D. Energetic Secrets

The technique of Energetic Secrets replaces Tuples with SealedEnvelopes as messages in the Joule communication model, incorporating public-key semantics into the communication foundations. This change simplifies the Joule semantics (answering questions like, “What is a message selector?”), incorporates the authentication and other security properties of Verifiers into the foundation, and improves the potential efficiency of an untyped Joule implementation by enabling C++-style dispatch. This Appendix introduces the Energetic Secrets concepts and uses. In later versions of this document, the concepts in this Appendix will be integrated into the main body of the text and specified in more detail.

In using Energetic Secrets, each potential operation (message selector) is represented by a pair of a Sealer and an Unsealer (which we will call an Un/Sealer pair), roughly corresponding to send and receive rights for messages of that operation. When a Sealer is applied to arguments, it seals them in a new SealedEnvelope that can only be opened by the corresponding Unsealer. The Unsealer is used by receivers of the SealedEnvelope to recognize the message and extract the arguments.

D.1. Sending Messages

Sealers and Unsealers are typically used implicitly: what had formerly been a Tuple expression (`foo: arg1 arg2`) implicitly applies a Sealer to arguments to produce an `Envelope`; **Switch** and **Server** constructs implicitly extract using Unsealers; and the **Type** form creates Sealer/Unsealer pairs. The statement

- `receiver anOperation: arg1 arg2`

which sends an envelope sealed with the Sealer for `anOperation:` containing `arg1` and `arg2`, is equivalent to

- `receiver (anOperation:sealer seal: arg1 arg2)`

in which the `seal:` operation, sent to `anOperation:sealer`, produces an Envelope which is then sent to `receiver`.

Energetic Secrets introduces a new syntactic convention, shown in the example above: labels implicitly refer to sealers and unsealers with a naming convention of appending `sealer` or `unsealer` to the end. (Operators append `:sealer` or `:unsealer`.) The identifier, `anOpera-`

tion:sealer, is just a normal identifier, which is bound to the Sealer for anOperation:. Note that the sending of the seal: message to the Sealer similarly invokes a Sealer and produces an envelope; seal:sealer is a primitively supplied Sealer (along with a few others like ::sealer) which serves to bottom out the mechanism for envelope creation.

D.2. Receiving Messages

Messages are received using **ForAll** and choose: as before. Recognizing and parsing are different for envelope messages, however. The code below shows the expansion of a **Switch** form, which is part of the expansion for a **Server** form.

```
Switch envelope
    case foo: a b
        scope in which a and b are visible
    case bar: c
        scope in which c is visible
endSwitch
```

The above **Switch** construct semantically expands into code involving the unseal: method, as shown in the following fragment for the foo:unsealer branch of the **Switch**:

```
Define num, fn
    • foo:unsealer unseal*: envelope num> fn>
endDefine
If num = 2
    Define a = fn :: 0, b = fn :: 1 endDefine
    scope in which a and b are visible
endIf
```

The num and fn revealed by the unseal*: operation are just like the num and fn used for “rest” arguments in Appendix C, *Optional Arguments* (and are in fact used to implement “rest” arguments).

The invocation of the foo:unsealer takes an envelope (received as a message to a **ForAll**, for instance) and, if the envelope really is an envelope sealed by the corresponding foo:sealer, reveals num which is the number of arguments in the envelope (presumably 2 in this case), and a server that will reveal each argument when called with an integer index and a result port. The remainder of the code invokes the supplied argument function to bind the arguments and executes the nested body. If the unseal failed, the revealed num would be –1.

D.3. Sealer and Unsealer Types

Sealers and Unsealers are methodical servers that respond to the protocol below.

```
There are no operations on Envelopes beyond the basic ones.
Type SealedEnvelope
    super Basic
endType
Type Unsealer
    super Basic
    Given a SealedEnvelope sealed by the corresponding Sealer, reveal the number
      of arguments in the envelope and a server that will reveal the arguments. The
      ‘fn’ may only be invoked once per argument.
    op unseal*: envelope num> fn>
endType
```

```
Type Sealer
    super Basic
    Given any number of args, create a SealedEnvelope that encapsulates them,
      and which can only be opened by the corresponding Unsealer.
    op seal: arg... envelope>
    Like seal: except the num and fn arguments are required and are a number and
      function so that users can supply a dynamic set of arguments computed at run
      time.
    op seal*: arg... num fn envelope>
endType

Type make-un/sealer
    op :: sealer> unsealer>
endType
```

D.4. Types and Virtual Un/Sealers

The Un/Sealer pairs are typically generated by the **Type** form. The straightforward expansion is to generate a different Un/Sealer pair for each selector. Instead, the **Type** form can expand to virtual Un/Sealers (unsealers implemented in Joule) to enable the use of a C++-style vtable implementation of message dispatch. Where the code:

```
Type T
    op foo: a b
    op bar: c
endType
```

would normally expand to include un/sealer creation as in:

```
Define   foo:sealer, foo:unsealer,
         bar:sealer, bar:unsealer
    • make-un/sealer :: foo:sealer> foo:unsealer>
    • make-un/sealer :: bar:sealer> bar:unsealer>
endDefine
```

it would expand instead to create a single Un/Sealer used for the whole type and would create virtual Un/Sealers that encode a vtable index (for example the operation's number in the Type) for each operation.

```
Define   foo:sealer, foo:unsealer,
         bar: sealer, bar:unsealer
    Define T:sealer, T:unsealer
        • make-un/sealer :: T:sealer> T:unsealer>
     endDefine
    • virtual-sealer :: T:sealer 0 foo:sealer>
    • virtual-unsealer :: T:unsealer 0 foo:unsealer>
    • virtual-sealer :: T:sealer 1 bar:sealer>
    • virtual-unsealer :: T:unsealer 1 bar:unsealer>
endDefine
```

When sealing, a virtual sealer would take all the supplied arguments, prefix the vtable index (0 for `foo` in the above case) and then seal with `T:sealer`. Servers without `T:unsealer` would be unable to open the envelope, so they couldn't discover the virtualization. The virtualized `foo:unsealer` would attempt to unwrap with `T:unsealer`, then check to see whether the first argument is 0 (the vtable index for `foo`), and only if both tests pass would it reveal the arguments of the envelope. The advantage of this scheme is that a compiler (or even a smart server) could expand the **Switch** statement shown in Section D.2 into a single unseal operation using `T:unsealer` (instead of one per case alternative) followed by an indirect jump through a vtable using the index. This will be described in detail (along with Joule implementations of virtual sealers and the fast-dispatch Type expansion) in future versions of this manual.

D.5. Certifying Requests

The last example application of Energetic Secrets presented in this Appendix is certifying properties of requests. A virtualized sealer can dynamically type-check arguments and refuse to produce a sealed Envelope unless the arguments type-check correctly. It can check other properties as well, including relations between the arguments, and extending to full pre-condition checking of the arguments. It can ensure durability by wrapping up, not the arguments, but rather the results of verifying the arguments, potentially reproducing them so that the receiver will be assured that the arguments will remain available.

Finally, modules can export different sealers for the same operation that implement different checks. The exporting module could give the sealers with fewer checks to clients who could prove they passed some trusted analysis that statically checks preconditions (such as argument types).