T-SYS: Timed-Based System Security for Real-Time Kernels

Brayden McDonald
bwmcdon2@ncsu.edu
North Carolina State University
Raleigh, USA

Frank Mueller
mueller@cs.ncsu.edu
North Carolina State University
Raleigh, USA

ABSTRACT

The increasing proliferation of cyber-physical systems in a multitude of applications presents a pressing need for effective methods of securing such devices. Many such systems are subject to tight timing constraints, which are poorly suited to traditional security methods due to the large runtime overhead and execution time variation introduced. However, the regular (and well documented) timing specifications of real-time systems open up new avenues with which such systems can be secured.

This paper contributes T-SYS, a timed-system method of detecting intrusions into real-time systems via timing anomalies. A prototype implementation of T-SYS is integrated into a commercial real-time operating system (RTOS) in order to demonstrate its feasibility. Further, a compiler-based tool is developed to realize a T-SYS implementation with elastic timing bounds. This tool supports integration of T-SYS protection into applications as well as the RTOS the kernel itself. Results on an ARM hardware platform with benchmark tasks including those drawn from an open-source UAV code base compare T-SYS with another method of timing-based intrusion detection and assess its effectiveness in terms of detecting attacks as they intrude a system.

KEYWORDS

Real-time systems, security, worst-case execution time

1 INTRODUCTION

Simple networked and embedded devices have become increasingly common throughout the wide range of applications as processors with the necessary capabilities have become cheaper and more plentiful. Increasingly, such systems are incorporated into critical infrastructure (ranging from a single traffic light to a municipal power grid) and autonomous vehicles, i.e., systems subject to hard real-time constraints. Failing to control such systems can result in loss of life or severe environmental damage. Meanwhile, cyber attacks have become widespread and are penetrating embedded systems as they are increasingly networked. Hence, it is becoming crucial to protect such cyber-physical systems (CPS) from attacks [7]. However, securing embedded, time-constrained systems presents a number of unique challenges beyond those of securing commodity compute systems [11]. Ordinary methods of protection, particularly kernel-level protection, are insufficient by themselves in embedded and real-time systems, since they focus on system functionality and tend to add significant execution overhead, yet lack the ability to ensure that a system operates within its timing constraints. In addition, some proposed protection methods are dependent on hypothetical specialized hardware [10], or require significant developer effort to configure protection based on known threats and system performance requirements. To fill this gap, methods need to be developed for implementing kernel-level protection into the RTOS, as well as allow for easy configuration based on elastic timing bounds.

Real-time systems require accurate timing information and predictable behavior with regards to execution time. This predictability can be leveraged to detect attacks by identifying timing irregularities. Such irregularities are indicative of system malfunction due to a cyber attack or excessive execution beyond specified WCET bounds of a task or code region. We assume the former hereafter.

This work contributes T-SYS, a monitoring method for intrusion detection that relies on inserting time checks (instrumentation points) along code paths with known WCET bounds. A compiler-based tool to allow the automatic integration T-SYS protection based on a user-defined Maximum Vulnerability Threshold is developed as well. This allows T-SYS to be configured, at compile time, according to expected threats, security requirements, or system performance. When T-SYS is implemented in both kernel- and user-level code, it is capable of providing end-to-end protection across the entire execution path. Its instrumentation complements other conventional security techniques by integrating WCET monitoring points along execution paths into code. Any intrusions resulting in execution time exceeding the WCET budget between two instrumentation points will be detected, which limits the code of such injections in length to a so-called “window of vulnerability” — correlated to the longest WCET path between instrumentation points. The Maximum Vulnerability Threshold defines the upper bound of this “window of vulnerability”, which will be tolerated by the compiler-based integration tool, and is determined by the user.

A number of WCET-based protection methods have been proposed [10, 34, 35]. We compare T-SYS to Bellec et al. [10], as they developed an algorithm to identify regions, for each of which timing is tracked in order to identify intrusion by detecting anomalies. However, the criteria used to divide code into regions, as well as the requirements for region structure, are vastly different. Where the Bellec algorithm utilizes single-entry/single exit nested regions, T-SYS allows for multi-exit regions; Bellec creates a hierarchy of nested regions requiring stack maintenance of timed context data while T-SYS neither requires regions nor a region stack. What’s more, T-SYS supports *elastic* timing requirements determined prior to compile time, facilitated by our ROSE compiler tool for placing instrumentation points. This elasticity allows the user to choose a desired level of protection based on application requirements instead of Bellec’s rigid one-sized regions determined by control-flow shape. We also develop transformations to loop structures to further reduce overhead.

Bellec relies on a hardware monitor to track the cycle count within a program, and thereby detect timing anomalies with zero performance overhead. With a similar hardware design, T-SYS’ overhead could also be reduced to zero. A further description of the requirements for T-SYS hardware is given in Section 4. However, as

such hardware does not exist in practice, experiments in this paper were conducted using software implementations of both T-SYS' and Bellec's methods assessing overheads for instrumentation points. This also provides an indication of cost for a realistic deployment of such intrusion protection.

The primary contributions of this work are:

• T-SYS is developed, a novel method for timing-based *protection across user and kernel boundaries*.

• A compiler-based tool for automatic integration of elastic T-SYS instrumentation points is algorithmically developed.

• A prototype implementation of T-SYS is realized in an existing Autosar/OSEK-compliant RTOS, as well as in a variety of existing real-world CPS benchmark task sets.

• Experiments comparing T-SYS to existing WCET-based security methods are conducted, comparing their ability to detect malware attacks, as well as their performance impact compared to the unmodified kernel and previous timing-based security method. They show clear benefits of T-SYS over prior work in terms of lower overhead, user-configurability and elasticity.

2 RELATED WORK

As cyber-physical systems have become increasingly important to 24/7 operations of critical infrastructure, so has the importance of protecting them against cyberattacks [13, 23]. In response to the increasing prevalence of and need for security in real time systems, a number of new ideas have emerged to meet the unique challenges of real-time security. This section aims to provide an overview of existing contributions to security in the domain of real-time systems, with particular focus on methods that incorporate timing information as part of their protection. The purpose of this overview is to determine what options currently exist for protecting real-time systems, and what problems in the field T-SYS is best suited for.

Prior work on intrusion detection in real-time systems has taken a variety of approaches [12]. Many of these methods are focused on increasing security at the network level, as the increasing use of networks in real-time systems presents an expanding attack surface [28]. While conventional and embedded network protection methods complement T-SYS, the most closely related methods are based on the principle of intrusion detection via timing anomalies. These methods leverage the unique timing constraints inherent to real-time systems as a means to identify attacks. Designing a real-time system inherently entails gathering timing information on the various components that comprise it [4]. Since sufficiently complex attacks are liable to generate timing anomalies, some protection methods incorporate this information into their intrusion detection strategies by identifying timing anomalies [31]. It is this category that T-SYS falls into.

Bellec et al. [10] created a protection method that employs a region-based approach, tracking the time spent executing regions. Their method employs specialized hardware to monitor execution. The regions used by Bellec are single-entry, single-exit nested regions. The hardware tracks execution through these regions by monitoring the a cycle count-down register initialized upon region entry and tracks nested regions via a stack structure with associated timer save/restore operations. They also provide an algorithm for automatically dividing target code into regions based on the control-flow graph of the code, which we compare T-SYS to in Section 7. T-SYS differs in its criteria for region selection (non-nested, single-entry/multi-exit), as well as in providing elasticity in its timing bounds through the MaxVuln parameter to determine the largest allowable region size.

Zimmer et al. [34] developed a set of methods for providing security in an RTOS exploiting precise timing information to detect attacks. T-Rex is a checkpoint based system that relies on fine-grained timing information (single clock cycle resolution) to detect buffer overflow attacks on function return and other straight-line execution paths of application code. T-ProT is a coarser-grained protection using synchronous checkpoints to validate for each task that a milestone in execution in reached by some expected time. T-AxT is integrated with the scheduler and supports asynchronous, periodic checks of a task's program counter value, to ensure that it is within the appropriate range.

Of these, T-SYS is most outwardly similar to T-ProT in that both use timers to bound a block of code. However, T-ProT implements its timer checkpoints via scheduler invocations, while T-SYS uses function calls to instrument code. This allows T-SYS to provide integrated protection *within both application and kernel code and across their intersection* instead of just application code, which creates novel challenges in that the control flow of a protected region may originate in the context of one task but lead to that of another task. T-SYS also supports *elastically sized vulnerability windows* as opposed to more the rigid constant sized regions of T-ProT.

Traditionally, the effect of kernel paths in real time systems has been estimated fairly pessimistically [22], taking the WCET of the syscall to be that of the longest path the call could possibly take through the kernel. Prior work [15] has modeled RTOS kernel paths using control-flow graphs (CFGs). These CFG models were then integrated with the existing CFG of the userspace programs (crossing the kernel-application boundary) to create a more complete CFG of the user task. By including kernel paths, previously-independent CFGs of different tasks could be connected, thereby creating a whole-system CFG.

Methods of WCET analysis have been developed to tighten bounds by incorporating system state information preceding system calls [16]. Information about system states is combined with prior analysis of individual kernel paths' WCETs as well as the conditions for taking these paths. In combination, such information yields tighter bounds on the response time of system calls and, transitively, application tasks.

3 ATTACK MODEL AND SCENARIO

There are a multitude of ways in which real-time systems can come under attack. Much of the research in real-time security focuses on identifying attacks at the network level [19, 21]. In this work, a general model is presented for both the attack that T-SYS is designed to detect, together with a model for the system itself, with a focus on defining how the kernel handles interrupts and what hardware features are made available.

We assume the existence of a high-precision monotonic counter provided by the hardware and available to be programmed by the kernel. This counter is write-protected and can only be modified via a kernel call preventing an attacker from being able to modify it without returning to the kernel.

We further assume that the attacker has managed to compromise the user data space. The attacker's goal is to hijack the control flow of the system in order to execute malicious code under elevated permissions, and then return undetected. We assume that the attacker cannot modify hardware factors or protected kernel memory. An attacker under these limitations may still be able to divert kernel execution, e.g., by triggering a buffer overflow within kernel code.

4 DESIGN

The primary objective of T-SYS is to detect intrusions, thereby allowing the system to respond rapidly to such an intrusion, e.g., by switching into a safe mode or shutting down a node entirely — but the exact response to the attack is outside the scope of this paper. Our approach to intrusion detection relies on tracking the execution time of code regions during runtime, and detecting when a region's execution time has exceeded its statically-determined WCET budget. Code regions are bounded by instrumentation points (IPs). The use of *worst case* execution time to construct these bounds (over less pessimistic estimates) is paramount, as by definition a code region's execution time will never exceed its WCET. Thus, we can assume that a region exceeding its WCET bound indicates the presence of an attack. The algorithms for generating regions (and thereby placing IPs) from a control-flow graph annotated with WCET information and with an elastic timing bound are discussed in Section 5.

In a software-based implementation, IPs are implemented as system calls when inserted into application-level code, and as simple function calls when added inside kernel paths. This provides the necessary level of data protection to the IP code by ensuring that important data (e.g., timing bounds or IP return addresses) reside in a different address space than application code, reducing an attacker's ability to tamper with this information. In a hardware implementation, a dedicated component tracks the program counter and executes all the functions of the IP (setting up timer, raising alarm) once the PC reaches an IP without extra code added to the application or kernel path. In this paper, we focus on a software implementation of T-SYS since it is applicable to today's hardware as implemented in our experimental evaluation.

4.1 Protection Model

T-SYS identifies timing anomalies along execution paths. Execution paths are represented as regions of contiguous basic blocks within the system's control flow graph, having a single entry and one or more exit points (in contrast to more constrained single-exit control flow [10], which does not match C/C++ control flow with `break` within loops). As every basic block within the CFG is associated with exactly one region, successors of an exit point of one region represent entry points for a subsequent regions. Execution time is tracked via IPs placed at region boundaries. Because regions are pairwise disjoint with an empty intersection in basic blocks (in contrast to nested regions [10]), each IP is associated with exactly one region. Figure 1 shows a sample CFG with 4 IPs and color-coded regions associated with each one.

At each IP, a timer with a deadline equal to the longest path through the associated region region (i.e., the longest time before reaching another IP) is set up. IPs are placed at the beginning of the first block in a region. Notice that program profiling/tracing [6, 8, 9, 20] places instrumentation in a basic block anywhere within a path, and often not at the top, which is one of several differences between T-SYS and profiling/tracing). Concrete rules for dividing a CFG into regions are discussed in Section 5.

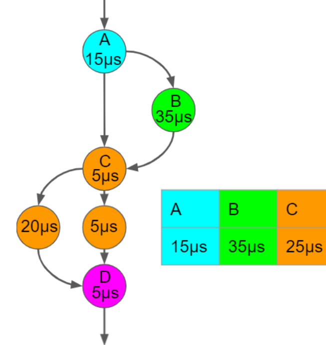

Figure 1: CFG with WCETs per block, regions denoted by color, with instrumented blocks labeled by letter, where pathWCET table contains timeout deadlines for each region.

Figure 1 shows a CFG with IPs *A*, *B*, *C* and *D*, along with WCETs per basic block. The table to the right shows the WCET bound for each IP. On encountering point *A*, the next IP reached could be either *B* or *C*, where it reaches in either case after 15ms (the length of *A*'s basic block). Both blocks directly following *A* contain IPs. As IPs are always placed at the start of a block, the length of the containing block is included. For *B*, the same case is seen with a WCET of 35ms. At point *C*, however, there are two paths to *D*, with WCETs of 10ms and 25ms, respectively. The longest of the possible paths defines the IP deadline, so the timer at *C* is set to 25ms.

Consider the effect of executing injected code of an attack that *diverts* from the expected control flow. Upon reaching an IP, a call is made to set up a timer, with a deadline equal to the WCET distance to the next IP. When the control flow is diverted off the path to the next IP, execution continues until the timer deadline is reached. When this happens, an interrupt is triggered, flagging an intrusion. With no diversion, the next IP would be reached before the deadline, and the timer would be reprogrammed with a new deadline. Also consider an attack using a suspend-and-resume strategy, where the attack is split up into multiple parts, suspending its own execution and returning to the diverted region to avoid allowing the total region execution time to exceed the timer deadline. In this case, every fragment of the attack would need to fit within the vulnerability window for the region. Given that the size of this window changes each time the region is executed (due to caching or control flow differences), the attacker would need to guarantee that they are always diverting back to the region with enough time left to finish execution within the deadline.

4.2 Interrupt Handling

Consider a user-level task executing in a system involving multiple tasks of varying priority in a preemptively scheduled system. The execution of application code may be interrupted and then temporarily suspended while execution is transferred to a higher-priority task. In general, the exact time and the location in the application where the interrupt occurs cannot be statically determined as preemptions may be asynchronously triggered.

To account for asynchronous actions, it is necessary for the operating system's interrupt handler to interface with the T-SYS timer when interrupting a T-SYS protected task. When an interrupt arrives during the execution of a T-SYS protected application, the remaining time left for the current timer is recorded in kernel (i.e., protected) memory. The timer is then canceled before the rest of the interrupt is processed.

Similarly, when returning execution to a protected task, the interrupt handler must reinstate the T-SYS timer with the recorded remaining time plus some constant to account for the overhead associated with handling the interrupt before the timer is paused, as well as for returning from the interrupt after resetting the timer. However, this overhead is constant and can therefore be directly credited to the T-SYS timer within the timer resume operation. If T-SYS is integrated into the kernel, the interrupt handler may contain an IP that sets up a new timer to protect a kernel region, e.g., to handle the interrupt or to call the dispatcher. This case is discussed further in Section 6.

4.3 Instrumentation Points

At each IP, the return address is read from the call stack and checked for validity against a table of known valid return addresses. If invalid, an attack is flagged by raising an alarm. Otherwise, the return address is used to extract the IP's unique ID by indexing into the `pathWCET` table, which contains the relevant region WCETs (and thus relative timer deadlines) for each IP. The `pathWCET` table is stored in protected kernel memory, and its contents are hard-coded at compile time by the ROSE-based implementation tool. A timer is then set up for this deadline. This timer setup operation also cancels the timer for the previous IP encountered. Pseudocode for implementation points is shown in Listing 1.

Listing 1: Instrumentation Point Pseudocode

```
void inst_point():
  ret_addr = get_return_address()
  if !is_valid_addr(ret_addr){
    alarm()
  }
  point_id = get_pid(ret_addr)
  current_time = get_timestamp()
  deadline = current_time + pathWCET[point_id]
  setup_timer(deadline)
```

IPs are represented in application code as system calls, and in kernel code by singular function calls. The return address checking prevents attackers from evading detection by inserting their own IPs into malicious code. Since the return address is unique to each IP, it can be extracted at compile time.

5 PLACEMENT OF IPS

A tool to support automatically implementing protection into arbitrary code, both user and kernel, is provided. To this end, the ROSE [27] compiler framework was utilized to create an instrumentation tool from a specification that incorporates previously acquired timing information, control-flow analysis and a vulnerability threshold, $MaxVuln$. This tool automatically divides the control flow graph of a given code base into regions based on the user-specified maximum vulnerability threshold, $MaxVuln$, and places IPs in desired locations throughout. Use of user-specified $MaxVuln$ parameter supports elasticity with respect to instrumentation granularity. Furthermore, this tool is capable of performing loop transformations to reduce the overhead of instrumentation.

A prerequisite for utilizing our instrumentation tool is that the developer has extracted worst-case execution time information for each basic block in the system. The difficulty of this process is largely dependent on which method is used to acquire basic block WCETs. Extraction of timing information was performed experimentally for this work, but other implementations of T-SYS may use any method available, including static WCET analysis tools [32]. T-SYS is agnostic to how basic block WCETs are extracted and will work with any method, so there is no need to specify a precise method for determining the WCET of a basic block.

Our tool provides elastic instrumentation, which takes the granularity of instrumentation as an input in terms of cycles to denote the vulnerability threshold. This allows the user to *directly* specify the minimum frequency of IPs rather than deriving this value indirectly from other user parameters, as is the case in other methods [10, 34].

The tool also supports basic block instrumentation (by simply treating every block as a separate region), which we used as a first-order approximation of WCET bounds, later refined in a second-order pass over regions with multiple blocks. This step achieves much tighter bounds on the WCET of each region.

5.1 Placement Algorithm

To place IPs, all basic blocks within a CFG are assigned into contiguous regions. Each region represents a section of code over which a given timer will be active. From here on, we refer to partitioning the CFG into regions as *coloring* it; blocks of the same region share a color, which is unique to that region. Regions created must follow a particular set of rules governing their structure to support instrumentation placement for timing protection:

- A basic block must share its color with either all of its children, or with none of them.
- A basic block must share its color with either all of its parents, or with none of them.
- A region may have only one entrance block.
- The WCET of the longest path through a region must not exceed the $MaxVuln$ threshold.
- $MaxVuln$ must be greater than or equal to the WCET of the longest basic block. (If finer granularity was needed, one could even dissect a block into multiple blocks.)

By these rules, a single basic block may constitute a region.

Once the CFG is partitioned into regions, an instrumentation point is placed at the beginning of the first basic block per region. Such a block exists because, as per the region structure requirements defined above, each region will have a single entry point. Placement is always performed at the top of a basic block for two reasons:

1) Placement in the middle of a basic block would divide the execution time between two regions, but given that timing information is stored at the granularity of single basic blocks, it would be unclear how this time should be divided up. 2) Placement at the end of a basic block would be complicated due to branch instructions whose time needs to be accounted for, yet the IP cannot be placed after them since they affect the program counter.

In order to properly handle loops within a given CFG, a preprocessing step is necessary during which each loop is represented as a single (compound) block. In the event that the loop's total WCET is larger than $MaxVuln$, the compound block will initially be treated as having an indefinite WCET. (We use this property in our algorithm to force it being treated as its own region when first creating regions.) Once the remainder of the CFG has been divided into separate regions, the loop's CFG will then be passed into our algorithm as a single compound object (without further internal analysis).

Loop bounds are expected to be statically bounded, either explicitly by a constant bound that can be statically evaluated at compile time or by user hints/pragmas to provide such a constant. For such constant number of iterations of a loop and a total WCET not exceeding $MaxVuln$, the compound block will be treated as having the same WCET as the loop it represents. In the event that the loop bounds are not available (and thus cannot be evaluated at compile time), the algorithm will assume the loop's total WCET is larger than $MaxVuln$ and thus follow the behavior for long loops outlined in the preceding paragraph. In addition, the loop structure may be transformed into a semantically equivalent one to ensure low instrumentation overhead, which is discussed in subsection 5.5.

Once this preprocessing step is complete, the CFG is partitioned according to the 3-step algorithm outlined below:

(1) Regions are created delimited by dominator and post-dominator blocks, which are uniquely colored with respect to other regions.[1]
(2) All interior blocks of a region beyond the delimiter blocks are colored with their region color.
(3) Regions are combined within the $MaxVuln$ threshold and region property requirements to reduce the total number of regions and thus instrumentation overhead.

Pseudocode of the placement algorithm is given in the appendix, along with a proof sketch for correctness and a complexity analysis.

Figure 2 depicts the coloration of a control flow graph after each step in the point placement process. The CFG displayed was taken from the ext_tsk kernel path, a portion of the scheduler within the Autosar/OSEK-compliant Toppers RTOS [33], which was instrumented as part of the evaluation in Section 7.

5.2 Partial Regions

In the $PartialRegions$ step, only some of the blocks in the CFG are colored in; others are left uncolored, with no region membership. The objective of this step is to generate single-entry, single exit regions within the CFG.

A depth-first traversal of the CFG is performed. At each uncolored block S, a list of the node's post-dominators, S_{post}, is acquired. Any block in S_{post} that does not have S as a pre-dominator is removed from S_{post}. The resulting pruned list is then sorted by distance from S (where distance represents WCET), with the furthest entry first (the remaining blocks, if any, can be ordered this way [26], as the furthest block will also be a post-dominator for

[1]A dominator block in a CFG indicates a prior block execution must have passed through to reach the current block, whereas a post-dominator indicates a block execution will have to pass though after leaving this block, i.e., these blocks denote must-information [5].

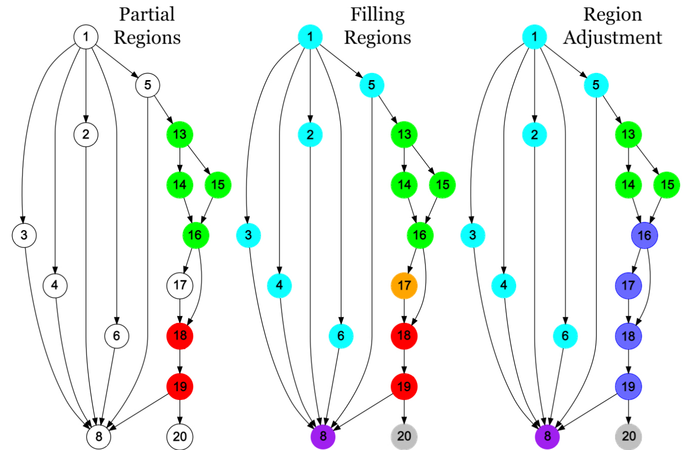

Figure 2: CFG after each step in coloring regions of the CFG for the Toppers scheduler

all earlier blocks in the list). For each remaining block in S_{post}, E, a depth-first algorithm is used to determine the longest path (in terms of worst case execution time) between S and E. If the computed region WCET is less than the $MaxVuln$ parameter, then all of the blocks between S and E are assigned a single color, and the depth-first traversal of the CFG continues from E.

In the event that none of the blocks in S_{post} pass the criteria above (i.e., S is not a dominator for any of its post-dominators, or no post-dominator is found with a longest path of less than $MaxVuln$) or S has no post-dominator, no blocks will be colored and the depth-first traversal of the CFG continues. This process is complete once every node in the CFG has been checked. As seen in the first graph of Figure 2, only some blocks are colored after step 1 (here, 40% of the blocks are colored. Uncolored nodes are shown as white, with a black background). Listing 2 in the appendix shows the pseudocode for the Partial Regions step.

5.3 Filling Regions

The Filling Regions step colors all remaining blocks that were left uncolored by the previous Partial Regions step. This eventually results in a fully cornered CFG. The methods begins with a depth-first traversal of the CFG. When an uncolored block is encountered, it is colored. After coloring a block, an attempt is made to grow the new region by painting all of its successors with the same color. This attempt can only succeed if, for every successor C,

- C is uncolored,
- C has no parents of a different color than S (including uncolored blocks), and
- adding C to the region will not create a path through the region that exceeds $MaxVuln$.

These rules ensure that any region created in this manner will (1) not interfere with the regions created in the previous step, and (2) will obey the rules for region structure defined previously. If the growth attempt succeeds, all successors obtain the predecessor's color, and growth attempts start for each newly-colored block in a breadth-first fashion. If the growth attempt fails, then the algorithm resumes looking for uncolored blocks to start new regions.

The second graph in Figure 2 depicts the state of the CFG after the Filling Regions step is complete. Note that every block in the graph has been colored at this point. The previously described process corresponds to the pseudocode in Listing 3 within the appendix.

5.4 Region Adjustment

The final phase of loop adjustment, Region Adjustment, optimizes the graph to reduce the number of IPs placed. This reduces the required size of the *pathWCET* table, which also reduces performance in a software implementation of T-SYS (where IPs have associated execution time overhead).

Region Adjustment uses the same dominator/post-dominator pair method from the Partial Regions step to identify potential regions. However, only region exit blocks that do not share a color with any sibling blocks (i.e., successors of the block's predecessor) are checked as possible new entry points. If a viable region is identified, then a check is performed to determine if creating the new region will reduce the total number of regions within the CFG. If so, then all blocks within the region are repainted a new color making them part of the new region. The check for reduction simply involves counting the number of unique colors identified among the prospective region's blocks. If it is more than 3, or if the new region contains a superset of the blocks in at least 2 regions (as is the case in Figure 2), then the check succeeds and the region is created. Pseudocode for this is shown in Listing 4 in the appendix.

We refer to this algorithmic approach of delimiting *maxVuln* as elastically sizing regions: Our automated process allows users to call the instrumentation tool with their preferred *MaxVuln* threshold, which could even differ from task to task depending on a task's real-time criticality.

5.5 Loop Transformation by Thresholding

The process of instrumenting loops opens up an interesting problem with regard to the cost of IPs. Specifically, how can a loop be efficiently instrumented when multiple loop iterations can pass within the *MaxVuln* time limit, but the total number of iterations makes the loop exceed *MaxVuln*? When a single loop iteration can be longer than *MaxVuln*, the loop's internal structure can be instrumented using the 3-step method from above. But the 3-step method does not allow IPs to occur on every k-th loop iteration due to the region constraints. Instead, each loop iteration would trigger an IP, which increases T-SYS' overhead.

Our solution to this problem is to transform the loop into a nested loop with a single IP on top of the outer loop and all of the logic of the untransformed loop placed in an inner loop. We bound the number of inner loop iterations such that it will not exceed *MaxVuln*. We limit loop transformations to loops with statically known iteration bounds so that the transformation can be performed at compile time.

An example of this transformation's effect of the loop CFG is depicted in Figure 3. The blue segment represents the original loop. The outer loop (orange segment) contains an additional IP (highlighted in yellow) and a conditional branch (enclosed in green) that determines the number of inner loop iterations to execute on a given iteration of the outer loop. The dynamic number of instructions increases slightly due to upper bounds calculations for the inner loop, but this overhead is easily compensated by the lower number of IPs.

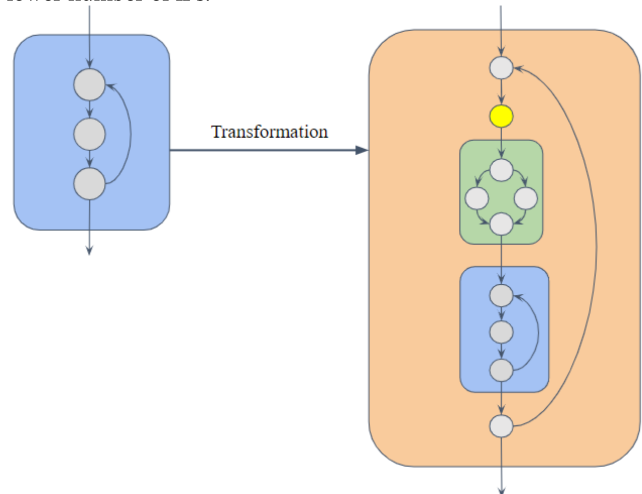

Figure 3: CFG of a loop before and after the loop thresholding transformation

Consider an untransformed loop with N iterations, where at least K iterations can execute within the *MaxVuln* time limit. After the loop transformation, the resulting outer loop will iterate N_{out} times, where $N_{out} = \lfloor \frac{N}{K} \rfloor + 1$.

The inner loop will iterate $N_{in} = K$ times, except for the final iteration of the outer loop, where $N_{in} = N - K \times (N_{out} - 1)$.

The value of N_{in} on the final outer loop iteration may be lower than during others to account for the case N is not an integer multiple of K. Handling this case is the reason for including the conditional statement within the outer loop.

Note that the calculation for K must take into account the additional time spent calling the IP and executing the branch statement as well as the outer loop return, and so will be smaller than the exact value of *MaxVuln* divided by the iteration WCET.

5.6 Generation of IPs

Once the process of transforming loops and partitioning the CFG into regions has completed, we may begin actually placing instrumentation points into the code. First, the completed CFG is re-formed by expanding any loops that were reduced to a single compound block and instrumented separately back into their original form. Subsequently, a single function call is inserted to the IP function at the beginning of every basic block that does not share a color with its parents (i.e., the beginning of each colored region). In addition, a table is generated relating each IP to its associated region's WCET. This WCET data is then populated into the *pathWCET* table (see Section 4), which is stored in protected kernel memory.

6 IMPLEMENTATION

We implemented the design using the ROSE [27] compiler to generate a T-SYS instrumentation for user/kernel source code and subject it to timing experiments with different *MaxVuln* parameters exploiting the elasticity of our tool.

We utilized an Autosar/OSEK-compliant [2, 3] RTOS, Topper [1, 29], that is commercially deployed by Suzuki (among others)

for automotive systems adhering to ISO 26262 [18] and MISRA-C [24] requirements. In particular, we employ the Toppers RTOS (EV3RT) version with available source code targeting a 32-bit ARM 9 processor clocked at 300 MHz featuring 16 KB instruction and 16 KB data caches in experiments [30].

We created a software implementation of T-SYS within Toppers, including all of the components outlined in Section 4. Syscalls for application-embedded IPs were added, along with reserved space for the pathWCET table. Modifications to the interrupt handler were made in order to handle pausing and resuming timers for interrupted tasks.

Actual tool-based integration of instrumentation points was performed on several CPS benchmark applications as well as within the Toppers kernel itself in order to ensure protection across the user-kernel boundary. Implementations were performed using various $MaxVuln$ levels. The task sets instrumented included a selection of tasks from PapaBench [14, 17, 25] with benchmarks from the open-source Paparazzi UAV codebase and selected Malardalen WCET Benchmarks. All instrumentation of IPs in kernel and application code was performed using our ROSE-based placement tool.

A high-precision, write-protected monotonic counter is a requirement for T-SYS, as discussed in Section 3. Most existing hardware platforms provide components that meet this need [12]. Toppers does not innately provide such a device, however, the AM1808 processor of the hardware platform used for testing features an eCAP (enhanced CAPture) module, which can be configured to act as a monotonic counter to generate an interrupt upon reaching a programmable deadline [30]. This device was used in our implementation of T-SYS within Toppers. In the general case, the difficulty of adding support for the T-SYS timer will depend on the precise details of the system being modified. In particular, if the timer hardware is already employed by the RTOS for another purpose, then additional modifications will be needed to multiplex it so as to add T-SYS support while retaining existing OS timers. In the Toppers case, the eCAP module was not being used, so modifications were straightforward.

Modifications to the Toppers interrupt handler were made to handle preemptions of T-SYS protected tasks. The task control block (TCB) structure was extended with a field to store the remaining timer budget at time of interruption. The timer is reinstated upon task resumption, using the remaining budget time plus a constant amount of additional time to account for the overhead associated with the interrupt handler diverting execution before processing the timer pause. The additional time required was measured at 25 cycles in our implementation. Using the TCB to store T-SYS related data is safe as the TCB is part of kernel (i.e., protected) memory.

In case that an interrupt initiated an instrumented kernel path, the T-SYS timer is recorded, and the diverted execution reaches an instrumentation point within the handler marking the entry into the protected section of kernel code. By recording the timer, we can credit the known execution time of the kernel path back to the task upon returning from the interrupt.

In the event that a context switch occurs during the kernel path (as would be likely during a scheduler interrupt), everything up to the dispatcher is considered part of the interrupted task's execution. Once the dispatcher is invoked, the timer's budget is recorded again, in order to be replaced (and credited with the needed extra time) once we return from the dispatched task. Another crediting operation is issued upon return from the interrupt back into the interrupted user task, using the recorded timer value from when the interrupt first arrived.

In addition to the minimum support required for handling protected applications (i.e. syscalls for instrumentation points & other modifications mentioned above), kernel paths related to mutex handling and those related to context switching were instrumented by applying T-SYS protection across the kernel/user boundary to ensure end-to-end protection across the runtime of an entire task set. The instrumented kernel paths constituted task entry/exit, mutex lock/unlock operations, and scheduler interrupts.

7 EXPERIMENTAL EVALUATION

The elasticity of the placement algorithm described in Section 5 supports experiments for a variety of applications with different timing requirements to be instrumented using a varying $MaxVuln$ parameter to conform to the timing bound requirements of each application. Our experiments focus on demonstrating the ability of T-SYS to detect timing anomalies using simulated intrusions. These experiments were performed using benchmark task sets and feature detection at both user and kernel levels.

We select benchmarks from the CPS PapaBench suite with minor modifications to adhere to the Toppers kernel API, and additional benchmarks from the Malardalen set. PapaBench is based on the real-world Paparazzi code base, an open-source framework for UAVs (e.g., quad-copters). It provides a good testing ground for emulating the protection methods' behavior in a realistic environment, particularly in the realm of cyber-physical systems. We modified PapaBench to make it compatible with the Toppers RTOS. PapaBench features precedence constraints, data exchange, synchronization and context switches between tasks, which allowed us to test the effectiveness of T-SYS' protection inside the kernel, as well as within user tasks. The Malardalen tasks were used for comparison to the Bellec algorithm (as it was tested using the same Benchmarks). PapaBench tasks were run together as a real-time task set, as they share mutex-protected data, while tasks from the Malardalen benchmark set were run separately (i.e., one task in the system at a time).

We further compare our T-SYS implementation with Bellec et al. [10]. Our analysis compares intrusion detection capabilities as well as number of instrumentation points executed at runtime for a software instrumentation of both. Notice that the Bellec algorithm is rigid while T-SYS supports elasticity in the maximum allowed vulnerability. Because of this, we use the rigid Maximum Attack Width (MAW) value by Bellec as a base $MaxVuln$ value for each benchmark. We then present additional data for multiples of this value to demonstrate benefits of T-SYS' elastic nature.

Tasks from PapaBench which were instrumented include *servo_transmit*, *send_data_to_autopilot* (shortened to *autopilot*), and *navigation*. These modified PapaBench tasks incorporate context switches and mutual exclusion locks to facilitate task communication. These properties were used to assess T-SYS' ability to detect intrusions to the kernel. Benchmark tasks from the Malardalen set were *fft, cnt, lms, st, edn, statemate, and qsort-exam* and *adpcm*. For both benchmark sets, we also assessed the sensitivity to timing

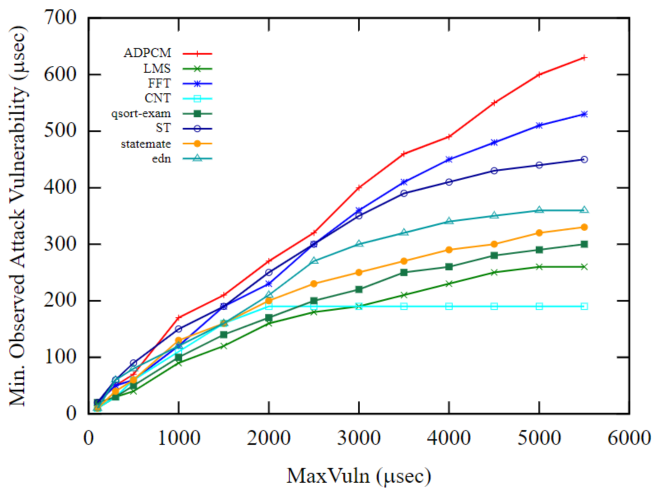

Figure 4: Minimum attack vulnerability vs. *MaxVuln* **for Malardalen tasks.**

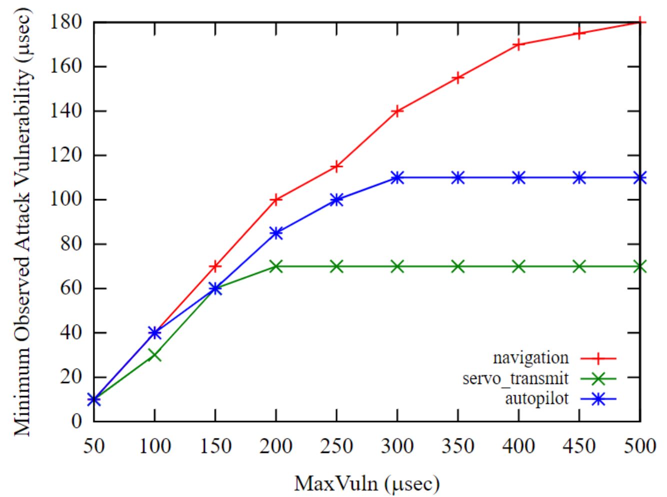

Figure 5: Minimum attack vulnerability vs. *MaxVuln* **for PapaBench tasks.**

overhead induced by T-SYS analyzing performance under different *MaxVuln* levels.

7.1 Attack Detection Experiments

These experiments demonstrate T-SYS' efficacy in detecting attacks while monitoring the time of protected regions under execution, both within and outside the kernel. As *MaxVuln* defines the upper bound on T-SYS timer deadlines, it is impossible for an attack with an execution time greater than *MaxVuln* to go undetected (see Section 4). These experiments focus on determining the longest attack that *is* capable of bypassing T-SYS for a given *MaxVuln* level. Simulated attacks were conducted against T-SYS protected tasks using various *MaxVuln* values. These simulated attacks were conducted by inserting function calls with known execution times into the tasks *after* instrumentation. Attacks were always placed immediately after the IP at the top of the longest region in the tested task or kernel path. This is the worst case for an attack to occur, as it gives the longest time window for the attacker.

By varying the length of the intruding function calls, we simulate attacks of different lengths. This was leveraged to assess how the *MaxVuln* parameter affects attack detection. Thus, for each value of *MaxVuln* shown in the table, the simulated attack length was increased in 5μsecs increments until an attack length was found that always resulted in intrusion detection after 100 attack attempts. The simulation of kernel attacks (Figure 6) followed the same principle.

The results of this experiment are depicted in Figures 4 and 5. The results show that increasing *MaxVuln* leads to an increase in the minimum observed detectable attack in most cases. This reflects an increase in the size of protected regions and their variability in execution time. If the gap between BCET and WCET is large, attackers have an easier time intruding as less time spent inside the loop provides more slack for the attacker to exploit: As long as the execution time of the original code plus that of the attack does not exceed the region's WCET, the attack will not trigger any alarms. As *MaxVuln* is increased, regions encompass more basic blocks with larger differences between BCET and WCET.

Sometimes, the minimum observed detectable attack stagnates after a certain *MaxVuln* value. This occurs once the entire task is contained within one region; increasing *MaxVuln* does not lead to an increase in region length after that, i.e., there is no further loosening of timing bounds as *MaxVuln* increases. This is seen in Figure 6, which details the maximum observed vulnerability for attacks that occur *within* the kernel, particularly during mutex releases/acquisition, and context switches as a result of task completion. Results for these kernel paths were obtained from the modified PapaBench task set representing protected user code. The kernel path attacks are graphed separately to indicate that they occur within the kernel (and not user code as previously for PapaBench attacks).

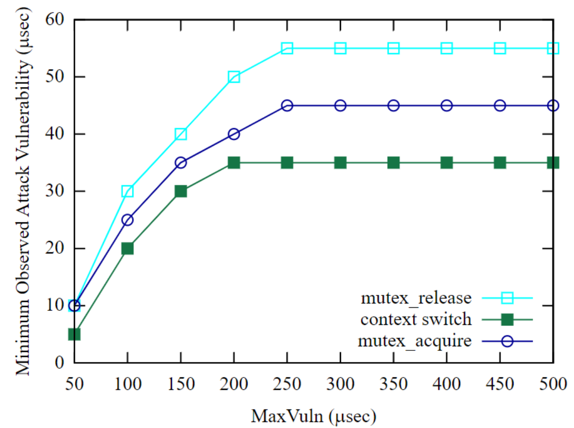

Figure 6: Minimum attack vulnerability vs. *MaxVuln* **for selected kernel paths.**

We also remark that one could establish a minimum guaranteed vulnerability if attach vectors were placed in each region and then gradually increased as in our experiment. Such an approach as linear complexity if the BCET can be triggered within a given region and would results in a tighter bound than a given *MaxVuln* value.

7.2 Performance Impact

The elastic nature of *MaxVuln* provides a customizeable tradeoff between vulnerability and performance for a software implementation of T-SYS. Raising *MaxVuln* allows for reduced overhead due to

less frequent IPs — at the cost of increased vulnerability due to laxer timing bounds. Similarly, lowering *MaxVuln* reduces the vulnerability of the system — at the cost of introducing more frequent IPs, and thus greater execution time overhead. In the case of hardware support for T-SYS, time overhead is zero (see Section 4). Execution times reported in Tables 1 and 2 were gathered experimentally and averaged over 50 runs.

As can be seen in Table 1, integrating T-SYS does induce some overhead in execution time as the unprotected control group always has the lowest execution time. The overhead is highest for the smallest *MaxVuln*. It consistently drops as *MaxVuln* is increased. An exception is `cnt`, which stays leveled for values of *MaxVuln* above 2,000. As `cnt` has a WCET less than 2,000, for all values above that, it still has only one IP at the start of the program, and thus has nearly no overhead. Table 2 displays similar behavior in most entries (for kernel and user tasks). It should be noted that, even for the lowest value of *MaxVuln* tested (corresponding to the highest overhead), all PapaBench tasks still completed before their deadlines, indicating that enough slack existed with in the original, unprotected code base to accommodate significant protection.

MaxVuln (μsec)	unprotected	1000	2000	3000	4000	5000
adpcm	321079	613574	484969	458921	423463	362904
lms	518362	989697	782991	741223	684509	585666
fft	68315	130266	103367	97615	90156	76695
cnt	1981	2601	2226	1991	1992	1990
statemate	295433	563840	446305	422211	390409	334027
edn	147086	280464	221775	209940	193686	166191
qsort-exam	6518	12180	9848	9659	8603	6871
st	426710	813607	642774	609665	562184	481264

Table 1: Average execution time (in μsec) of Malardalen benchmarks for different values of *MaxVuln*.

MaxVuln (μsec)	unprotected	100	200	300	400	500
navigation	614	1162	921	863	821	685
servo_transmit	186	262	199	201	197	198
autopilot	292	426	385	342	301	305
context_switch	157	197	176	157	158	156
mutex_acquire	245	297	278	246	244	245
mutex_release	221	271	245	223	221	220

Table 2: Average execution time (in μsec) of PapaBench tasks and kernel paths for different values of *MaxVuln*.

7.3 Comparison with Bellec

We compare T-SYS to Bellec in terms of number of regions created during instrumentation and number of regions entered during execution, analogous to instrumentation points executed. For purposes of comparison, the *MaxVuln* parameter used was the corresponding maximum attack width (MAW) determined by Bellec. For each task, we use this value as a baseline for four separate T-SYS instrumentations: (1) base *MaxVuln* (T-SYS), (2) $\frac{1}{2}$ of base *MaxVuln* (T-SYS(0.5x)), (3) 2X base *MaxVuln* (T-SYS(2x)), and (4) 5X base *MaxVuln* (T-SYS(5x)). This allows us to analyze how T-SYS compares when taking advantage of its elasticity.

Table 3 reports the number of regions created per algorithm. T-SYS creates fewer regions than Bellec for Malardalen tasks for an equivalent *MaxVuln* (baseline), ranging from ≈ 3%-28% depending on code shape. When *MaxVuln* is cut in half (T-SYS(0.5x)), significantly more regions are created than for base T-SYS or for Bellec. This is expected, as reducing *MaxVuln* limits the length of regions.

Task	Base MAW	Bellec	T-SYS	T-SYS (0.5x)	T-SYS (2x)	T-SYS (5x)
adpcm	9007	36	31	74	23	6
lms	1210	47	34	68	17	11
fft	1117	41	38	72	19	12
cnt	274	15	9	17	5	2
statemate	2970	21	19	34	13	7
edn	3155	32	26	49	18	10
qsort-exam	614	25	23	62	14	9
st	8001	18	16	28	9	5
navigation	121	5	5	9	3	1
servo_transmit	93	3	3	5	1	1
autopilot	134	7	6	10	4	1

Table 3: Comparison of Bellec vs T-SYS algorithms, by number of regions created.

When *MaxVuln* is increased above the base value, the number of regions created drops compared to base instrumentation of both Bellec and T-SYS. With 2X *MaxVuln*, the drop in region count varies widely (between 23% and 50%) as T-SYS has more rules for region structure than maximum length requirements. Thus, granularity does not always scale linearly with *MaxVuln*. When increasing *MaxVuln* to 5X, a consistently large drop is observed in most cases. Notice that smaller tasks from PapaBench are entirely contained within a single region at 5X.

Next, the number of regions encountered dynamically during execution is compared, each of which corresponds to a timer update for the software implementation of both algorithms. In experiments, the Malardalen benchmarks were run to completion while the PapaBench task set ran for 3 seconds, constituting 6 hyperperiods.

Task	Base MAW	Bellec	T-SYS	T-SYS (0.5x)	T-SYS (2x)	T-SYS (5x)
adpcm	9007	14256	12275	24912	6240	1504
lms	1210	407	351	906	241	191
fft	1117	2017	1736	3302	960	580
cnt	274	534	498	1011	278	101
statemate	2970	791	754	1294	452	239
edn	3155	1125	1052	1926	618	348
qsort-exam	614	971	956	1835	572	320
st	8001	640	601	1209	384	198
navigation	121	521	513	1017	221	71
servo_transmit	93	312	254	531	61	61
autopilot	134	548	457	1102	246	87

Table 4: Comparison of Bellec vs T-SYS algorithms, by number of regions entered during execution.

The results of this experiment are depicted in Table 4. T-SYS was observed to have an equivalent or lower number of regions encountered dynamically than Bellec for all benchmarks. Note that the percentage difference between T-SYS and Bellec is lower in runtime than in the static case. This is due to T-SYS incorporating code paths into adjacent regions in some cases that are less frequently executed. Bellec sometimes creates separate regions for such paths, but since they are hardly executed, the dynamic counts remain nearly the same. Overall, the dynamic region count follows the same trend as the static one — reducing *MaxVuln* increases it while increasing *MaxVuln* reduces it. Note that the dynamic region count also stagnates for *servo_transmit* once the number of regions hits 1, as there is no difference between instrumentation under T-SYS(2x) and T-SYS(5x) for that task.

Overall, the elasticity of our T-SYS approach provides significant savings over Bellec as $MaxVuln$ is increased, which makes T-SYS far more flexible and user-configurable.

8 CONCLUSION

This work has contributed T-SYS, a method for securing real-time applications via monitoring execution time. We have implemented a compiler-based tool for integrating T-SYS into user and kernel code. Timestamp checks are automatically placed at specific locations according to an elastic, user-specified MaxVuln parameter. We have implemented support for T-SYS into a commercial operating system and used the compiler tool to implement protection for benchmark tasks as well as the kernel itself, crossing the user/kernel boundary.

We have compared T-SYS with another state-of-the-art timing-based security method and found that T-SYS is competitive in terms of regions created, as well as in terms of region entry operations executed during runtime, while providing the unique ability of unified protection outside and inside the kernel as well when crossing kernel boundaries. Overall, T-SYS provides a versatile, user-friendly and elastic environment for enhancing real-time tasks with timed protection, which can complement conventional security means in safety-critical environments with lower overhead than prior work.

ACKNOWLEDGMENT

This work was supported in part by NSF grants 1813004 and 1525609.

REFERENCES

[1] [n. d.]. Toppers: Toyohashi Open Platform for Embedded Real-time Systems. https://www.toppers.jp/en/atk2.html.
[2] 2005. *OSEK/VDX Group. Operating System Specification 2.2.3.* Technical Report. OSEK/VDX Group. http://portal.osekvdx.org/files/pdf/specs/os223.pdf.
[3] 2013. *AUTOSAR. Specification of Operating System (Version 5.1.0).* Tech Report GbR. Automotive Open System Architecture.
[4] 2016. Using timing-based side channels for anomaly detection in industrial control systems. *International Journal of Critical Infrastructure Protection* 15 (2016), 12 – 26. https://doi.org/10.1016/j.ijcip.2016.07.003
[5] A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley.
[6] G. Ammons, T. Ball, and J. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*. 85–96.
[7] C Arnold, J Butts, and K Thirunarayan. 2013. Strategies for Combating Sophisticated Attacks. *Journal of Information Warfare* 12, 1 (2013), 11–21. https://www.jstor.org/stable/26486995
[8] T. Ball and J. Larus. 1992. Optimally Profiling and Tracing Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 59–70.
[9] Thomas Ball, Peter Mataga, and Mooly Sagiv. 1998. Edge Profiling versus Path Profiling: The Showdown. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. Association for Computing Machinery, New York, 134–148.
[10] Nicolas Bellec, Simon Rokicki, and Isabelle Puaut. 2020. Attack detection through monitoring of timing deviations in embedded real-time systems. In *ECRTS 2020 - 32nd Euromicro Conference on Real-Time Systems*. Modena, Italy, 1–22. https://doi.org/10.4230/LIPIcs.ECRTS.2020.8
[11] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. 2011. Timing Analysis of a Protected Operating System Kernel. In *IEEE Real-Time Systems Symposium*. 339–348.
[12] I. Butun, S. D. Morgera, and R. Sankar. 2014. A Survey of Intrusion Detection Systems in Wireless Sensor Networks. *IEEE Communications Surveys Tutorials* 16, 1 (First 2014), 266–282. https://doi.org/10.1109/SURV.2013.050113.00191
[13] T. M. Chen and S. Abu-Nimeh. 2011. Lessons from Stuxnet. *Computer* 44, 4 (April 2011), 91–93. https://doi.org/10.1109/MC.2011.115
[14] Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 330–335. https://doi.org/10.1109/MICRO.1997.645830
[15] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. 2015. Cross-Kernel Control-Flow–Graph Analysis for Event-Driven Real-Time Systems. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES'15)*. 6:1–6:10.
[16] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann. 2017. SysWCET: Whole-System Response-Time Analysis for Fixed-Priority Real-Time Systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*. 37–48.
[17] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010) (OpenAccess Series in Informatics (OASIcs), Vol. 15)*, Björn Lisper (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 136–146. https://doi.org/10.4230/OASIcs.WCET.2010.136 The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.
[18] ISO. 2011. Road vehicles – Functional safety.
[19] Georgia Koutsandria, Reinhard Gentz, Mahdi Jamei, Anna Scaglione, Sean Peisert, and Chuck McParland. 2015. A Real-Time Testbed Environment for Cyber-Physical Security on the Power Grid. In *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy* (Denver, Colorado, USA) *(CPS-SPC '15)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/2808705.2808707
[20] J. Larus. 1983. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice & Experience* 13, 8 (Aug. 1983), 671–685.
[21] C. Lin, Q. Zhu, C. Phung, and A. Sangiovanni-Vincentelli. 2013. Security-aware mapping for CAN-based real-time distributed automotive systems. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 115–121. https://doi.org/10.1109/ICCAD.2013.6691106
[22] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. 2009. A Survey of WCET Analysis of Real-Time Operating Systems. In *2009 International Conference on Embedded Software and Systems*. 65–72. https://doi.org/10.1109/ICESS.2009.24
[23] Cyntia Vargas Martinez and Birgit Vogel-Heuser. 2019. A Host Intrusion Detection System architecture for embedded industrial devices. *Journal of the Franklin Institute* (2019). https://doi.org/10.1016/j.jfranklin.2019.03.037
[24] MIRA Ltd. 2004. MISRA-C:2004 Guidelines for the use of the C language in Critical Systems. www.misra.org.uk
[25] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. 2006. PapaBench: a Free Real-Time Benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06) (OpenAccess Series in Informatics (OASIcs), Vol. 4)*, Frank Mueller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. https://doi.org/10.4230/OASIcs.WCET.2006.678
[26] Daniel Prokesch, Benedikt Huber, and Peter Puschner. 2014. Towards Automated Generation of Time-Predictable Code. In *14th International Workshop on Worst-Case Execution Time Analysis (OpenAccess Series in Informatics (OASIcs), Vol. 39)*, Heiko Falk (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 103–112. https://doi.org/10.4230/OASIcs.WCET.2014.103
[27] Dan Quinlan, Markus Schordan, Rob Mazke, Pei-Hung Lin, Jim Leek, Justin Too, Chuhua Liao, Craig Rassmussen, and USDOE National Nuclear Security Administration. 2019. ROSE Compiler Framework. https://doi.org/10.11578/dc.20190515.9
[28] Anil Singh, Nitin Auluck, Omer Rana, Andrew Jones, and Surya Nepal. 2017. RT-SANE: Real Time Security Aware Scheduling on the Network Edge. In *Proceedings of The10th International Conference on Utility and Cloud Computing* (Austin, Texas, USA) *(UCC '17)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/3147213.3147216
[29] Hiroaki Takada. 2003. Introduction to the TOPPERS Project - Open Source RTOS for Embedded Systems. In *International Symposium on Object-Oriented Real-Time Distributed Computing*. 44–45.
[30] Texas Instruments Incorporated 2013. *AM1808/AM1810 ARM Microprocessor Technical Reference Manual (Rev. C).* Texas Instruments Incorporated.
[31] David Wagner and Paolo Soto. 2002. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (Washington, DC, USA) *(CCS '02)*. Association for Computing Machinery, New York, NY, USA, 255–264. https://doi.org/10.1145/586110.586145
[32] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. 2008. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems* 7, 3 (April 2008), 1–53.
[33] Lixiao Yi, Yutaka Matsubara, and Hiroaki Takada. 2017. EV3RT: LEGO Mindstorms Real-time software platform for EV3. *Journal of Computer Software* 34, 4 (2017), 4_91–4_115. https://doi.org/10.11309/jssst.34.4_91
[34] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan. 2010. Time-Based Intrusion Dectection in Cyber-Physical Systems. In *International Conference on Cyber-Physical Systems*. 109–118.
[35] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan. 2015. Intrusion Dectection for CPS Real-Time Controllers. In *Cyber Physical Systems Approach to Smart Electric Power Grid*, Siddhartha Kumar Khaitan, James D. McCalley, and Chen Ching Liu (Eds.). 195–217.

A FUTURE WORK

Future work on T-SYS will focus on expanding support for additional platforms. In particular, the current T-SYS models restricted to single-core systems. However, multicore processors are increasingly common in embedded systems, making expanding the T-SYS model to multicore systems a valuable prospect. The difficulty of doing so is dependent on the particular hardware available, particularly the timer module used to keep track of region execution time. If there is a separate module available for each core and tasks are pinned to cores, then expansion is relatively straightforward with the main challenge being the complexity of WCET analysis on multicore systems. Difficulties arise with a single timer shared amongst all cores, as this would require a timer queue to ensure the shortest deadline amongst all currently-executing regions is tracked by the timer. Handling this queue would add some overhead to instrumentation points, as well as interrupts where the timer is accessed.

B POINT PLACEMENT ALGORITHMIC COMPLEXITY

The three steps in the IP placement process are performed sequentially. Therefore, its complexity is bounded by the most complex step. As each step is a graph traversal process, we determine the complexity in terms of the number of basic blocks (i.e., nodes) in the control flow graph.

The Partial Regions step has a worst-case complexity of $O(n^2)$. The `step1_dfs()` function is recursively called on all basic blocks, however, it is possible for basic blocks to be visited multiple times. All cycles in the CFG have been eliminated by the preprocessing step, so the CFG is now a directed acyclic graph. Therefore, even if we assume the worst case, where the current node has all other nodes as children (and thus the loop will iterate $n-1$ times), each recursive call can iterate at most $n-2$ times, and so on. Thus, the total complexity is the sum of integers 1 to n, which is bounded by $O(n^2)$.

The Region Filling step has a worst-case complexity of $O(n^2)$. The outer `step2()` function is $O(n^2)$ (for similar reasons as `step1_dfs()`). The `expandRegion()` function's recursion is bounded by the MaxVuln parameter (a constant), which is $O(1)$, giving us a total complexity bound of $O(n^2)$.

The Region Adjustment step has a worst-case complexity of $O(n^2)$, as it is structurally identical to the Partial Regions algorithm, with additional constant-time operations.

Thus, as the IP placement algorithm is comprised of three $O(n^2)$ algorithms run sequentially, with an overall complexity of $O(n^2)$ as well.

C POINT PLACEMENT PSEUDOCODE

Listing 2: Instrumentation Placement Partial Regions Step

```
step1_dfs(P):  for S in P.children
                 skip = false
                   if S.hascolor == false
                     S_post = S.postDoms
                     for E in S_post
                       if S not in E.Doms
                         S_post.remove(E)
                     sortByDom(S_post)
                     for E in S_post
                       rWCET = longestPath(S, E)
                     if rWCET < MaxVuln
                         paintRegion(S, E)
                         skip = true
                         skipToBlock = E
                         break
                   if skip == true
                     step1_dfs(skipToBlock)
                   else
                     step1_dfs(S)
```

Listing 3: Instrumentation Placement Filling Regions

```
step2(P):      for S in P.children
                 if !S.hasColor
                   paint(S, newcolor())
                   expandRegion(S)
                 step2(S)

               expandRegion(P):
                 for S in P.children
                   if S.color != P.color
                     return
                   if longestPath(P, S) > MaxVuln
                     return
                   for S_p in S.parents
                     if S_p.color != P.color
                       return
                 for S in P.children
                   paint(S, P.color)
                 for S in P.children
                   expandRegion(S)
```

Listing 4: Instrumentation Placement Region Adjustment

```
step3_dfs(P):  for S in P.children
                 skip = false
                 if S.children[0].color != S.color
                   S_post = S.postDoms
                   for E in S_post
                     if S not in E.Doms
                         S_post.remove(E)
                       sortByDom(S_post)
                   for E in S_post
                       rWCET = longestPath(S, E)
                       if rWCET < MaxVuln
                           paintRegion(S, E)
                               skip = true
                               skipToBlock = E
                               break
                 if skip == true
                   step3_dfs(skipToBlock)
                 else
                   step3_dfs(S)
```

D SKETCHED CORRECTNESS PROOF

We establish the correctness of several components of the placement algorithm.

Lemma D.1. *Regions created via pre/post dominator pairs are correctly formed.*

There are three ways in which a region can be incorrectly formed:

• By having a WCET that exceeds $MaxVuln$. This cannot happen as $MaxVuln$ compliance is explicitly checked at region formation.

• A block in the region shares a color with a successor and has a successor that it does not share a color with. This cannot happen: As the region's exit is a post-dominator for the entry point, all paths from the entry must eventually reach the exit block, and every block included in one of these paths is colored for the region. If a block within the region has a successor outside the region and a successor inside the region, then this would imply that there is an execution path from the start that does not lead to the exit. This is a contradiction, as the exit is a post-dominator for the start block. Therefore, this case cannot happen.

• A block in the region shares a color with a predecessor and has a predecessor that it does not share a color with. This is also impossible, with a proof similar to the previous one, this time relying on the fact that the start is also a dominator for the end block.

Given this, we have shown that regions built using the pre/-post dominator algorithm used in the Partial Regions and Region Adjustment steps are correctly formed.

Lemma D.2. *Regions created via the Filling Regions step are correctly formed.*

The correctness of the Filling Regions algorithm is not fully proven due to space concerns. It can be shown that the regions generated are correct by noting that the correctness of the region is checked at each stage of growth (analyzing the new predecessors and successors created after each change), and growth only occurs if no rules were violated.

Based on these assumptions, we can show that the Region Adjustment step will also produce correctly-formed regions, provided the regions it starts with are correctly formed, as they should be according to Lemmas D.1 and D.2.

There are 4 ways in which the Region Adjustment step can violate the rules for region structure laid out in Section 5:

• By creating a region with a WCET exceeding $MaxVuln$. This cannot happen as the region's WCET is checked against $MaxVuln$ before creation.

• By creating a region that violates the structural rules for region shape. This does not happen, as the regions are generated via pre/post dominator pairs, which generate correctly shaped regions, according to the Lemma D.1.

• Recoloring nodes causes the regions they were previously a part of to no longer obey the shape rules. This cannot happen: For the new region to break existing regions, the old regions would need to have at least one block with a successor that was recolored, and a successor that was not recolored (or likewise for a pair of predecessors). However, for this to be true, the recolored block, part of the new region, would need to have a predecessor or successor color violation as well. Because we know that the new region is properly formed, we can infer that regions with blocks removed (due to being recolored and added to the new region) are still properly formed.

• Recoloring nodes causes the WCET of the regions they were previously a part of to exceed $MaxVuln$. This will not happen: The region WCET is calculated as the sum of block WCETs along the longest path through the region. Assuming absence of hardware anomalies, removing a block can only reduce WCET (if the block was on the longest path), or leave it unaffected (if the block was not).

Based on these proof steps, the algorithms described in Section 5 properly generates regions according to the region shape assumptions that our protection is based on.