: Expected 3 arguments, but got 2.
src/kernel/liveslots.ts(534,7): error TS2322: Type '(value: any) => void' is not assignable to type '() =>
src/kernel/liveslots.ts(536,7): error TS2322: Type '(value: any) => void' is not assignable to type '() =>
src/kernel/liveslots.ts(756,17): error TS2554: Expected 7 arguments, but got 1.
src/kernel/state/kernelKeeper.ts(279,18): error TS2554: Expected 1 arguments, but got 0.
src/kernel/state/kernelKeeper.ts(301,15): error TS2345: Argument of type '(text: string, reviver?: (this: 
  Types of parameters 'reviver' and 'index' are incompatible.
    Type 'number' is not assignable to type '(this: any, key: string, value: any) => any'.
src/kernel/vatManager/manager-local.ts(73,48): error TS2554: Expected 1 arguments, but got 2.
src/kernel/vatManager/manager-local.ts(99,48): error TS2554: Expected 1 arguments, but got 2.
src/kernel/vatTranslator.ts(155,51): error TS2554: Expected 2 arguments, but got 1.
src/worker-protocol.ts(31,28): error TS2345: Argument of type 'Buffer' is not assignable to parameter of t
```

This only surfaces the problem; adding type annotations would definitely uncover more issues. The real problem with JavaScript is that it is too liberal, and allows modifications to the code locally, without considering the far-reaching effects of those changes. As a result, the programmer is forced to manually do the job of a type checker, which can be in fact easily automated to prevent serious bugs.

Recommendation

Consider switching the SwingSet kernel to TypeScript or another strictly-typed language, or tracking the type information automatically by other means.

IF-AGORIC1-03: Undocumented function format change

Category	Code Structure
Severity	Low
Issue	agoric-sdk#3174

Involved artifacts

- agoric-sdk/packages/SwingSet/src/parseVatSlots.js

Description

Function parseVatSlot is documented to parse a string and return the object with this signature:

```
/**
 * Parse a vat slot reference string into a vat slot object:
 *   {
 *      type: STRING, // 'object', 'device', 'promise'
 *      allocatedByVat: BOOL, // true=>allocated by vat, false=>by the kernel
 *      id: Nat
 *   }
*/
```

At the same time, the function actually accepts the extended format `{ type, allocatedByVat, virtual, id, subid }` with `id` and `subid` separated by `/`:

```
  const delim = idSuffix.indexOf('/');
  let id;
  let subid;
  let virtual = false;
  if (delim > 0) {
    assert(type === 'object' && allocatedByVat, X`invalid vatSlot ${s}`);
    virtual = true;
    id = Nat(BigInt(idSuffix.substr(0, delim)));
    subid = Nat(BigInt(idSuffix.slice(delim + 1)));
  } else {
    id = Nat(BigInt(idSuffix));
  }

  return { type, allocatedByVat, virtual, id, subid };
```

Moreover, the non-extended format seems to be used everywhere in the documentation as well.

This may pose multiple problems:

- User of that function or of the documentation will be unaware of the extended format. E.g., they may implement their own parsing for the non-extended format, for example, and then the input data in the extended format will crash them.
- Vat slots seem to play one of the central roles in the whole system, as they are part of CapData structure. Allowing two incompatible formats for that may be a source of non-determinism.

It should be noted that this issue would be caught easily by migrating the code base to TypeScript or tracking the type information by other means; see issue IF-AGORIC-06.

Recommendation

- Update the documentation to reflect the extended format.
- Inspect all call sites of `parseVatSlot` function to ensure that the extended format doesn't pose problems.

IF-AGORIC1-04: Problematic pattern in processQueueMessage

Category	Code Structure
Severity	Low

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/kernel.js

Description

The function processQueueMessage() has an implicit assumption that it is called with a message struct that is not `undefined`. This does indeed hold in both places where it is invoked:

- in step()

  ```
  if (!kernelKeeper.isRunQueueEmpty()) {
    await processQueueMessage(kernelKeeper.getNextMsg());
  ```

- in run() there is a similar fragment with the `while` loop.

While this is OK with the current codebase, the implicit assumption may become problematic later, i.e. if in some place `processQueueMessage(kernelKeeper.getNextMsg())` happens without a preceding check for queue emptiness; the `processQueueMessage()` will throw an exception then, and the whole kernel will probably get terminated. Besides, the above pattern seems to be quite inefficient, because the whole run queue is first fetched and parsed from the storage in `kernelKeeper.isRunQueueEmpty()`, only to check its length; and later again fetched and parsed in `kernelKeeper.getNextMsg()`.

Recommendation

- Document the assumption that `processQueueMessage` accepts only a defined message, or check if it is indeed defined.
- Refactor the above fragments along the following lines:

  ```
  let msg = kernelKeeper.getNextMsg();
  if(msg !== undefined) {
      await processQueueMessage(msg);
  ```

IF-AGORIC1-05: Unrestricted number of vat slots per vat

Category	Implementation
Severity	Medium
Issue	agoric-sdk#3243

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/

Description

Notice: As agreed with Agoric, we performed the analysis under the assumption of possible bugs in the liveslots implementation. In that case, userspace vat code together with the liveslots layer operate as a whole against the kernel, which is the necessary condition for triggering this issue. Correctly functioning liveslots implementation should prevent this issue from being exploitable. We will perform assessment of liveslots correctness in the subsequent phases of our analysis.

The kernel allocates a lot of storage in kvStore for each unknown vat slot ID that it encounters in a syscall. E.g. the function translateSend:

```
const target = mapVatSlotToKernelSlot(targetSlot);
const argList = legibilizeMessageArgs(args).join(', ');
// prettier-ignore
kdebug(`syscall[${vatID}].send(${targetSlot}/${target}).${method}(${argList})`);
const kernelSlots = args.slots.map(slot => mapVatSlotToKernelSlot(slot));
```

The code above calls mapVatSlotToKernelSlot, which, in turn, calls addKernelPromise:

```
function addKernelPromise(policy) {
  const kpidNum = Nat(BigInt(getRequired('kp.nextID')));
  kvStore.set('kp.nextID', `${kpidNum + 1n}`);
  const kpid = makeKernelSlot('promise', kpidNum);
  kvStore.set(`${kpid}.state`, 'unresolved');
  kvStore.set(`${kpid}.subscribers`, '');
  kvStore.set(`${kpid}.queue.nextID`, `0`);
  kvStore.set(`${kpid}.refCount`, `0`);
  kvStore.set(`${kpid}.decider`, '');
  if (policy && policy !== 'ignore') {
    kvStore.set(`${kpid}.policy`, policy);
  }
  // queue is empty, so no state[kp$NN.queue.$NN] keys yet
  incStat('kernelPromises');
  incStat('kpUnresolved');
  return kpid;
}
```

This allocates *a lot* of kernel storage as a result of a single mentioning of a vat slot in a call from a vat to the kernel. Moreover, the number of vat slots in the arguments of a syscall is unrestricted. This makes it possible to fill the kernel storage with promises, and their respective data.

Recommendation:

Restrict the number of vat slots per vat.

IF-AGORIC1-06: Unrestricted length of vat slot IDs

Category	Implementation
Severity	Low
Issue	agoric-sdk#3242

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/parseVatSlots.js

Description

Notice: As agreed with Agoric, we performed the analysis under the assumption of possible bugs in the liveslots implementation. In that case, userspace vat code together with the liveslots layer operate as a whole against the kernel, which is the necessary condition for triggering this issue. Correctly functioning liveslots implementation should prevent this issue from being exploitable. We will perform assessment of liveslots correctness in the subsequent phases of our analysis.

Currently the kernel accepts vat llot IDs as BigInts. E.g., in the function parseVatSlot:

```
if (delim > 0) {
  assert(type === 'object' && allocatedByVat, X`invalid vatSlot ${s}`);
  virtual = true;
  id = Nat(BigInt(idSuffix.substr(0, delim)));
  subid = Nat(BigInt(idSuffix.slice(delim + 1)));
} else {
  id = Nat(BigInt(idSuffix));
}
```

As BigInts don't have any limit, it should be possible to kill the kernel with a single syscall, by passing a very-very large number, that would eat all the memory when trying to convert from a string representation into BigInt representation.

Recommendation

Restrict the size of vat slot IDs to some sufficiently large number, e.g. to 32 bits. As an easier to enforce measure, the string representation of the number could be restricted to e.g. 10 characters.

IF-AGORIC1-07: Scattered updates to kernel tables

Category	Code Structure
Severity	Informational
Issue	agoric-sdk#3312

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/

Description

The changes to the kernel storage are scattered in a multitude of unexpected places throughout the codebase.

For example, the following chain of calls exist: vatSyscallToKernelSyscall -> translateResolve -> deleteCListEntriesForKernelSlots -> mapKernelSlotToVatSlot.

In the above chain, a translation phase from vat to kernel, involves both *deletion* (in deleteCListEntriesForKernelSlots) and *creation* (in mapKernelSlotToVatSlot) of entries in the kernel tables.

Recommendation

Clearly separate state updates from other kinds of logic; also document the functions that (by transitivity) do state modifications.

IF-AGORIC1-08: Possible cycles in promise resolutions

Category	Implementation
Severity	Medium
Issue	agoric-sdk#3313

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/kernel.js

Description

In the code of deliverToTarget in `kernel.js`, the function contains an await of an asynchronous call to itself:

```
async function deliverToTarget(target, msg) {
  insistMessage(msg);
  const { type } = parseKernelSlot(target);
  if (type === 'object') {
    const vatID = kernelKeeper.ownerOfKernelObject(target);
    if (vatID) {
      await deliverToVat(vatID, target, msg);
    } else {
      resolveToError(msg.result, VAT_TERMINATION_ERROR);
    }
  } else if (type === 'promise') {
    const kp = kernelKeeper.getKernelPromise(target);
    if (kp.state === 'redirected') {
      // await deliverToTarget(kp.redirectTarget, msg); // probably correct
      // TODO unimplemented
      throw new Error('not implemented yet');
    } else if (kp.state === 'fulfilled') {
      const presence = extractPresenceIfPresent(kp.data);
      if (presence) {
        await deliverToTarget(presence, msg);
      }
```

If a promise can be resolved to be self-fulfilled (i.e. resolved to a data that contains a slot pointing to itself), then this will cause infinite recursion in the kernel in deliverToTarget. In a less critical case the promise may be not self-fulfilled, but a cycle may exist in a promise resolution graph.

Recommendation

Carefully inspect the code to catch all cases of intra-vat and inter-vat cycles in promises; also take an effort to prove that the protocol excludes the possibility of cycles. As an intermediate protective measure, maybe break the above recursive code by introducing a real asynchrony (e.g. putting into a queue), such that cycles have a chance to be detected without immediate consequences.

IF-AGORIC1-09: Possible loss of vat termination events

Category	Implementation
Severity	Medium
Issue	agoric-sdk#3315

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/kernel.js

Description

The function setTerminationTrigger is as follows:

```
function setTerminationTrigger(vatID, shouldAbortCrank, shouldReject, info) {
  if (shouldAbortCrank) {
    assert(shouldReject);
  }
  if (!terminationTrigger || shouldAbortCrank) {
    terminationTrigger = { vatID, shouldAbortCrank, shouldReject, info };
  }
}
```

It can be seen that if `terminationTrigger` is set already, then the second call to that function will not have any effect, i.e. the second termination event will get lost; this seems like a dangerous scenario.

We have not checked all possible sequences of events that may lead to the problem, in particular because this involves the liveslots code, which are outside of the scope of the assessment. One potential problematic sequence is when a vat requests to exit, which calls `setTerminationTrigger(vatID, false, !!isFailure, info)`, in particular setting the `shouldAbortCrank` to `false`; then a letter call to `setTerminationTrigger` with `shouldAbortCrank` set to `true` will get ignored.

While this probably doesn't constitute an error with the current code base, it may become a source of a bug at later stages, when more calls to `setTerminationTrigger` are added.

Recommendation

Modify the code of setTerminationTrigger to check whether the termination trigger has been set already, to compare vat ids and other parameters, and to take the appropriate action upon second function invocation.

IF-AGORIC1-10: Inconsistencies in vat bookkeeping

Category	Code Structure
Severity	Medium
Issue	agoric-sdk#3316

Involved artifacts

- agoric-sdk/packages/SwingSet/src/kernel/

Description

The conditions under which a vat is considered alive/present are numerous and spread across multiple files/datastructures/functions. Here are a few examples:

- in function terminateVat, the vat being alive is checked via a call `kernelKeeper.vatIsAlive(vatID)`:

```
function vatIsAlive(vatID) {
    insistVatID(vatID);
    return kvStore.has(`${vatID}.o.nextID`);
}
```

- In `kernelKeeper.js` the VATs state is kept in `ephemeral.vatKeepers`; consider e.g. the function evictVatKeeper does this:

```
function evictVatKeeper(vatID) {
    insistVatID(vatID);
    ephemeral.vatKeepers.delete(vatID);
}
```

- in `vat-warehouse.js` the state is kept *both* via `kernelKeeper` and in `ephemeral.vats`; consider e.g. the function evict

```
async function evict(vatID, makeSnapshot = false) {
    assert(!makeSnapshot, 'not implemented');
    assert(lookup(vatID));
    const info = ephemeral.vats.get(vatID);
    if (!info) return undefined;
    ephemeral.vats.delete(vatID);
    xlate.delete(vatID);
    kernelKeeper.closeVatTranscript(vatID);
    kernelKeeper.evictVatKeeper(vatID);

    // console.log('evict: shutting down', vatID);
    return info.manager.shutdown();
}
```

Thus, there are at least three storage places for VATs: `kvStore`, `ephemeral.vatKeepers` from `kernelKeeper.js`, and `ephemeral.vats` from `vat-warehouse.js`. This creates a danger of having inconsistent state views or updates. E.g. the function deliverAndLogToVat contains the fragment below, where the vat existence is checked via one API, and the `vatKeeper` is then extracted via another API:

```
assert(vatWarehouse.lookup(vatID));
const vatKeeper = kernelKeeper.provideVatKeeper(vatID);
```

Recommendation

Hide vat bookkeeping behind a single API that is used consistently at all places throughout the codebase.