DSN'06, IEEE, pp 23-44. 2006.

Static Analysis to Enforce Safe Value Flow in Embedded Control Systems

Sumant Kowshik Grigore Rosu Lui Sha
University of Illinois at Urbana-Champaign
{kowshik, grosu, lrs}@cs.uiuc.edu

Abstract

Embedded control systems consist of multiple components with different criticality levels interacting with each other. For example, in a passenger jet, the navigation system interacts with the passenger entertainment system in providing passengers the distance-to-destination information. It is imperative that failures in the non-critical subsystem should not compromise critical functionality. This architectural principle for robustness can, however, be easily compromised by implementation-level errors. We describe SafeFlow, which statically analyzes core components in the system to ensure that they use non-core values communicated through shared memory only if they are run-time monitored for safety or recoverability. Using simple, local annotations and semantic restrictions on shared memory usage in the core component, SafeFlow precisely identifies accesses to unmonitored non-core values. With a few false positives, it identifies erroneous dependencies of critical data on non-core values that can arise due to programming errors, inadvertent accesses, or wrong assumptions regarding the absence of difficult-to-detect implementation errors such as data races and synchronization. We demonstrate the utility of SafeFlow by applying it to discover critical value flow dependencies in three prototype systems.

1 Introduction

Modern embedded control systems are composed of multiple software components communicating with each other to control a physical device or the environment (termed the *plant*). The most critical requirement of such a system is safety, i.e., enforcing that the plant remains within a continuous state space, which is acceptable by system requirements. A typical control system, however, provides various other features and functionality, including performance enhancements for more efficient control, user interfaces, and complex modes of operation. These features are collectively deemed non-critical. With the evolution of such a system, the complexity and the relative size of the non-critical features steadily increases, making the system unwieldy and unverifiable in its entirety. As a result, guaranteeing the critical functionality (safety properties) of the complex system becomes a significant challenge.

The dichotomy between the demand for increasing functionality and the need to guarantee the safety requirements of the system, necessitates software architectural solutions. A key architectural principle that has emerged is the separation of critical functionality and non-critical functionality among a set of *core* and *non-core components* respectively. The architectural design aims to enforce that non-core component failures do not compromise core subsystem functionality. In general, a small set of simple, well-tested components constitute the core subsystem, while newer, untested components and non-essential features form the non-core subsystem.

We illustrate the ideas in this work using the running example of the inverted pendulum control system shown in Figure 1. The pendulum is balanced by the controller, which periodically reads the track position and pendulum angle from the sensor and outputs a voltage value between -5V and $+5$V to the actuator moving the supporting trolley left or right with different accelerations. The critical functionality of the system is to keep the pendulum upright at all times. This can be achieved by the core system consisting of the core controller and the sensor/actuator implemented in hardware. A controller, which minimizes the jitter of the pendulum while balancing the pendulum using more complex control algorithms, is implemented separately as a non-core component, which communicates with the core controller using shared memory. Similarly any user interface, which displays the status of the controllers by reading the shared memory is a non-core component.

In general, there is a two-way data flow between core and non-core components. For instance, in Figure 1, the core controller communicates sensor readings to the non-core controller, which, in turn, communicates its computed control outputs to the core controller. A unique attribute of values in control systems is that they quantify properties of the physical world. This hybrid nature of control systems, due to the continuous dynamics of the plant and the environment ensures that values exhibit continuity. This enables the core subsystem to maintain a conservative model of the physical system, which can be used to verify safety and system recoverability properties of unreliable values generated by the non-core subsystem. As in any other domain, it is difficult, if not impossible, to check the correctness of values such as control outputs (e.g. output voltage of the inverted pendulum controller) if they are in a permissible range (e.g.

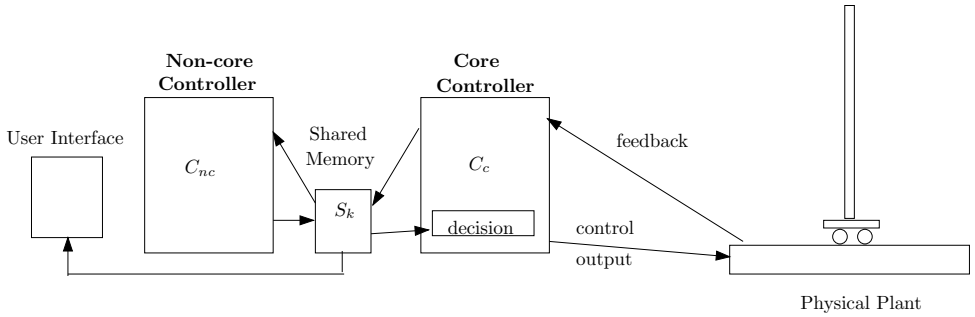

Figure 1: Running Example: Inverted Pendulum

[−5,+5]V). However, the core subsystem can take advantage of the model to verify that the system remains in a recoverable state if a non-core value is utilized by the core subsystem. This kind of a safety check is termed a *monitor*.

Examples of such monitors abound in control system design. In Figure 1, the core component can use the Lyapunov stability envelope proposed by the Simplex architecture [22] as a run-time monitor to check that the system remains in a recoverable state if a non-core control output is applied to the plant. The ability to check recoverability has been exploited by Cunha *et al* [3] for disaster prediction and avoidance in control systems. In our own experience with autonomous car controllers at UIUC [11], control outputs are monitored for potential collisions with other cars or obstacles before being applied to a car actuator.

The broad goal of this work is to enforce the following property in the actual system implementation (in C):

> **Safe Value Flow**: *All non-core values flowing into a core component should be monitored before use in critical computation*

This verifies that the core subsystem does not have any dependencies on non-core values. Ding and Sha have described this requirement as a prerequisite to the core component *using* the non-core component, but not *depending* on it [8]. This is a refinement of the conventional notion that there exists a failure dependency if there is *any data flow* between two component [18].

Lampson has defined three channels of value flow between components in the security context: legitimate, covert, and storage [14]. While legitimate channels of information involve communication through variables in the local process context, storage channels use environment objects and covert channels use means that are not reflected in values stored anywhere in the system (e.g. execution time). In this work, we only address legitimate channels of value flow from non-core to core components. Particularly, through the paper, we assume that all communication is performed through shared memory (Message passing extensions are briefly discussed in section 3.4). This is a reasonable assumption since shared memory communication is frequently used in embedded control systems for its fine-grained flexibility and the potential run-time efficiency. While covert channels are not relevant in the context of dependability, value flow through storage channels needs to be addressed in future work.

There are three distinct sources of low-level implementation errors that can violate the safe value flow property above. Firstly, programming bugs can cause the core component to use non-core values without monitoring along some path. This is particularly important as the number of paths in the core component increases, making it difficult to inspect manually. Secondly, programmers can use unmonitored non-core values due to inadvertent accesses to non-core values in shared memory. The primary reason for such accesses is that, in C, pointers to shared memory are not locally distinguishable from other pointers. Interprocedural propagation of shared memory pointers can result in inadvertent accesses to shared memory locations in the core component. The third source of implementation errors is subtler in nature. Core component developers often make assumptions regarding the safety of certain shared memory locations and rely on the encapsulation of the non-core component, synchronization and atomicity, data format compatibility and other pre-conditions that are difficult to verify. Previous work that verifies these properties (see section 5) have been best-effort in nature. Violation of any of the above properties results in the propagation of unreliable values to the core component that should be run-time monitored. In this sense, the safe value flow property is foundational and offers a "last line of defense" against many difficult-to-detect interaction errors.

We have designed *SafeFlow*, a static analysis tool that detects potential value dependencies of core components on the non-core subsystem. Static analysis offers the benefits of incurring no run-time overheads and early error detection, which are attractive advantages for embedded systems (run-time error dependency detection incurs performance penalties). Assuming that monitors are correctly implemented, SafeFlow relies on a few simple and local programmer annotations to describe semantic information about monitors, critical data, and shared memory initialization in the core component. Also, C programs have many language features that make them difficult to analyze statically. Hence, SafeFlow takes advantage of the domain-specific uses of shared memory and imposes a few reasonable semantic restrictions on shared memory pointer usage.

The SafeFlow analysis precisely identifies all uses of un-

monitored non-core values communicated through shared memory to core components at development time. Importantly, it reports errors for all unmonitored non-core values that affect critical data within the core component. A secondary contribution of SafeFlow is the simple and succinct annotation language that is useful in representing semantic information in embedded C programs. We applied SafeFlow to three prototype control systems and discovered five critical erroneous value dependencies of the core subsystem on non-core values. There are two main limitations to SafeFlow. First, SafeFlow can produce a few false positives due to control dependence on non-core values not used in critical data computation and due to imprecision of the static analysis. We currently require that the errors are verified using the value flow graphs manually. Secondly, erroneous annotation of an inaccurate or incomplete monitoring function can cause it to miss real dependencies (false negatives). This problem is nearly impossible to eliminate, though we mitigate it by designing a simple annotation language.

The remainder of the paper is organized as follows. The next section discusses the basic principle of our approach. The analysis details are presented in Section 3. We show the experimental results of applying our analysis in Section 4. Section 5 discusses related work and Section 6 concludes.

2 Basic Approach

Consider a core component, which communicates with non-core components using a set of shared variables. For each shared variable, S_i, we define the following mutually exclusive predicates:

- $\texttt{noncore}(S_i)$: holds if x can be written by any non-core component
- $\texttt{core}(S_i)$: holds if it can be verified that x is only written by core components. ($\texttt{core}(S_i) \Rightarrow \neg\ \texttt{noncore}(S_i)$ and $\texttt{noncore}(S_i) \Rightarrow \neg\ \texttt{core}(S_i)$)

Strictly applying the `noncore` predicate to shared variables that can be written by non-core components helps us detect dependencies on non-core values that arise due to difficult-to-detect bugs in inter-component interaction such as data format compatibility and synchronization. For a local value in the core component, x, we define the following mutually exclusive predicates:

- $\texttt{safe}(x)$: holds if x is defined by the core component or if its value is not dependent on any non-core values
- $\texttt{unsafe}(x)$: holds if x is dependent on any non-core values. ($\texttt{unsafe}(x) \Rightarrow \neg\texttt{safe}(x)$ and $\texttt{safe}(x) \Rightarrow \neg\ \texttt{unsafe}(x)$)

We now define the following operational rules for shared variable access in core components:

- Shared memory read: For $x = \texttt{read}(S_i)$, $\texttt{noncore}(S_i) \Rightarrow \texttt{unsafe}(x)$ and $\texttt{core}(S_i) \Rightarrow \texttt{safe}(x)$
  Reading a non-core shared variable returns an unsafe local value. Also, reading a core shared variable returns a safe local value.
- Shared memory write: $\texttt{write}(S_i, x)$
  Writes to a shared variables, S_i, using a local value in the core component *does not modify* the truth values of $\texttt{core}(S_i)$ and $\texttt{noncore}(S_i)$. This is because non-core shared memory locations are assumed to be accessible to non-core components throughout their lifetime. Verifying the absence of data races and the correctness of synchronized accesses is difficult and cannot be assumed.

Using the above operational semantics, all reads of non-core shared variables by the core component only return unsafe values. However, as discussed in section 1, core components in control systems contain run-time monitors for non-core values, which read non-core values and check them for safety before using them. In order to handle this, the programmer identifies functions where the non-core shared variable, S_i, is monitored before use (termed a *monitoring function* for S_i), specifying that $\texttt{core}(S_i)$ holds within this function. Thus, using our operational rules, reading S_i within this monitoring function returns safe local values. The programmer is expected to verify that the monitoring function correctly checks the non-core values for safety (or recoverability) before storing it in local variables that escape the monitoring function or using it in computation of critical data. This facility enables core components in control systems to safely read non-core values.

In the following section, we use the above principles to design a simple annotation language, which encodes semantic information in the core component, and statically analyze the core component to detect unsafe non-core value propagation to critical core component data.

3 SafeFlow Analysis

In the previous section, we described our basic approach in identifying monitoring functions and using this information to analyze the core component for unmonitored non-core value accesses that propagate to critical data. In this section, we first describe the annotations that we require from the developer, which provide semantic information about monitoring functions and critical data. In order to use this information in our analysis, we need to address two key issues that arise in weakly typed languages such as C, which makes sound and precise analysis of programs challenging: (a) memory errors and (b) aliasing and type-unsafe language constructs. To address (a), we leverage our prior work on guaranteeing memory safety and enforcing the semantics of the pointer analysis results in the presence of memory errors. To address (b), we impose a reasonable set of restrictions on shared memory pointer usage, by exploiting the limited ways in which embedded control systems use shared memory. These restrictions enable us to statically analyze the core component to precisely identify

unmonitored non-core value access as well as erroneous dependencies with few false positives.

```
  SHMData *noncoreCtrl;
  SHMData *feedback;

  float decision(Feedback* f,
                 float safeControl,
                 SHMData *noncoreCtrl)
  /***SafeFlow Annotation
      assume(core(noncoreCtrl, 0,
                  sizeof(SHMData))) ***/

1:   if (checkSafety(feedback, noncoreCtrl))
2:      return noncoreCtrl->control;
3:   else
4:      return safeControl;
  }

  main()
  {
1:   void *shmStart;
     /* Initialize shared memory */
2:   shmid = shmget(SHMKEY, SHMSize, flags);
3:   shmStart = shmat(shmid, 0, 0);
4:   feedback = (SHMData *) shmStart;
5:   noncoreCtrl = feedback + 1;

6:   Lock(shmLock);
7:   while (1)
8:   {
9:      float *output;
10:     getFeedback(feedback);
11:     computeSafety(feedback,
                      &safeControl);
12:     Unlock(shmLock);
13:     wait(tsecs);  /* Wait for non-core
                component to compute value */
14:     Lock(shmLock);
15:     output = decision(feedback,
                  safeControl, noncoreCtrl);
        /***SafeFlow Annotation
            assert(safe(output)); ***/
16:     sendControl(output);
17:   }
  }
```

Figure 2: Example: Core Controller Code

In order to explain our analysis, we use the example in Figure 2, which is a simplified version of the core controller in the Simplex architecture implementation for the inverted pendulum [21] from figure 1. In each period, the core controller dispatches a control output to the actuator in order to balance the pendulum. The output of the non-core controller is dispatched, if it can be checked to maintain the system in a recoverable state by the `decision` module. Otherwise a safe control output, computed by the core controller, is applied to the actuator. The routine, `main`, allocates the shared memory using the UNIX system call (lines 1-3) and initializes the global variables, `noncore` and `feedback`, to point to shared memory (lines 4-5). Within the loop, during each period, the core component receives feedback about the position of the pendulum, which is published in shared memory (line 10), computes the safe control output (line 11), waits for the complex controller to publish its computed control output (line 13), checks the non-core component output for recoverability in the function, `decision`, and sends the appropriate control output to the actuator (line 16). The goal of our analysis is to detect unmonitored non-core values in the core component and enforce that the critical value in the core component, `output`, does not depend on any unmonitored non-core values.

3.1 Annotations

One of our major goals in this work is to incur minimal or no burden on the programmer in verifying safe value flow in the system. Thus, we ensure that our approach requires minimal programmer annotations. Moreover, we require that annotations are local, succinct, and intuitive to the programmer. Our analysis requires two kinds of semantic annotations from the programmer: (a) Identifying critical data and (b) Characterizing monitoring functions. In general, annotations are undesirable since they preclude using the technique on legacy code without some porting effort and incur a burden on the programmer. However, annotations in our approach are unavoidable since they describe semantic information only known to the developer. On the flip side, imposing annotations on the programmer has the advantage that it enforces a discipline with respect to identifying critical data and the behavior of monitoring functions.

Our annotations are enclosed within C comments which begin with the special string, *SafeFlow Annotation*, as shown in figure 2. There are two kinds of annotations: `assume` annotations that provide semantic information about monitoring functions that can be used as facts by the analysis and `assert` annotations that specify the property that must be checked or validated by the analysis.

Monitoring functions need to specify that certain memory locations in shared memory can be assumed to be core in the function and in any function invoked recursively by the monitoring function. For this purpose, the `assume` annotation is declared using the predicate `core`, which is applied to a shared memory pointer, `shmptr` as follows: `core(shmptr, offset, size)` , which denotes that the shared memory locations accessible using `shmptr` from offset, `offset`, for `size` bytes can be assumed to contain core values. Offset and size values should span an entire array in shared memory, since an array is treated as a single unit by our analysis; otherwise, the annotation becomes ineffective.

In other words, the values read from these locations will be safe according to our operational rules in section 2. This annotation is illustrated in the function, `decision`, above the function body in figure 2, which specifies that the shared memory pointer, `noncoreCtrl`, can be dereferenced safely between offsets `0` and `sizeof(SHMData)`. This annotation is local and can be specified in terms of local or global shared memory pointers. Also, the monitoring function developer clearly knows the locally shared memory locations that are being monitored and used in that function.

The other semantic information we require for our analysis is the identification of critical data in the core component. Here, we need to specify the requirement that this

critical data is safe i.e. it does not depend on non-core values. For this purpose, we employ the `assert` annotation on the `safe` predicate, which takes a primitive typed variable such as `char`, `int`, `float`, or `double`). Its syntax is simply the following:
`safe(x)`, which denotes that the local value `x` is safe.
This is illustrated in the annotation preceding line 16 in `main` in figure 2. In general, annotations that identify the critical data in the core component are inserted at program points preceding communication with another component through an I/O operation. Similarly, the arguments to system calls such as the process-id argument to `kill` are asserted to be critical data in the core component. The `assert` annotations required by our analysis are much simpler than the assertion invariants in [17], which attempt to capture the functionality of the system (That work has shown that specifying the latter is not easy for all programmers).

In addition to the above annotations, in our implementation, we need some annotations to describe the shared memory pointers returned by untyped shared memory initializing functions. This is described in section 3.2.1.

3.2 Language Restrictions

Generally, precisely checking safe usage of unreliable values in components written in weakly typed languages like C is statically undecidable. While C is extensively used in programming embedded system components, it permits many type-unsafe constructs including pointer arithmetic, arbitrary casts and even memory errors, due to which it is impossible to precisely identify shared memory accesses statically in generic C programs.

The first issue posed by type-unsafe constructs and memory errors is that the local pointers in the core component could access shared memory in arbitrary and unpredictable ways via bounds violation, dangling pointer dereference or uninitialized pointer dereferences. In order to counter this, we propose to leverage our previous work on memory safety. In [7], we proposed a restricted type-safe subset of C, which we statically analyze to guarantee memory safety. We essentially propose a combination of static analysis, minimal (often zero) run-time checking and some system support (only to minimize the run-time overhead) to ensure that uninitialized pointers, type-unsafe casts, dangling pointers to freed memory or stack memory, and array bounds violation do not overwrite any data area not allocated by the component. In our case, since shared memory is allocated through the shared memory libraries, our analysis and language restrictions enforce that local pointers in the core component cannot access any shared memory locations. In a follow-up work [6], we describe the additional run-time checks required to guarantee memory safety (in fact, the stronger property of guaranteeing the semantics of the pointer analysis results) for practically the full generality of C.

The second consequence of type-unsafe constructs is imprecision in tracking shared memory location accesses leading to a greater number of false positives in a conservative analysis. To statically analyze value flow precisely, we impose a few semantic restrictions on shared memory pointer usage, which enables better precision , ensures expressiveness to program practical embedded systems, and handles legacy systems with minimal porting effort.

Broadly, the goals of our restrictions enforce that a pointer to shared memory cannot be aliased through memory locations, and arrays in shared memory cannot be indexed in ways that make it difficult, if not impossible, to analyze statically. Also, we require that shared memory is not deallocated or destroyed until the end of the program (the end of function `main`), in order to prevent dangling pointers to shared memory in the program. Dereferencing dangling pointers to system-defined shared memory can have unpredictable consequences such as crash failures of the core component. The restrictions are as follows: (**P1**) Shared memory cannot be deallocated until the end of the main function; (**P2**) Taking the address of a pointer to shared memory is disallowed; and (**P3**) Casts between pointers to incompatible types in shared memory and casts from shared memory pointers to integers is disallowed.

In addition , we adapt the restrictions on arrays in our previous work [7] to the arrays in shared memory:

(**A1**) Indices used to access arrays within shared memory must lie within the bounds of the array

(**A2**) If an array in shared memory, `A`, is accessed inside a loop, then: **(a)** the bounds of the loop must be provably affine transformations of the size of `A` and outer loop index variables or vice versa;**(b)** the index expression in the array reference, must be a provably affine transformation of the vector of loop index variables, or an affine transformation of the size of `A`; and**(c)** if the index expression in the array reference depends on a symbolic variable `s`, which is independent of the loop index variable (i.e., appears in the constant term in the affine representation), then the memory locations accessed by that reference have to be provably independent of the value of `s`.

Rule **P1** above prevents dangling pointers to shared memory. Rule **P2** disallows aliasing shared memory pointers by storing them in memory. **P3** ensures that the shared memory is used in a type-safe manner. The array rules **A1** and **A2** verify that the arrays in shared memory do not violate their bounds. In general, inability to distinguish statically between array locations results in an array index operation conservatively assumed to be anywhere within the span of the memory locations represented by the array.

3.2.1 Shared Memory Initialization

In addition to the `assume` and `assert` annotations described in section 3.1, we require one other annotation in order to facilitate our analysis. This is due to shared memory allocation through systems calls (in UNIX for instance) being untyped, which necessitates shared memory pointer

```
  ...
  initComm(key_t SHMKEY, size_t SHMSize,
           FLAGS flags)
  /***SafeFlow Annotation
      assume(shminit); ***/
  {
1:   void *shmStart;
     /* Initialize shared memory */
2:   shmid = shmget(SHMKEY, SHMSize, flags);
3:   shmStart = shmat(shmid, 0, 0);
4:   feedback = (SHMData *) shmStart;
5:   noncoreCtrl = feedback + 1;
     /***SafeFlow Annotation
         assume(shmvar(feedback,
                       sizeof(SHMData))) ;
         assume(noncore(feedback));
         assume(shmvar(noncoreCtrl,
                       sizeof(SHMData)));
         assume(noncore(noncoreCtrl));
     ***/
     InitCheck(shmStart, SHMSize,
               feedback, sizeof(SHMData),
               noncoreCtrl, sizeof(SHMData));
  }
  main()
  {
1:   initComm(SHMKEY, SHMSize, flags);
2:   Lock(shmLock);
     ...
```

Figure 3: Initialization function

casting and pointer arithmetic in order to initialize shared memory. These constructs violate the restrictions on shared memory pointer usage described above. Moreover, due to the type-unsafe constructs, the sizes of arrays in shared memory need to be made explicit. We also require annotations to specially designate that these initialization routines do not require to adhere to our language restrictions.

For instance, lines 1-5 of `main` in figure 2, the pointer returned by the call to `shmat` is cast to a pointer to a structure of the returned type (violating rules **P3**). In order to overcome this, one option is to write and invoke typed wrappers for these system calls for each required type. This is clearly onerous on the developer. In our approach, we designate that the initializations are performed in a special function known as the *initializing function*, identified by the `assume` annotation using the predicate `shminit`. The predicate, `shminit`, permits **P3** to be violated.

To overcome the type-unsafe initialization of shared memory, the initializing function needs to be annotated to identify the shared memory variables and their respective sizes. For this purpose, we employ `assume` annotations on the predicate `shmvar`, which takes the typed shared memory pointer as the first argument and the total size of the shared memory locations that can be accessed through the pointer as the second argument:
`shmvar(shmptr, size)`
The size of the array pointed to by the shared memory pointer can be inferred by dividing the size of the shared memory, `size`, by the size of the type pointed to by `shmptr`. It is important to verify that the locations pointed to by individual shared memory pointers are non-overlapping and the entire size span of each of these pointers be valid. To annotate that the set of locations that can be accessed by a shared memory pointer are non-core, we employ the `assume` annotation on the predicate, `noncore`, if the shared memory locations can potentially be overwritten by a non-core value. This predicate is applied to a shared memory pointer as follows:
`noncore(shmptr)`

In figure 3, we show an annotated version of `initComm` with all the above annotations. The `shminit` annotation is written just below the function declaration and applies to the function and any function invoked recursively by it. The `shmvar` and the `noncore` annotations are written at the end of the function and are post-conditions of the initializing function. In this case, `feedback` and `noncoreCtrl` are declared to be two shared memory variables of size `sizeof(SHMData)`. In order to assist the programmer in writing correct size annotations, we automatically insert a run-time check:
`InitCheck(void *SHMStart, size_t SHMSize, ...)`
which verifies that the variables in shared memory do not overlap with each other. If this check fails, the core component is terminated before it bootstraps. While this is a run-time check, it is only executed once during shared memory initialization.

3.3 Static Analysis

Our static analysis algorithm operates in three phases on the core component implementation: (i) Identification of pointers to shared memory interprocedurally; (ii) Enforcing language restrictions **P1**-**P4**, **A1**, and **A2**; (iii) Identifying non-core shared memory accesses and determine if critical data is control or data dependent on unsafe local values. The analysis is implemented on low-level virtual machine (LLVM) byte-code [16], which is a typed intermediate format that is in static single assignment (SSA) form.

Before the three phases of our analysis, we execute a pre-processing pass on the C code which converts `assume` and `assert` annotations to calls to external dummy functions. Annotations on monitoring functions and initializing functions are specified at the entry point of the function. The post-condition of the initializing function is generated at all the exit points of the function. In the first phase, we discover the initializing functions in the program and identify the shared memory pointers initialized. We then propagate these pointers interprocedurally using a bottom-up and top-down analysis on the strongly connected components (SCCs) of the call graph. These pointers escape the functions as globals, function arguments or return values. The post-conditions of initialization functions identify the pointers to shared memory that escape the function. In the bottom-up pass, these pointers are propagated to the callers until the root of the call graph i.e. `main` is reached. Within each function, a standard global data flow algorithm is used on the basic blocks in the control flow graph (CFG). At merge points (due to conditionals or loops), a pointer is conservatively assumed to point to shared memory if it

is initialized on some path. Within SCC's, the pointers to shared memory are propagated to the callers until the shared memory pointer information for each function stabilizes. A top-down pass on the call graph propagates the pointers to shared memory from the root of the call graph to the callees.

The second phase of our analysis enforces the language restrictions **P1**-**P3**, **A1**, and **A2**. According to rule **P1** pointers to shared memory in each function should not be arguments to shared memory deallocation functions (`shmdt` for instance). This is easy to analyze by examining all the uses of the pointer by following def-use chains. **P2** and **P3** can be similarly verified by examining the uses of pointers to shared memory within each function. Casting between types in LLVM necessarily uses the `cast` instruction. Casting a pointer to shared memory to a pointer to an incompatible type is disallowed (**P3**). **P2** is verified by enforcing that a pointer to shared memory is never stored in any pointer using the `store` instruction in LLVM.

Our restrictions on array indexing are derived from a previous work [7]. Constraints **A1** and **A2** are verified by generating constraints on each array index expression in the program interprocedurally. The constraint propagation algorithm is exactly the same as the one developed previously. The size of the array in shared memory is provided by the annotation in the initializing function. The set of affine constraints are given to a integer programming solver such as Omega [13], which checks that there are no array bounds violations.

The final phase of our algorithm processes the `assume` annotations in each function (except the initialization function) and determines the core set of shared memory locations accessible within each function. This is compared to the non-core shared memory locations accessed in the function to determine the unsafe values read from unmonitored shared memory. A warning is reported for each unsafe access to shared memory, without any false positives or false negatives. Finally, we enforce that critical data is not data or control dependent on unsafe shared memory accesses using an interprocedural value flow analysis on the critical data. An error is reported when the analysis detects dependency of critical data value in the core component on unmonitored non-core values.

We propagate unsafe values through the program using a standard value flow graph [4]. We verify the assertion for critical functionality, by checking if the critical data depends on an unsafe value. We use an alias analysis like Data Structure Analysis (DSA) [15], which maintains points-to graph and a typed representation of the memory in the program. DSA is a context-sensitive, field-sensitive, and flow-insensitive analysis.

The critical data analysis algorithm checks for dependencies on unsafe data read by the core component from shared memory using an interprocedural, context-sensitive, and flow-sensitive algorithm. Currently, each function in the core component is analyzed multiple times for different call sequences leading to it, making the implementation exponential in run-time complexity. In practice, the core component in an embedded system is simple and has relatively fewer paths than system software. Moreover, the overhead due to static analysis time for safety violation detection is not a significant factor in most development and testing efforts.

In the example in figure 2, the shared memory pointer, `feedback`, is used in the function, `decision` (which only annotates `noncoreCtrl` as being safe). Thus, any values generated by `decision`, which depend on `feedback` are unsafe. This includes the return value, `output`, which violates the critical functionality requirement of the component. The dereferencing of `feedback` in `decision` is reported as unsafe. One way to eliminate this dependency is to use a local copy of the feedback as an argument to `decision`, rather than the pointer to the shared location.

The algorithm can be made more efficient by analyzing each function only once and summarizing the data dependencies in the functions using *value flow graphs* developed in ESP [4]. This ability to summarize procedures means that we can carry out a single bottom-up pass on the SCC's in the call graph, inlining the value flow graphs in the callers and using these graphs to determine if critical data depended on any unsafe accesses to shared memory.

3.4 Discussion and Extensions

3.4.1 False Positives

Our analysis detects false positives, due to two reasons: the imprecision in our analysis and control dependence on shared memory variables. The merging of unsafe predicates on variables upon branch merges and the path-insensitivity in the final phase of our analysis can result in false positives, due to errors flagged by infeasible paths. In the future, our analysis could be combined with path-sensitive value flow algorithms such as ESP [4]. Similarly, the imprecision of the pointer analysis used to check if any unsafe data is reachable from critical pointer data can also lead to false positives. While improving the precision of the pointer analysis results using more aggressive analyses (e.g. making it flow-sensitive) increases precision, eliminating false positives in all cases is difficult.

The second source of false positives is due to critical data being control dependent on unmonitored non-core shared memory values. For instance, in one of our test-cases, the configuration of the system is present in shared memory. The core component reads the configuration without monitoring it and computes critical data differently based on the presence or absence of a non-core component. In one path of execution, the critical data uses the non-core values after monitoring it. In the other path of execution, the core component outputs safe data computed by it. The critical data is computed correctly in either path of execution, but the control dependence on the non-core configuration data reports an erroneous dependency. The source of this false positive is due to the inability of the analysis to automatically infer whether the non-core variables modifying the control flow

affect the critical data computation in the core component. In these cases, manual inspection of the reported errors is required. However, it is important to realize that in these scenarios, a superior design would be to restructure the non-core components by separating out an additional core component that writes the configuration in shared memory.

3.4.2 Non-core component encapsulation

Due to our conservative model of the non-core component, the core component cannot rely on atomicity properties such as writing and reading a shared variable in sequence and expecting the written value to be read. In Section 1, we motivated our conservative model of the values produced by the non-core component as being due to the difficulty of detecting many complex errors in non-core component behavior. In special cases, more fine-grained model of non-core component behavior could be exploited, due to guaranteed absence of errors such as synchronization errors or data compatibility errors. This can easily be expressed by using more `assume` annotations in the core component to declare shared memory locations to be `core` within certain functions. For instance, in the example in figure 2, the function `decision` could be further annotated with `assume(core(feedback, 0, sizeof(SHMData)))`, thus declaring feedback to be safe to dereference in `decision` and all the functions recursively called by it. A limitation of this approach is that annotations in our current approach can only be applied at the function level and might necessitate restructuring the code to further modularize the program appropriately.

3.4.3 Message Passing and I/O calls

In this section, we briefly discuss extending our approach above to communication through message passing and I/O library calls. We define a trusted set of library calls we can use for receiving messages and performing I/O reads. We illustrate the required annotations on the library calls using the example of the `recv` call on sockets:

```
ssize_t recv(int socket, void *buffer,
             size_t length, int flags);
```

First, we use the predicate, `noncore(socket)`, to specify that socket file descriptor, `socket`, is used to communicate with non-core components. Otherwise, the socket is assumed to be used for communication with core components. Additionally, we require that these descriptors are not used in any computation and merely passed by value across procedures. Socket file descriptors not annotated as non-core are assumed to communicate with core components. In practice, the sockets communicating with core components need to contain run-time authentication to check that the peer components are indeed a part of the core subsystem. Secondly, we use `assume` annotations to define that it is safe to dereference received non-core data within the function. This is exactly the same as the `assume(core(...))` annotations in the monitoring functions, except that it is applied to a local pointer. All other pointers to received data are assumed to be unsafe, being from non-core components.

4 Results

Through our experiments, we intend to answer the following questions:

- Is the restricted language expressive enough to develop embedded control systems?
- Are the annotations onerous on the programmer?
- Does the analysis successfully detect erroneous dependencies on non-core values in our tests?

Table 1 shows the results of applying our analysis on three laboratory control systems. All three systems implement the robust architectural design of isolating the core components and monitoring for non-core values flowing to the core component. The first system is a Simplex architecture for an inverted pendulum (IP) controller used to balance an inverted pendulum as seen in our running example (figure 1). The second is a generic Simplex architecture implementation for simple plants with a configuration file that can be customized for different plants. The third system is a double inverted pendulum control system that is based on the inverted pendulum controller code, albeit with changes to enable additional control modes. The first two systems have been used in the real-time systems laboratory at UIUC for three years and have been extensively tested. In particular, these systems were designed as a demonstration of the Simplex architecture for core component isolation. Much effort has gone into the development of these two systems, particularly in protecting against non-core values in the core component. The double IP controller is a relatively new system, whose implementation is currently being refined in our laboratory. For running our analysis, we used a preliminary version of the double IP controller.

In order to apply SafeFlow, we needed to annotate the core components of the systems with information about shared memory initialization. The critical data in the core components is the control output being sent to the actuator and the first argument (process id) of the `kill` system call invoked (annotated using `asserts`). Finally, we required annotations on the monitoring function specifying the non-core accesses that are safe to access within the function. The number of lines of annotation is small in all cases. In particular, majority of the annotations (9 of 11 lines in IP control, 15 of 22 in generic Simplex, and 15 of 23 in double IP control) were used to annotate initializing functions. Notably, no source changes were necessary for the systems to adhere to our language restrictions. A very small number of source changes were required in two experiments to separate the monitoring function, which was a part of a larger function. This was necessary because the annotations for the monitoring functions can only be specified at the function level. In table 1, the number of actual lines of source changed and

System	LOC (total)	LOC (core)	Source Changes (LOC)	Annot. lines count	Error Dependencies	Warnings	False Positives
IP	7079	820	7 (86)(1 func)	11	1	7	2
Generic Simplex	8057	1020	0	22	2	7	6
Double IP	>7188	929	7 (88)(1 func)	23	2	8	2

Table 1: Applying SafeFlow to Control Systems: Results

the `diff` output of the modified program with respect to the original program are listed.

The SafeFlow analysis detected several warnings or unmonitored non-core value accesses in all three systems. These warnings contain no false positives are a very useful output of the analysis since it makes explicit all unmonitored non-core values read by the core component. SafeFlow detected two erroneous value dependencies of the core component on the non-core component in the generic Simplex system, two in the double IP system and one in the inverted pendulum controller. In the generic Simplex implementation, one erroneous dependency was caused by a shared memory variable (the sensor feedback value) being written by the core component and read later by the core component. This potential value dependency on non-core values would be fatal, if the non-core component replaced the sensor feedback with a hand-crafted value that would "rig" the recoverability check to permit an erroneous non-core value to be used by the core controller. This could occur due to an erroneous non-core component implementation, which overwrites the feedback value (which is supposedly read-only, but not enforced) or violation of the synchronization on the feedback value in shared memory due to data races.

In all the three systems, the first argument of a `kill` system call invoked by the core component was dependent on an unmonitored non-core value. This could be easily used to bring down the core component if the non-core component overwrote the value with the process id of the core component itself, causing the core component to kill itself! One error in the double IP controller is a result of accessing an unmonitored non-core value assuming that this value does not propagate to the critical data in the core component. Our analysis discovers that this assumption is invalid. It is important to note that the three systems tested were designed and implemented to provide safe value flow from non-core to core components. The five errors we found are very subtle, in that they capture implementation oversight and erroneous assumptions.

SafeFlow returns a few false positives among its errors. All false positives returned in our tests were due to control dependence on non-core values that do not affect critical data computation. These needed to be manually identified with the aid of the value flow graphs representing the flow of values from unmonitored non-core values to the critical data. False positives can be reduced by using the `assume` annotation to declare such non-core values as being safe to access within certain functions, only after reliably verifying this fact.

5 Related Work

As described in section 1, there have been several architectural designs that employ monitors to protect core components against non-core values [22, 3, 11]. The notion of isolating critical component functionality has been formally modeled by Arora and Kulkarni [1] in order to build masking fault-tolerant systems. Further, Jhumka *et al* [12] have proposed the design of accurate and complete detectors in such an architectural design. However, none of these works have addressed enforcing these principles in low-level implementation and have verified their designs only at the model level, which is a much higher level of abstraction. We have shown that implementation errors can easily violate even the simple architectural principle of safe value flow.

The SafeFlow approach is closest to previous work in taintedness analysis and secure value flow. The taintperl [25] package employs data flow techniques to quarantine data potentially contaminated by malicious users (completely preventing the use of such data). Being an interpreted language, the taintedness information is tracked at run time. In the context of security, Denning and Denning [5] first proposed a type system enhanced with security attributes in order to verify confidentiality and integrity of variables in the implementation. This has been further refined by Volpano *et al* [24] and in the design of JIF [19, 20]. None of these works are suited for the safe value flow property in embedded systems, for two main reasons: heavy-weight annotations and the lack of a notion of monitoring functions. Moreover, prior work in security typed languages has not practically addressed applying the techniques to C programs. In contrast, the key contribution of SafeFlow is the design of a succinct and light-weight annotation language to specify monitoring function properties and language restrictions that enables statically detecting erroneous dependencies. While secure programming languages can be justified to guarantee confidentiality and integrity in security-critical applications, safe value flow demands legacy code compatibility and ease of use by embedded system developers. Declassification in JIF is the closest analogue to monitoring in literature, although the semantics are different. We borrow our value flow graphs to propagate the `unsafe` predicate on a local variables through the program from the propagation of security attributes towards guaranteeing non-interference, for instance. This is also similar to Cqual [23] which contains mechanisms to annotate interface variables with attributes and propagate these attributes through the program (and potentially to assertion or interface violations).

Leveson [18] has developed a manual technique to analyze lines of code to generate software fault trees for each line of an Ada program. The manual technique and the large size of the fine-grained trees preclude the scalability of the technique. AFPI [2] attempts to determine failure dependencies across components through fault injection and testing methodology, without offering any guarantees

Enforcing locking protocols and detecting data races statically has been studied extensively, most recently in [9]. Data format compatibility and access control can be enforced by encapsulating the shared memory reads and writes by the non-core component and through assumption specifications in AADL [10]. These techniques are best-effort and do not attempt to capture all errors. Thus, assuming the absence of such errors is unjustified in general, motivating SafeFlow to use the conservative model for the non-core component in its analyses.

6 Conclusions and Future work

We described SafeFlow, an annotation-based static analysis that verifies that the core components guaranteeing critical functionality do not depend on unmonitored non-core values in the system. While monitoring has been used extensively in robust architectural design, various implementation errors can violate these architectural principles. The analysis tool enables core component developers to explicitly track non-core values in shared memory that are used without being monitored in core components. Further, the tool reports erroneous dependencies if these non-core values can affect critical data computation along some path in the system. Applying SafeFlow found critical, erroneous or inadvertent value dependencies in mature critical systems designed for safe value flow, with a few false positives. The SafeFlow experiments confirm the motivation behind the analysis tool as being a "final line of defense", complementing other best-effort error detection tools.

In the future, SafeFlow needs to address other channels of value flow between non-core and core components, significantly *storage channels*. Further attention is required on eliminating the false positives generated by SafeFlow.

References

[1] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Trans. Softw. Eng.*, 24(6):435–450, 1998.

[2] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 132, Washington, DC, USA, 2003. IEEE Computer Society.

[3] J. Cunha and M. Rela. On the use of disaster prediction for failure-tolerance in feedback control systems. In *International Conference on Dependable Systems and Networks*, Washington D.C., USA, June 2002.

[4] M. Das, S. Lerner, and M. Seigle. Esp: Pathsensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, 2002.

[5] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[6] D. Dhurjati, S. Kowshik, and V. Adve. Enforcing alias analysis for weakly typed programs. In *Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation*, page To Appear, 2006.

[7] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, Feb. 2005.

[8] H. Ding and L. Sha. Dependency algebra: A tool for designing robust real-time systems. In *RTSS*, pages 210–220, 2005.

[9] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[10] P. Feiler, B. Lewis, and S. Vestal. The sae avionics architecture description language (aadl) standard. In *RTAS Workshop on Model-driven Embedded Systems*, 2003.

[11] S. Graham. *Issues in the convergence of control with communication and computation*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 2004.

[12] A. Jhumka, M. Hiller, and N. Suri. An approach for designing and assessing detectors for dependable component-based systems. In *HASE*, pages 69–78, 2004.

[13] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Computer Science Dept., U. Maryland, College Park, Apr. 1996.

[14] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[15] C. Lattner and V. Adve. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Nov 2003.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Jose, Mar 2004.

[17] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Trans. Software Eng.*, 16(4):432–443, 1990.

[18] N. G. Leveson, S. S. Cha, and T. J. Shimeall. Safety verification of ada programs using software fault trees. *IEEE Softw.*, 8(4):48–59, 1991.

[19] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow, 2001.

[20] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[21] L. Sha. Dependable system upgrades. In *Proceedings of IEEE Real Time System Symp.*, 1998.

[22] L. Sha. Using simplicity to control complexity. *IEEE Software*, July/August 2001.

[23] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, pages 201–220, 2001.

[24] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.

[25] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly, 1991.