Appears in the proceedings of the *16th International Conference on Runtime Verification* (RV 2016).

TrackOS: a Security-Aware Real-Time Operating System

Lee Pike[1] and Pat Hickey[2], Trevor Elliott[1], Eric Mertens[1], Aaron Tomb[1]

[1] Galois, Inc. {leepike|trevor|emertens|atomb}@galois.com
[2] Helium pat@helium.com

Abstract. We describe an approach to control-flow integrity protection for real-time systems. We present *TrackOS*, a security-aware real-time operating system. *TrackOS* checks a task's control stack against a statically-generated call graph, generated by an abstract interpretation-based tool that requires no source code. The monitoring is done from a dedicated task, the schedule of which is controlled by the real-time operating system scheduler. Finally, we implement a version of software-based attestation (SWATT) to ensure program-data integrity to strengthen our control-flow integrity checks. We demonstrate the feasibility of our approach by monitoring an open source autopilot in flight.

1 Introduction

Cyber-physical systems are becoming more pervasive and autonomous without an associated increase in security. For example, recent work demonstrates how easy it is to gain access to and subvert the software of a modern automobile [5]. In this paper, we focus on software integrity attacks aimed at modifying a program's control flow. Traditional methods for launching software integrity attacks include code injection and return-to-libc attacks.

Control-flow attacks are well known, and protections like canaries [6,11] and address-space layout randomization [22] have been developed to thwart them. However, for each of these protections, researchers have shown ways to circumvent them, using techniques such as return-oriented programming [5].

More recently, *control-flow integrity* (CFI), originally developed by Abadi *et al.* [1], is more difficult to exploit. CFI implements run-time checks to ensure that a program respects its statically-built control-flow graph. If the control stack is invalid, then some other program is being executed; modulo false positives, it is a program resulting from a malicious attack.

Consequently, the CFI approach to security has been favored recently as the way forward in protecting program integrity. For example, Checkoway *et al.* demonstrate how to execute return-to-libc attacks without modifying return addresses [4]. In reference to traditional kinds of defenses, the authors write:

> What we show in this paper is that these defenses would not be worthwhile even if implemented in hardware. Resources would instead be better spent deploying a comprehensive solution, such as CFI.

The traditional technique for implementing CFI requires program instrumentation (the instrumentation can be done at various levels of abstraction, from the source to the binary). Instrumentation is not suitable for critical hard real-time systems code for at least two reasons. First, instrumentation fundamentally changes the timing characteristics of the program. Not only can instrumentation introduce delay, but it can introduce jitter: CFI checks are control-flow dependent. Second, safety-critical or security critical systems are often certified, and instrumenting application code with CFI checks may require recertification. Our approach allows real-time CFI without instrumenting application code.

The question we answer in this paper is how to provide CFI protections for critical embedded software. Our answer is a CFI-aware real-time operating system (RTOS) called *TrackOS*.

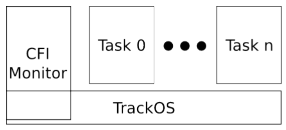


Fig. 1. *TrackOS* RTOS integration.

TrackOS has built in support for performing CFI checks over its *tasks*, as processes on an RTOS are generally known. *TrackOS* tasks do not require any special instrumentation or runtime modifications to be checked. *TrackOS* overcomes the delay and jitter issues associated with CFI program instrumentation: rather than instrumenting a program, CFI checks are performed by a separate *monitor task* as shown in Figure 1. This task is responsible for performing CFI checks on other untrusted tasks. The monitor task is scheduled by the RTOS, just like any other task. However, the task is privileged by the RTOS and is allowed access to other tasks' memory (this is why we show the task overlapped with the RTOS in Figure 1).

An insight of *TrackOS* is that RTOS design already addresses the problem of real-time scheduling, and CFI monitoring in a real-time setting is just an instance of the task scheduling problem. Furthermore, as an instance of the real-time task scheduling problem, the user has the freedom to decide how to temporally integrate CFI into the overall system design, given the timing constraints. For example, a developer could decide to make CFI monitoring a high-priority task if there is sufficient slack in the schedule or instead monitor intermittently as the schedule allows.

Summary of Contributions

1. *Static analysis*: Before execution, we analyze a task's executable to generate a call graph that is stored in non-volatile memory (program memory). We implement a lightweight static analysis that is able to analyze a 200KB machine image (compiled from an approx. 10kloc autopilot) and generate a call graph in just over 10 seconds on a modern laptop.
2. *Control-flow integrity*: At runtime, a monitor task traverses the observed task's control stack from the top of the stack, containing the most recent return addresses, to the bottom of the stack. The control stack is compared against the static call graph stored in memory. In our approach, we do not assume frame pointers, so the analysis must parse the stack. We make optimizations to ensure checks have very low-overhead. Most importantly, the overhead is completely controllable by the user using the RTOS's scheduler, just like any other task.

   This approach implements callstack monitoring rather than just checking well-formedness of function pointers, like many rootkit dectection mechanisms [12, 15, 16]. The approach supports concurrency (i.e., multiple tasks can be monitored simultaneously).
3. *Program-data integrity*: Our CFI approach is only valid as long as it is executing. An attacker that can reflash a microcontroller can simply overwrite *TrackOS* and any of its tasks. Consequently, we need a check that the program memory has not been modified. We implement a *software-based* attestation framework to provide evidence to this effect. The framework is not novel to us; we borrow the *SoftWare-based ATTestation* (SWATT) approach tailored to attestation in embedded systems [20]. Our full implementation therefore answers a challenge by the authors of SWATT, in which they note that "software-based attestation was primarily designed to achieve code integrity, but not control-flow integrity" [14]. As far as we know, this is the first integration of software-based program-data integrity attestation with control-flow integrity; de Clercq *et al.* previously combine CFI and data integrity relying on hardware support [7].

Assumptions and Constraints Regarding system assumptions, while not fundamental to our approach, we assume execution on a Harvard or modified Harvard architecture in which the program and data are stored in separate memories (e.g., Flash and SRAM, respectively). Return-oriented programming is still feasible on a Harvard architecture [9]. We do not assume the hardware supports virtual memory or provides read-write memory protections. We do not assume that programs have debugging symbols. We also do not assume the existence of frame pointers.

We assume the attacker does not have physical access to the hardware. However, she may have perfect knowledge of the software including exploitable vulnerabilities in the software, including the bootloader. She may have unlimited network access to the controller. We assume that the microcontroller's fuses allow all memory, including program memory, to be written to. Furthermore, *any* control-flow transfer technique is in-scope by the attacker.

2 Static Analysis

TrackOS compares the control stack against a statically-generated call graph of each monitored task. The call graphs are generated via binary static analysis tool called *StackApprox*; no sources or debugging symbols are required. *StackApprox* currently targets AVR binaries.

StackApprox is similar in spirit to a tool developed by Regehr *et al.* [17], although the use cases are different. In Regehr's case, the focus is on statically determining control-stack bounds, whereas our primary use case is to generate representations of call graphs as C code, although *StackApprox* approximates stack sizes, too. *StackApprox* uses standard abstraction interpretation techniques to efficiently generate a call graph; for the sake of space, we elide details about the tool's design and implementation.

Like in Regehr *et al.* [17], *StackApprox* analyzes direct jumps automatically but requires the user to explicitly itemize indirect jumps. Doing so ensures that all indirect jumps are specified and not the result of unintended or undefined (with respect to C source semantics) behavior. Moreover, large number of indirect jumps are not common in hard real-time systems (we itemized 30 targets for a 10K LOC autopilot, including interrupts).

For the purposes of CFI checking, we generate four tables or maps from the generated call graph. Only values for functions reachable from the start address are generated. Typically, the start address is the entry point for an RTOS task.

– *Loop map*: A mapping from return addresses to callers' return addresses associated with their call-sites.
– *Top map*: A mapping from call-targets (usually the start of a function definition) to the set of return addresses associated with the functions' call-sites.
– *Local stack usage map*: A mapping from call-targets to the maximum number of data bytes pushed on the stack, not including callees' stack usage.
– *Contiguous region map*: Pairs representing the start and stop address that define a contiguous region.

Our build system calls *StackApprox*, which generates C sources containing the four maps, and then integrates the generated C files into the build automatically. The basis of *TrackOS*, FreeRTOS (see Section 3), like many embedded RTOSes, statically links the operating system and its tasks. Consequently, there is a circular-dependency problem: because the call-graph data is statically linked into the program, it is needed to build the program, but the program binary must be available to generate the call-graph data. Our solution is to split compilation into two rounds. First, we generate dummy call-graph data that contains empty structures but provide the necessary definitions for building an ELF file. This ELF is then analyzed to extract the actual call-graph data, which is linked with the target program to produce the final ELF file.

Note that this approach requires that the call-graph data be located after the program it is linked with (i.e., the `.text` segment) to ensure the addresses are not modified by populating the call-graph data.

3 TrackOS Architecture

Before describing the CFI monitoring algorithm in the following section, we highlight here the aspects of integrating the CFI checker with the RTOS, including the definition of task control blocks, context switching, and finally, a scheduler addition we call *restartable tasks*. Our prototype of *TrackOS* is a derivative of FreeRTOS, an open source commercially-available RTOS written in C and available for major embedded architectures.[3]

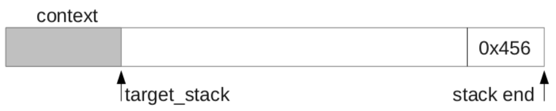


Fig. 2. Stack layout for a swapped out task. The saved context is on the top, `target_stack` points to the beginning of the saved control stack, and a fixed address, `0x456`, marks the bottom.

TrackOS *Task Control Blocks* *TrackOS* extends FreeRTOS's task control blocks with the following additional state:

1. *Stack location*: a pointer to the portion of a stack that comes after its saved context is added to the TCB. When a task has been swapped out by the scheduler, its control stack will first contain its saved context (i.e., its saved registers and a pointer to its task control block). The saved context is a fixed size. On the top of the stack is the task's saved context; on the bottom of the stack is a return address to the task's initialization function. A hypothetical task control stack is shown in Figure 2.
2. *Timing*: timing variables are used to track the timing behavior of the observed task to provide *TrackOS* with the duration the task has executed in its most recent time slice. This can be used, for example, to control when stack checking is run (e.g., it might be delayed until after initialization) or even to have time-dependent stack-checking properties (e.g., "after 500ms of execution, function `f()` should not appear on the stack").
3. *Restarting*: "restarting" variables allow the CFI task to be restarted as necessary; we explain the concept in Section 4.2. To do this, we save a code pointer to the CFI intialization code and its initial parameters as well as a pointer to a shared "restart mutex" with the observed task.

[3] `http://www.freertos.org/`

Context Switching In Figure 3 (top right), we show FreeRTOS's context switching routine (ported to the AVR architecture), together with the extensions necessary for *TrackOS*. This routine is used to swap the context of all tasks (including the monitoring task), whether they are checked or not by the monitor task, and it may be called from the timer interrupt during preemption or explicitly by a task during a cooperative yield (interrupts are disabled when `vPortYield()` is called). After saving a task's context, *TrackOS* updates its pointer to the top of the stack, after the saved context. Additionally, it saves the execution time of the saved task. After scheduling a new task in (Line 10), all that has to be done is record the execution start time for the newly-scheduled task.

m

4 Control-Flow Integrity

In this section, we overview the control-flow integrity algorithm implemented in *TrackOS*, which is the heart of the approach. We begin by describing the basic algorithm in Section 4.1, then we describe two extensions to basic real-time stack checking in Section 4.2.

4.1 Basic Algorithm

The CFI algorithm described below is the heart of *TrackOS*. There are two main procedures: first, we find the top return address in the stack, resulting from an interrupt or an explicit yield by the task. Second, once a valid return address is found, it serves as an "entry point" to the rest of the control stack. The second procedure walks the control stack, moving from stack frame to stack frame.

We describe each procedure in turn. Pseudo-code representations of the two procedures are in Figure 3. For readability, we elide details from the implementation, including hooks for performing restartable checks (see Section 4.2), helper functions (e.g., binary search), memory manipulations, type conversions, error codes, special-cases to deal with hardware idiosyncrasies, and other integrated stack checks for aberrant conditions. In addition, for the sake of readability, utility functions in pseudo-code listings that are underlined are described in the text without being defined.

In the following, we assume the maps generated by the *StackApprox* static analysis tool are available to the CFI checker. We do not assume that frame pointers are present, so the stack must be parsed by the CFI algorithm to distinguish data bytes from return addresses.

Yield Address Algorithm While a task is in the task queue waiting to be executed, its context is saved on its control stack. The CFI checker's entry point is just after the saved context, pointed to by the `target_stack` variable. (The `stack_t` type is the size of stack elements, which are one byte in our implementation.)

```
void check_stack(stack_t *target_stack) {
  current = target_stack;

  // Preemptive yield
  if(preemptive_yield_ret(current)) {
    current = preemptive_stack(current);
    stack_loop(current);
  }
  // Cooperative yield
  else if(coop_yield_ret(current)) {
    stack_loop(current);
  }
  // Cooperative yield from an ISR
  else if( search_ret_isrs (current) {
    current++;
    current = preemptive_stack(current);
    stack_loop(current);
  }
  else { error (); }
}

// Check a preemptive function
void preemptive_stack(stack_t *current) {
  current++;
  func = find_current_func(current);
  if(interrupt_in_main(func, current))
    done(SUCCESS);
  else
    return find_caller_ret (func, current);
}
```

```
void vPortYield( void ) {
        portSAVE_CONTEXT();

  #ifdef TRACKOS
    pxCurrentTCB->pxStoredStack =
      pxCurrentTCB->pxTopOfStack
        + portSP_TO_RET_ADDR;
    saveTime();
  #endif

        vTaskSwitchContext();

  #ifdef TRACKOS
    newStartTime();
  #endif

  portRESTORE_CONTEXT();
        asm volatile ( ''ret'' );
}
```

```
void stack_loop(stack_t *current) {
  while(!(inside_main(current)) {
    stack_t *valid_rets =
      lookup_valid_rets(current);
    if(NULL == valid_rets) { error(); }
    else {
      current =
        loop_find_next(current, valid_rets );
      if(NULL == current) { error(); }
    }
  }
  if(at_stack_end(current)) {
    done(SUCCESS);
  }
  else error ();
}
```

Fig. 3. Left: CFI procedure to discover the task's yield location. Top right: Context switch in *TrackOS*. Bottom right: *TrackOS* CFI procedure to walk the stack.

The entry point to the stack checker algorithm is `check_stack()`, shown in Figure 3, left. The invariant that holds after calling `check_stack()` is that either the check has been aborted due to an error, or the function returns a stack pointer to the first proper stack frame on the stack (pointing to the frame's return address). `check_stack()` is executed within a critical section, ensuring that the CFI checker, whenever it executes, always checks that the current location of the observed task's execution is valid.

There are three cases to consider at the entry point of the stack: a preemptive yield, a cooperative yield, and a cooperative yield from an interrupt service routine. These cases correspond to the three cases in the body of `check_stack()` in Figure 3, left.

Preemptive Yield In this case, the RTOS scheduler preempts the task via a timer interrupt. Inside the interrupt service routine (ISR), there is a call to a function that performs a preemptive context switch; if this is a preemptive yield, the top of the stack should contain the return address inside the ISR from that function. (The return address is found by *StackApprox* at compile time.) The function `preemptive_yield_ret()` performs this check.

In the case of a preemptive yield, we call `preemptive_stack()` (Line 22 in Figure 3, left). In that function, we first increment the stack pointer: the next value on the stack following the return address inside the timer ISR is the interrupt address for the task. The function `find_current_func()` takes an arbitrary address and searches through a map containing the address ranges of reachable functions generated by *StackApprox*. If a function that contains the interrupt address cannot be found, the procedure returns an error. Assuming a reachable function is found, `interrupt_in_main()` checks that the function is not the initialization function for the task. If it is the initialization function, then there are no further stack frames to check, since no function calls have occurred. (Additionally, the function checks that the distance to the bottom of the stack is less than the maximum number of data bytes the task's initialization function pushes onto the stack.) The CFI checker completes successfully (`done(SUCCESS)`.

If there are additional stack frames to check, from the interrupted function, the algorithm searches for the first return address on the stack. `find_caller_ret()` finds on the stack a return address for some caller of `func`. Using *StackApprox*'s *top map* (see Section 2), `find_caller_ret()` finds the set of return addresses associated with the call-sites for `func`; we determine the maximum stack usage for `func` that is also generated by *StackApprox*; call this value *max*. Then, `find_caller_ret()` searches for a return address appearing in the *top map* that is no more than *max* bytes from the current location in the stack, which are assumed to be data bytes. If a match is found, it is returned. At this point, we have found a return address on the stack belonging to the monitored task, and we are ready to enter the `stack_loop()` function in Figure 3, bottom right.

`find_caller_ret()` is a heuristic for finding a valid return address. It is possible for a data byte to have the same value as a valid return address. If by malicious behavior, then the attacker may be able to cause the CFI algorithm to trace data bytes as return addresses, but these data bytes would still have to conform to *StackApprox*'s static call-graph.

Cooperative Yield In a cooperative yield, the target task has yielded to the RTOS scheduler by directly making a `yield()` system call, which the function `coop_yield_ret()` expects to find on the top of the stack. We increment the stack pointer and call `stack_loop()`.

Cooperative Yield From an ISR This is a case in which the target task is preempted by an ISR, and then that ISR directly yields to the scheduler. We assume ISRs mask interrupts, so while an ISR should not be preempted, it can yield directly. Also, we assume an ISR only calls `yield()` just before returning, after all its local data has been popped from the stack. For each ISR that can preempt the target task, *StackApprox* generates a lookup table mapping ISRs to the return addresses for their calls to `yield()`. The function `search_ret_isrs()` searches for a match between the top of the stack and a return address from the ISR tables.

If a match is found, then after incrementing the stack pointer (Line 14), we can treat the stack the same as in the preemptive case in which we handle an interrupt to a task.

The Stack Loop Algorithm At the entry to `stack_loop()` in Figure 3, bottom right, `current` points to a known return address on the stack. `stack_loop()` "walks down" the stack in its main loop (Lines 1-10), from stack frame to stack frame.

The motivation for checking the stack in the reverse order of calls is to determine if the current location of the program is in an unexpected program location. Unexpected return addresses further down the stack represent latent vulnerabilities in which the program may return to an unallowed program location as it pops return addresses off of its stack.

The algorithm breaks out of the loop when it reaches a return address for the entry point to the task, relying on the convention that the task entry has exactly one caller, checked by `inside_main()`. Once outside the main loop, there is a final check by `at_stack_end()` that return address of the task's `main()` function is indeed the last return address on the stack and that there are exactly the number of data bytes between the bottom of the stack and the first call by `main()`.

Inside the loop, for each return address `ret` pointed to by `current`, the function `lookup_valid_rets()` looks up the set of return addresses of calls to the function `func` containing `ret` based on the *loop map* generated by *StackApprox*. If there are known callers of `func` found, then `loop_find_next()` searches the stack for another valid return address for a call to `func`. For each return address `ret'`, `loop_find_next()` depends on knowing the number of data bytes to be expected on the stack between `ret'` and `ret`, which is provided by *StackApprox*.

4.2 Extensions

Below we describe three extensions to the basic algorithm described above.

Restartable Monitoring The monitor task as it has been described is not reentrant. If it is swapped out by the RTOS scheduler while checking task A's stack, and task A then executes, its stack changes. When the monitor is swapped back in, the control stack it was previously checking is stale. Thus, we have designed the monitor task so that it is *restarted* when it is swapped in by the scheduler, meaning that its state is automatically reinitialized to its initial state when it is scheduled; in particular, the monitor restarts checking an observed task from the top of the stack.

The portion of the algorithm to determine a task's yield location and discover the first return address (Figure 3, left) on the stack is executed inside of a critical section in which interrupts are disabled. Thus, each time the CFI task is enabled by the RTOS, it is guaranteed to at least perform the initial checks on the stack. This initial check is small and the execution time is fairly constant, requiring just a few thousand clock cycles in our experiments. The motivation is to ensure that if the CFI monitor is scheduled, it is not prevented from checking that the current control location is valid. While the algorithm could allow this portion to also be interruptable, it provides the attacker with the opportunity to starve the monitor.

Blacklisting Sometimes, a code block might be reachable in a statically-generated call graph, but under nominal conditions, it should not appear on a tasks's control stack. For example, after startup, initialization code should not be executed. Similarly, error-handling code should not be executed under normal conditions. While this code cannot be eliminated from the program, it represents a security risk similar to *libc* insofar as it contains additional instructions for use in return-oriented attacks [18].

Consequently, we extend the CFI algorithm with a *blacklisting* capability. The user specifies at compile time a list of code blocks that can be called (usually these are function entry addresses), and *StackApprox* generates an array of all return addresses for callers of those blocks. The array is stored in non-volatile memory as well. Then during the execution of the CFI algorithm, for each return address found on the control stack, the algorithm makes an additional check to see whether the return address appears in the blacklist array. If it is found, then a blacklisting error is returned.

Timing Analysis Finally, a task's control block contains hooks to keep track of a task's total execution time. This allows the programmer greater flexibility to determine when checks should occur with respect to a task's total execution time or to state control-flow properties in terms of timing behavior.

4.3 Implementation

The implementation of the CFI algorithm requires around 150 lines of code (LOC) of extensions to the RTOS, together with the implementation of a CFI monitoring task. The CFI monitor task is a privileged task, with access to the state of other tasks (and in particular, their control stack memory). Its implementation is approximately 500LOC. The monitoring task can be assigned any schedule priority level; of course, this will affect the frequency of the CFI checks.

Compiled, our implementation of the CFI task requires approximately 2000 bytes of program memory. The call-graph is stored in a special section after the `.text` segment so that instruction addresses do not change when by linking call-graph data (i.e., by "pushing" program instruction addresses down), thereby rendering the analysis on the original program useless. The size of the call graph and TCB pointers are hard-coded into the task. CFI tasks are cheap; in our implementation, adding an additional CFI task adds only 28 additional bytes to the text segment of the resulting ELF file and requires only 200-250 bytes of stack space, as noted above.

Most importantly, this approach does not require any modifications to the CFI monitoring algorithm and we can simply use the RTOS scheduler to schedule the individual CFI monitors.

5 Program-Data Integrity

As TrackOS currently targets the Harvard architecture AVR processor, it gains some measure of program protection through the separation of program and data memory spaces; typically, the program memory is flashed once at programming time, and used as read-only memory during its execution. However, assuming a bootloader is installed, the bootloader can write to program memory during execution. Francillon *et al.* demonstrate how to install malware on a Harvard architecture by exploiting the bootloader to write malicious code into program memory during execution [9]. This sort of attack can be used to simply overwrite the CFI monitor or even the entire RTOS! Even more problematic is that for embedded RTOSes on small microcontrollers, there is no memory isolation between the tasks and the RTOS itself. So a malicious task can potentially modify OS code. Consequently, for increased security, control-flow checking should be augmented by a data attestation approach.

The problem of remote attestation of low-cost embedded devices is addressed in both *Secure Code Update By Attestation* (SCUBA) [19], and *SoftWare-based ATTestation* (SWATT) SWATT [20]. SCUBA strives to provide a safe execution environment for a firmware update, while SWATT attempts to establish the state of a remote system. For our implementation of the remote-verification checksum function, we have drawn from both SWATT and SCUBA. From SWATT, we have taken the idea of verifying the entire program, and from SCUBA we have taken the implementation of a high-performance checksum function.

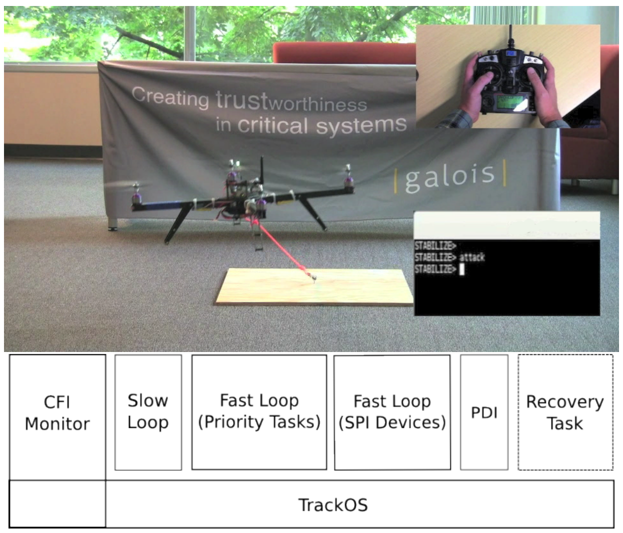
The checksum function itself is implemented as a simpler version of the ICE primitive from SCUBA [19] that omits the program counter and status register from the hash to simplify the implementation.

The advantage of software-based attestation (SBA) is that it requires no new hardware, which is particularly important in embedded systems with size, weight, and power constraints. SBA was thought to be impractical until the publications of SWATT and its successors, such as SCUBA. While there are shortcomings, e.g., [3], it is a comprehensive approach to ensuring data-integrity without requiring additional or modified hardware.

6 Experimental Results

In our work, the runtime overhead that is typically introduced by CFI is collected into a single RTOS task that can be scheduled at a user-defined priority, and system scheduability analysis is no different than if a new user task were introduced into the system. Because the system is general and highly-dependent on user configuration, general benchmarking is not particularly informative.

Still, to show the feasibility of our approach, we describe a case-study in which we use *TrackOS* to detect instrumented latent software vulnerabilities in ArduPilot, a popular open source autopilot [2]. ArduPilot is a full-featured autopilot that executes (at the time of the experiments) on an 8-bit ATMega2560 AVR microcontroller running at 16MHz with 256KB of flash, and 8KB of SRAM. A custom board has been designed for the autopilot that contains sensors including GPS, an accelerometer, gyroscope, and a barometer and sonar to determine altitude. The ArduPilot can be used with fixed-wing and multi-rotor aircraft. It provides stabilization, GPS-guided waypoint navigation, autoland, position loitering, and communication with a ground station over a Zigbee protocol-enabled radio transmitter.



```
STABILIZE>
STABILIZE> attack
STABILIZE>
```

Fig. 4. Top: Attack launch configuration. Bottom: Ported ArduPilot architecture on *TrackOS*.

Architecture The ArduPilot code base is just under 10K LOC of C/C++, not including standard libraries. The ArduPilot runs “bare” on the AVR hardware. Consequently, we ported it to run as a set of tasks on *TrackOS*. Its architecture is shown in Figure 4. The infrastructure is decomposed into the following tasks:

- The CFI checker, integrated with *TrackOS*.
- A “slow loop” that reads GPS data, updates navigation information, updates altitude and throttle data.
- A “fast loop” that reads the pilot’s radio controller, updates attitude, and writes to the servos.

– Another “fast loop” reading SPI-bus devices, the gyroscope and barometer.
– A program-data integrity task that responds SWATT challenges sent to it over a SPI bus interface.
– Finally, a recovery task. The recovery task only implements throttle control from the radio controller and is enabled if *TrackOS* detects malicious behavior. This task is not enabled until an attack is detected, at which point the slow loop task is disabled (the micro-controller does not have enough memory to support both tasks simultaneously). Thus, the recovery task is shown in the figure as a dashed component.

The CFI checker runs at priority 2, the slow loop runs at priority 1, the fast loops and recovery tasks run at priority 3, the highest priority, and the program-data integrity task runs at priority 1.

The SWATT server is implemented on an ARM Cortex M3, clocked at 60MHz. The server has 4MB of external flash memory, with a 50MB/second interface to the memory over a SPI bus.

We manually annotate just under 30 indirect jumps, including the interrupt vector table.

In our experiments, we implemented two kinds of attacks. First, we implement a buffer overflow vulnerability in which an array is allocated on the stack allowing an attacker to overwrite bytes out-of-bounds. Overwriting a return address, the attack jumps to a function that is unreachable without modifying the control flow. Second, we implement a blacklist attack in which we instrument the code with a function that is supposed to be unreachable during stack checking (e.g., the function could be part of a start-up or an error-handling routine).

The experiment setup is shown in Figure 4. The attacks are launched from a ground station (i.e., a laptop) communicating with the autopilot over the MAVLink protocol[4]. When an attack is detected on board, the recovery task begins, which ignores all radio signals except to reduce thrust to land the craft.

Even though we have scheduled the CFI checker to run at a *lower* priority than the fast loop, it detects the buffer overflow and blacklisting attacks we instrument. (Indeed, on the Atmega2560, the fast loop must be the highest priority to be schedulable.) Our work shows that CFI checking can happen intermittently and still discover control flow vulnerabilities. To evade detection, an attacker must either (1) exploit a vulnerability, perform an attack, and cleanup before the scheduler swaps the task out; or (2) starve the CFI monitor task indefinitely.

7 Related Work

Research in run-time control-flow protections for embedded software is nascent. In particular, there are few approaches that take into consideration the real-time and memory constraints present in embedded control systems. In the following,

[4] `http://qgroundcontrol.org/mavlink/start`

we focus specifically on the dynamic monitoring approaches. We omit related research in static analysis and software-based attestation; while *TrackOS* depends on them, we did not make novel research contributions there.

As noted in the introduction, work by Abadi *et al.* [1] addresses many of the shortcomings with earlier protection approaches. An approach that is similar in spirit to our is work by Petroni and Hicks for monitoring control-flow attacks to detect Linux kernel rootkits [15]. Their work is inspired by CFI checks as describe by Abadi *et al.* but addresses environments in which some of the assumptions made by Abadi *et al.* do not hold. Petroni and Hicks also periodically monitor the OS to reduce the timing overhead; in their case monitoring is done from a separate virtual machine hosted by a hypervisor. They focus specifically on rootkit attacks; empirically, many Linux rootkits work by modifying function pointers found in the heap. Therefore, they do not check stack-based software attacks like we do. Furthermore, their work is not focused on real-time systems. Hofmann *et al.* take a similar approach to Petroni and Hicks, also looking to detect kernel rootkits [12] in Linux. Hofmann *et al.* do consider stack-based attacks by checking return addresses on the stack for property violations (i.e., that they point to valid kernel code regions). Their approach is not suitable for checking general return-to-libc attacks, like ours is. Note though that generating a call graph for something as complex as the Linux kernel is much more difficult than traditional embedded code given the prevalent use of heap-based function pointers, dynamic linking, and sheer complexity.

Two works combine CFI and data integrity checks, like ours. These include de Clercq *et al.* [7] and Zeng *et al.* [23], using hardware support and sandboxing, respectively.

With respect to CFI in embedded systems, Francillon *et al.* propose hardware extensions that support a distinguished control stack and data stack [10], and corresponding instruction-based memory access control. They implement a prototype hardware simulator. While hardware support like they envision simplifies the control-flow security problem, our approach works with conventional, unmodified hardware. Reeves *et al.* present Autoscopy, an in-kernel tool for detecting CFI violations targeted at SCADA systems [16]. Autoscopy has a five percent overhead. Their approach differs from ours insofar as we do not assume reliance on an advanced operating system mechanism, which is not available in small embedded RTOSes. Furthermore, Autoscopy learns a call graph by executing the system during a learning phase, an approach that can lead to false positives if any control paths are missed. Like the rootkit-specific approaches already described, Autoscopy focuses specifically on function-pointer hijacking rather than arbitrary CFI violations.

8 Conclusions and Future Work

We have described *TrackOS*, a unique implementation of CFI monitoring targeted at real-time embedded systems. Our research demonstrates the feasibility of CFI monitoring for low-level systems, relying on the operating system to handle scheduling, making the approach suitable even in hard real-time systems. Many research opportunities remain in the area of CFI checking for embedded systems; below, we describe research questions specifically left open in our work.

We have not addressed the resteering problem. One framework for resteering is the *Simplex architecture*, originally designed to increase the reliability of complex control systems by providing a safe and simple fallback controller [21]. Mohan *et al.* show how to adapt the Simplex architecture for control-flow attacks. The idea is to monitor with high fidelity the execution time of a control system, with the idea that deviations from the expected execution time are the result of malicious behavior [13]. The approach relies on having accurate timing bounds on normal execution. Our approach does not require timing analysis of the monitored task.

Our use of data attestation is partly because we there is no memory isolation between the RTOS and the tasks executing on it. On a microcontroller and kernel supporing virtual memory, this is less problematic.

With a control-flow graph and timing information available to a dynamic monitor, high-level properties can be checked at run-time. For example, temporal logic analyses might be written about control flow, which can be useful for both testing as well as run-time protections of the system. For example, we might query that an authentication routine always follows updated waypoints being read from a ground station over the radio. One of the authors discusses other potential temporal logic properties in related work [8].

Acknowledgments

This work is supported in part by Air Force contract FA8650-11-C-1003. All findings herein are the authors' alone. Pat Hickey performed the work while at Galois, Inc.

References

1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13(1):1–40, 2009.
2. Source code available at `http://code.google.com/p/ardupilot-mega/`. Retrieved December, 2012.
3. Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Computer and communications security (CCS)*, pages 400–409. ACM, 2009.
4. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Computer and communications security (CCS)*, pages 559–572. ACM, 2010.
5. Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security*, 2011.

6. Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*. USENIX Association, 1998.
7. Ruan de Clercq, Ronald De Keulenaer, Bart Coppens, Bohan Yang, Pieter Maene, Koen de Bosschere, Bart Preneel, Bjorn de Sutter, and Ingrid Verbauwhede. SOFIA: Software and control flow integrity architecture. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, 2016.
8. Iavor Diatchki, Lee Pike, and Levent Erkök. Practical considerations in control-flow integrity monitoring. In *Proceedings of the The Second International Workshop on Security Testing (SECTEST'2011)*. IEEE, March 2011.
9. Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *Computer and Communications Security (CCS)*, pages 15–26. ACM, 2008.
10. Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, SecuCode '09, pages 19–26. ACM, 2009.
11. Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conff on USENIX Security Symposium*, 2001.
12. Owen Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. In *Architectural support for programming languages and operating systems (ASPLOS)*. ACM, 2011.
13. Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. S3A: Secure system simplex architecture for enhanced security of cyber-physical systems. *CoRR*, 2012.
14. Adrian Perrig and Leendert van Doorn. Refutation of "on the difficulty of software-based attestation of embedded devices", 2010. Unpublished. Available at `https://sparrow.ece.cmu.edu/group/publications.html`.
15. Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 103–115. ACM, 2007.
16. Jason Reeves, Ashwin Ramaswamy, Michael E. Locasto, Sergey Bratus, and Sean W. Smith. Lightweight intrusion detection for resource-constrained embedded control systems. In *Critical Infrastructure Protection V - 5th IFIP WG 11.10*, pages 31–46. Springer, 2011.
17. John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computer Systems*, 4(4):751–778, November 2005.
18. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.
19. Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security (WiSe 2006)*, September 2006.
20. Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.
21. Lui Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

22. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307. ACM, 2004.
23. Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011.