TinyOS Call Graph Verification

Philip Levis

January 28, 2002

Overview

The TinyOS programming model constrains the call graph of a TinyOS component: TinyOS commands cannot signal events. Originally, the TinyOS build process assumed this rule was followed. We have implemented a system that checks TinyOS source for this rule. The system is very flexible and can be easily extended to include more constraints on TinyOS source. We explain how this system works, how it is used to check call graphs, and it could check an additional TinyOS call graph constraint that we are considering incorporating.

`cqual`

Call graph checking is performed by a customized version of `cqual`, an application developed by Jeff Foster at UC Berkeley. `cqual` allows a user to specify new type qualifiers for C programs, establish rules for their conversion, then check that these rules are not violated.

For example, one can add a pair of qualifiers, `$tainted` and `$untainted`. Some function parameters can only take `$untainted`, or trusted data; the format string of `printf()` is an example. Arbitrary `printf()` format strings (e.g. user-provided) are a known security problem. Some functions produced `$tainted`, or untrusted data; an example is `getenv()`, as it's a user specified string. One can define a rule that `$untainted` data can be converted to `$tainted`, but not vice-versa. This rule is written by placing both qualifiers in a type hierarchy: `$untainted` is a subtype of `$tainted`. If a variable is untainted, one can also considered it tainted, just as Volvo can be considered a car; however, a tainted variable is not necessarily untainted, just as a car is not necessarily a Volvo.

If one annotates functions properly, `cqual` can detect when the result of a `getenv()` is passed as the format string to `printf()`, and throw an error accordingly.

For a more in-depth explanation of description of the `cqual` system, please refer to its web site: `http://www.cs.berkeley.edu/Research/Aiken/cqual/`. Please note that we have modified `cqual` for TinyOS, and so bug reports or issues should be sent to the TinyOS team, not Jeff Foster, the `cqual` maintainer.

Modifications

We have modified `cqual` to also apply qualifier rules on procedure invocation. A procedure can only invoke procedures of the same type or supertype. By specifying an event as a subtype of command, the TinyOS call graph constraints are enforced; an event can invoke commands (as command is its supertype), but commands cannot signal events (as they are a subtype).

We have also removed type checking from `cqual`; if there are type errors, they will be discovered and dealt with during compilation.

Qualifier Lattice

The TinyOS qualifier lattice is this:

```
partial order {
$aevent [level=value, color = "pam-color-tainted", sign = neg]
$sevent [level=value, color = "pam-color-tainted", sign = neg]
$command [level=value, color = "pam-color-untainted", sign = pos]

$aevent < $command
$sevent < $command
}
```

Refer to the cqual documentation for an in-depth description of the file format. The important part is that three qualifiers are defined: aevent, sevent, and command. Currently, only aevent and command are used, to represent events and commands. The difference between aevents and sevents is described in the final section of this document, in regards to more fine-grained type qualification system we are considering that will hopefully reduce race conditions in TinyOS code.

Both event qualifiers are subtypes of the command qualifier. This allows both events to call commands (their supertype), while commands cannot signal events and the two types of events cannot signal one another.

`cqual` propagates qualifiers. For example, if one defines a simple C helper function that signals an event, then it will be given the event qualifier can cannot be called by commands.

Checking Process

Makefile.qual checks a TinyOS application call graph through several steps. Executing Makefile.qual builds the file lists needed for the application, storing them in a separate file. It then executes Makefile.check, which imports this file and uses the file lists. Because a traditional C compiler will not understand `cqual` qualifiers, copies are made of all the necessary files, which are then annotated with the necessary qualifiers. The steps taken are as follows:

- Make a local `cqual` directory to store files in.
- Walk the component description graph and compute which preprocessed source files will be used by `cqual` in the local `cqual/` directory.

- Walk the component description graph and compute which source files are to be copied into the local `cqual/` directory.
- Walk the component description graph and compute which header files are to be copied into the local `cqual/` directory.
- Store these lists in `cqual/cqual.objs`.
- Execute `Makefile.check`
- `Makefile.check` imports `cqual/cqual.objs`.
- Remove all relevant component header files from the TinyOS source directories.
- Build new annotated copies of relevant component header files.
- Move relevant header files into the local `cqual/` directory.
- Copy relevant source files into the local `cqual/` directory.
- Add qualifiers to function definitions in the source files in the local `cqual/` directory.
- Run the qualified source files through the C preprocessor.
- Run `cqual` on the preprocessed source files.

`aevents` and `sevents`

Two kinds of events are defined in the TinyOS qualifier lattice: `aevent`, or asynchronous events, and `sevents` or synchronous events. Currently, the TinyOS C annotations do not distinguish between the two, and the call graph checker only uses `aevents`.

The lattice distinction exists because of the observation that a TinyOS component currently has no way of knowing whether events is handles execute asynchronously (in an interrupt context) or synchronously (from a task). Distinguishing the two is important for determining possible race conditions and knowing which state variables must be protected from interrupts.

By having two separate sub-types, the task boundary between the two can be made explicit. Because tasks are executed through function pointers (and this modified version of cqual does not propagate qualifiers through function pointers), a task that fires synchronous events can be enqueued by an asynchronous event. An asynchronous event cannot signal a synchronous event, or vice versa.