SAFETY IN OUR EMBEDDED SYSTEMS

WHITE PAPER

Enforced in Hardware by CoreGuard® Technology

WRITTEN BY:

Jothy Rosenberg

Founder & CEO

SEPTEMBER 2018

DOVER MICROSYSTEMS

RADIO CITY
New York-Presbyterian
ARGENTINA
WINDSOR
ONE WAY

safety | ʹsāftē |

noun (plural **safeties**)

1 the condition of being protected from or unlikely to cause danger, risk, or injury

Introduction

Safety protocols for embedded systems mandate that developers follow standards, adhere to design and coding guidelines[1], enforce the checking and re-checking of procedures, and conduct testing extensively. Yet despite these best efforts, even the most carefully-engineered systems are vulnerable to the unanticipated events of Mother Nature, mechanical or electrical failures, human errors, and malicious cyberattacks. Life happens, and in our cyber-physical IoT world, these inevitable system failures or malfunctions aren't just frustrating—they can be catastrophic, as well.

Consider just a few examples:

Oct 2012	Security researcher Barnaby Jack creates a well-publicized video demonstrating his ability to gain access to pacemakers from a number of manufacturers. In the video, Jack uses just a laptop and is no more than 50 feet away from the devices when he causes them to deliver fatal shocks of 830 volts. Luckily, he uses mannequins for his demonstration and no individuals are hurt.
Dec 2014	A cybercriminal group causes extraordinary collateral damage to a German steel mill by disrupting control systems to gain access to the plant. Upon doing so, the group accidentally sends a blast furnace into overdrive, causing it to self-destruct—putting workers lives at risk.
Jul 2015	Fiat Chrysler Automobiles issues a recall of 1.4 million vehicles following a successful hack by security researchers Charlie Miller and Chris Valasek. Miller and Valasek gained access to a 2014 Jeep Cherokee over the internet and were able to control the car's radio, A/C, windshield wipers, steering, brakes, and transmission.

1 *The Power of Ten - Rules for Developing Safety Critical Code*, Gerard J. Holzmann, NASA/JPL Laboratory for Reliable Software.

The increasing complexity of our embedded systems make safety breaches even more likely because of one simple fact: There are an average of 15 bugs per 1,000 lines of source code. More complicated systems mean more code, and more code equals more bugs. Additionally, many of these systems are connected to the internet (and to each other) often making them more accessible and therefore more vulnerable to attack.

Until recently, the only way to secure these embedded devices was through the aforementioned best practices: adhere to standards and guidelines, abide by the latest policies, and test, test, test.

But now there's a new way to look at and approach cybersecurity for embedded devices with CoreGuard™ .

Dover Microsystems' CoreGuard solution is a unique, patent-pending technology that protects a device by checking each and every instruction handled by the host processor, and blocking any instruction that violates a safety (or security or privacy) policy.

A Quick Review of CoreGuard

CoreGuard technology takes a hybrid approach to safety, with silicon IP that enforces software-defined policies. It has two main components:

- **Micropolicies**: CoreGuard micropolicies define the instruction-by-instruction guarantees for a particular system. Micropolicies can enforce a wide range of customer-defined security requirements, such as protecting memory against buffer overflows, preventing control flow hijacking (like Return-Oriented Programming attacks), and managing information integrity and information confidentiality. Micropolicies can also ensure safety rules unique to a device and the applications it runs.

  Every micropolicy consists of two parts: **metadata** maintained about each word in the host processor's accessible memory, and **rules** that determine whether or not each instruction is allowed by the set of installed micropolicies. Micropolicy rules are defined in a proprietary domain-specific language (the Dover Policy Language) that is designed to support analysis and verification. Because the Dover Policy Language is optimized for writing micropolicies, it allows for the most efficient code possible; most micropolicies are just a dozen or fewer lines of code.

- **Policy Enforcer**: The policy enforcer is CoreGuard's hardware mechanism for controlling the host processor. It allows CoreGuard to check every instruction for compliance with micropolicies. When an instruction violates any micropolicy, the policy enforcer blocks it from executing. The policy enforcer also maintains a strict separation between the code and data used by the micropolicies and the application software and operating system being protected; only CoreGuard hardware can run CoreGuard software. CoreGuard hardware logic is added to an existing SoC (System on a Chip) design by integrating it with the processor or microcontroller that is running all applications for that SoC.

Legal Transitions for Finite State Machines

CoreGuard's policy language can express the allowed (safe) states of a protected system by defining a **finite state machine** (FSM). An FSM describes the legal (safe) configurations of a system and the allowed transitions from one safe state to the next.

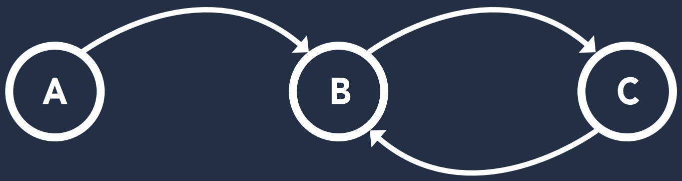
CoreGuard can apply this State Machine approach to a majority of safety policy enforcement cases.

Some cases are best handled with another powerful CoreGuard policy approach called Information Flow Integrity—or with a combination of both approaches.

Let's consider a simple safety assertion:

Traffic Light Controller: A traffic light must never have adjacent directions showing the green signal simultaneously (fast-moving cars might crash).

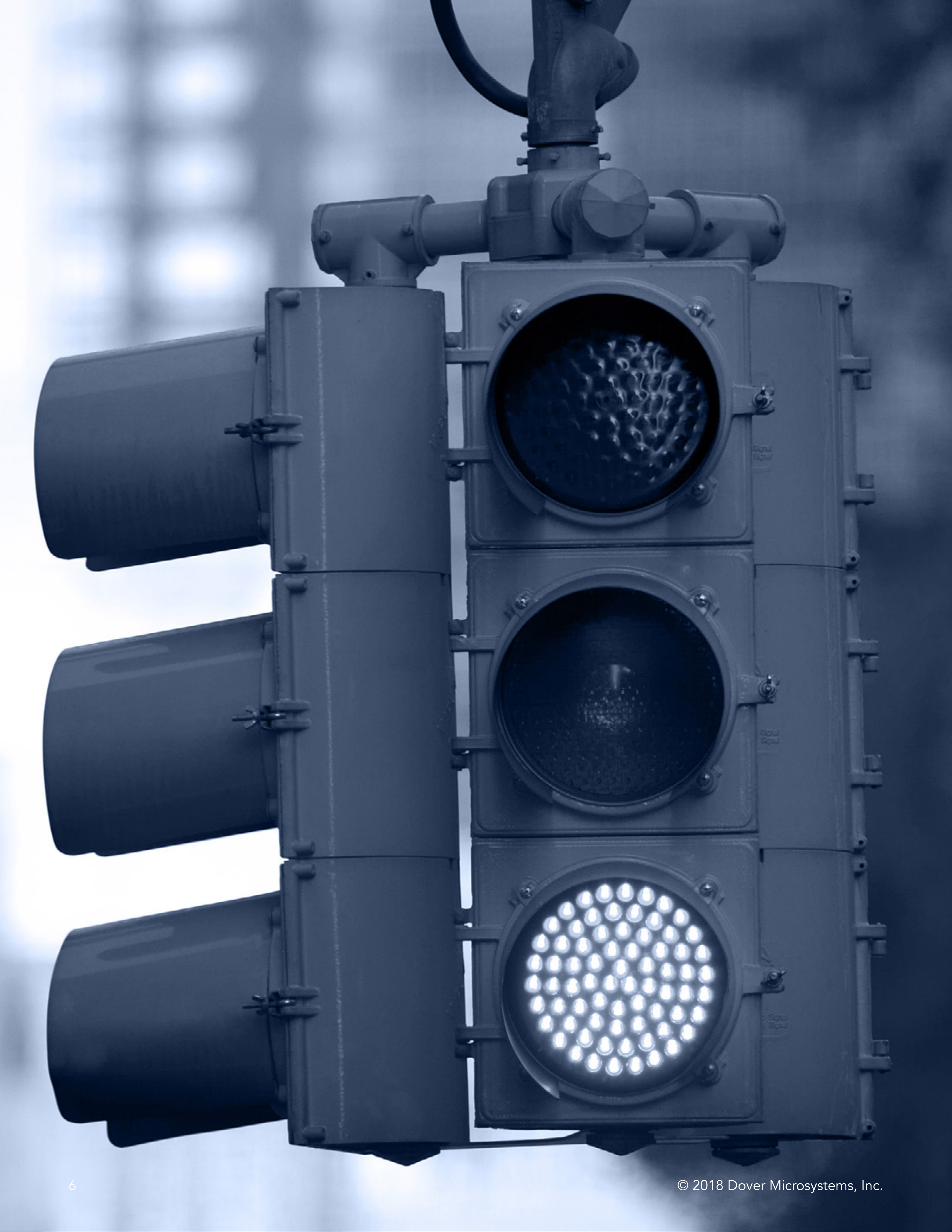
Next, we'll take a closer look at this case to show you how CoreGuard implements and enforces a Traffic Light Safety micropolicy.



$$\begin{tikzcd} A \arrow[r, bend left] & B \arrow[r, bend left] & C \arrow[l, bend left] \end{tikzcd}$$

What is a finite state machine?

A finite state machine, or more simply just state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The state machine can change from one state to another in response to some external inputs; this change from one state to another is called a transition. A state machine is defined by a list of its states, its initial state, and the conditions for each transition. Real-world examples of state machines are abundant: vending machines, elevators, combination locks, and traffic lights to name just a few. In this paper we will use a traffic light controller as the example state machine whose safety properties we want to enforce in CoreGuard's hardware.



Traffic Light Safety Policy

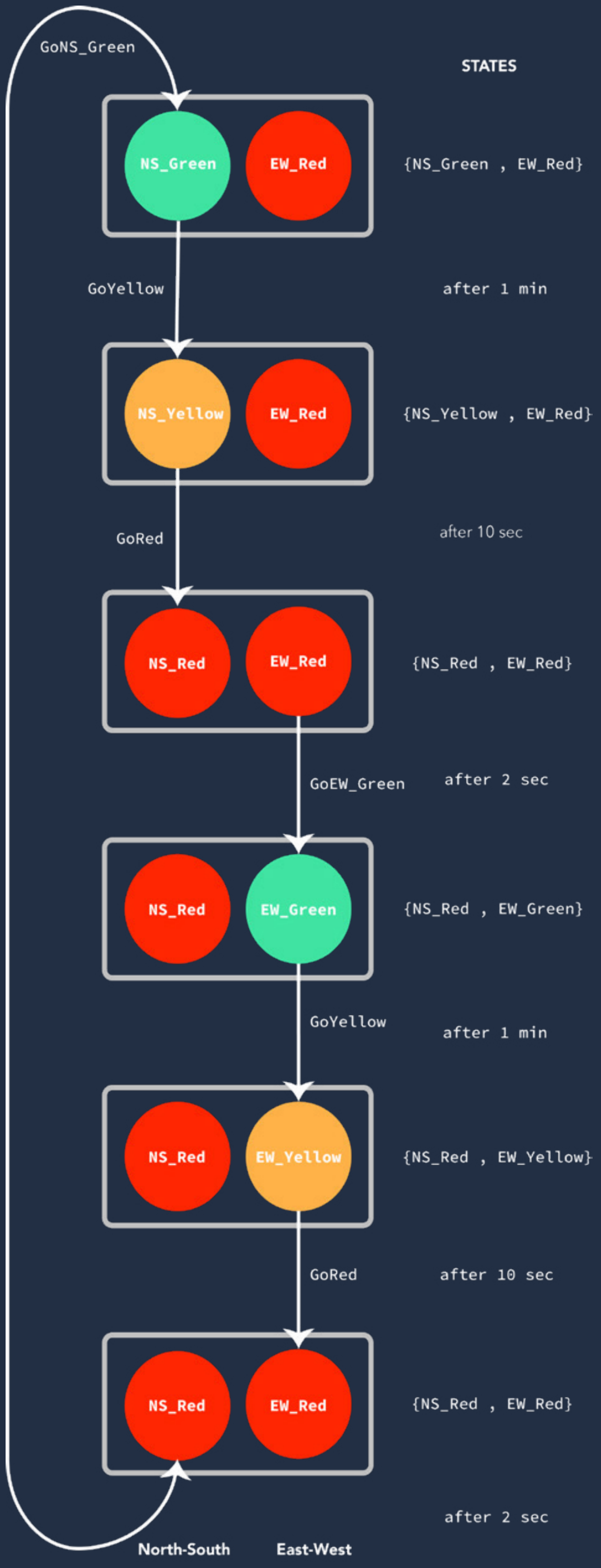
Our simple traffic light controller controls 12 lights: three each facing east and west, and three each facing north and south. Each of the four directions has a red, yellow, and green light. The lights facing east and west are wired to show identical colored lights, as are the pairs of lights facing north and south. Figure 1 shows a finite-state machine diagram for the traffic light controller. A state is a pair of north-south and east-west colors. Each state is highlighted in a gray box, and the arrows show the allowed transitions from one state to another. There are five unique states with our traffic light controller (the all-red color state is shown twice in the diagram).

The metadata for our Traffic Light Safety Policy

As described earlier, CoreGuard uses metadata to determine whether an instruction complies with the set of micropolicies installed on the SoC. The Traffic Light Safety Micropolicy metadata labels two things:

1. **Current state of the system.** How metadata should label current state will depend on the application. With our traffic light application, a metadata value represents each combination of direction and color—for example, "NS_Red," "EW_Green," and so forth. There are five unique states:
   - NS_Green, EW_Red
   - NS_Yellow, EW_Red
   - NS_Red, EW_Red
   - NS_Red, EW_Green
   - NS_Red, EW_Yellow

FIGURE 1: TRAFFIC LIGHT STATE DIAGRAM



2. **The first instruction of each transition routine.** Within the traffic light application, transition routines are critical because they include the instructions that change the machine from one state to another. CoreGuard uses this type of metadata to label the first instruction in each of the four transition routines:

   - GoGreenEW
   - GoGreenNS
   - GoRed
   - GoYellow

   When the traffic light controller application calls a transition routine, and the host processor attempts to process the first instruction in that routine, the metadata on that instruction activates the CoreGuard policy mechanism and evaluates the rules in the Traffic Light Safety Micropolicy (explained later).

FIGURE 2: TRAFFIC LIGHT SAFETY MICROPOLICY METADATA

```
metadata:
   // Metadata to represent the light
colors
      NS_Red
      NS_Yellow
      NS_Green
      EW_Red
      EW_Yellow
      EW_Green
```

These metadata values represent each direction-color combination.

A state is a pair of these direction-color combinations. For example: NS_Green & EW_Red

```
   // Metadata to label the application
transitions
        GoGreenEW
        GoGreenNS
        GoRed
        GoYellow
```

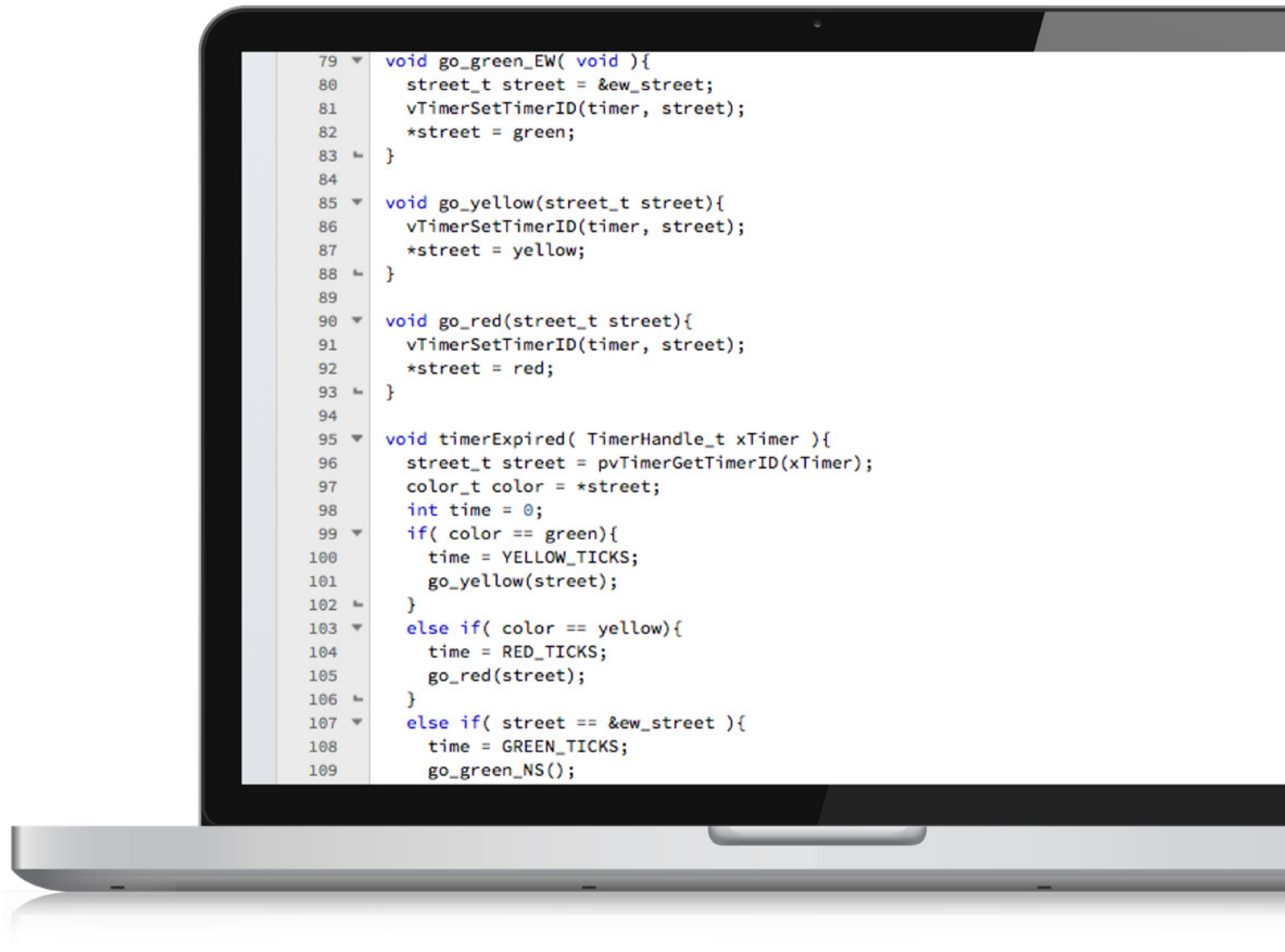
These metadata values are used to label the first instruction of each transaction routine in the application.

The traffic light application code

The CoreGuard SDK (Software Development Kit) includes a Policy Linker tool that associates metadata tags with the transition routines in the application code. The values for these metadata tags can then be used to construct a traffic light safety micropolicy that is triggered when the application goes to execute those transition routines.

First, let's look at the application code.

Figure 3 shows a snippet of the C-code for the traffic light application. Note how the routine names in the traffic light code correspond to the metadata labels for transitions.



```
void go_green_EW( void ){
  street_t street = &ew_street;
  vTimerSetTimerID(timer, street);
  *street = green;
}

void go_yellow(street_t street){
  vTimerSetTimerID(timer, street);
  *street = yellow;
}

void go_red(street_t street){
  vTimerSetTimerID(timer, street);
  *street = red;
}

void timerExpired( TimerHandle_t xTimer ){
  street_t street = pvTimerGetTimerID(xTimer);
  color_t color = *street;
  int time = 0;
  if( color == green){
    time = YELLOW_TICKS;
    go_yellow(street);
  }
  else if( color == yellow){
    time = RED_TICKS;
    go_red(street);
  }
  else if( street == &ew_street ){
    time = GREEN_TICKS;
    go_green_NS();
```

FIGURE 3: C-CODE FOR TRAFFIC LIGHT APPLICATION

FIGURE 4: TRAFFIC LIGHT SAFETY MICROPOLICY RULES

```
lightPol =        Transition              Current State                          New State
Rule 1  (code == [+GoGreenNS]  env == [NS_Red, EW_Red]     -> env = {NS_Green, EW_Red})
Rule 2  (code == [+GoGreenEW], env == [NS_Red, EW_Red]   -> env = {NS_Red, EW_Green})
Rule 3  (code == [+GoYellow] , env == [NS_Green, EW_Red] -> env = {NS_Yellow, EW_Red})
Rule 4  (code == [+GoYellow] , env == [NS_Red, EW_Green] -> env = {NS_Red, EW_Yellow})
Rule 5  (code == [+GoRed]    , env == _                  -> env = {NS_Red, EW_Red})
Rule 6  (code == [-GoGreenNS, -GoGreenEW, -GoYellow, -GoRed], env == _ -> env = env)
```

The micropolicy that enforces safety for the traffic light

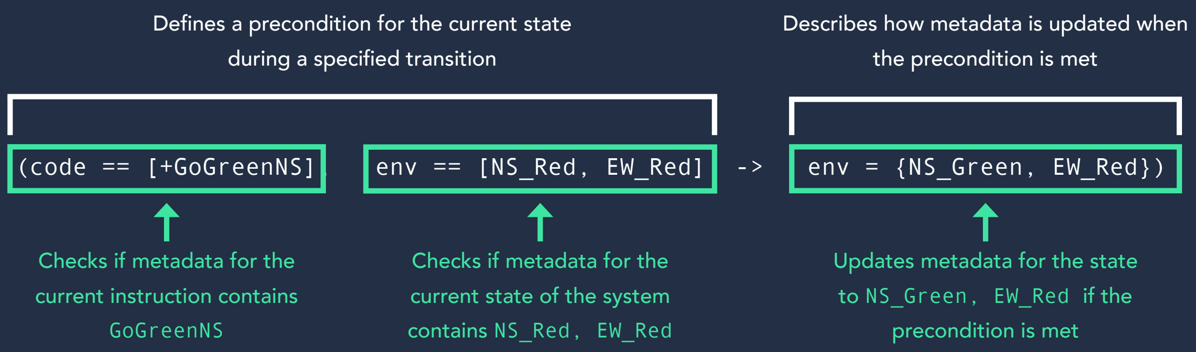
Now let's look at the micropolicy code to see how it relates to the application's C-code.

Figure 4 shows the rules for the traffic light safety micropolicy, written in the Dover Policy Language. **Rules 1 - 5** describe the legal transitions for each state by first evaluating a precondition and then defining how metadata should be updated when the precondition is `true` (that is, when the rule has a "match"). These preconditions are `true` only for the first instruction in a transition routine (for example, the instruction with a metadata value of `GoGreenNS`). **Rule 6** allows all subsequent instructions in the transition routine to execute because its precondition is met with instructions that are NOT tagged with `GoGreenNS`, `GoGreenEW`, `GoYellow`, or `GoRed` metadata. CoreGuard evaluates the policy rules sequentially until it finds a match.

If none of the rules match, then it is a **policy violation**; CoreGuard blocks the instruction from executing and issues an exception. The application will be programmed to handle this exception safely. For example, it may activate a safety mode that puts all lights in a blinking red state.

Figure 5 breaks down Rule 1 for a closer look at how CoreGuard evaluates a rule.

FIGURE 5: BREAKDOWN OF A MICROPOLICY RULE



```
(code == [+GoGreenNS]   env == [NS_Red, EW_Red]  ->  env = {NS_Green, EW_Red})
```

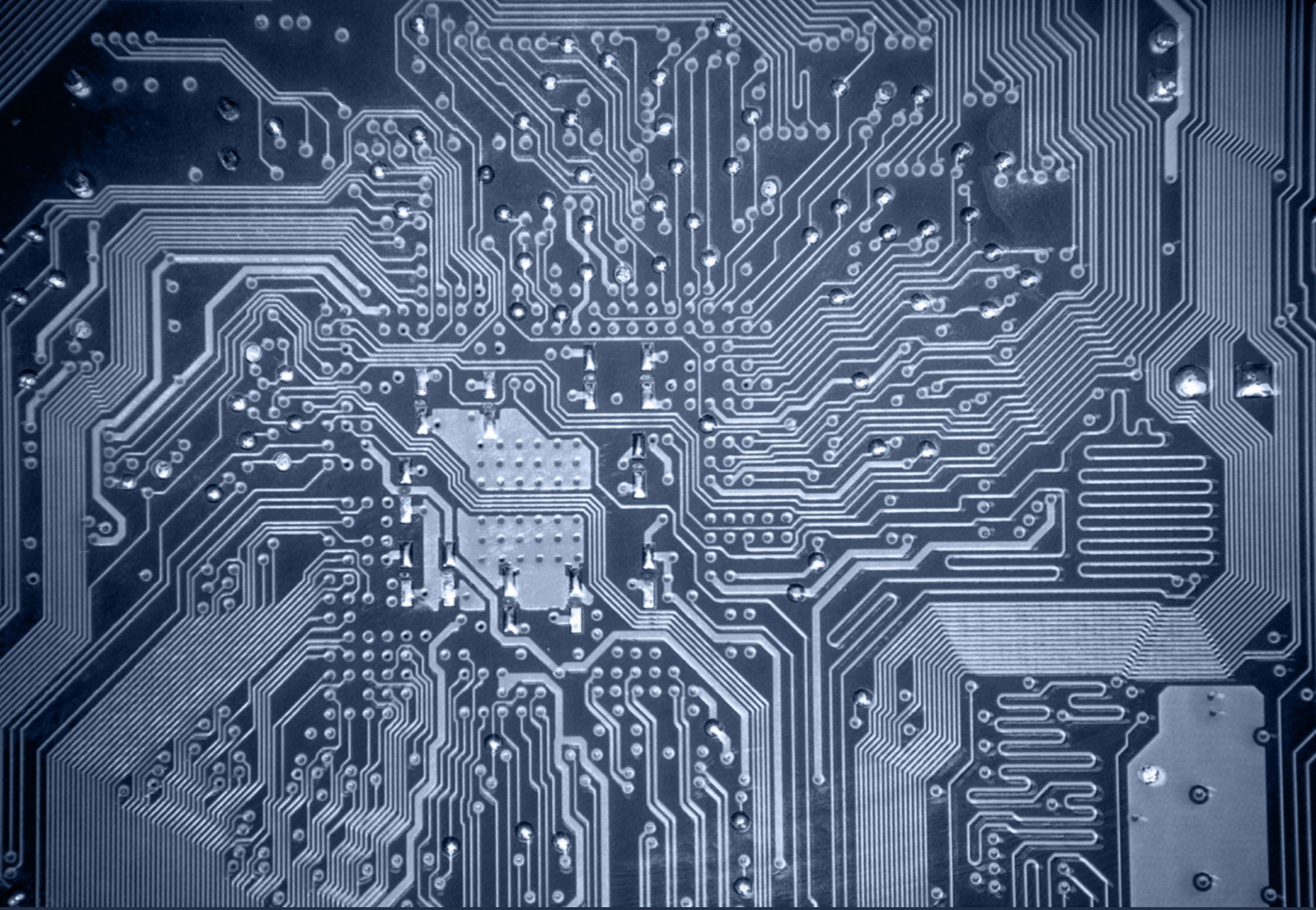
Defines a precondition for the current state during a specified transition
Describes how metadata is updated when the precondition is met
Checks if metadata for the current instruction contains GoGreenNS
Checks if metadata for the current state of the system contains NS_Red, EW_Red
Updates metadata for the state to NS_Green, EW_Red if the precondition is met

The First and Only Flexible Hardware-Based Safety Enforcement

Safety policy enforcement in hardware is part of CoreGuard's IoT Trust Triad: Security, Safety, and Privacy. To enforce safety, CoreGuard uses the same hardware, policy language, and policy tools that it uses to enforce security and privacy. This consistency of approach enables embedded systems developers to implement true defense in depth with a set of micropolicies that meets the unique safety, security, and privacy requirements of their SoC.

Using standards and adhering to policies, following coding guidelines and implementing extensive testing—these are all important practices when it comes to the safety of our cyber-physical embedded systems. These measures alone, however, are not enough to guarantee that a system malfunction or failure (whether intentional or unintentional) does not result in serious, or even deadly, consequences.

Whether it's a pacemaker, a piece of industrial equipment, or a major city's traffic infrastructure, CoreGuard's powerful, flexible, and unassailable hardware safety policy enforcement can stop it.



Learn More: info@dovermicrosystems.com

About Dover Microsystems

Dover's lineage began in 2010 as the largest performer on the DARPA CRASH program. In 2015, Dover began incubation inside Draper before spinning out in 2017.

Based in Boston, Dover is the first company to bring real security, safety, and privacy enforcement to silicon. Dover's patented CoreGuard solution integrates with RISC processors to protect against cyberattacks, flawed software, and safety violations.

www.dovermicrosystems.com