Proceedings of

the 1st Workshop on
Security and Dependability of
Critical Embedded Real-Time Systems

November 28th, 2016 in Porto, Portugal

in conjunction with

IEEE Real-Time Systems Symposium
29th November 2016 – 2nd December 2016

Editors:
Marcus Völp
Paulo Esteves-Veríssimo
António Casimiro
Rodolfo Pellizzoni
Contents

Message from the Chairs 3
Program Committee 4
Program 5
Session I: Learning how to survive 6
  embSFI: An Approach for Software Fault Isolation in Embedded Systems
    Andreas Ruhland, Christian Prehofer and Oliver Horst 6
  CAML: Machine Learning-based Predictable, System-Level Anomaly Detection
    Jiguo Song, Gerald Fry, Curt Wu and Gabriel Parmer 12
  Enforcing Safety and Security Through Non-Intrusive Runtime Verification
    Inês Gouveia and José Rufino 19
Session II: Secure and dependable vehicles 25
  MERgE: Technology Advancement for Cohesion of Concerns in System Engineering
    Charles Robinson, Jérôme Pequery and Sam Michiels 25
  Towards Comprehensive Threat Modeling for Vehicles
    Mohammad Hamad, Marcus Nolte and Vassilis Prevelakis 31
  Towards a Fail-Operational Intrusion Detection System for In-Vehicle Networks
    Clinton Young, Joseph Zambreno and Gedare Bloom 37
Session III: Be secure, but not too late 39
  ScheduLeak: An Algorithm for Reconstructing Task Schedules in Fixed-Priority Hard
    Real-Time Systems
    Chien-Ying Chen, Amiremad Ghassami, Sibin Mohan, Negar Kiyavash, Rakesh B. Bobba and Rodolfo Pellizzoni 39
    Fardin Abdi, Monowar Hasan, Sibin Mohan, Disha Agarwal and Marco Caccamo 47
  Timing Analysis of Secure Communication between Resource Managers in DREAMS
    Gautam Gala, Thomas Koller, Daniel Gracia Pérez, Gerhard Fohler and Christoph Ruland 55
  A Server Model to Integrate Security Tasks into Fixed-Priority Real-Time Systems
    Monowar Hasan, Sibin Mohan, Rakesh Bobba and Rodolfo Pellizzoni 61

© Copyright 2016 University of Luxembourg.
All rights reserved. The copyright of this collection is with University of Luxembourg.
The copyright of the individual articles remains with their authors.
Message from the Chairs

Welcome to Porto, Portugal and welcome to CERTS 2016, the 1st workshop on Security and Dependability of Critical Embedded Real-Time Systems. We invite you to join us in participating in a workshop of lively discussions and exchanging ideas related to the security and dependability of real-time and embedded systems. The workshop brings together researchers from industry and academia from the security and dependability, distributed systems and real-time communities to discuss and promote new and exciting research ideas and initiatives and to identify and discuss the grand challenges that lie ahead of us for such critical systems.

CERTS 2016 received a total of twelve submissions. All papers were peer-reviewed and ten papers were finally accepted. Each paper received at least three individual reviews. The papers will be presented in three sessions. The first session includes three papers that explore fault isolation, learning and runtime verification to survive faults and attacks in critical real-time systems. Session 2 focuses on security and dependability of vehicles and similar cyber-physical system. Session 3’s main theme is timeliness of critical systems while ensuring security.

CERTS 2016 would not have been possible without the support of many people. The first thanks are due to Marco Caccamo and Frank Mueller, the RTSS 2016 organizers and to Song Han, RTSS 2016 workshop chair for entrusting us with organizing CERTS 2016, and for their continued support of the workshop. Our special thanks go to the program committee, a team of 18 experts from ten different countries, for volunteering their time and effort to provide useful feedback to the authors, and of course to all the authors for their contributions and hard work. Last, but not least, we thank you, the audience, for your participation. Through your stimulating questions and lively interest you help to define and improve CERTS. We hope you will enjoy this day.

The CERTS 2016 Workshop Chairs,

Marcus Völp  
SnT, University of Luxembourg

Paulo Esteves-Veríssimo  
SnT, University of Luxembourg

António Casimiro  
University of Lisboa

Rodolfo Pellizzoni  
University of Waterloo
Program Committee

Lorenzo Alvisi  
Iain Bate  
Danny Dolev  
Antonio Augusto-Fröhlich  
Virgil Gligor  
Zbigniew T. Kalbarczyk  
Sibin Mohan  
Ravi Prakash  
Sasikumar Punnekkat  
Robert van Rennesse  
Guillermo Rodriguez-Navas  
José Rufino  
Lui Raymond Sha  
Elad Schiller  
Ulrich Schmid  
Marisol Garcia-Valls  
Heechul Yun  
Sebastian Zug

University of Texas Austin  
University of York  
Hebrew University  
Federal University of Santa Catarina  
Carnegie Mellon University  
University of Illinois  
University of Illinois  
University of Dallas  
Mälardalen University  
Cornell University  
Mälardalen University  
University of Lisboa  
University of Illinois  
Chalmers University of Technology  
Technische Universität Wien  
Universidad Carlos III de Madrid  
University of Kansas  
University of Magdeburg
<table>
<thead>
<tr>
<th>Time</th>
<th>Session/Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:30</td>
<td>Welcome</td>
</tr>
<tr>
<td>9:40</td>
<td><strong>Setting the Scene: the future of CERTS as seen by the organizers</strong></td>
</tr>
<tr>
<td>10:30</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>10:00</td>
<td><strong>Session I: Learning how to survive</strong></td>
</tr>
<tr>
<td>11:00</td>
<td>embSFI: An Approach for Software Fault Isolation in Embedded Systems</td>
</tr>
<tr>
<td></td>
<td>Andreas Ruhland, Christian Prehofer and Oliver Horst</td>
</tr>
<tr>
<td>11:20</td>
<td>CAML: Machine Learning-based Predictable, System-Level Anomaly Detection</td>
</tr>
<tr>
<td></td>
<td>Jiguo Song, Gerald Fry, Curt Wu and Gabriel Parmer</td>
</tr>
<tr>
<td>11:40</td>
<td>Enforcing Safety and Security Through Non-Intrusive Runtime Verification</td>
</tr>
<tr>
<td></td>
<td>Inês Gouveia and José Rufino</td>
</tr>
<tr>
<td>12:00</td>
<td>Lunch</td>
</tr>
<tr>
<td>13:30</td>
<td>Open issues in Security and Dependability of Critical Embedded Real-Time Systems</td>
</tr>
<tr>
<td>14:00</td>
<td><strong>Session II: Secure and dependable vehicles</strong></td>
</tr>
<tr>
<td>14:20</td>
<td>MERgE: Technology Advancement for Cohesion of Concerns in System Engineering</td>
</tr>
<tr>
<td></td>
<td>Charles Robinson, Jérôme Pequery and Sam Michiels</td>
</tr>
<tr>
<td>14:40</td>
<td>Towards Comprehensive Threat Modeling for Vehicles</td>
</tr>
<tr>
<td></td>
<td>Mohammad Hamad, Marcus Nolte and Vassilis Prevelakis</td>
</tr>
<tr>
<td>14:40</td>
<td>Towards a Fail-Operational Intrusion Detection System for In-Vehicle Networks</td>
</tr>
<tr>
<td></td>
<td>Clinton Young, Joseph Zambreno and Gedare Bloom</td>
</tr>
<tr>
<td>15:00</td>
<td>Coffee Break</td>
</tr>
<tr>
<td>15:30</td>
<td><strong>Session III: Be secure, but not too late</strong></td>
</tr>
<tr>
<td>15:50</td>
<td>ScheduLeak: An Algorithm for Reconstructing Task Schedules in Fixed-Priority</td>
</tr>
<tr>
<td></td>
<td>Hard Real-Time Systems</td>
</tr>
<tr>
<td></td>
<td>Chien-Ying Chen, Amiremad Ghassami, Sibin Mohan, Negar Kiyavash, Rakesh B. Bobba</td>
</tr>
<tr>
<td></td>
<td>and Rodolfo Pellizzoni</td>
</tr>
<tr>
<td></td>
<td>Systems</td>
</tr>
<tr>
<td></td>
<td>Fardin Abdi, Monowar Hasan, Sibin Mohan, Disha Agarwal and Marco Caccamo</td>
</tr>
<tr>
<td>16:10</td>
<td>Timing Analysis of Secure Communication between Resource Managers in DREAMS</td>
</tr>
<tr>
<td></td>
<td>Gautam Gala, Thomas Koller, Daniel Gracia Pérez, Gerhard Fohler and Christoph</td>
</tr>
<tr>
<td></td>
<td>Ruland</td>
</tr>
<tr>
<td>16:30</td>
<td>A Server Model to Integrate Security Tasks into Fixed-Priority Real-Time Systems</td>
</tr>
<tr>
<td></td>
<td>Monowar Hasan, Sibin Mohan, Rakesh Bobba and Rodolfo Pellizzoni</td>
</tr>
<tr>
<td>16:50</td>
<td>Closing Remarks</td>
</tr>
</tbody>
</table>
embSFI: An Approach for Software Fault Isolation in Embedded Systems*

Andreas Ruhland  
fortiss GmbH  
Munich, Germany  
andreas.ruhrland@tum.de

Christian Prehofer  
fortiss GmbH  
Munich, Germany  
prehofer@fortiss.com

Oliver Horst  
fortiss GmbH  
Munich, Germany  
horst@fortiss.com

Abstract—SoftwareFaultIsolation(SFI)isatechniquetosandboxsoftwarecomponentsbasedontransformationandchecks on the assembly code level. In this way, software components can only access memory within specific fault domains. This paper presents embSFI, which applies selected SFI techniques to embedded systems in order to increase dependability and security, complementing or replacing a memory management unit. Our approach is designed to use SFI techniques which can be validated efficiently, even on embedded devices. Furthermore, we show that the overhead in performance is low, however depending on the scenario.

I. INTRODUCTION

Software Fault Isolation (SFI) modifies code at the assembler level with instructions to guarantee that the code does not violate the restrictions required by the system. This can, for example, be instructions that ensure the target address of a memory instruction is valid. This concept was introduced more than 20 years ago by Wahbe et al. [1]. This approach does not target embedded ARM systems but some general concepts were used within embSFI. Another recent approach, NaCl [2], targets ARM hardware but requires virtual memory addresses and is not built for small embedded systems without full posix support. Concepts from both approaches are used to build embSFI which provides a fast online validation on embedded systems and a LLVM [3] compiler extension to generate suitable code. The concrete SFI patterns that make fast validation and execution on embedded systems possible were newly designed for embSFI. A more detailed description of this work can be found in [4]. In this paper software component and app are used interchangeably.

For secure and dependable critical systems it is important to keep the fault domains as small as possible. This means that should an error occur in one software component, other software components are unaffected. Small fault domains are one of the main motivations for microkernels, which are still extensively researched for critical systems [5]. There are various methods of creating fault domains, such as utilizing a memory management unit (MMU), virtualization or the aforementioned SFI. EmbSFI uses SFI to enforce fault domains, and so we highlight the general advantages of using SFI for dependable critical embedded real-time systems in the following sections.

If a MMU or a memory protection unit (MPU) is not present within the embedded system, SFI can substitute their functionality. Should a MMU be available, SFI can still provide finer granularity on memory accesses. For example, with SFI it is possible to allow a software component to directly access only one byte in a memory area in a hardware memory mapped area. This can be necessary if the access to different hardware should be controlled to only grant access to specific software components. This could also be implemented by using a MMU and different execution levels and issuing a system call for access, but this approach would take control from the software developer. In addition, system calls can make the time analysis of software more difficult and make the system as a whole more complex. Because of that, SFI was used to decrease the context switch time for hardware drivers in microkernels [1].

When enforcing SFI, it is important to validate that every instruction in the binary is checked, and that the system only accepts binaries that fulfill the SFI invariants. This means that it is not possible to hide instructions, as can be done with self-modifying code. This is an important security feature as hardware cannot always be trusted and be susceptible to faults as shown with the rowhammering attack [6]. Within this attack, an instruction normally available in the user level of execution was used to change the expected behavior of the hardware. Through the use of SFI, such instructions can be disallowed. This is one reason why SFI is used inside the Chrome Browser to execute untrusted code [7].

The rest of the paper is organized as follows: Section II will describe the assembler properties of the embSFI system. The implementation of embSFI, including the compiler extension that generates code according to these properties, and the validator, which validates that the assembler code fulfills these properties, are discussed in Section III. An evaluation of the implementation with regards to its general limits and benchmark performance of the modified compiler are presented in Section IV.

II. ASSEMBLER PROPERTIES OF EMBFSI

In this section, we describe the properties of the validatable assembler. embSFI combines concepts from [2] and [1] and
adopts them to the ARMv7 architecture. These concepts are additionally made compatible for embedded systems without utilizing a MMU. A software component referred to as app, a consists of a code area and a data area. The code area of the app must be of a fixed size, \( C \), that must be a power of two. The data area includes a variety of data such as heap and stack, and similarly must have a fixed predefined size \( D \), which is also a power of two. The execution environment for the app, similar to an operating system, can provide special functions that are callable from within the app. It is possible to allow the app to directly access specific memory areas outside of the private memory of the app. The concrete set of assembler properties were selected to be quickly validatable and allow for code that adheres to these properties to be easily generated. The following example code shows valid assembler code compared to code generated by an unmodified compiler:

**Listing 1: without SFI**

```
push {r4, s1, fp, lr}
add fp, sp, #8
movt r0, #0
bl 0 <print>
...
```

**Listing 2: with SFI**

```
bfi sp, r9, #24, #8
push {r4, s1, fp, lr}
add fp, sp, #8
mov r0, #0
mov r0, #0
nop {0}
nop {0}
bl 0 <print>
...
```

The different additional instructions in Listing 2 enforce the properties, no matter what the machine state is when entering this code.

We first describe the general properties of the modified code, and follow these with concrete patterns that are accepted by the embSFI validator.

A. embSFI approach

The main goal is to prevent the code of an app to jump somewhere out of its own code or to access memory outside of its own private memory area. However, there exist properties not directly connected to these goals, which are still necessary to be validatable.

1) Bundling: Bundling is a concept utilized by [2], that ensures every target of an indirect or direct instruction jump must be the start of a bundle. A bundle consists of multiple consecutive instructions with every instruction belonging to exactly one bundle. The binary size of the bundle \( B \) must be a power of two \( B = 2^n \) for \( n \in \mathbb{N} \). The bundle size is given by the environment and is the same for all apps. It can be assumed that the first instruction is located at the beginning of a bundle and all other instructions are consecutive and kept in the same order. Possible jump targets are normally function entries or branches of the app or selected function entries of the operation system. Assuming that hexadecimal addresses are used that address bytes, that a single instruction is 4 byte long as in the A32 Armv7 instruction set and that 4 consecutive instructions should always form a bundle, the resulting bundle size would be 16 bytes. This was the bundle size of Googles NaCl described in [2]. This means the start address of each code bundle should end with 0x0. For example, 0xaaa0 and 0xfffff are possible jump targets but 0xbbbb1 is not.

Therefore, in Listing 2 it is not possible to jump directly to the push in line 1 and skip the sandboxing instruction in line 0.

2) Restricted Instruction Set: For embSFI only ARMv7 A32 instructions were considered which means no thumb instructions are currently possible like in [2]. Multiple instructions are forbidden completely, for example, store memory operations, which use a register as base address and as an index. All instructions that lead to undefined behavior or a fault are also not allowed.

3) Position independent code: The code of the app cannot assume that it will be located at a specific address during execution. This is also assumed by other binary loading concepts on embedded systems [8]. Therefore, the code must be position independent, which means that the code must either use program counter dependent addresses or use linker symbols instead of hard coded addresses.

4) Memory read and write: As realized in [1] the memory address where any instruction reads from or writes to must be inside the fixed size private data area of the app or an explicitly allowed memory address. The allowed memory access addresses do not change during the execution. The code segment of the app is not readable or writable. This is a difference to [7] where read access to the code segment is allowed. In the SFI method proposed by embSFI the stack and the predefined app data are both in the same consecutive fixed size private data area. The size of the private data segment must be a power of two which means \( D = 2^n \) for \( n \in \mathbb{N} \). Also the start address of the data memory \( S \) area must be a multiple of \( D \) which means \( S = D \times n \) for \( n \in \mathbb{N} \)

5) Guard zones: As proposed in [1] guard zones are used to make certain accesses possible. For example in Listing 2 the push instruction manipulates 16 bytes and the base address is only guaranteed to be anywhere inside the private memory area of the app. This means that a small amount of memory around the data memory area must be reserved for the app. For example if the data memory areas goes form 0xaaaaa00 to 0xaaaaaff and the guard zone \( G \) is 40 bytes big the address ranges 0xaaaaa9d8 to 0xaaaaa00 and from 0xaaaaaff to 0xaaaaab27 have to be reserved exclusively. In difference to [1] no error is generated if the guard zones are accessed. They can contain for example parts of the stack.

6) Dedicated registers: Like proposed in [1] for MIPS dedicated registers can also be used in the ARM Architecture. In the register r9 the start address of the data segment is stored. It is shifted \( log_2(D) \) bits to the right. This is only for an efficient implementation and is not used in [7] because the NaCl implementation relies on the memory management unit for separating code from data and it is not used in [1] because it is an optimization specific for certain ARM instructions as shown in the following sections. For example if the data segment starts at 0xdeadbe00 and is 256 bytes big the content of r9 is 0x00deadbe. The code can always assume
that \( r9 \) which is set by the environment contains the shifted address of the current data segment. Also the code is not allowed to use any instruction that could possibly change \( r9 \). As code and data memory areas are separated, the dedicated register \( r8 \) is used for the code and must always contain a valid jump address for the app. This means bundling must be respected and the size of the code segment. The app can assume that \( r8 \) always contains a valid jump address because it is set by the environment and enforced with the validation. The app can change \( r8 \) only if it is guaranteed that after the modification \( r8 \) again contains a valid jump address. This can be tricky and one way to do it is shown in Section II-B2.

B. Valid Patterns

To keep the validator very simple and small so it can be executed on an embedded system the patterns that are accepted are very simple. One simplification is to always look only at a bundle at a time. The validator will not accept code that applies to the principles but uses different patterns.

1) Memory access instructions: In this section possible valid A32 instruction sequences for memory accesses that can easily be validated are discussed. All instructions sequences are always in the same bundle. The most straightforward approach would be the direct access of data at a fixed address in the data memory area of the app. The address could be set by the linker.

\[
\begin{align*}
\text{mov} & \ r1, \#\text{dataaddress} \\
\text{ldr} & \ r2, [r1] \\
\text{str} & \ r3, [r1]
\end{align*}
\]

If the linker replaces \( \text{dataaddress} \) with a valid constant address in the data segment of the app the validation of this code would be successful. The validator accepts several of move, load and store combinations.

If the memory access is the result of a complex calculation, the validation cannot track the value of the variable for example because the calculation is spread in multiple bundles. In this case, the register used for the access must be sandboxed as done by multiple other approaches [2] [1]. However, embSFI does a special optimized sandboxing for ARM using dedicated registers. We assume that \( D_a = 2^4 = 16 \) and look at the following instruction sequence:

\[
\begin{align*}
\text{bfi} & \ r1, r9, \#4, \#28 \\
\text{ldr} & \ r3, [r1]
\end{align*}
\]

Because of the properties we can assume that \( r9 \) contains the address of the start of the data memory area \( S_a \) shifted \( \log_2(D_a) = 4 \) bits to the right. Now the \( \text{bfi} \) instruction copies the bits 1 to 28 of \( r9 \) to the bits 4 to 32 of the register \( r1 \). In the instruction 4 is the start target bit and 28 is the number of bits copied [9, p. 2616]. Now we can assume \( r1 \) to be in the data range. As \( \text{ldr} \) loads 4 bytes the access is valid if the guard zone \( G \geq 4 \).

As ARM allows also constant offset the maximal constant depends on the guard zone size \( G \). For example instead of \( [r1] \) using \( [r1, \#10] \) requires \( G \geq 14 \).

The sandbox approach as described above has the disadvantage that a faulty app instead of immediately shutting down keeps on running using wrong values. An alternative approach which is also described in [1] is based on comparing the variable and jumping to an error handling routing if the address is out of range.

2) Branch instructions: In this section possible instruction sequences for branches that can easily be validated are discussed. It can again be assumed that all instructions are in the same bundle. During the validation, the absolute address of every single instruction is already known. So the A32 branch instruction \( \text{b} \) which uses a constant offset to the current program counter can easily be validated by calculating the constant target of the branch and checking if it is inside the app’s code memory area and if it targets the beginning of a bundle.

For indirect jumps where the target address is normally stored in a register also valid instrumentations exist. We assume \( B = 2^4 = 16 \) and \( C_a = 2^8 = 256 \)

\[
\begin{align*}
\text{bfc} & \ r1, \#0, \#4 \\
\text{bfi} & \ r8, r1, \#0, \#8 \\
\text{bx} & \ r8
\end{align*}
\]

The first instruction is a bit field clear which sets the bits 0 to 3 to zero [9, p. 2614]. Now the register \( r1 \) is bundle aligned. After that, the last 8 bits of the bundle aligned \( r1 \) are copied to the last 8 bits of the dedicated \( r8 \) register. This is valid because we can assume that \( r8 \) already contains a valid jump target and as the code segments starts 256 byte aligned and is 256 bytes big for all possible values of \( r1 \) \( r8 \) is still a valid jump target after the \( \text{bfi} \) instruction. This makes the \( \text{bfi} \) instruction valid. The instruction \( \text{bx} \ r8 \), which jumps to the address in \( r8 \), is always valid because \( r8 \) always contains a valid jump target as required by the dedicated register invariant.

For function calls normally the A32 instruction \( \text{bl} \) is used which pushes the address of the next instruction onto the stack and then jumps into the function [9, p. 2627]. This makes it easy to return to the callee. But also the return must follow the invariants which means that the target address to return to must be the start of an aligned bundle. This can be realized by filling up the bundle with operations that do nothing like \( \text{nop} \). Assuming one bundle contains 4 instructions this would lead to:

\[
\begin{align*}
\text{nop} \\
\text{nop} \\
\text{nop} \\
\text{bl} \ #\text{function}
\end{align*}
\]

This moving of the link instructions down at the bottom of a bundle was also used by Google and is already implemented in LLVM [2]. With these instrumentations a validator can easily check if the invariants are fulfilled by the code by only looking at one bundle at a time. Also by using always the same instrumentation, the validator gets simpler.
III. IMPLEMENTATION

The implementation of embSFI consists of different components like seen in Figure 1. In this section, the implementation of the different components will be described.

A. ELF loading

As binary format the ELF format was chosen because it is already used by other embedded systems [8], it is well documented for ARM [10] and supported by common compilers like llvm [3] and gcc. As embedded systems often have only a small amount of memory available other approaches exist which use smaller file formats that reduce the overhead of the ELF format [11]. Because of simplicity reasons embSFI uses the plain ELF format. The app must be a relocatable and not an executable. This is required so that the app can be linked easily to a specific hardware address during loading and does not depend on virtual memory addressing. Shareable ELF files would be harder to validate than already statically linked relocatables. To keep the code small and simple only the relocations R_ARM_ABS32, R_ARM_CALL, R_ARM_MOVW_ABS_NC and R_ARM_MOVT_ABS were implemented. There are over 100 different relocation types defined for the ELF format [10]. Some of them are not relevant for the SFI proposed here because they are used only in T32 ISA, which is not allowed so for the code generation these relocations were sufficient. The loader and linker included in embSFI requires the possibility to allocate aligned memory otherwise the allocation is very inefficient and takes up to double of the required memory like described in Section IV-A2.

B. Validation

The validator is implemented in C and so can be integrated into different execution environments or operating systems, which are often also written in C in the embedded environment. The validator expects the already linked code section and the data sections. The validator works directly on the binary machine code to optimize its memory footprint. The running time is $O(\text{number of instructions})$. In the evaluated examples described in Section IV the validation time was always below one second.

C. Compiler

As in [2] Clang was used for embSFI which already supports bundling. Other benefits of using clang are the decoupling of frontend language and output machine code. So the embSFI extension for LLVM can be also used with other languages that are not C. The embSFI extension for LLVM requires a parameter to specify the size of the required data area, which is fixed as explained in Section II.

IV. EVALUATION

The advantages of preferring SFI over MMU were already discussed in the Section I. These general advantages also apply to embSFI. In this section the general limits and drawbacks of embSFI will be discussed, followed by an evaluation of the performance of embSFI.

A. Limits and Drawbacks

1) No dynamic memory allocation: The embSFI principles require a fixed data size which does not change during the execution which means that the used memory cannot grow. Also all the required memory is reserved during the execution of the app. This is not a problem in for example high confidential real-time systems because there it should be guaranteed that the app is able to be executed and does not stop working in the middle of the execution because of a lack of available memory.

2) Allocated size: The size of the used memory can be the double of the original app because the code and the data memory areas should both have a size which is a power of two. So for example if an app uses 257 bytes of data the OS has to allocate $D_a = 512$ bytes. Therefore, the double size is the worst case scenario. Additionally the guard zones must also be allocated around the data area.

3) Aligned memory: The memory for the app must be allocated aligned. If no aligned memory allocation is available on the OS it could be required to allocate double of the needed $D_a$ bytes of memory and then set the latest $\log_2(D_a)$ bits of the aligned block to zero to get a valid $S_a$. This holds also for the memory used for the code.
4) Bigger code size: To ensure the invariants sometimes new instructions have to be added compared to the execution without SFI. The worst case size of the code can be multiplied with the instructions per bundle. So if every bundle consists of 4 instruction the code can be 4 times larger for SFI than without SFI.

5) No self-modifying code: As the code and the data segment are clearly separated it is not possible to use self-modifying code. This should not be needed in the most systems with high confidentiality because it makes code often more vulnerable to exploits. This restriction can also found in other SFI approaches like [2] and [1].

6) No function pointers to the OS: This is theoretically possible but not practical because it would make instrumentation much harder. The r8 register as described above can easily be used in the range of the own code of the app but not outside. One possibility of solving this is the use of trampolines like done in [2].

7) No usage of compiled code: It would be possible to transform already compiled code that does not fulfil all principles into instrumented SFI code. This can be done for example with binary rewriting tools like [12]. However, as the embSFI principles make assumptions about the dedicated registers r8 and r9 such a rewriting would be hard to implement and the resulting code would be very slow. So in general no compiled code without the source can be used.

8) No return exit: This means that when the app is finished it cannot do a normal return because this return would jump eventually into the OS where it is not allowed to jump. Either an OS function which is allowed for the app and terminates the app must be called or an endless loop must be inserted at the end of the app as last instruction.

9) Error detection: The current patterns guard memory access, which means that a memory access for an invalid address is changed to a valid memory access but an error is not detected. This was done because of performance reasons. It would also be possible to detect the error and handle an error routine provided by the OS. This would increase the instructions to add and so decrease the performance in regards to memory size and run-time.

B. Performance

As in ARMor [13] to evaluate the performance MiBench [14] was selected which is well suited for embedded system because it does not rely on OS functions like file handling and is implemented in C. The benchmark was executed on an ARM Cortex A-53 processor [15]. As compiler Clang with the modified LLVM [16] in the version 3.7 was used including the embSFI extension. The results were compared to binaries emitted by an unmodified Clang/LLVM 3.7. The source code to reproduce the benchmark results is available at http://embsfi.de.

1) qsort: The qsort small example from MiBench [14] was chosen to be evaluated because it heavily accesses the main memory. The test case uses 5000 strings and the qsort from the standard c library. The tested app consists of all parts necessary for the benchmark, which means that the qsort and all other used library functions are part of the app. The implementation from newlib [17] was used for that. The binary size overhead for all different optimization levels was around 90%. The overhead in the execution time is very different for the different optimization levels. Without optimization O0 the overhead of the SFI variant is 143% for the time optimized variant O3 the overhead is 82% and for space optimized Os variant the overhead is 39%.

2) bitcount: The bitcount large example from MiBench [14] was selected because it is not as memory bound as qsort. The bitcount test uses different algorithms for calculating the number of bits in an integer. As bit counting is very fast every algorithm which is represented in an own function is called in a loop multiple times. This means some SFI overhead for calling a function and for returning from a function. Also some of the bit counting algorithms rely on lookup tables in the main memory which means memory access which has an overhead in SFI. The highest percentage of binary size overhead has the not optimized variant with 81%. Nevertheless, the differences between the variants are not very high as the speed optimized variant O3 has 76% and the size optimized variant Os has 77%. One reason for the high overhead are the high number of small functions, which all need a safe return. Still the number of instrumented data accesses is in every optimization level more than twice as often used as the instrumented jump to a register.

The time is not as probably expected because in the case of the optimized binary size Os the SFI variant is nearly 9% faster than the non SFI variant. This could be because of a better cache behavior resulting from inserted nosps. Still the both fastest variants are the O3 variants without SFI and with SFI which has a run time overhead of 35%.

V. RELATED WORK

Small fault domains in general can be achieved by sandboxing [18]. Sandboxing means that the untrusted application can only access restricted parts of the whole system to increase the safety and security of the system. It can be implemented in different ways for example with virtual machines (VM) or Software Fault Isolation (SFI). Process VMs can be implemented very efficiently also for small embedded systems which was demonstrated with Maté [19]. Another possibility which would not introduce any run-time overhead is proof-carrying code (PCC) [20]. This means that beside the code an encoded proof is delivered to the run time environment, which can check if that proof holds and if it fulfils specific requirements. Compared to PCC and VMs the approach of SFI which was introduced 1993 by Wahbe et al. [1] is simpler. The specific solution targets for example MIPS hardware but not ARM. Still some general concepts were used from that approach.

One more modern approach used by Google for their Native Client [7] to execute non trusted binaries is also based on SFI [2]. The overhead is as low as 5% for the ARM architecture. However, the system requires virtual memory and
so the usage of a MMU. In addition, it does not target small embedded systems. There are also small SFI systems like ARMor [13] which are fully verified and optimized for the use in an embedded system. ARMor guarantees for example that all memory accesses are in a certain predefined area and that the program counter only goes to instructions in the predetermined control flow graph (CFG). ARMor uses the link-time rewriting framework DIABLO [12] for changing the binary and afterwards the higher order logic framework HOL [20] for verifying this. The verifying process is more complex than a simple online validation like in embSFI. Therefore, the verifying is done on separate machine and can take several hours.

Control Flow Integrity (CFI) can be enforced using methods similar to SFI and can be extended to full SFI solutions like shown by Abad et al. [22]. The system XFI [23] shows an approach for x86 systems. BGI [24] shows that such isolation like shown by Abad et al. [22]. The system XFI [23] shows an isolation that all memory accesses are in a certain predefined area and afterwards the higher order logic framework DIABLO [12] for changing the binary and afterwards the higher order logic framework HOL [20] for verifying this. The verifying process is more complex than a simple online validation like in embSFI. Therefore, the verifying is done on separate machine and can take several hours.

VI. CONCLUSION

As demonstrated by embSFI, reducing the fault domains with SFI is achievable in embedded systems, including fast online validation. This also increases the security of systems because different separated software components cannot read private memory of other components. Even when a MMU is available, there are multiple reasons to prefer embSFI if a small run-time and memory size overhead is acceptable. In addition, embSFI is easy extensible. For example, if a software component should only be allowed access one bit directly in a hardware mapped memory this is achievable with SFI and not with an average MPU. The performance analysis of such an adoption for embedded ARM is left to future research. To create a secure system, the embedded validator must be implemented correctly. To guarantee that it would be necessary to proof the validator for example using HOL [21]. If this is achievable with acceptable effort needs still to be researched. The adoption and performance of embSFI to ARMv7 thumb and ARMv8 A64 instructions needs to be evaluated in the future. However, without these extensions embSFI can increase the dependability and security in embedded systems with a small effort and is easy to integrate in already existing C solutions, which consist of separable software components.

REFERENCES


CAML: Machine Learning-based Predictable, System-Level Anomaly Detection

Jiguo Song*, Gerald Fry†, Curt Wu†, Gabriel Parmer*
*The George Washington University
{jiguos, gparmer}@gwu.edu
†Charles River Analytics
{gfry, cwu}@cra.com

Abstract—Security challenges are increasing in distributed cyber-physical systems (CPSs), which integrate computation and physical processes. System security is complicated by both the temporal and safety constraints of CPSs. In this paper, we investigate the potential for using system-level anomaly detection in a component-based RTOS to detect system compromises and aberrant behavior. We investigate a machine learning-based anomaly detection framework, CAML, which monitors for and identifies cyber attacks in system-level services within bounded time. We leverage past work in system fault recovery to predictably recover the system to an uncompromised state. We also evaluate the effectiveness of CAML in an avionics simulator-based CPS environment with injected cyber attacks. Our results and analysis indicate that CAML has promise to effectively enhance CPS robustness by securing the underlying RTOS against system-level cyber attacks with only small performance degradation.

I. INTRODUCTION

Cyber-physical systems (CPSs) are integral to many domains such as health care, military, and industrial control. Due to strict timing constraints and limited resources, they must carefully balance between cost, size, weight, energy and safety. Threat and vulnerability mitigation is emerging as an extremely important concern regarding CPS design, as these systems often interact with the physical world through the integration of computation, actuation, sensing, and physical processes. The threats in CPSs are not only from the design errors or software bugs in the increasing complexity of system, but can also be from malicious attacks against the system by attackers who intentionally want to create chaos. Much research [1][2] has been done in the past to address different kinds of cyber-attacks, including deception attacks, DoS attacks, and even physical attacks against the actuator or plant. It is important for safety critical CPSs to continue to function even under the presence of malicious attacks. CPSs also must often meet temporal constraints and defend against cyber-attacks that can cause deadline misses to maintain the system correctness (i.e., avoid unbounded priority inversion). An intrusion detection system (IDS) is an effective way to detect the malicious attacks in the embedded system and respond without missing deadlines, before the control system is affected leading to physical system failure.

Unfortunately, the security of the system can also be compromised due to vulnerabilities within the underlying real-time operating system (RTOS), on which the application-specific components of the control system typically rely. For example, if a malicious attacker compromises the RTOS scheduler, erroneous behavior in the control system might lead to physical system failure (e.g., a vehicle could lose control of its brakes and fail to stop before an accident occurs).

In this paper, we focus on how to detect and recover from cyber-attacks in low-level system services by the malicious adversary, while continuing to maintain operationally correct system behavior within bounded time. The contributions in this work include:
- a machine learning-based, predictable anomaly detection framework for cyber-attacks in system-level services, and
- an evaluation of the effectiveness of fault detection and recovery using a flight simulator under the presence of injected attacks in a component-based RTOS.

II. THREAT MODEL

A typical cyber-physical system (CPS) consists of physical, control, actuation and sensing subsystems. The computing stage in the control system performs processing on the collected data and calculates the control commands that can be sent to the actuators to manipulate the physical system. A Real-Time Operating System (RTOS), on which the application (e.g., a PID controller) relies, is often used in the computing stage. Figure 1 shows a CPS that uses a component-based RTOS in its computing stage, which contains system-level components such as the scheduler, memory management, file system (FS), synchronization, event management, and timer. The PID controller, as an application component, communicates with the physical system through the network component (e.g., processes sensor data and sends control signals to a remote avionics system).

A malicious adversary can use various techniques to attack the CPS at any stage. For example, the adversary might intercept the information over the communication channel, compromise the key, forge messages to the operator, or prevent normal requests from being processed, causing Denial of Service (DoS). The adversary can also attack services within the RTOS. For example, in Figure 1, a PID controller periodically receives incoming data from the sensing subsystem and needs to access the FS to log the information, before it can send the computed control signal to the remote avionics system. A malicious attacker, Noise (shown in the figure as a red demon), could compromise the FS, resulting in corrupted information, or could delay the delivery of actuation control commands. For example, Noise could delay the execution in the FS (e.g., by executing an infinite loop) to prevent other

*This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675, ONR Award No. N00014-14-1-0386, and ONR STTR N00014-15-P-1182. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.
applications from accessing the FS service, causing deadline misses. The attacks in system-level services could eventually cause the physical system to deviate from its expected behavior, which could result in a catastrophic failure of the safety-critical system. To address this problem, we propose CAML, a machine learning-based intrusion detection infrastructure that focuses on predictably detecting and recovering from system-level anomalies due to cyber-attacks in a component-based RTOS. CAML monitors the system control flow and timing behavior using extensions to our previous work on C’MON [3] (Section IV), uses machine learning algorithms to detect anomalies (Section III), and uses C$^3$ [4] techniques to predictably recover system-level services (Section IV). Though our initial analysis focuses on attacks that affect timing behavior, potentially leading to physical system failure (e.g., an aircraft crash), future work will expand the detection capability of CAML for other types of security threats.

III. MACHINE LEARNING ALGORITHMS

In this section, we describe two unsupervised learning algorithms used for CAML to identify system anomalous behavior: algorithmic sequence learning (ASL) and a density-based clustering algorithm, density-based spatial clustering of applications with noise (DBSCAN).

Algorithmic Sequence Learning. The ASL algorithm is based on the episode mining technique in [5]. ASL can be applied to temporal, sequentially ordered datasets to predict events or describe frequent patterns. ASL identifies patterns in sequences based on frequency and consistency by examining all sub-sequences of events within a specified window and enumerating associative IF-THEN rules that map antecedents to consequents (i.e., sub-sequences of events that are likely to co-occur). The algorithm calculates the confidence (probability) that the sequence identified by the THEN-clause will follow the observation of the sub-sequence identified by the IF-clause. For example, in a component-based RTOS, if the sequence of components that a thread invokes follows a pattern, then ASL generates rules that predict that sequence, given an observed subsequence. If a component is compromised (e.g., an attacker causes the FS component in the sequence to invoke an additional component before returning), then these rules will be violated. Parameters to the rule generation algorithm include minimum support and minimum confidence thresholds. These parameters affect the number of resulting rules and their confidence. Minimum support specifies the minimum number of times a pattern must occur for it to be of interest, and minimum confidence specifies the minimum percentage of the time that the consequent is observed given that the antecedent is observed. ASL generates association rules which can then be used as conditions that incoming events are tested against. If an event sequence is recorded that matches an antecedent, but the events that follow do not match the rule’s consequent, then we flag this as a potential anomaly. The likelihood that this sequence of events is actually anomalous is proportional to the confidence level associated with the rule: the greater the confidence of the rule, the stronger the expectation that it will hold, and the greater the likelihood that a deviation represents a significant anomaly. The frequency of false positives can be controlled by only selecting rules that meet a certain minimum confidence level; this avoids problematic situations where, for example, a rule that holds only 60% of the time (has relatively low confidence) would identify anomalies the remaining 40% of the time. We extended the original ASL algorithm to detect novel events. Our updated ASL algorithm adds a rule whenever a new event is detected that includes a novel event. For example, a thread invokes the FS component for the first time. These novel rules are somewhat different from the standard rules because they are violated immediately; however they are still subject to the same thresholds (i.e., frequency and support), so multiple violations may be required to trigger an anomaly detection. We call them “null rules” because the rule would take the form of \( \{X\} \) implies \( \{null\} \), where \( X \) is a novel event. The “null rules” ensure that we have a way to track novel events and trigger notifications when the frequency and support of novel events exceeds thresholds.

Density-based Spatial Clustering of Applications w/ Noise. Another algorithm, DBSCAN, finds clusters based on how densely packed the data points (i.e., events like component invocations with time stamps) are [6], and we have successfully applied these algorithms to detect anomalous behavior in sensor networks. One key advantage of density-based algorithms over a more traditional k-means clustering approach is that the number of clusters does not need to be defined a priori. A cluster is also defined by local relationships between data points, so its overall shape can be quite irregular provided its constituent data points satisfy the density requirements. Density-based clustering can be applied to continuous multi-dimensional data and is especially well-suited for data that
does not conform to a regular distribution. For example, the execution time of different threads in a specific component can form a cluster even if the time does not follow any known distribution. There are two primary parameters in the algorithm can be defined by the user: \( \varepsilon \), which defines the neighborhood radius around a point for grouping points into a cluster, and \( \minPts \), which defines the minimum number of points to form a cluster. In the detection model, data points that cannot be classified into any existing cluster are labeled as anomalies. DBSCAN checks each event (data point), and if more than \( \minPts \) exist within a \( \varepsilon \)-neighborhood, then a normal cluster can grow by collecting reachable points. After all data points are considered, the events that could not be assigned to any cluster will be treated as anomalous events.

IV. System Design

In this section, we will first briefly review COMPOSITE and C’MON [7], upon which CAML is built. Then we will discuss the CAML infrastructure.

A. COMPOSITE and C’MON background

CAML is built on top of the COMPOSITE component-based OS. Components in COMPOSITE are user-level, hardware isolated (via page-tables) code and data that implement some functionality and export an interface of functions through which other components can harness that functionality. Components implement system policy and low-level services such as scheduling, physical memory management and mapping, synchronization, and I/O management, as shown in Figure 1. C’MON [7] is a system-level latent fault monitor located between the kernel and rest of system components. It tracks system-level communication events and timing, such as invocations (IPC) between components, interrupts and thread dispatching. By harnessing the event logging interface code, C’MON enables monitoring of all interactions between each component and the rest of the system beyond.

B. CAML infrastructure

CAML extends the C’MON event logging infrastructure and focuses on more general system-level anomaly detection by incorporating additional event pre-processing and machine learning techniques. As Figure 2 shows, CAML consists of three components: the event logging component, SMC, the event multiplexing component, EMP, and the machine learning component(s), ML.

SMC component. The System Monitor Component (SMC) tracks and logs system events. It does so by maintaining per-component shared ring buffers with all other system-level components. As ① shows, when the system executes and components communicate, the communication events are published into these buffers as raw events that include the event type, the cycle-accurate time stamp, and thread and component information. A task, \( \tau_{smc} \), in the SMC then copies all raw events into a large buffer shared between the SMC and EMP components (as ② shows). Copying events happens either periodically or when any per-component shared buffer is full (defined as the time-trigger activation and buffer-trigger activation in [7]). The EMP component pre-processes and aggregates raw events into more abstract typed event streams that are fed into different machine learning algorithms. EMP-based event pre-processing logic reduces the code complexity and the memory footprint of the SMC component. It also avoids running complex computation at a high priority level.

EMP component. A task, \( \tau_{emp} \), executing in the EMP component is responsible for multiplexing the raw events in the large shared buffer. The task, \( \tau_{emp} \), periodically consumes and processes the raw events in the following steps: (1) examine the raw event; (2) extract the desired information (i.e., thread and component identifier, order and location of the events, etc.); and (3) aggregate the events into multiple, different event streams. Event streams are categorized as per-thread or per-component, and they provide an abstraction of the system behavior and its execution characteristics. For example, an event stream can be a sequence of component invocations and return actions with timing information for a particular thread. An event stream can also be specified within a time window of fixed length. For example, a stream could provide the number of times that any thread is interrupted in a specific component within a period of time. Event streams will be copied into per-stream buffers shared between the EMP component and a set of ML components, and are eventually used as the training data for machine learning algorithms (as ③ shows).

ML components. A periodic task, \( \tau_{ml} \), runs a machine learning algorithm to detect anomalies (as ④ shows). Note that different event streams can be directed to the different
ML components. To learn the pattern, $\tau_{ml}$ extracts the high-level system execution features from event streams and builds a set of rules. The violation of the rules will be identified as anomalies in the system using machine learning algorithms. The execution of the tasks, $\tau_{emp}$ and $\tau_{ml}$, could be incorrectly scheduled or delayed if an anomaly is presented in the scheduler. Therefore, $\tau_m$ must ensure that $\tau_{emp}$ and $\tau_{ml}$ can be activated correctly and on time. CAML supports recovering the affected system service (e.g., faulty scheduler, memory manager, and/or FS) using C³ [8]. One challenge is to accurately localize the faulty component before the proper response can be made (e.g., recover the affected service or quarantine the malicious user). The localization process must determine which system service is actually affected and the identification of the faulty service must be as accurate as possible to reduce the false positives/negatives and their impact on system-wide schedulability. A detailed approach to fault localization is a work in progress.

V. SYSTEM TIMING ANALYSIS

In this section, we will present the response time analysis for CAML and show that CAML can effectively detect and recover from system-level anomalies without missing deadlines.

A. Response Time Analysis (RTA) Model

The tasks, $\tau_m$, $\tau_{emp}$ and $\tau_{ml}$ are responsible for logging events, multiplexing events, and running the machine learning algorithm, respectively. The tasks’ periodicities are denoted as $p_m$, $p_{emp}$, and $p_{ml}$, and the tasks’ WCETs are $e_m$, $e_{emp}$, and $e_{ml}$ respectively. Note that $p_m < p_{emp} < p_{ml}$ < periodicity of other tasks in the system. We define $B_{smc\rightarrow emp}$ as the shared buffer between the SMC and EMP components, and $B_{emp\rightarrow ml}$ as the shared buffer between the EMP and ML components. When a per-component event logging buffer is full, an invocation must be made to the SMC component to start consuming the raw events. The worst-case cost of this activation, $INV_m$, and the maximum number of such activations, $M_m$, are both defined as in [7]. The overall structure of the RTA is based on a recurrence that finds a fixed point less than the task’s deadline. If no fixed point is found, the task is not schedulable:

$$R_{i+1}^{n} = R_{i}^{n}(RTA) + R_{i}^{n}(C^3) + R_{i}^{n}(MON) + R_{i}^{n}(EMP) + R_{i}^{n}(ML) + R_{i}^{n}(F)$$

where $R_{i}^{n}(RTA)$ is the traditional response-time analysis [9] and $R_{i}^{n}(C^3)$ is the contribution from the C³ fault recovery [8]. The contribution from $\tau_m$ logging and copying events in the SMC component is given as

$$R_{i}^{n}(MON) = \left[ \frac{R_{i}^{n}}{p_m} \right] (M_m \times INV_m + M_{emp} \times INV_m + e_{emp})$$

where $M_m \times INV_m$ is due to the buffer-triggered activation [7], and $M_{emp}$ is defined as $\frac{1}{\sum_x B_{smc\rightarrow emp}^{nosync} B_{emp\rightarrow ml}} - 1$. $B_{nosync}$ is the buffer for a component, $c_x$, for which no synchronous buffer-triggered activation could arise. The term, $M_{emp} \times INV_m$, represents when the $B_{smc\rightarrow emp}$ is full, in which case we must switch to $\tau_{emp}$ and let it consume the events. The last term, $e_{copy}$, is the total overhead of copying events from the SMC per-component buffers to the buffer $B_{smc\rightarrow emp}$ within $p_m$. This overhead is proportional to the number of events generated in $p_m$. The contribution due to $\tau_{emp}$ multiplexing events in the EMP component is given as

$$R_{i}^{n}(EMP) = \left[ \frac{R_{i}^{n}}{p_{emp}} \right] e_{emp} \tag{3}$$

where $e_{emp}$ includes the overhead of processing events and creating event streams. This is proportional to the number of events generated within $p_{emp}$. The contribution due to $\tau_{ml}$ running the machine learning algorithm is given as

$$R_{i}^{n}(ML) = \left[ \frac{R_{i}^{n}}{p_{ml}} \right] e_{ml} \tag{4}$$

where $e_{ml}$ is the total overhead of consuming event streams and detecting the anomalies. Though the size of the $B_{emp\rightarrow ml}$ could affect the system schedulability, to make the analysis simple we make an assumption: the size of the $B_{emp\rightarrow ml}$ is sufficient for holding all streamed events. This is a reasonable assumption given the event streams contain the aggregated information such as the thread WCET in the component, the execution time since last thread activation, and component invocations. Compared to tracing system-wide events, the aggregated information requires much less memory space. The contribution due to anomaly localization and wasted computation is given as

$$R_{i}^{n}(F) = \left[ \frac{R_{i}^{n}}{p_{mt}} \right] (e_{ml}(localization) + p_{ml} + w_f) \tag{5}$$

where $e_{ml}(localization)$ is the fault localization overhead. The sum of $p_{ml}$ and $w_f$ is the wasted time due to the anomaly since there is a period of computation within the faulty component which cannot be trusted. This wasted computation is the period of the machine learning task, $p_{ml}$ plus the WCET of the system-level component, $w_f$.

B. Schedulability Evaluation

We conducted schedulability evaluation in a system with 25 components and 50 tasks with average periodicity of 100 ms and an attack that occurs every 500 ms. Utilization and schedulability are both represented as percentages. The memory constraint is relaxed by assuming that there is enough memory to hold all logged and streamed events. Figure 3 shows how the CAML infrastructure affects system schedulability while varying the event rate (evts/ms), which is defined as the total number of events that occur in one millisecond. The green line is the reference system without CAML. The red, blue, and black lines show the schedulability of the system with event rates of 200 evts/ms, 400 evts/ms and 800 evts/ms, respectively. A higher event rate has more impact.

1We assume that the anomaly can be detected within one $p_{ml}$. To relax this assumption, Eq.5 must be modified to accommodate the situation in which multiple $p_{ml}$ are required for detection.
to the simulator to adjust the aileron positions of the aircraft to achieve and maintain a fixed target heading.

B. Experimental results

To analyze the effectiveness of the machine learning algorithms for anomaly detection, we tested them on the sequences of events that correspond to the observed sequences of component invocations by the autopilot thread. We generated rules based on event sequences under normal system behavior. We then applied these rules to the system under faulty behavior, during which we introduced a malicious call into the FS component that the component which implements the autopilot PID controller needs to access.

VI. EXPERIMENTS

A. FlightGear Simulator

To simulate the CPS, we used a physics-based flight simulator, FlightGear (http://www-flightgear.org/). FlightGear includes a visual interface and multiple flight dynamics models for realistic simulation of aircraft interacting with the physical environment. In our experiments, we simulate a Cessna 172P Skyhawk (1981 model) taking off from a landing strip with minimal pilot interaction. The simulation environment supports remote interaction with custom software communicating over network connections. We use this communication feature to implement an autopilot program that executes on a machine running our CAML framework on top of COMPOSITE on an Intel i7-2760QM (2.4 Ghz with only single core enabled). The autopilot program communicates over TCP connections with the remote simulator, periodically receiving messages at a rate of 3 Hz from the simulator containing data about the current status (e.g., heading) of the aircraft. Based on the information it receives, the autopilot sends commands back to the simulator.

<table>
<thead>
<tr>
<th>DBSCAN cluster radius</th>
<th>Anomalies in normal scenario</th>
<th>Anomalies in faulty scenario</th>
<th>Anomaly rate increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.015</td>
<td>102</td>
<td>123</td>
<td>21</td>
</tr>
<tr>
<td>0.02</td>
<td>42</td>
<td>90</td>
<td>114</td>
</tr>
<tr>
<td>0.025</td>
<td>58</td>
<td>73</td>
<td>26</td>
</tr>
</tbody>
</table>

TABLE I: DBSCAN anomaly results
Table I shows the aggregate anomaly scores for each radius and scenario with the anomaly rate increases in the presence of fault. 7 of the 10 radii showed no statistical difference. The radii (0.015, 0.02, 0.025) showed some differences, and we repeated experiments with these three radii two more times to determine its effectiveness at anomaly detection.

![Histogram of detection algorithm execution time](image)

**Fig. 6:** Machine learning execution time comparison

To assess the use of the ML algorithms in a real-time system, **Figure 6** shows the histogram of how task $\tau_{ml}$'s execution time varies for two different machine learning algorithms. The results show that the execution time of the DBSCAN algorithm is mostly distributed between 0.1–0.3 ms, and the ASL algorithm is spread between 0.3–2.7 ms.

<table>
<thead>
<tr>
<th>Number of Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE II:** Task period and measured execution time

<table>
<thead>
<tr>
<th>period (ms)</th>
<th>$\tau_m$</th>
<th>$\tau_{emp}$</th>
<th>$\tau_{ml}$</th>
<th>$\tau_{noise}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec time</td>
<td>event copy</td>
<td>0.00002</td>
<td>110</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>event process</td>
<td>0.0002</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3 (ASL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\tau_{ml}$ recover the system with the setup from Section VI-A and the system parameters from Table II. The normal behavior of $\tau_{pid}$ is shown in Figure 7 from the beginning to 27s. The moment that $\tau_{noise}$ is activated and starts delaying itself by spinning 500 ms in the FS component is depicted as “fault” with a red arrow in Figure 7. The execution time of $\tau_{pid}$ increases quickly and results in unbounded priority inversion (shown as the green line) without proper detection and recovery. In contrast, the system with CAML successfully identifies the anomaly (e.g., the execution time in FS has deviated from its normal pattern and breaks the established rules). CAML recovers the FS from the attack and $\tau_{pid}$ quickly returns to its normal execution without missing deadlines.

**VII. RELATED WORK**

Operating System Monitoring and Intrusion Detection. There has been much work done on OS monitoring in the past. For example, [10] [11] [12] are the tools to trace timing of execution within monolithic operating systems. OS intrusion detection through system call patterns has been studied in [13], [14], and virtual machine introspection for intrusion detection is studied in [15]. CAML differs from these related efforts in that it applies statistical learning methods to analyze system execution behavior in a component-based RTOS and focuses on enabling the system to be resilient to cyber attacks in low-level OS services without missing deadlines.

Machine learning-based anomaly detection. There have been several efforts that have applied software-based machine learning or statistical methods to detect anomalies and/or failures in general purpose and real-time systems. In [16], a probabilistic model-driven approach is used to detect intrusions in CPSs and to develop mitigating responses to malicious attacks. A sequence matching approach is used in [17] to detect anomalous user behavior in UNIX-based systems. The authors in [18] investigated a hardware implementations of a Support Vector Machine (SVM) and clustering algorithms for anomaly detection in NoC-based systems. The work most closely related to ours is [19], in which the authors proposed Real-time Calculus [20]-based inter-arrival curves and applied a semi-supervised sliding window-based classification technique on a sequence of events for anomaly detection. In contrast, CAML detects and recovers from cyber attacks on software components that perform OS functions with real-time constraints. The algorithms (ASL and DBSCAN) used for CAML are unsupervised. CAML extracts high-level features from the event traces with cycle-accurate time stamps, which allows the events (i.e., event streams for the thread or component) to be associated with the timing information for more effective anomaly detection.

**VIII. CONCLUSIONS**

In this paper, we presented CAML: a machine learning-based anomaly detection framework that predictably monitors, identifies, and recovers from cyber-attacks in a component-based RTOS. We evaluated the effectiveness of CAML using two unsupervised machine learning algorithms to detect injected attacks and showed that CAML shows promise for enhancing system resilience to cyber-attacks in system-level services without missing deadlines.
REFERENCES


Abstract—The recent extensive development in Cyber-Physical Systems (CPSs) has lead to the emergence of new concerns regarding timeliness, safety and security properties. For decades, numerous vulnerabilities have put systems and applications at risk and CPSs are no exception. Noteworthy recurring issues are, for example, Buffer Overflows (BOs). We intend to deal with some types of BOs, other accidental faults and intended attacks by means of Non-Intrusive Runtime Verification (NIRV), to be accomplished through the design of a black-box observer and monitoring entity. Tackling security hazards can be enforced at different levels or granularities depending on how detailed our knowledge of the inner workings of the system and applications running on it is. We introduce solutions to detect and handle explicit attacks and accidental faults, focusing on completely null understanding of the analyzed environment’s specificities, but also discussing scenarios where program mechanics and engineering are completely known.

I. INTRODUCTION

The field of cyber-physical systems has been identified as a key area of research and it is part of contemporary technologies that are, themselves, receiving attention for their innovative nature, such as smart grids, autonomous mobile systems, medical monitoring and robotics.

Due to their criticality, these systems are a target of multiple concerns, namely in the timeliness, safety and security domain. Vulnerabilities are relevant security defects, being, as such, an open door for intentional attacks and accidental faults. The large majority of Operating Systems (OSs), libraries and applications possess plenty of vulnerabilities and their exploit or even just presence may place a system at a risky or erroneous state and jeopardize its operation at multiple levels, for example, forcing it to miss deadlines, writing outside the allocated address space, corrupting data, etc.

At the same time, some of these vulnerabilities have persisted for years without being fully dealt with, namely Buffer Overflows, which have not ceased being one of security’s major problems, affecting all kinds of systems. As such, since cyber-physical systems are not an exception to the rule, we wish to bring forth ways to enforce their safety and security.

Our approach to the problem is plainly naive. We simply assume most (if not all) applications are made up of function calls and from there we elaborate a scheme to verify if each function is executing correctly and that no violation to the specification has occurred, namely with respect to the assigned addressing spaces, at function-level granularity. The solution relies on non-intrusive runtime verification, comprising of: observation, for capturing events; and monitoring, to perform their analysis and evaluate correctness. We focus on situations where practically nothing is known about the system and it’s applications, zero-knowledge environments, but also providing a subsection on the broader types of faults that could be detected in more detailed scenarios.

Guaranteeing safety and security properties non-intrusively, at runtime and with no knowledge of the system, libraries and/or applications being monitored presents some challenges, namely in the types of faults and attacks the observer is able to detect and at which granularity.

Existing non-intrusive solutions, either require changes in the system architecture, storage of observed events for offline analysis (which is not ideal, specially for storage limited systems, a characteristic of most CPSs) or are incapable of performing monitoring in, for example, zero-knowledge environments, where access to source code is absent and binary files are stripped of symbolic information.

The contribution of this paper is the provision of the initial ideas to enforce safety and security in zero-knowledge environments, enabling some common attacks and unintentional faults to be prevented, to will be implemented in the near future.

II. SYSTEM MODEL

Our solution to enforce safety and security in cyber-physical systems is inserted in the realm of reconfigurable logic. By taking advantage of System-on-a-Chip (Soc) architectures, we are able to perform observation directly in hardware, at low level, rejecting techniques such as code instrumentation and, thus, eliminating its associated overhead and effort. Function grained code instrumentation is difficult using only binary files, as disassembly operations would be required offline. On the other hand, system calls and well defined execution points (such as read/write of input/output ports) could be intercepted online. Figure 1 depicts a generic SoC architecture.

A. Assumptions

The system model was conceived taking into consideration a set of assumptions.
First, as previously mentioned, we assume all applications consist of function calls. Executing function calls pass their parameters to the called function, save their return address and allocate memory space for the local variables.

Secondly, we assume the observer can always be inserted in the SoC platform and connected to the SoC bus. Then, as already mentioned, we assume, for the scope of this paper, that the observer is inserted within a zero-knowledge environment, where no access to source code or binaries (specially non-stripped ones) is given. However, we briefly describe in subsection V-D what other faults could be detected in case there is more information available in the system.

Moreover, the cache needs to be write-through, meaning data is written into the cache and the corresponding main memory at the same time. If information is only written into the cache than it is of no interest for the observer since it is not capable of accessing the Processing Element and, thus, will not be able to detect any errors.

Finally, in case the observer is faced with binary-based symbolic information that allows finer granularity observation, then we consider the ELF (Executable and Linkable Format) format is used.

B. The Observer Entity

In order to perform the observation and monitoring of the system and applications running on it, an Observer Entity (OE) component will be inserted onto the SoC platform. It will consist of a black box, connected to the SoC Bus in the same fashion as the other components present in Figure 1, such as the Interrupt Controller. Throughout the paper, observer and observer entity will be used interchangeably.

The observer entity will be specified in VHDL (VHSIC Hardware Description Language), a language used to describe digital and mixed signal systems, such as Field-Programmable Gate Arrays (FPGAs), containing an array of re-programmable logic blocks.

The reason why the observer needs to be a black box, that is, be viewed in terms of its input but not its internal functioning, relates to the necessity of maintaining the observer entity’s inner workings occult to prevent malicious entities from hijacking its functioning. Furthermore, it has to be coupled directly to the SoC bus, since inserting the observer inside the processor would require its modification, which is not always possible, feasible or desirable. Also, changes in the processor would possibly mean changing the observer as well. As it is, we are not concerned about changes in the individual SoC components, only with specificities in the Instruction Set Architecture (ISA), given that the observer operation will be ISA-dependent. The reason behind this matter has to do with the fact that different ISAs treat function calls, data storing and endianness differently, among others. They also differ in the set of available instructions. Therefore, the observer needs to be adaptable to the different ISAs.

The Application Binary Interface (ABI) defines the low-level binary interface between two or more pieces of software on a particular architecture, that is, how an application interacts with itself, with the kernel and with libraries. It also normalizes how function calls, argument passing, results return and stack space allocation are performed, including layout and alignment of data types. So, because some ABI decisions are done based on the ISA, the observer will possibly need to be ABI-dependent as well.

Figure 2 represents, once more, the SoC architecture, but now with the observer entity connected to the bus. This OE includes observing capabilities as well as verification against a specification for the purpose of discovering if any violation occurred.

C. Monitoring and Fault Detection

Faults can be tackled at a coarse or fine grained degree depending on the knowledge level. By knowledge level we mean how much is known about the system, libraries and applications we want to monitor. Say, if one was to build his own system, running its own applications, monitoring could be done at a really fine grained degree due to the huge level of detail, such as access to source code and binaries’ symbolic information and even the familiarity with the types of input and output each function is supposed to receive and return, respectively.

However, such scenarios are unrealistic, usually that level of detail is not present or is inaccessible. Binaries are commonly
stripped to reduce storage requirements, to prevent reverse engineering or to resist analysis. Moreover, normally we would not even be able to access these files in order to perform disassembly operations on them aiming to extract variables’ memory location, check performed system calls, etc. Therefore, a final assumption dictates the observer architecture is based on the premise that nothing is known, that is, that we operate in a zero-knowledge environment.

Now, considering such a scenario, the types of errors we can detect are somewhat limited to generic vulnerabilities and the set of attacks that can exploit them. Then, the observer will be able to detect anomalies such as writing to read-only memory and some types of Buffer Overflows, such as writing outside of the space reserved for a specific application, i.e., outside the data section or, more specifically, outside the bss (Block Storage Space) section and most notably function-grained stack frames. The data segment is a read-write chunk of a binary file, or the corresponding virtual address space of a program, that contains initialized global variables and static local variables. On the other hand, the bss section, also known as uninitialized data (or content-less section), is usually adjacent to the data section and contains all global variables and static local variables that are initialized to zero or do not have explicit initialization in the source code.

It is possible to detect this sort of faults since we are dealing with well delimited memory zones. In sum, the observer will detect errors at function-level granularity for these described zero-knowledge environments. If more is known about the applications being monitored, then detection can be done at a much finer granularity (and, for example, the great majority of BO types could be detected).

It is also viable to detect when a malicious entity is trying to alter some data on the registers. Normally the function call should be responsible for writing the necessary data, such as input parameters, in conformity with the ABI specification, which is well known for each system. Some architectures avoid using the stack and store as much data as possible in the registers until eventually they run out of registers and the Operating System proceeds to reclaim them. As such, through SoC bus observation, we can detect when any other instruction is trying to write to them. Readings are less important, since they will not alter the data; they might, however, be relevant for confidentiality reasons. With this, we can detect attacks such as return pointer substitutions.

1) Buffer Overflows: As verified by [1], Buffer Overflows are still a major concern nowadays. A BO is an anomaly where a program, while writing data to a buffer, overruns the buffer’s boundary and overwrites adjacent memory locations, such as the situations described above. Due to the limited knowledge scenario and importance of such vulnerabilities, we would like to initially focus our efforts on BOs, even if restricted to more generic variants of this vulnerability like, as stated before, writing outside some well-known memory zones like the data and bss section and, most notably, function-grained stack frames. In a zero-knowledge environment it is not possible to detect variable-level BOs, since we do not possess detailed information regarding the specific memory placement each one of those variables. We do not refrain from reminding that, given more details about the system and applications, the observer is still capable of detecting application specific faults.

Finally, we also intend to protect against Denial of Service (DoS) attacks. Subsection V-B yields greater details on how to detect them. [2] presents an architecture plus a compiler-driven solution to tackle both Buffer Overflows and DoS, which we will later analyze.

As for now, we solely intend to address applications running natively in the OS. We leave for future work to approach applications that require the use of Virtual Machines to run, such as the Java Virtual Machine (JVM).

III. OBSERVER ENTITY ARCHITECTURE

Basically, the observer can be divided into two major components, the System Observer and the System Monitor. The first is responsible for checking points of interest like a read, write or simply access operation to memory. On the other hand, the System Monitor will analyze those events so as to verify if any deviation from the expected behavior occurred, such as the ones previously described, for example, detecting write operations outside the limits of the data section. Figure 3 shows a simplified representation of the observer entity, detailing the relation between these two components.

![Simplified Observer Entity Architecture](image)

The red arrow represents a retroaction, that is, the System Monitor’s response in case it came to the conclusion that some erroneous behavior has taken place. The idea is to prevent the malicious action from doing any damage. So, our first idea was to raise exceptions, catch those exceptions and have handlers take the appropriate measures. However, handlers represent a convenient attack point, they can be replaced or modified. Some techniques can be used to tackle this issue, such as defining more than one handler, to act in case another gets attacked.

Thus, afterwards, we thought of instantly killing the process to refrain the malicious actions from occurring or propagating, in situations where there would be no serious repercussions in doing so.
The zero-knowledge environment prevents the observer from taking a more specific action to correct the incident, its reaction has to be generic. Nevertheless, killing a process directly in hardware is hard, which leads to the conclusion that using exception handlers is most likely the best option after all.

A. Extended Observer Architecture

This subsection provides greater details on the observer entity architecture and clarifies its inner workings.

On previous models [3], the observer was fed by the system clock, provided to all SoC components. However, in order to reduce the number of entry points and consequently increase its notion of a black box, an internal clock will provide time to the observer. Synchronization with the system clock may be required for precise timing of system behavior.

At first, the observer as being generic, that is, operate equally independently of the architecture. Yet, each processor or processor family has its own ISA and each ISA works differently. Thus, the observer would need to recognize specific instructions such as function calls and returns and process information accordingly. As a result, it needs to be ISA-dependent. With that said, the addition of another module to the observer architecture is in place, an ISA-dependent Call/Return Detection component. Some ABI decisions are made based on ISA specificities, which leads us to the conclusion that the observer may possibly need to be dependent on the ABI as well.

Fig. 4: Complete Representation of the Observer Entity Architecture

In addition, a History Manager needs to be employed in order to save output from the Call/Return Detection module. Possibly, the Call/Return Detection-History Manager relationship may bring scalability issues, namely regarding the number of processes and chained function calls.

Finally, the System Monitor presented in Figure 3, should have self-learning capabilities incorporated within, so that it needs no further configuration. The History Manager will provide these capabilities. The use of configuration files would be a risk, since, again, they may be replaced by malicious versions aiming to deviate its behavior. Albeit providing more flexibility, observation point extraction for configuration files is also relatively time consuming if done online. Hence, the self-learning capability will be advantageous not only from a security point of view but also time wise.

Figure 4 defines the complete observer architecture. The System Observer depicted in Figure 3 is an abstraction to the set comprising of the System Observer and the ISA-dependent Call/Return Detection, shown in Figure 4. In a similar way, the System Monitor in Figure 3 contains the System Monitor and the History Manager in Figure 4. Schematically, the observer entity is comprised of:

- System Observer
  - System Observer
  - ISA-dependent Call/Return Detection
- System Monitor
  - System Monitor
  - History Manager
- Bus Interfaces
- Observer Clock

IV. ARCHITECTURAL DIFFERENCES

We considered three distinct architectures for the observer implementation: Sparc, ARM and Intel x86. A brief analysis on the current state of these architectures was one of the motivations for the insertion of an ISA-dependent component. For example, Sparc and ARM are RISC architectures, while x86 is CISC, meaning they differ, say, in the way arguments to functions are passed. Therefore, the observer needs to be able to adapt to each of them. Also, we came to the conclusion that not all of these architectures are technologically ready for the observer concept. As of today, Sparc is the most suitable platform for implementing the observer, due to its architectural design and easiness in connecting extra components to the system bus [4].

In principle, it should be possible to use the observer entity in ARM as well [5]. Although it cannot cover all ARM-based SoC devices available, those which include CoreSight [6], a configurable validation, debug and trace component, included on ARM’s SoC are suitable for our purpose. There is the possibility to place an FPGA connected to a CoreSight output so that it would filter CoreSight’s results and act accordingly.

As for x86, a built-in Altera FPGA integration with an Intel Xeon processor has already been announced [7]. The concept was created with the intent of accelerating algorithms and taking workloads off the processor(s). The current Intel Xeon architecture does not seem to be the most appropriate for the integration of the observer allowing it to perform monitoring according to our system model. In this Intel and Altera solution, the processors and co-processor (the FPGA) are connected via a dedicated bus. Our approach depends on the observer being connected to the systems bus, or, in this case where we want to address SoC architectures, the SoC bus. Probably, we will have to wait for technology to evolve towards the observer entity’s needs. Other proposed approaches rely on a Front-Side Bus (FSB) architecture but require the substitution of the processor with a very specialized module.
V. Fault Detection

In section II we shortly described what kinds of vulnerabilities we were capable of addressing. Due to the environment characteristics, that is, the lack of understanding of the system, libraries and applications’ inner workings, the observer is solely able to perform runtime observation and monitoring at a coarse granularity. Here, we provide greater detail on some of the faults the observer will be detecting and how, as well as a small discussion on other solutions. We also provide a small subsection on how to perform monitoring at a finer granularity, provided additional information is accessible.

A. Return Pointer Access Protection

A fault easy to detect in a zero-knowledge environment is the overwriting of return pointers saved on the stack, for instance, to hijack the path of execution. [8] presents StackGhost, a hardware facilitated stack protection mechanism implemented through a kernel modification to OpenBSD 2.8, under the Sparc architecture. StackGhost intends to transparently and automatically guard function return pointers. It protects against Buffer Overflows and primitive format string attacks. In order to prevent corrupt pointers from exploiting the code, the authors suggest the use of a reversible transform to be applied to the legitimate return address and the result written to the process stack. The idea is based on the supposition that if an attacker has no knowledge of the transform or its key then execution cannot be affected intentionally. It requires bit inversion (two least significant bits) or keeping a return-address stack. Additionally, they consider encrypting the stack frame.

This solution has some drawbacks. One, even though the performance impact is negligible, less than one percent, it exists. Two, the transform can be discovered. Sparc usually stores the return address in a specific register (%i7) belonging to the register window in use. As such we propose detecting return address substitution attacks by verifying if writes to this register are being performed by something else rather than the function call instruction. Our approach has the advantage of being simpler, less error prone and more secure.

B. Denial of Service

On the other side, [2] presents a solution that isolates code and data at a function level - the granularity required in our solution - aiming to provide protection against possibly untrusted code such as plugins and open-source software. A slight downside is that it requires identification of untrusted functions or groups of functions at compile time (through the project’s makefile, for example) or load time. In addition, this solution depends on compiler support and requires processing core modification, since the authors devise architectural alterations that are to sit between the processor and the cache. These modifications also work as a basis for their Denial of Service defense. While being a minimally invasive approach, we are seeking a completely non-invasive methodology.

As such, for Denial of Service prevention, given that the observer has access to the instructions being fetched, including function calls, it has the information required to establish time limitations for function execution. This can be done without the changes or compiler support required in [2], since only temporal protection is guaranteed, independently of the function call hierarchy.

C. Buffer Overflows

Due to the lack of knowledge on applications’ contents, the observer cannot prevent Buffer Overflows such as writing outside the space allocated for a single variable, array or structure as a result of, for example, misusing functions like string copy (strcpy) in C without checking boundaries. Without symbolic information present on binary files or disassembly data extracted from stripped binaries [9] it is not possible to know the address space boundaries of a certain variable and, consequently, if that space was overflown. Thus, the observer is only capable of monitoring overflows on well delimited memory zones, like the data or bss sections and function-grained stack frames. This is the reason why we keep track of stack frames in the History Manager component.

With that, the whole stack frame of one function can be protected from can be protected from being overwritten, for example, by another function.

Additionally, it is possible to protect memory zones reserved through the malloc system call. This primitive has an argument the size of the memory buffer to be allocated and returns a pointer to the allocated memory space. With this information, the observer entity is able to verify if memory access is within the allowed boundaries. The free system call, returns the memory chunk back to the system and forbids further access from the program, which should be verified by the observer entity.

D. Full-knowledge Vulnerabilities

This subsection briefly describes what sorts of other vulnerabilities the observer could provide protection against, in case there was more information available about the system and its applications, that is, if we were in a full-knowledge environment. Here the term full-knowledge describes (significantly) more information, it does not necessarily mean we know everything about the subject under observation.

If the location and sizes of global and local variables was known, more specific types of buffer overflows could be detected and not only section/function-grained violations, as described in the subsection above. Global variables’ addresses and sizes could be directly extracted from the binaries’ symbol tables. Local variables, however, require binaries to be compiled with the right debug options so that their description can be accessed.

Also, if great details were known about the running applications, the observer would even be able to detect incorrect output and input, format string and possibly path traversal attacks, etc. So that the observer is able to perform this sort of monitoring, specific knowledge of the system is required,
which would happen if, for instance, we were the system and/or application designers or if the source code is available. Nevertheless, the bottom line is that these mechanisms can be used by the observer entity if the situation allows it. Figure 5 shows the common placement of fine grained objects of interest, like variables.

Enforcing safety and security in applications running on virtual machines will also be left for future work.

Finally, we intend to convey more attention to the observer entity’s implementation in ARM and, in case technology eventually becomes ready, in Intel x86.

VIII. CONCLUSION

Guaranteeing safety and security properties non-intrusively, at runtime and with zero knowledge of the system, libraries and/or applications we wish to monitor presents some challenges, namely in the types of faults and attacks the observer is able to detect and at which granularity. Since we assume there is no specific knowledge available like binaries’ symbolic tables or even stripped binaries on which to perform disassembly operations, we are limited to some basic but nonetheless important vulnerabilities, such as some sorts of Buffer Overflows on well delimited areas, denial of service and unwanted register access. For finer grained monitoring, extra information would need to be provided.

Thus, in sum, the observer reacts to information captured from the SoC bus, if it deems it erroneous; and functions as a non-intrusive black box, providing an extra layer of security for this reason. We are well aware that security by obscurity is not advised. Nevertheless, by omitting design details we are preventing the probability of some attacks.

REFERENCES


VI. RELATED WORK

Both offline and online NIRV approaches have been previously introduced for embedded systems [10] and cyber-physical systems [11], generally having an external infrastructure for observed information processing. Offline NIRV methodologies still have serious limitations such as being impracticable to store all the observed data over an arbitrary observation time due to the high observation data bandwidth and the discrepancy between observation data output and processing bandwidth. From these issues, online NIRV was born, bringing forth ways to process data on the fly, allowing debugging and RV without any interference, as presented in [12]. Our approach is inserted within this category and works as a monitoring and verification infrastructure. The approach in [13] also addresses both concepts. NIRV has already been used in safety-critical environments as well [14].

NIRV approaches have recently been applied to Time- and Space- Partitioned (TSP) systems to improve safety and decrease the computational cost of timeliness adaptability [15], [16].

Overall, to the best of our knowledge, OE architectures do not tend to be black boxes and generally do not comprise a self-learning module for automation and adaptation, generally relying on configuration files. Additionally, several solutions require changes in the architecture; [14] is an example of one that does not need modifications to the underlying system.

VII. FUTURE WORK

The observer will be designed in VHDL, using a Xilinx XUPV5-LX110T development board as a proof of concept prototype. Additionally, monitoring methods will be enhanced and new ones introduced, in order to detect as much deviations as possible in the described zero-knowledge environment.
**MERgE: Technology Advancement for Cohesion of Concerns in System Engineering**

Charles R. Robinson  
Thales Research & Technology, France  
firstname.lastname@thalesgroup.com

Jérôme Pequery  
Obeo, France  
firstname.lastname@obeo.fr

Sam Michiels  
iMinds-DistriNet, KU Leuven, Belgium.  
firstname.lastname@cs.kuleuven.be

**Abstract**—MERgE is a project that has been funded by the ITEA, under their “Engineering Support” roadmap, to advance technology for multi-concern interactions in system engineering. The applicability and benefits of the work has focused particularly on Safety and Security. These system properties are usually treated and certified separately, however increasing complexity requires urgently the ability to track such global qualities through the product lifecycle and consider their relationships. Driving factors here include legacy management, cost reduction, increasingly open systems and the need for resilience. This is a project paper, providing an overview and industrial return of the technology developed to better address these challenges. The work has been applied to Radio Communication, Automotive, Space and Industrial Control domains. The collaborative work has considered co-engineering, multi-viewpoint technology, expert systems, variant architectures, process enactment & recovery and operational system assessment all developed on a platform with domain tailoring capabilities.

**I. INTRODUCTION**

We are faced with increasingly complex cyber-physical systems as technology advances to meet the demands of our society. Engineering typically takes place over several phases: Requirements Gathering, System Specification, System Design, Implementation and Testing from unit to system level. Within each phase you frequently have specialists that focus on particular concerns like safety or security. These are often treated separately, sometimes expressly so, despite research for over two decades indicating trade-offs need sufficient treatment [1]. The increasing rise in digital services, complexity and open systems is rapidly driving up this priority on industrial agendas. This is also evidenced by increased research, for instance co-engineering was visible in several workshops at the SAFECOMP conference in late 2015. The challenge addressed by MERgE for over three years has been to provide improved cohesion between concerns within engineering phases, like architecting, but also between the different phases. This provides improved traceability, reduction of redundancy and better vision of the effects of decisions taken by specialists. An example of multi-concern interaction is provided in Figure 1 where concerns are separated into ‘viewpoints’. An extensive state-of-the-art of security-safety co-engineering was carried out during MERgE [2].

The MERgE Project has approached the problem of improved concern interaction from several angles. At the concept level, the analysis techniques and processes have evolved and include a proposed plan for aiding the convergence of safety and security. Tooling has also been developed, including a customisable platform integrating tool builders, modelling tools and system analysis tools. These will be discussed in more detail shortly.

Four different use case environments have been provided on which MERgE technology has been developed and tested. The Automotive and Aerospace domains have a strong safety background, with security becoming a bigger issue recently with the uptake of wireless technology. This is also the case for the Industrial Control System (ICS) domain, although this demonstrator assesses operational systems whereas the two prior domains consider the system engineering. The ICS also developed a reconfigurable physical simulator – such systems are rare, illustrated by a recent call for proposals to build such systems [3]. The Communication use case provides a complementary reflection of the other three where security is dominant but with increasing safety requirements.

The following sections present an overview of the work that has taken place during MERgE, beginning with some of the global and backbone efforts. This is followed by a focus on the four domain demonstrators and finishes with the conclusion.

**II. TOOL PLATFORM AND SYSTEM LEVEL**

A MERgE tool platform was implemented by Obeo. The platform provided a tailoring capability for users to select the MERgE tools most relevant to them. It also the capability to create new tools to fill gaps in one’s tool chain. The differing requirements of the use cases needed this customisable capability by the domain user. Eclipse provides a foundation with the tool Oomph! [4] bringing functionality to enable automatic updates from partner tools. Capella [5] from within Eclipse forms a baseline of the platform across the use cases. Capella
is a model driven engineering (MDE) toolkit recently released to the open source community by Thales. It contains core elements Kitalpha [6] (for viewpoint modelling of engineering specialties) and Sirius [7] – the tool builder allowing domain-specific MDE. This work has been realised in the MERgE Platform Configurator released at www.merge-project.eu.

At the engineering level, UPMC advanced work on process enactment across several demonstrators. Their monitoring tool provides capability, via models, to follow development teams in their tasks, identify deviations from procedures and recommend solutions [8]. It was noted that this type of technology will be beneficial particularly for development of interconnected systems and also the safe and secure operation of systems. However sufficient formalisation ‘digitally’ of processes is needed and many domains are still in an early transformation stage for describing their processes in this way. KUL and Inria collaborated on tooling to support development teams in their choice of solutions (and implications) to meet requirements in software product line engineering [9]. The solution space exploration proposed here provided benefits such as catalogues and instances management, conformance checking, synchronization and versioning.

The engineering practices were investigated by Oulu and JYU for several demonstrators. For instance, “Design for Excellence” (DfX), specifically sets out to define best practices to ensure requirements are properly managed and coordinated during the product lifecycle in the most cost-effective manner [10]. Another focus relates to “technical debt” that can accumulate during development [11]. This can occur when trade-offs are taken to minimise delivery time or development costs, at the expense of quality. Also the time between fusing work of different developers such as between safety and security has an associated technical debt – the longer the period before combining work, the higher the technical debt that needs to be treated. This debt should be managed particularly for critical systems where quality often has a higher priority than the physical cost. The next sections describe four demonstrators developed during MERgE.

III. RADIO COMMUNICATION

The backbone of safety and security for a community or system relies on communication. Thales Communications & Security (TCS) provide a use case environment for investigating software-defined radio (SDR). Traditionally, radios were defined by their hardware, usually specified for particular standards and functionality. However there is an increasing demand for the same piece of equipment to support many standards. This can be accomplished by adding more hardware to a radio but it is costly. It is a pressure point that SDR alleviates where hardware functionality is handled by software. The ability to intercept or interrupt SDR communications however, opens the door to security attacks that can have severe implications on safety. When communication is compromised life and resources have increased risk of being lost.

The uptake of SDR can be seen increasingly used in civilian and military applications. For instance, any “state-of-the-art, commercially available mobile phone now supports at a minimum several standards, including LTE, W-CDMA, GSM, Bluetooth® and WLAN” [12]. Solutions from SDR will be necessary across many domains in the near future (those of MERgE included), due to the increasing openness of systems. It is predicted by the Gartner Group there will be 250 million connected vehicles by 2020 [13] – the SDR concept is even being investigated now for extension to complete car systems – software defined cars. Both NASA and ESA are currently developing satellites to test SDR such as their respective projects NanoRacks-GOMX-3 and OPS-SAT. There is less public discussion concerning SDR in the industrial control domain, where re-configurability is kept to a minimum – but wireless is increasingly being used in this domain and SDR can offer other advantages such as through policies for segregating the different communication channels.

Through this use case TCS has investigated with partners (re-)programmable software, secure communications, open system architecture and architecture evaluation. An overview of the work is shown in Figure 2. The Comms. Demonstrator focuses on the engineering stage, where the system architecture model was described by the MDE tool Capella. A key solution here is to partition communication channels based on security and safety priorities. This allows parts of a system to be more open and their isolation prevents their influence on the more secure areas.

One axis of investigation, at the higher level of abstraction, was modelling for safety and security viewpoints. Prototyping for a security-safety co-engineering (CE) tooling solution was advanced by Thales Research & Technology (TRT), All4Tec and Onera. From the system architecture, safety and security models are applied with the tool Safety Architect (All4Tec) which provides the ability to generate fault-trees and attack trees to determine vulnerabilities. The safety/security specialists also define the interdependent failure modes. This approach was taken according to the SotA in MERgE [2], where it was believed to advance CE for these domains, the initial step should allow specialists continue with their
processes, but introduce a third 'translation' model, rather than everyone converging at once on combined definitions. The language Alloy was integrated with Safety Architect by Onera [16] to allow a safety/security specialist to provide a formal proof that their models will resist feared events.

At lower levels of abstraction there can be many more ways to satisfy the requirements for a product. For higher complexity one can find themselves faced with hundreds of thousands of potential solutions. Tooling was brought to the Demonstrator to assist the process of choosing a solution. The tool KCVL (INRIA) explores the solution space and derives architecture alternatives from the system components available. To improve speed and refine the choices provided, pattern technology (Thales Global Services -TGS) [14] was integrated with the process [15]. Patterns can be referenced from a catalogue of previous successful implementations to subsections of the current solution space. This approach was applied to the Comms. Demonstrator to consider safety and security as well as latency and cost. The patterns also considered options in terms of both physical and logical partitioning. Additionally, the Pattern Technology integrated the object constraint language (OCL). This was to provide guidance for when a model is updated to check it continues to respect safety and security requirements.

Following these steps there may still be many architectural configurations from which to choose. This is where TRT applies their Evaluation Tool [17], an expert system, to further refine the choices from the architecture variants and guide the developer to a final decision. The expert system adds a means of qualitative assessment based on a database of specialist responses to relevant choices under given conditions.

Finally, an instantiated version of the use case was subjected to brute force attacks via the tool Defensics (Codenomicon). The approach called fuzz testing sends a wide range and systematic series of unexpected or invalid inputs to the system to find vulnerabilities. The use case with the partitioned approach proved to be resilient to the tests.

IV. AUTOMOTIVE

In this domain embedded systems are being used at an increasing pace to improve functionality and reduce costs. This advancement opens up significantly more system capabilities, but also complexity. At the extreme end of complexity is the current advances towards autonomous vehicles, where functions including automatic steering on highways and parallel parking are already available in the public, for instance:

- Tesla has already up to 60,000 vehicles on the road with this functionality according to a Financial Times report in October 2015 [18].
- With the proliferation of car software and especially use of wireless technology, new vulnerabilities need to be considered for both safety and security. For instance, "Fiat Chrysler has recalled 1.4m vehicles over a vulnerability that could allow hackers to disable them on highways" reported in July 2015 [19].

The importance of managing the many engineering concerns is clear. Within this domain there has been much recent activity related to the ISO 26262 safety standard which is particularly important for implementing safety-critical functions of embedded systems. It has itself has been evolving to consider also security inter-dependence. The Automotive use case expertise has been provided by Melexis. The main focus was on multi-concern engineering for device configurability and traceability particularly for safety and performance. This development was carried out ensuring strong conformance with ISO 26262 and involvement at standardisation meetings. Within the automotive demonstrator (Figure 3), at the requirements stage there was collaboration between E2S, KUL and Inria related to the configuration and constraints of a specific product. The offered capabilities of a product are handled by the FeatureIDE tool [20] providing a high-level description with graphical models. The formalism in this tool is naturally extended by the technology Familiar [21] which eases the development of models through early alerts on inconsistencies and the number of configurations that are possible. The tool ATO by E2S [22] is also applied here and provides verification that the choices offered respect the constraints (through the object constraint language).

Related to the architecting and simulation phase, work was conducted on the auto-generation of variants and metrics. From FeatureIDE, the product configuration is converted to a suitable template that is read by the KCVL tool [23] to establish the range of appropriate architectural models. KCVL ensures the resulting choices meet required ASIL-level safety levels. The pattern technology was investigated here again with particular advantages related to consistency of design and enabling the respect for safety rules to be made more explicit.

This was followed by developing technology for the analysis of the variant possibilities and procedures for selection of the optimum choice [24]. Here ATO offers additional analysis options through generated metrics and includes timing analysis. The tool Familiar provides complementary functionality to validate the generated models. Some preliminary investigation
also took place into the use of the TRT Evaluation tool to aid with further refining the selection of optimal architecture variants.

V. AEROSPACE

The company Space Applications (SASNV) contributed their knowledge in the development and testing of the Aerospace Demonstrator. Their focus was mostly on safety and traceability concerns in development, although some aspects of security were also discussed. Security has played a lesser role in this domain and there have not been many publicised breaches in security. However the scene is changing with intrusions becoming more common:

- “Hackers from China breached the federal weather network recently, forcing cybersecurity teams to seal off data vital to disaster planning, aviation, shipping and scores of other crucial uses” reported in November 2014 [25].
- A few years ago “the Sri Lankan government noticed that propaganda for the Tamil Tigers, the rebel group, was being broadcast regionally from an Intelsat satellite over the Indian Ocean.” reported in April 2015 [26].
- A hacking group with apparent Russian government links “has been accused of hijacking vulnerable commercial satellite communications, using hidden receiving stations in Africa and the Middle East to mask attacks on Western military and governmental networks.” reported in September 2015 [27].

While the market landscape will be forced to significantly augment security measures, the means of defence will mostly filter down to SME organisations through standards.

Related to the tool chain investigated and developed (see Figure 4), at the start of the development cycle DOORS is one of the predominant tools used in the Space Industry for collecting requirements and used by SASNV. ReqIF was provided as an Eclipse plugin by Obeo to facilitate the exchange of requirements with later architecting stages. Similarly the tool CAMEO was used aiding indirect traceability via UML models.

In the architecting phase, a modelling tool used by SASNV is MagicDraw (MD) and this was linked to Safety Architect [28]. The latter tool was improved by All4Tec to support the standard exchange formats of MD, generate SFMECA reports using ECSS standards and provide failure propagation analysis based on MD models.

For the implementation and testing phase VectorCAST was evaluated and integrated providing functionality for code quality, metrics, unit testing and validation. The code generation tool of KUL was advanced to proof of concept, being able to generate skeleton structures of code based models coming from MDE.

VI. INDUSTRIAL CONTROL SYSTEMS

These systems involve a significant part of a nation’s infrastructure with environments ranging from energy to manufacturing and water treatment. Safety is a high priority however security measures are different between closed systems and ones connected to the outside world. Increasingly industrial control systems (ICSs) are becoming more open, particularly to gain efficiencies between supply chains, however frequently insufficient security measures are implemented. The vulnerabilities are increasingly being exploited:

- “Attackers successfully compromised U.S. Department of Energy computer systems more than 150 times between 2010 and 2014” – reported in September 2015 [29]. In the same period there were 19 successful attacks experienced by the National Nuclear Security Administration responsible for weapon stockpiles.
- It was reported in January 2016 [30] that in the last year cyber-attacks had doubled in the U.S. manufacturing domain.
- At the end of 2015 in the Ukraine, power outages took place as a result of cyber-attacks on the energy grid leaving 225,000 people without electricity[31].

In MERgE the ICS Demonstrator had a focus on operational systems. There were two axis of investigation. One of the scenarios developed a reconfigurable physical industrial control simulator (POHTO) while the other provided a services environment used in the domain (nSense). The technologies are represented in Figure 5.

For the operational system investigation, Codenomicon applied their tools Defensics [34] and AppCheck [35] to the physical ICS environment in order to highlight the vulnerabilities of particular setups. System behaviour was monitored and static binary analysis performed. The business process was provided by nSense related to service provision. JYU and Oulu had a particular focus here on requirements management.

The ICS use case expertise was used as an opportunity for exploring MDE. This style of system architecting and modelling of processes is only beginning to enter their domain, although it is underway (for instance at AREVA [32] and
Mitsubishi [33]. Obeo and UPMC in particular brought their competence in MDE to look at the value that may be found here. It is believed significantly improved connectivity of safety/security could be achieved if MDE was used in this domain both for system design, but also for system operation and training. However ICS vendors have a lot of autonomy and lobbying is needed for large organisations and standards bodies to appreciate fully the benefits of MDE and call for wider usage. A preliminary investigation was made into MDE for this demonstrator with nSense with very promising prospects for both engineering and processes. For example, safety processes in the nuclear domain involve large volumes of information to absorb and understand - models can significantly improve training and understanding of safety and security processes here.

![Fig. 5. ICS Demonstrator.](image)

<table>
<thead>
<tr>
<th>Software Engineering Assessment (Ouler-JYU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform &amp; Automated Updating (Edpale &amp; Donphh) (Obeo)</td>
</tr>
<tr>
<td>Modelling for Process Enactment &amp; Recovery (UMPC+nSense)</td>
</tr>
<tr>
<td>Security ICS Training (nSense)</td>
</tr>
<tr>
<td>POHTO+nSense Use Case Requirements</td>
</tr>
<tr>
<td>POHTO+nSense Use Case Architecting/Simulation</td>
</tr>
<tr>
<td>POHTO+nSense Use Case System Implementation</td>
</tr>
<tr>
<td>POHTO Use Case System Operation</td>
</tr>
</tbody>
</table>

### VII. Conclusion and Future Development

An overview of the MERgE technologies has been presented in this project paper, providing interested parties with the references to delve into further detail. The four domains had differing needs and priorities, evidenced by the varied technologies selected across the demonstrators. The work has focused on non-functional properties and in particular safety and security, co-engineering, tool-chain improvement and traceability.

Especially for safety/security it can be difficult to bring processes and models into the public domain. This slows down their formalism and can block the use of current state of the art tools. Also costs can be a hindrance even given acceptability of technology by experts. Another challenge is the exchange of information between the phases of the product lifecycle. For instance, dependability flaws discovered in completed systems feed back to the design stage often only after standardisation, which can take some years.

Tied with the de-risking TRL levels, sometimes called the valley of doom, tools traverse these levels to achieve sufficient maturity for confident industrial and commercial use. Bugs at this point should be few and far between for general functions. However, even a technology that works well needs efficient training mechanisms in place and support for user queries. Some constraints that SMEs face for technology uptake can only be relieved by large organisations or standards bodies. This is particularly a case with engineering technology used in large-scale projects with many partners. For instance, partitioned procurement approaches can be a hurdle to overcome - a commonality across our use case domains, but particularly for Industrial Control and Space. Another hurdle to surmount is getting companies across industry to describe their resources with sufficient formalisation of processes. This is needed to fully harness the capability of some tools (projects for the industrial internet of things are likely to help here). Another challenge included determining the right levels of abstraction when introducing MDE for safety/security.

The uptake of state-of-the-art technology by industry and especially SMEs is often a challenge. One of the roadblocks here, given a new technology, is the evident lack of expertise available on the market when recruiting. Mechanisms are needed to facilitate and maximize the efficiency of training programs. This can be extended more generally to one’s ability to recruit new personnel and get them up to speed on company specific business processes or system design. Furthermore such mechanisms also help explaining to clients the significance of a company’s offerings. Model-driven engineering (MDE), although used primarily for the architecting of systems and processes, is also an enabler for the needs mentioned above. For over the last decade MDE has been maturing in various domains. The workbench Sirius is a further evolution of the existing technology opening up the ability for particular tailoring of MDE for domain-specific or company-specific requirements. This is especially useful for the safety and security sectors given the diversity of problems and solutions for both system design and system operation.

Variants of the MERgE Platform have been developed based on the use cases and are available to download or by request from the website www.merge-project.eu.

### Acknowledgments

The project MERgE was conducted thanks to funding from the ITEA2. The good level of collaboration within this project has been supported in particular thanks to the efforts of the demonstrator leads TCS, Melexis, KUL, SASNV and nSense.

### VIII. Acronyms

- **ECSS** - ESA Procedures Standards and Specifications
- **ICS** - Industrial Control System
- **JYU** - University of Jyväskylä
- **KUL** - Katholieke Universiteit Leuven
- **MD** - MagicDraw
- **MDE** - Model Driven Engineering
- **MERgE** - Multi-Concerns Interactions System Engineering
A Viewpoint-Based Approach for Formal Safety & Security Assessment

SASNV - Space Applications Services NV
SDR - Software-defined Radio
SFMECA - Software Failure Modes, Effects and Criticality Analysis
SME - Small to Medium Enterprises
SPL - Software Product Line
TCS - Thales Communications & Security
TGS - Thales Global Services
TRT - Thales Research & Technology

**REFERENCES**


Towards Comprehensive Threat Modeling for Vehicles

Mohammad Hamad*, Marcus Nolte†, Vassilis Prevelakis*

*Institute of Computer and Network Engineering, TU Braunschweig
†Institute of Control Engineering, TU Braunschweig
{mhamad,prevelakis}@ida.ing.tu-bs.de
nolte@ifr.ing.tu-bs.de

Abstract—Over the past few years, significant developments were introduced within the vehicular domain. The modern vehicle becomes a network of dozens of embedded systems which collaborate together. While these improvements have increased functionality of vehicle systems, they have introduced new potential risks. Threat modeling has gained a central role in identifying the threats that affect different subsystems inside the vehicle. In most cases, threat modeling was implemented either for one subsystem or based on a specific perspective such as the external threat surfaces only. In this work, we try to revise the existing threat modeling efforts in the vehicular domain. We reassemble them and extract their main characteristics to build a comprehensive threat model. This general model could be used to identify the different threats against the vehicular domain. Furthermore, reusable attack trees could be derived from this general model.

I. INTRODUCTION

Recently, vehicle manufacturing has changed significantly. These changes are reflected in the increased use of automotive embedded systems and the large amount of embedded software applications which are integrated into each single vehicle. A modern vehicle may contain up to 100 microcontroller-based computers, known as electronic control units (ECUs), which run millions of lines of codes (LOC) [1], [2]. Each ECU relies on a set of sensors and actuators to serve one or more of the E/E-systems or -subsystems in a vehicle. Different types of communication buses (e.g. CAN, FlexRay, etc.) are used to interconnect the distributed ECUs inside the car. The increase of connectivity within the vehicles is a double-edged sword. On the one hand, it extends the vehicle functionalities and capabilities, but on the other hand, it opens the door for several cybersecurity threats and makes the vehicle a more attractive target to adversaries [3].

The safety critical nature of the vehicle requires the adoption of high-security measures when developing vehicular IT systems. A good understanding of security requirements, which can be concluded from threat modeling, is a primary step toward contriving sufficient security countermeasures. Threat modeling helps to identify and address most of the potential threats. In fact, threats identification would likely reduce the life cycle cost of achieving security objectives when it is considered during the design process. Furthermore, threat modeling provides relevant information about the attack vectors which threaten the system. Such data can be used as a reference during the test process to avoid the omitted threats.

Different researchers scrutinize threat modeling in the vehicular domain. However, most of the researches examine the potential threats only partially. By looking at threats which affect a particular sub-system, then by creating attack vectors, and by suggesting appropriate mitigation mechanisms. Practically, the lack of a general threat model, within the vehicular domain, makes threats analysis of the different subsystems a resource consuming task. Additionally, it increases the possibility of inconsistencies between the interacting subsystems and it causes redundancy while defining the attack vectors.

In this work, we revise the existing vehicle-related threat modeling efforts to develop a comprehensive threat model. We define various potential attackers’ groups, the nature of the attack, potential targets and security requirements of the vehicular domain. Then, we propose an abstract model which can be used to classify all conceivable attacks against the vehicular domain. The abstract model is used as an aid to construct general attack trees [4] which illustrate attack vectors which threaten a particular sub-system of the vehicle.

The rest of the paper is organized as follows: In section II, we review existing threat models in the vehicular domain and reassemble them. We propose a general model in section III to identify possible threats within the vehicle. In Section IV, we used our general model to identify threats within an automated obstacle avoidance use-case. Related work is presented in section V. Finally, we present our conclusion in section VI.

II. THREAT MODELING

Threat modeling is a systematic approach for describing and classifying the security threats which affect a system. Moreover, it provides significant information that would help to safeguard the target (sub)system against attacks. Effective defense against threats requires addressing all existing security flaws in the target system and identifying threats which exploit these vulnerabilities. In addition, it demands good comprehension of the prospective attackers, their capabilities, and their objectives. Therefore, we start exploring threat modeling in the vehicular domain by defining the potential attackers’ profiles.
A. Attacker profile

Different groups of attackers are attracted to attack vehicles. These groups vary from the owner of the car to an expert hacker with sophisticated tools. Each one of these groups typically has its own motivations:

1) **Falsification:** An attacker (who could be the owner) may like to misrepresent actual vehicle information such as changing the tachograph or odograph measurements to sell the car with false mileage reading.

2) **Illegal profit:** An attacker could make profit by stealing the vehicle or by selling the attack capability to a different organization. Some attacks could be driven by a commercial competitor of the target vehicle’s vendor to sabotage their product and gain share in the market.

3) **Insane fun and vandalism:** Revenge and vandalism could motivate some attacks as the case of a dismissed employee who sought to punish his ex-company by brick the sold cars from this company [5].

4) **Research and test purposes:** Attacks and penetration tests could be performed by security experts or test teams. The attackers, in this case, have benign motivations. They try to discover security flaws in different components of the vehicle systems before they get exploited by third parties.

5) **Accidental:** In some circumstances, an attack could happen without any intention. Such attack could be performed while upgrading an existing system or by unintentionally reading malicious data as in the case of the malfunction in the vehicles GPS, climate control and front console radio systems in Toyota Lexus vehicles [6].

6) **Overlap:** Sometimes, multiple motives could stand behind a single attack.

However, motivation alone is not enough. An attacker needs sufficient technical skills and different sets of equipment to achieve his targets. The disparity of skills, capabilities, technical equipment, and financial resources could be used as indication to classify the attackers into different groups [7]:

1) **Unsophisticated attackers (script kiddie):** Attackers with limited financial resources and insignificant knowledge about the vehicle architecture belong to this group. Such attackers lack the ability to use complicated tools. Regular thieves, owners who would like to install or replace a component within their cars, an attacker who tampers with highway signals for gaining reputation in their community are good examples of this group members.

2) **Hacker:** This group includes highly skilled experts who have adequate tools and equipment to perform the attack. The members of this group could use their experience to get profit such as black-hat hackers. Mechanics and security researchers belong to this group.

3) **Organization:** These organizations have multiple members of the above group who work together. Typically, massive financial support enables them to obtain the sophisticated tools and attract experts. Security research groups could be one sample of this class.

B. Attackable objects

Attackers may focus in different parts of the vehicle components such:

1) **Data:** attackers could target stored data in some ECUs; this data could be cryptographic private keys, digital certificates, or private vehicle and driver activities (e.g., vehicle location, navigation destination, etc.). Or they could threaten transferred wired/wireless data within the vehicle. This data includes:

   a) In-vehicle exchanged data between different components and one component and its sensors. Spoofing the transferred data between the on-board system and the pressure sensors on the tires is an example of the vulnerability of such data [8].

   b) Transferred data between the vehicle and the external world; such as V2V communication data, V2I communication data, etc.

2) **In-Vehicle Hardware:** Generally, attacking the hardware infrastructure (i.e., ECUs, sensors, and On-Board Units) requires direct access to the target devices. Attacking In-Vehicle hardware could occur by replacing a device with a malicious one, or even installing new hardware which performs mischievously. Sometimes, the attacked hardware may not be a part of the vehicle. It could be 3rd party devices plugged to the vehicle, such as driver’s mobile phone [9]. The attacker could target to degrade the performance of the vehicle’s component or even lead them to produce misleading results intentionally (e.g. Volkswagen’s Emissions Scandal [10]).

3) **Surrounding infrastructure:** Some attacks could target the surrounding environment of the vehicle. A typical example of such an attack is the modifications to the electronic road signs such as “Zombies Ahead”, where an attacker figured out how to alter the text on electronic road signs warning of Zombies attack. Even such a ridiculous attack could create public safety issues for the drivers on the roadway [11].

4) **Software and framework:** The massive amount of the integrated software on each vehicle and the different levels of security auditing between the different vendors make them more susceptible to attacks. The framework which controls the ECU could be a target for various attacks; some attackers could tamper with this framework of the ECU to achieve superior performance [12]. A malicious update of one application or of internal parts of the framework could open the door for the attacker to inflict damage to the vehicle.

C. Attack requirements

1) **Direct access:** Some attacks are based on direct access to the target vehicle. Direct access could be achieved while a vehicle is parked. Then, attackers could have a chance to attach a GPS device to track the vehicle later or target the vehicle’s immobilizer and electronic locks [13]. In some circumstances, taking the car to the service station to check it could become an avenue for direct access from attackers. In such cases, an attacker has full access to the vehicle, and he could get the benefit of using existing physical interfaces to have
direct access to the internal network. On-board Diagnostic port (OBD-II) is one physical interface which was already employed in many attacks [3].

2) Remote access: Other attacks do not require direct access to the target vehicle. Attackers could target the vehicle remotely. Such attacks take advantage of the integrated wireless features of modern cars. These features include Bluetooth, a cellular connection, wireless tire pressure monitoring, etc. The entertainment system is another point which could be remotely hacked. Playing a song laced with Malware able to emit malicious messages to the CAN bus [3].

3) Mixed access: Direct access to the vehicle could be an introduction to remote attacks. Indeed, some attackers, even with rapid direct access to the vehicle, could install some devices inside the vehicle (such as a cover USB, malicious DVD, malicious component connected via OBD-II port, etc.) or outside it (communication sniffing devices). Later on, they could employ those parasitic devices to target the vehicle remotely. Attackers may use other people to install these devices, such as a valet who parks the victim’s car, a mechanic at a service station [3], etc.

D. Attack effects

For this contribution, we classify attacks based on their effect:

1) Limited attack: The ultimate target of some attacks could be a single part of the vehicle. The effect of such attacks will stay bounded in the attacked ECU(s) and not propagate any further. The targeted system will define the jeopardy of the attack.

2) Stepping stone attack: The attack can start by compromising one component or subsystem. Later, the attacker uses this subsystem as an attack surface to plague all related subsystems. The same process could be repeated for the newly infected components. Koscher et al. [3] showed that an attacker who can control one ECU is able to attack other connected ECUs.

E. Security Requirements

1) Authentication and Integrity: Providing the integrity within the vehicular systems is comprised of:
   - Providing data integrity to safeguard against any modification of data during a transaction.
   - Providing message source authentication to enable the verification of both ends of the communication.
   - Providing framework and software integrity to ensure the use of only trusted code and prevent the influence of malware.
   - Providing hardware integrity to prevent hardware fraud.

2) Privacy and Confidentiality: While providing authentication for the exchanged messages in the vehicular domain is vital, providing confidentiality often is less important. For example, there is no critical reason to encrypt the exchanged messages between the different ECUs inside the vehicle. Enforcing confidentiality for the exchanged data should not be mainly to prevent vehicle identification detection. The ability to identify the vehicle is feasible already by different mechanisms without the need to snoop the exchanged messages such as (identify vehicle by color, number plate, etc.) The primary goals should be preventing the leak of the driver’s critical data (such as driver behavior, previous location) as well as to guarantee that any observer is not able to efficiently link different messages coming from the same source. In some scenarios, confidentiality is required; for example, leaving the valuable stored information (e.g. private keys) without any confidentiality protection may leave the whole vehicle security at stake if an attacker is able to extract this data.

3) Availability: Availability is required particularly for safety-related applications which are integrated into the vehicle. Such applications should be available even if the vehicle is under attack.

III. ABSTRACT MODEL

A. Proposed model

In this section we try to extract the main characteristics of the reassembled threat model in order to create an abstract model. This model shall be suitable for adoption by security experts to identify and classify the majority of threats against vehicular systems. Such classification could reduce redundancy and inconsistency while applying defense techniques against homogeneous threats. In addition, it provides the basis for defining generic attack trees.

The proposed model shown in Fig. 1 uses three layers to identify and classify threats:

1) Target Domains: A vehicular system contains various assets (e.g. hardware, software, data, or surrounding infrastructure). Each asset may include several hidden vulnerabilities. A motivated attacker could target each of these assets by generating suitable conditions to exploit one or more of these vulnerabilities. We use these various assets as the first layer for identifying the potential threats by defining the flaws within each asset.

2) Requirements violation: The exploitation of an existing vulnerability in any asset will lead to a violation in one or more of the security requirements (i.e. confidentiality, integrity, or availability). We can further identify and classify the potential threats based on the violated requirement(s).

3) Accessibility: Eventually, the way of accessing the vehicle (i.e., remote, direct, or mixed access) in order to exploit a specific vulnerability is used as the last level for compartmentalization.

Applying this model to the whole vehicle system will identify most of the threats. The achievement of each one of these threats will be used as a root of a general attack tree which explains how an attacker could be able to exploit a defined vulnerability. Manipulating data and disabling hardware parts in the vehicle are examples of such general attack trees.

These trees will turn into distinct ones gradually reflecting the various studied subsystems. The accomplishment of one tree could open the door to fulfill other trees as we explained within the stepping stone attack.
Within the context of a vehicular system, many researchers used attack trees to illustrate attack vectors which threaten a particular (sub)system of the vehicle. However, general attack trees seem to be indispensable for avoiding redundancy and the interference between the high number of integrated sub-systems within the vehicle. The general trees will be derived from threats which were identified by our proposed threat model.

B. Attack Trees

Threat analysis describes who are the potential assaulters, what are the motivations behind an attack, and which components could be threatened. Describing how an attack could be executed is the mission of attack trees. An attack tree is used to explain attacks in a tree structure as shown in Fig. 2. The root of the tree represents the attacker’s ultimate goal, while the intermediate nodes of the tree (sub-goals) define different stages of the attack. In case a node in an attack tree requires achieving all of its sub-goals, the sub-goals are combined by an AND branch. In case a node requires achieving any of its sub-goals, the sub-goals are combined by an OR branch. Leaf’s nodes represent atomic attacks. Attack scenarios are generated from the attack tree by traversing the tree in a depth-first method [14]. Each attack scenario will contain the minimum combination of leafs. In classical attack tree models the attacks’ chronology is disregarded. However, in many cases, the success of an attack depends on the subsequent success of interrelated attack steps. Arnold et al. [15] propose sequential AND- and OR-gates (SAND, OR) to handle sequential occurrence of attacks.

C. Review Risk Analysis

Attack trees have been used to evaluate the security risk of the system and calculate the probability of a successful attack. This possibility depends on aspects, proposed by ISO/IEC 18045 [16], such as the required time for an attack, the desired attack tools, etc. However, regarding the risk analysis within the vehicular domain, the calculation of the probability of potential attacks based on associating numeric values with each level of these factors as proposed by ISO/IEC 18045 is not adequate anymore. Elapsed time, for example, has a different effect regarding the way of carrying out the attack, whether it is a remote attack or one with direct access to the vehicle. Moreover, the overlap between expertise and used tools also has a different effect; even inexpert attackers could launch an attack by using sophisticated tools. Eventually, stepping stone attacks should be considered during the calculation of probability of an attack. An attack could be unlikely, while achieving one attack goal in a different subsystem might increase this possibility.

IV. USE-CASE - AUTOMATED OBSTACLE AVOIDANCE

A. Description

In the CCC project, the Institute of Control Engineering (ICE) contributes the full-by-wire research vehicle MOBILE [17] as a demonstrator. MOBILE serves as a platform for research in the fields of E/E-systems and vehicle dynamics. It features four close-to-wheel electric drives (4x100 kW), as well as individually steerable wheels, and electro-mechanic brakes [17]. The vehicle features a FlexRay communication backbone for inter-ECU-communication and additional CAN bus interfaces, which are used for communication with vehicle sensors and actuators. The ECUs responsible for vehicle control are programmed in a custom-designed MATLAB/Simulink tool chain. Combined with detailed vehicle dynamics models, the tool chain serves as a means to establish a rapid-prototyping process for vehicle control algorithms.

Within the scope of the project, a use case in the form of automated obstacle avoidance will be implemented in the
experimental vehicle MOBILE. The basis for this use-case is a trajectory following stability control system. In general, stability control systems only steer the vehicle into the direction given by driver input at the steering wheel angle. However, as the driver does not always perform safe steering maneuvers, particularly in critical driving situations, such as fast obstacle avoidance, this extended stability control will follow a safe pre-planned trajectory instead of following a potentially unsafe path, steered by the driver.

The research vehicle is equipped with a variety of environment sensors to perceive the static and dynamic vehicle environment. Three lidar scanners, a radar sensor and a camera monitor the environment around the car. This data will be used to create a map of the static environment, which provides the basis for a model-based trajectory planning utilizing all actuators (particularly all-wheel steering) for maximal maneuverability.

A hardware architecture showing the perception system, as well as a simplified network for vehicle control is depicted in figure 3. With regard to environment perception, the required actions will be performed in a distributed system of three nodes. GPS and inertial data is fed via CAN to a node which is responsible for vehicle localization and motion estimation. Lidar sensors and a camera are streamed via UDP to a node responsible for environment perception (sensor data processing, data fusion, environment modeling). Data from a radar sensor is acquired via a CAN bus connection.

Trajectory planning will be performed on the "Vehicle Control" node, utilizing aggregated data from vehicle and environment sensors. The planned trajectory is then converted to reference values for the six vehicle control ECUs (three main control units, three hot stand-by nodes), which are connected with the already mentioned FlexRay backbone. As the research vehicle is not permitted to drive in public traffic, the use-case will be verified and validated on a closed testing ground only. However, the sensor setup and sensor data processing architecture is very similar to the research vehicle Leonie [18], also built and maintained by the ICE, so that at least parts of the identified attack vectors could be transferred to a vehicle with a driving permit for public roads.

**B. Threat modeling for the use-case**

We used our model to identify the potential threats within the automated obstacle avoidance use-case. We started our investigation by defining all components which could include vulnerabilities, and identify the security requirement that could be violated in the case of exploiting these vulnerabilities. Lidar, Camera, Radar, and GPS are possible attack surfaces in our use-case. We tried to construct attack trees for each one of them, as shown in figure 4; these trees are derived from the general ones (i.e., disabling a hardware and confusing the proper function of a component). Detailed explanation about attacking the camera and lidar in the vehicle can be found in [19].

The manipulation of the surrounding infrastructures has a direct effect on the functionality of different components in our use-case (such as the Camera). Figure 5 illustrate a general attack tree for the surrounding environment which affect our use-case’s components. On the other hand, crashing the framework of an ECU will lead to preventing the ECU from doing its function and disable it, even, temporarily.

**V. RELATED WORK**

Threat analysis of modern vehicles has remained a hot topic, and will continue. As modern vehicle architecture is getting more complex, the potential threats are increasing too. Various researchers have tried to point out the vulnerabilities within the vehicular system based on different perspectives; Checkoway et al. [20] looked at potential attack surfaces which could be exploited by attackers externally. On the other hand, Koscher et al. [3] studied the attack surfaces on the underlying system structure. They demonstrated that attackers could leverage...
direct access to the CAN bus to control various functions adversarially. Petit and Shladover, in [21], investigated cyber-attacks for the automated and connected vehicle. Attack trees have been used as a tool to illustrate the attack steps for individual attack scenarios within the vehicular system [9]. Aijaz et al. [22] tried to create a reusable attack tree for V2V communication threats. In our work, we try to provide an abstract model which helps creating a general attack tree for the whole vehicular domain.

Many threat model schemes, such as STRIDE [23] and SDL [24], are used to characterize cybersecurity threats in different environments. However, McCarthy et al. [25] claimed that these models may not be fully applicable in the automotive cybersecurity analysis. Therefore, they proposed the use of a threat model which is a hybrid of various models. We worked in the same direction by adopting an existing model (i.e. the Confidentiality, Integrity, and Availability (CIA) information security model) in our approach.

VI. CONCLUSION

In this work, we created a comprehensive threat model based on the existing vehicle-related threat modeling efforts. Our model classifies and identifies the threats based on target assets, the violated security requirements, and the accessibility of the threats. General attack trees can be linked to each of the identified threats. We explored the automated obstacle avoidance use-case while trying to classify the potential threats against it, based on our model. Future work will define mitigation mechanisms based on this model.

ACKNOWLEDGEMENT

This work was supported by the DFG Research Unit Controlling Concurrent Change (CCC), funding number FOR 1800. We thank the members of CCC for their support. We thank Mischa Möstl for his helpful discussions, and Marions Tsantekidis for his comments that improved the manuscript.

REFERENCES

Abstract—The landscape of automotive in-vehicle networks is changing driven by the vast options for infotainment features and progress toward fully-autonomous vehicles. However, the security of automotive networks is lagging behind feature-driven technologies, and new vulnerabilities are constantly being discovered. In this paper, we introduce a road map towards a security solution for in-vehicle networks that can detect anomalous and failed states of the network and adaptively respond in real-time to maintain a fail-operational system.

I. INTRODUCTION AND PROBLEM STATEMENT

Recent trends in industry towards smart and interconnected devices, referred to as the Internet of Things are now also appearing in the automotive world. Passenger comfort and infotainment features continue to progress through the advancement of in-vehicle networks and connectivity of the automotive system with its environment. However there is a lack of security to prevent comfort and efficiency features from compromising safety-critical control systems. Examples of such security vulnerabilities include the now infamous Jeep Grand Cherokee hack [1], and more recently the remote hack of a Tesla Model S covered on Wired.com.

The automotive industry has slowly replaced much of the mechanical couplings between car components with electronics and software, because electronics are cheaper and lighter than their mechanical counter parts. In conjunction with the hardware, an automobile also contains over 100 million lines of software [2], and consumers are constantly seeking features that allow for more interaction between smart phones and car. Automotive in-vehicle networks are rapidly changing because of consumer demand for the interconnectivity of devices, and what had previously been a network isolated from attackers due to the limited possible access points is now more vulnerable. Having a wireless access point in the in-vehicle network allows for remote diagnostics and over-the-air firmware updates to thousands of vehicles simultaneously, but at a cost: safety and security vulnerabilities are introduced to what was once a closed system.

Our primary goal is to solve security problems inherent with in-vehicle networks that mix safety-critical packets with non-critical messages, such as those for ADAS and infotainment.

We propose a fail-operational intrusion detection system (FO-IDS) that identifies potential attacks and causes the cyber physical system (CPS) to transition to an operational safe state.

II. RELATED WORK

The idea of intrusion detection is not new, however it is only recently being applied to automotive in-vehicle networks. IDSs work by monitoring messages and frequencies of the network and system to identifying anomalies and respond accordingly. Currently in safety-critical CPSs, when an anomaly is encountered, the system enters a fail-safe mode; this mode maintains operations while handling the anomaly. The system will return to normal operating modes after the system is clear of any anomalies. However there is a lack of cyber anomaly detection and response in in-vehicle networks. False positives are a major issue in IDSs, and several groups have demonstrated IDS with low to zero false positives [3]. Other work specifically on CPS IDSs focuses on different methods of anomaly detection. Few of these necessarily perform well for automotive in-vehicle networks, and a method of detecting anomalies using the fail-operational system needs to be developed.

SecureCore [4] and Secure System Simplex Architecture [5] introduce time-based anomaly detection for CPS IDS that revisits the Simplex architecture by using redundant hardware controllers. Their solution works at the processor level, and it can detect and recover from attacks that under- or over-use processor time. Translating this solution to defend in-vehicle networks is not straightforward, because malicious packets can take the same amount of time to process as benign ones.

Sasaki et al. [6] introduce a framework to detect man-in-the-middle attacks and switch to a backup control system, which could be useful to help generate a fail-operational mode.

We aim to leverage previous work in regards to IDS and suggest features for anomaly detection while continuing to explore other designs and better understand the characteristics of an IDS in real-world automotive CPS. We propose an FO-IDS that identifies security and safety anomalies and finds a fail-operation mode that the system can transition to when an intrusion has been detected.
III. OUR APPROACH

FO-IDS is an improvement to classical automotive IDS solutions that will incorporate information flow tracking and sensor data provenance. Response of the FO-IDS will cause a mode-change in the automotive system to enter a safe, degraded, yet operational state that prevents the detected attack, and then initiate recovery to restore degraded services. Figure 1 shows a high-level conceptual design of our proposed system.

A. Design Classifiers for Sequential Anomaly Detection

An FO-IDS needs to be able to identify anomalies that occur in sequences and combinations of messages observable at in-vehicle network gateways and ECUs. Sequential anomalies are important for detecting semantic attacks that span multiple state transitions of a CPS, i.e., attacks that may induce seemingly valid individual transitions but invalid, unsafe aggregate transition behavior. A Bluetooth pairing that leads to a malicious firmware upgrade is an example of a sequence-based semantic attack, where infiltration precedes compromise with multiple intervening CPS state transitions. The Bluetooth pairing itself is not anomalous, and neither is a firmware upgrade, but the two together need to raise suspicion.

In-vehicle networks use shared bus protocols that mix predictable, high-priority, periodic control messages with less predictable, lower priority messages. The IDS will partition the incoming messages to the IDS to examine control messages using algorithms that exploit the highly predictable nature, while non-control messages will require algorithms that can tolerate more randomness. Kernel-based machine learning appears well-suited to sequential anomaly detection [7] and the high dimensional feature spaces of vehicular data, but can have substantial time complexity. We will investigate whether kernel-based approaches can be efficient enough for IDS at in-vehicle network transmission rates, and will explore sequence mining with heuristic pruning of the state space to include limiting history and prioritizing search paths using physical constraints of the in-vehicle network to parameterize the machine learning algorithms.

B. Improve Classification by Enhancing Feature Collectors

The unmodified in-vehicle network traffic will provide a wealth of data for anomaly detection. However, to improve classifier performance we will increase the available data by tracking, within the constraints of network bandwidth and control unit resources, data provenance and information flows. We will explore the use of device identification in addition to (logical) timestamps that would assist in resisting replay and false data injection attacks. Information flow control and taint tracking will associate device tags with data that propagates through the in-vehicle network to enable traditional information security approaches such as Biba and Bell-LaPadula to be used between automotive subsystems.

C. Recoverable Fail-Operational Modes

Much work exists in fail-operational vehicular control to ensure fail-operational safety in the transition to fly-by-wire avionics [8]. Our aim is to understand the nature of each in-vehicle network component with respect to its capability to fail gracefully, which components are naturally able to fail into an operational state, and which require additional support from fault tolerance mechanisms. We will create a taxonomy of in-vehicle components (gateways, ECUs, and subnetworks) with respect to their fail-operational capability, and investigate mechanisms such as real-time microrebooting and network partition-tolerance to enable fail-operational responses when anomalies are detected.

REFERENCES


Fig. 1. Project Concept
SchedUleak: An Algorithm for Reconstructing Task Schedules in
Fixed-Priority Hard Real-Time Systems

Chien-Ying Chen\textsuperscript{1}, Amir Emad Ghassami\textsuperscript{2}, Sibin Mohan\textsuperscript{1}, Negar Kiyavash\textsuperscript{2},
Rakesh B. Bobba\textsuperscript{3} and Rodolfo Pellizzoni\textsuperscript{4}

\textsuperscript{1} Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
\textsuperscript{2} Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, USA
\textsuperscript{3} School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA
\textsuperscript{4} Dept. of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada

Email: \{cchen140, ghassam2, sibin, kiyavash\}@illinois.edu, rakesh.bobba@oregonstate.edu,
rodolfo.pellizzoni@uwaterloo.ca

\textbf{Abstract}
In real-time embedded systems, failures due to the lack of security can cause serious damage to the system or even injury to humans. Until recently security was an afterthought in the design of such systems. Even less understood are attack mechanisms that can successfully target real-time systems. In this paper we present a novel attack model and algorithm to extract the exact schedules of real-time systems that are built using fixed priority scheduling algorithms.

pp. 39-46

Fardin Abdi†, Monowar Hasan†, Sibin Mohan‡, Disha Agarwal‡ and Marco Caccamo†
†Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
‡Dept. of Computer Science and Engineering, PES University, India
Email: {‡abditag2, †mhasan1,l,sibin,mccacmo} illinois.edu, ‡agarwal.disha21@gmail.com
†∗These authors have made equal contribution to this work.

Abstract—In this paper we present ReSecure, a framework that uses the concept of system restart to secure hard real-time systems (RTS). ReSecure is used to improve the security of RTS without violating safety or temporal constraints. We also show how designers of systems can customize (or even optimize) system parameters to achieve the best trade-offs between security and control system performance. We demonstrate our concepts using a prototype on an ARM-based embedded platform as well as a 3 degree of freedom (3 DOF) helicopter system.

I. INTRODUCTION

Physical systems have limitations and restrictions that need to be respected during operation. Otherwise, the system itself or the environment around them, including humans, can be damaged. These physical systems are usually controlled by embedded real-time controllers and require consistent availability and reliable execution in order to operate safely. These limitations, therefore, need to be taken into account when designing the controllers for these systems. With the growing complexity of these controllers due to the use of sophisticated operating systems (OS), vendor developed drivers and open source libraries as well as increased connectivity makes it very challenging to verify that those physical restrictions are always respected.

Despite the fact that sensors and actuators that interact with the physical world may open up composite sources of vulnerabilities, security issues in real-time systems (RTS), however, have received relatively little attention by academia or industry. The sophisticated attack landscape, limited computing resources due to the embedded architecture as well as exceptionally high reliability and availability requirements make it harder to build effective intrusion detection and prevention mechanisms in RTS.

Recent RTS are more interconnected and even controlled over unreliable mediums such as the Internet. In addition, there is more monetary and adversarial incentives for malicious activities. The Stuxnet worm that highlighted the possibility and effectiveness of an attack on critical infrastructure [1] and malicious code injection into the telematics units of modern automobiles [2], [3] are some instances of such attacks. Without considering the safety-critical and embedded nature of these systems, simply adding security mechanisms (such as encryption, message authentication, etc.) will not be effective to ensure both safety and security.

Hence we intend to provide designers a flexible yet unified framework that allows to customize system parameters (security policies and preference over security and performance). In this paper we present ReSecure, a runtime restart-based protection approach to ensure security in RTS. In traditional computing systems (e.g., servers, smart phones, etc.), software problems are often resolved by restarting, either the application process or the platform [4]–[6]. We use a similar concept to improve security guarantees in RTS. In particular, we propose to restart the platform periodically/asynchronously and load a fresh image of the applications and OS after each reboot with the objective of wiping out the intruder or malicious entity.

Unlike conventional computing systems restart-based recovery mechanisms are not straightforward in RTS due to additional real-time constraints, as well as interactions of the control system with the physical world. Besides, most physical systems need consistent actuation with tight timing constraints (e.g., a helicopter can quickly destabilize if the controller restarts). The ReSecure framework is specifically designed to improve security of safety-critical RTS with physical systems in need of constant actuation. In order to guarantee the physical safety requirements, we use the Simplex architecture [7], [8], a method that utilizes a minimal, verified controller as backup when the complex, high-performance controller is not available or malfunctioning. In particular, ReSecure is a set of software and hardware mechanisms that enable a trade-off between the security guarantees and control performance while guaranteeing the safety of the physical system at all times. Specifically, the contributions of this work can be summarized as follows.

• We introduce ReSecure, a restart-based architectural framework to improve security in RTS. We discuss how triggering restart events through several system components can enhance security without sacrificing safety (Section IV).
• To configure system parameters based on the design requirements, we provide an analytical framework that calculates the best trade-off between security and controller performance (Section IV-C).
• We evaluate the system with a proof-of-concept implementation on an ARM-based development board and embedded real-time Linux (Section VI).

The ReSecure framework proposed in this paper is based on Simplex [7], [8] that enables the safe restart mechanisms. Specifically, we utilize a variant of Simplex, viz., system-level Simplex [9]. We present first a brief overview of Simplex and our assumptions on adversarial capabilities before we elaborate the design details of ReSecure.

II. OVERVIEW OF SIMPLEX ARCHITECTURE

Simplex [8], [10]–[12] is a well-known architecture that uses a simple verified subsystem (Fig. 1) to ensure safety of the plant. This conservative safety subsystem is then complemented by a high-performance complex control subsystem that is concerned with mission-critical requirements. A decision module then uses the high-performance complex controller whenever possible, but will switch to the safety controller when system safety is jeopardized. Using Lyapunov stability properties from control theory, there exists a set of states that

†By the term ‘safety’ we refer to ensure that the limitations and restrictions of the physical system (e.g., actuator limits, maximum pressure, maximum temperature, etc.) are always respected.
within those set, safety control is always able to stabilize the system and keep it safe. The goal of Simplex method it to guarantee that under any behavior of the complex subsystem, the physical system would remain safe.

Fig. 1. Logical view of Simplex architecture. Decision module chooses the control that does not jeopardize the safety.

Our proposed approach is based on variants of Simplex such as system-level Simplex [9] and reset-based recovery [13] that have moved the safety subsystem and the decision module into a dedicated hardware unit. This isolates the trusted components from the faults and misbehavior of the complex subsystem.

III. ADVERSARY MODEL

RTS face threats in various forms depending on the system and the goals of an adversary. For instance, adversaries may insert, eavesdrop on or modify messages exchanged by system components. Besides, attackers may manipulate the processing of sensor inputs and actuator commands, could try to modify the control flow of the system as well as extract sensitive information through side-channels. The adversarial capabilities we consider in this work are as follows.

i) Integrity violation: We assume that the adversary can compromise all the software components on the complex unit including underlying real-time OS (RTOS) as well as the real-time applications. For example, an adversary may insert a malicious task that respects the real-time guarantees of the system to avoid immediate detection, compromise one or more existing real-time tasks and/or may override parts of the underlying RTOS itself that may eventually threaten overall safety of the system.

ii) Denial of Service (DoS): The attacker may take control of the real-time task(s) in the complex controller and perform system-level resource (e.g., CPU, disk, memory, etc.) exhaustion. In addition, an advanced attacker may capture I/O or network ports and perform network-level attacks to tamper with the confidentiality and integrity (viz., safety) of the system.

iii) Information leakage through side-channels: The adversary may also aim to learn important information by side or covert-channel attacks by simply lodging themselves in the system and extracting sensitive information. For example, the intruder may utilize side-channels to monitor the system behavior and infer certain degree of system information (e.g., hardware/software architecture, user tasks and thermal profiles, etc.) that eventually leads to the attacker actively taking control, manipulating and/or crashing the system.

In addition to these adversarial capabilities, we also make the following assumptions.

i) Sensor reading manipulation: We consider that an intelligent adversary can only attack the external sensors via physical access. Since this may not be the obvious attempt in practice, we assume that the sensor readings are not compromised.

ii) Safety unit integrity: Safety unit has simple and verified software that has no interaction with the outside world except reading the complex controller command and sensor values in every cycle, and sending the final command to the actuators. Therefore, we assume that safety unit remains uncompromised during system operation.

iii) Read-only memory unit: We assume that the attacker cannot modify the content of the read-only memory unit that stores the uncompromised image of the RTOS and real-time applications. For example, this can be achieved by using an E²PROM storage unit.

IV. RESTARTABLE DESIGN: APPROACH AND OVERVIEW

We consider a real-time control application, viz., a system consisting of a set of sporadic, fixed priority, independent tasks executing on the complex unit. Each real-time task \( \tau_i \) is characterized by \( (C_i, T_i, D_i) \), where \( C_i \) is the worst-case execution time, \( T_i \) is the minimum inter-arrival time (or period) between successive releases and \( D_i \) is the relative deadline. Although we limit ourselves to fixed-priority scheduling in this paper, the concepts of restarting to secure RTS can be extended to dynamic priority schemes without loss of generality. The schedulability of the real-time application tasks are assumed to be guaranteed by offline analysis [15]. Our design goal is to enable the complex subsystem to restart and reload an uncompromised image containing all the applications and the underlying RTOS/firmware while guaranteeing that the physical plant stays safe during restart.

The high level overview of ReSecure is illustrated in Fig. 2. As mentioned earlier, we use the system-level Simplex [9] as the basis of our design. In this architecture, safety and complex subsystems are on isolated hardware units such that faults or exploits in the complex subsystem does not affect the functionality of the safety unit. The only interaction of the safety subsystem has with the outside world is limited to reading the complex controller commands and sensor values in a controlled format and sending the control commands to actuators. Therefore, we do assume that the safety unit always remains trusted during system operation and there is no malicious entity embedded in it. The entire security framework can be collapsed if the safety unit itself is compromised in the first place.

The decision module switches the control to the safety controller when the commands from the complex controller may jeopardize safety or when there is no command from the complex controller (e.g., during a restart). The Safety controller maintains (a somewhat degraded) performance until a safe command is available from complex controller (e.g., the restart completes and the original controller can resume its operation).

One may argue why complex unit does not receive the same level of protection as the safety unit. The complex unit, however, is often exposed to external sources (e.g., user input, I/O or network peripherals, software update, etc.) that opens it up to the potential sources of vulnerabilities. As mentioned earlier, the binary or control logic in the safety unit will not often change. Therefore, it makes sense to harden security mechanisms in the complex unit.

2This assumption is clarified further in Section IV.

3One approach to assign priority could be using the Rate Monotonic algorithm [14].

4Similar assumptions are not uncommon in literature [16], [17].

5For the detailed description of safety controller and decision module design procedure we refer the readers to [9] and [15].
from a compromised state, we choose the following set of safety controller. In order to recover the complex controller may not made progress towards the mission with only the of the decision module and safety unit. However, the system complex unit but cannot violate the physical safety because

**A. Triggering Restarts**

In ReSecure platform, the attacker can compromise the complex unit but cannot violate the physical safety because of the decision module and safety unit. However, the system may not made progress towards the mission with only the safety controller. In order to recover the complex controller from a compromised state, we choose the following set of events to trigger a full restart of the complex unit and reload an un compromised image of *both OS and control applications* from read-only memory: *i) failure of critical components, ii) at predefined periodic times and iii) upon detection of an intrusion by the monitoring unit.* In the following we elaborate the above choices and discuss the mechanisms to implement each one.

1) **Watchdog Timer:** Watchdog timers ensure that the components in the complex unit are alive and have not crashed. Hence, the complex controller and monitoring unit should update a watchdog timer at every execution loop. Thus, the platform needs to provide at least one watchdog timer. The watchdog must be independent of everything else in the system (except power). It is important to mention that most watchdog timers are “no way out”, meaning that once activated, they cannot be deactivated. However, this must be verified on the specific platform which is being used. If the watchdog can be disabled after activation, an attacker might be able to disable the watchdog and prevent a restart.

2) **Monitoring Unit:** Monitoring unit can act as an attack detector and look for signs of malicious behavior in the system. Alternatively, it can look for signs of fault in the system before they occur. For instance, it can perform tests that can cover resource availability (e.g., sufficient memory and file handles, reasonable CPU time, etc.), evidence of expected process activity (e.g., system processes running, specific files being presented or updated, etc.), overheating, network activity, system-specific tests as well as monitoring for specific attack or security threats. For each one of the above, there is a large body of work in the literature that can be plugged in. Thus, we do not consider the specific design/functionality of the monitoring unit. Our flexible framework enables designers to implement their own monitoring methods to improve security depending on system requirements.

3) **Periodic Timer:** No matter how effective the monitoring strategy is, it can never detect all compromises in the system. A sophisticated attacker may even be able to disable the monitoring unit. Besides, not all the intrusions necessarily crash the complex controller and/or initiate a restart through watchdog timers. For example, an attacker may just modify the complex controller logic. The external timers trigger periodic restarts and reload un compromised system image independent of all the components and events in the system. This allows the system to recover from the unforeseen attacks (e.g., zero-day vulnerabilities) that all the other restart-triggering mechanisms (e.g., watchdog timers and monitoring unit) fail to detect.

Note that ReSecure does not make any assumptions with regards to the detectability of attacks. An attack can remain undetected by monitoring unit or even may compromise the monitoring unit. However, even such sophisticated attacks will be removed when the periodic timers trigger the restart. Needless to say that a more effective monitoring mechanism leads to a faster detection and recovery and consequently a lesser loss of control performance.

**B. Restartable Components**

Since aforementioned triggers can occur asynchronously at any time during operation, all the components on the complex unit must be designed to be restarted at any arbitrary time. In conventional software systems that are not designed for restartability, restarting the component or the entire system at states where the restart is not expected might result in unrecoverable data-dependent failures. This can occur in two main ways: *i) due to logic data corruption, and ii) due to file system corruption. The case of logic data corruption may occur if the system is being restarted while an application was updating a unit of storage (e.g., a file). In this case, the semantics of the partially updated storage may be affected, leading to the permanent inability to restart an application or conversely a functional but compromised state.* Conversely in our framework, the system is forcefully restarted after the execution of disk-cached I/O operations and before cached file changes are synchronized with the disk.

For this paper, we make the following considerations about the restartability of software components. First, many microcontroller-based systems as well as many embedded OS provide no support for persistent storage and thus provide no file systems. In these class of systems, a fresh image of the OS and applications is loaded into main memory from a read-only memory resource (typically a flash memory) during a restart. Second, for those platforms with support for file systems, restartability can be achieved if a read-only partition is designated for the storage of the OS and control application binaries. This will ensure that the OS and application images will not be impacted by file system corruption during restart.

**C. Securing RTS by Restarting**

Restarting the entire complex unit and loading the fresh image of all the framework and applications restores the system into a functional and uncompromised state. However, it does not fix the vulnerabilities that were exploited in the first place by the attacker. That is why that a single restart cannot be considered as an effective attack prevention mechanisms by itself.

The main idea of restart-based protection is that, if we restart the system frequently enough, it is less likely that the attacker will have time to become effective and cause meaningful damage to the system. After every restart, there will be a predictable down time (during the system reboot), some operational time (before system is compromised again) and some compromised time (until the compromise is detected or periodic timer expires). The periodic timer sets an upperbound on the compromised time of the system.

The length of each one of the above intervals depends on the type and configuration of the platform, adversary models, effectiveness of the monitoring unit and complexity of the exploits. As a general rule, the effectiveness of the restarting mechanism increases *i) as the time to re-launch the attacks increases or ii) the time to detect attacks and trigger a restart decreases.*
method\textsuperscript{6} to evaluate the expected unavailability due to restarts and attacks and the expected damage from the attacks/exploits given a certain restart configuration. We also show how to find an optimal configuration to balance lack of availability and damage.

V. ReSecure and Attack Resiliency

Most of the existing security mechanisms protect the system from specific categories of attack and providing high security assurance requires a mixture of multiple mechanisms. Even then, there are always unknown vulnerabilities that can be exploited by an attacker to perform \textit{zero-day} attacks. On the other hand, patching the OS in embedded controllers and programmable logic controllers (PLCs) against (un)known bugs and vulnerabilities is very challenging. Due to the safety-critical nature of the applications in industrial controllers, any updates to the software needs to undergo extensive testing and verification before being deployed. ReSecure on the other hand, guarantees complete safety of physical unit along with the presence of any type of exploits.

In the following, we review some examples of security threats that ReSecure can recover.

1) Malicious Software: Sophisticated adversaries can create malware that seeks to infect the external media used for transfers and updates. This opens the opportunities for possible exploits and may infect the system unknowingly to the human operator \cite{19}. Some malicious code injections require time, resources and sometime \textit{physical intervention} to spread through systems. For example, the Stuxnet worm \cite{1} was able to successfully subvert industrial control systems and infect PLCs. The worm was able to get in once an infected peripheral device was connected to the main system. After gaining access, the malware gradually inflicted damage to the physical plant by substituting legitimate actuation commands with infected ones over a period of time. Flame \cite{20} is another example of malware designed specifically for espionage that spreads through USB devices and local networks.

Such malwares get into the system from removable drives that requires physical access and later slowly spread over the network by replicating themselves. In practice, however, it is unlikely that malicious entities can gain frequent physical access to the system after every reboot.

2) Stealing Sensitive Information: An adversary may want to track side-channel signals generated by the complex controller tasks with the intention of stealing sensitive information and secrets of the system. As a matter of fact, due to the \textit{deterministic schedules} of most RTS, an attacker can extract certain system information using signals such as timing parameters of the task \cite{21}, thermal profiles \cite{22}, secret keys \cite{23} and cache access pattern \cite{24}. Such side channel attacks require the adversary to perform monitoring activities for a \textit{certain period of time} to \textit{improve} the accuracy of the extracted information. Restarting make it possible to prevent from \textit{stabilizing} the attacker task before it can extract a certain degree of \textit{useful} information.

3) Denial-of-Service (DoS) Attacks: DoS attacks is a category of attacks where the attacker exhausts the resources of the system thereby preventing the access for the legitimate user. DoS attacks can occur in two forms; \textit{i.e.}, system-level and network-level. In system-level attacks, attacker tries to exhaust all the memory, consume all the processor utilization or fill all the disk spaces. These eventually prevent the real-time applications from functioning. The network-level DoS relies on sending an overwhelming influx of request to overflow the target’s resources. Flood of TCP/SYN packets, overflowing the data buffer and reflective pings are examples of network DoS attacks.

Restarting can always claim all the stale resources back and enable the system to continue operation. In the RTS context, in order to perform a system-level DoS attack, the attacker may need to monitor the execution profile for the tasks using, say, side-channel attacks and then override the task code with malicious DoS routines. Besides, the DoS routines also need some time to saturate the system resources. It is worth noting that combining all these steps to launch successful attacks requires \textit{considerable} time.

With a correct estimation of the capabilities of the attacker (how much traffic they generate or at what rate they can exhaust system resources), the system designer can formulate the cost function for the attack and find an optimal restart time (refer to Appendix). Thus we assert that the restart-based method can be a potential solution to prevent (as well as recover) the system from many DoS attacks.

VI. IMPLEMENTATION AND EVALUATION

In this section we evaluate ReSecure with a proof-of-concept implementation. Although our prototype is implemented on an ARM-based development board and embedded Linux, the proposed method can be ported into different RTOS and platforms without loss of generality. Source codes used in the experiments are available online\textsuperscript{7}. Table I lists the details about the implementation and parameters used in the experiments.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Artifact/Parameter} & \textbf{Values} \\
\hline
Complex unit platform & ARM Cortex-A8 1GHz, 512MB RAM \\
Safety unit platform & Intel Core i7, 8GB RAM, Real-time Linux kernel \\
Complex unit OS & 4.4.12-rtt-r30 \\
Complex unit reboot time, $T_R$ & 13 s \\
Periodic restart timeout, $T_{Timer}$ & 120 s \\
Period of the controller tasks, $T_C$ & [10 ms, 100 ms] \\
Watchdog timeout & Control task period \\
\hline
\end{tabular}
\caption{IMPLEMENTATION PLATFORM AND EXPERIMENTAL PARAMETERS}
\end{table}

A. Experimental Setup

Our prototype implementation is developed in C and compiled using the GNU C compiler with optimization level 2 (\textit{e.g.}, option \texttt{-O2}). We do not use any nonstandard external libraries and the implementation has no dynamic memory allocations or recursion.

1) Physical Plant: As a physical system we consider a 3 degree of freedom (3 DOF) helicopter. We use a known model of the system obtained from prior work \cite{25} and perform hardware-in-the-loop (HIL) simulations \cite{26} to capture the exact behavior of the sensors and physical plant. Pitch, elevation and travel angle are reported to the safety and complex controller as sensor readings.

This plant is considered \textit{safe} as long as the wings of the fans do not hit the horizontal surface on which the 3-DOF helicopter is fixated. The linear inequality in Eq. (1) defines the set of states in which system is \textit{safe}.

$$\text{elevation} \pm 1/3 \times \text{pitch} > -0.3.$$ (1)

\textsuperscript{6}A similar probabilistic analysis is performed in literature \cite{18} to find the optimal restart time for software rejuvenation purposes.

\textsuperscript{7}https://github.com/mnwrhsn/restart_n_secure_cps.
By applying the Simplex method to the linear model of the system and the safety condition, we obtain the safety controller (i.e., a proportional feedback controller) and the decision logic for decision module.

2) Complex Unit: We use a BeagleBone Black (BBB) development board (ARM Cortex-A8 1GHz processor, 512MB RAM, 4GB on-board eMMC storage) as our complex unit. The complex unit uses an embedded Debian GNU/Linux console image as the OS. Since the vanilla Linux kernel is poorly suited for executing real-time applications directly, we enable the real-time capabilities by applying the RT-PREEMPT patch [27] (kernel version 4.4.12-ti-rt-r30) that is known to respect real-time constraints in embedded platforms such as ARM [28].

The OS in the complex unit uses priority-based preemptive scheduling powered by RT-PREEMPT patch. The highest priority task that is eligible to run in the system is always scheduled by the OS. Each real-time controller task has a period $T_{\text{sys}} \in [10 \text{ ms}, 100 \text{ ms}]$. The period controller tasks are suspended after the completion of corresponding instances using the `clock_nanosleep()` system call. Also we set the program executable as a `startup cron job` (using `crontab` utility) so that it can execute as soon as the system reboots. After reboot we load the fresh (viz., uncompromised) OS image from the BBB’s on-board eMMC storage. From our experiments we find that, it takes at most 13 seconds (on average 12.1379 seconds with 0.3509 standard deviation) to restart the BBB and make the complex controller’s code operational.

In order to prevent unauthorized disk access and protect the file systems from corruption during the asynchronous restart events, we mount the file systems of the complex unit’s OS as read-only. For this, we modify the `/etc/fstab` file and ensure that the root file systems will always be mounted as read-only after each reboot.

3) Safety Unit: The safety unit (e.g., safety controller and decision module) is implemented on a x86 machine (Intel Core i7 processor, 8GB RAM) running Ubuntu Linux 14.04. In our current implementation, the complex unit communicates with the safety unit using a fixed IP. The sensor and actuation commands (viz., travel, elevation and pitch angle of the helicopter) are fed into both safety and complex controller. We implement the decision module as a multi threaded application that toggles the control between safety and complex controller according to $i)$ availability of complex unit and $ii)$ the decision logic obtained from applying Simplex method to the system model and the safety conditions.

4) Monitoring Unit: As we have mentioned in Section IV-A2, the flexible architecture of ReSecure allows the designer to integrate the desired security methods in order to monitor the system resources/components that they care about. For illustration purposes only, we develop a minimalistic monitoring unit. The monitoring unit is implemented as a periodic task of its own (with a period of 2 seconds). Specifically, the monitoring unit in our current implementation stores the legitimate kernel modules on startup. Later, during each of its periodic invocation at time $t$, it scans the currently running kernel modules at time $t$ (e.g., available through `/proc/modules` interface) and compares with the stored information. In the event when any mismatch or anomaly is detected the monitoring unit issues a restart request through SysRq command (e.g., using `/proc/sysrq-trigger`).

5) Implementation of Restart Events: As mentioned in Section IV-A, the restart events in the complex unit are triggered by the watchdog and periodic timers. We implement the watchdog timer using `ularm()` system call and update the timer in each periodic invocation of controller tasks. Once activated, the watchdog timer is periodically accessed. Failing to update the watchdog within the timeout duration will force the system to restart.

External periodic hardware timers can be implemented by using the BBB’s general purpose input/outputs (GPIO) pins. For instance, one can interface a hardware timer with BBB’s GPIO interface and generate an interrupt to trigger the restart when the timer expires. Our current prototype, however, does not use any external hardware timers. Instead, we use Linux high resolution timers (viz., hrtimers [29]) to implement the periodic restart. In particular, we develop a loadable kernel module that activates the timer on startup and restarts the system as soon as the timer expires. In our experiments we set the timeout for periodic restart $T_{\text{Timer}} = 120$ seconds.

B. Experience and Evaluation

In the following we analyze the effectiveness of ReSecure by mimicking some of the attacks described in the preceding section. In particular we illustrate how different restarting strategies can be useful to recover from these attacks.

1) Recovery by Watchdog Timeout: As mentioned in Section III, an attacker can capture the control tasks by tracking the execution profile using side channel attacks [21] that exploit the deterministic behavior of the real-time scheduling. We perform system-level DoS by launching a fork bomb attack on the complex controller. This DoS attack utilizes Linux `fork()` system call and recursively replicate itself. Due to the CPU overload some (or all) of the control tasks may miss their deadlines. In this experiment we assume that the attacker hijacks the highest priority real-time control task and is able to execute malicious codes with the privilege similar to those of legitimate tasks (viz., `root`). Recall that we update the watchdog timer once during every invocation of controller tasks. Since the highest priority controller task is itself compromised with self-replicating malicious codes, the watchdog timer is not updated and a restart event is triggered. The system is recovered from the attack within less than 14 seconds (including $T_{\text{R}} = 13$ seconds reboot time). Again, the safety constraints in Eq. (1) are not violated since the safety controller takes the control during restart.

2) Recovery by Monitoring Unit and Periodic Restart: Similar to previous experiment, we assume that attacker can hijack the control tasks and is able to inject malicious codes. However, unlike Section VI-B1 where the attacker overrides the highest priority task, here the attacker may take control over any arbitrary priority task (referred to as victim task). This is particularly useful for the attacker to simply lodge itself in the system and glean sensitive information [21]. To demonstrate the attack, we inject malicious code (viz., a kernel-level malware [30] that intercepts every `read()` system call) inside victim task. Since the original malware was developed for the x86 architecture, we modify the source code and port it for ARM. We also slightly modify the malware logic so that instead of making any changes, it silently lodges in the

---

3https://beagleboard.org/black.

4Technically it is possible to override/corrupt the on-board OS image of BBB. However, this requires physical intervention to hard-flash the board by using the reset (viz., USER/ROOT) switch and may not be easily doable in practice. Hence, we consider on-board eMMC as a secure storage to load the OS image after each restart.

5By some careful engineering it could be possible to make file system writable even if it is mounted as read-only. In our current prototype, however, we do not consider this issue. As we have mentioned in Section IV, the OS in complex unit can also be loaded from read-only physical disc that can only be modifiable by external access.

6We intend to port the safety unit in an embedded platform in future work.

---

system and extracts information (i.e., representing side channel attacks).

The system can recover from this attack by two means, either the monitoring unit detects the presence of unauthorized entity and/or by periodic restart. We experiment with both alternatives. In our experiments we consider that monitoring unit remains uncompromised and periodically (with period $T_{MU} = 2$ seconds) scans the currently running kernel modules and triggers a restart if any malicious entity is found. In both cases, the system recovers from the intrusion within $T_R + \delta$ seconds of attack where $\delta = T_{MU} - t$ if the restart is triggered by monitoring unit or $\delta = T_{Timer} - t$ when system reboots due to periodic timeout where $t \in [0, T_{Timer}]$ represents relative point of intrusion.

VII. DISCUSSION

Despite the fact that ReSecure provides an integrated approach to guarantee safety and security in real-time CPS, this is obviously not a silver bullet for solving all security problems. In what follows we briefly discuss the defense mechanisms of ReSecure against different threat model as well as the limitations of the current framework with possible directions of improvements.

A. Threat Model

The underlying detection algorithm used in the monitoring unit may have false positives that will trigger unwanted restarts. A few false positives will not have a significant impact on the performance. False negatives are also mitigated by the periodic timers that set an upper bound on the detection time and restart the system independent of the monitoring unit or watchdog timers. Hence, the basic safety and security premise of ReSecure does not depend on having a perfect monitoring unit.

There may be cases where an advanced attacker performs a series of attacks with a view to destabilizing the system. Whether the attack is being detected by the monitoring unit and/or by watchdog timeout, this will cause the system to be restarted frequently. Despite the fact that frequent restart may affect (or degrade) system performance, ReSecure ensures that the system will still remain safe during the sporadic unavailability of the complex unit. This kind of attack, however, will require the adversary to intrude into the system and launch series of attacks as soon as the fresh OS image is loaded. Again, we can argue that this will be difficult and unlikely in practice.

B. Limitations and Improvement

ReSecure requires a dedicated computing module as a safety unit due to periodic/asynchronous unavailability of complex unit. However, this overhead and performance loss comes with increased security guarantees – that might be acceptable for many security-critical RTS.

Although ReSecure can recover from certain types of DoS, code injection and side channel attacks, the restart mechanism does not prevent intrusion. As we have illustrated in the Appendix, the frequency of periodic restarts is calculated based on the prior knowledge of exploits. Therefore, ReSecure may not work as expected for zero-day vulnerabilities. For instance, the attacker may launch successful attacks between two consecutive restart events. However, compared to non-restart based mechanisms, the proposed framework ensures a clean (e.g., uncompromised) system state every once in a while with guaranteed safety at the cost of restarts.

Given that certain side channel attacks are less effective if we are utilizing the restart mechanism, ReSecure in its current form, however, does not ensure zero information leakage from such attacks. This is because, there is always the possibility that the attacker can extract sensitive information between consecutive restarts. This issue can be mitigated by randomizing execution patterns and system configurations after every restart. For example, researchers have proposed schedule obfuscation method aiming at randomizing the real-time task schedule [31], cache random-eviction and cache random permutation [32] as well random key distribution [33]. Such techniques can be utilized along with ReSecure to randomize system parameters after each restart and further reduces the chance of information leakage. We intend to incorporate such techniques on top of the restart-based recovery mechanism in future work.

VIII. RELATED WORK

The Simpex architecture [10], [34] has been used extensively to provide verified design with unverified logic. Variants of Simpex have been proposed to account for physical system failures [35], as well as faults in OS or processor [9]. These works have focused on fault-tolerance and are not concerned with the security issues in RTS.

The notion of restarting has been used in some studies to recover from faults in safety-critical systems. System-level Simpex is the first architecture that is able to restart the complex controller [9] with formal safety guarantees. Reset-based recovery for safety-critical systems [13] has further analyzed restarting as a recovery mechanism. Both of these approaches utilize extra hardware to isolate safety subsystem from faults in complex subsystem. The objective is to recover the system safely from non-adversarial software faults in the controller and underlying layers (OS and firmware). To our knowledge, ReSecure is the first work that extensively considers enabling systematic restart-based recovery with guaranteed safety to secure the RTS.

Although sophisticated adversaries and malware developers are able to overcome air-gaps in recent years, security issues considering hard real-time constraints have not been studied comprehensively in literature. Recent work [16], [17], [36] on dual-core based hardware/software architectural frameworks aim to protect RTS against security vulnerabilities. A similar line of work exists [37] where authors leverage the deterministic execution behavior of RTS and use Simpex architecture to detect intrusion while guaranteeing the safety. However the solution proposed by that work is limited to time-based intrusion detection methods only. In contrast, ReSecure utilizes the concept of restarting on top of Simpex and able to prevent and/or recover from large classes of attacks.

IX. CONCLUSION

The evidence from recent successful attacks on UAVs [38], automobiles [39] and industrial control systems [1] indicate that security violations are becoming more common in RTS. In this work, we are moving towards developing an integrated security-aware RTS and illustrate restart as a viable mechanism to ensure security in such safety-critical systems. Designers of such systems can now evaluate the necessary trade-offs between control system performance degradation and increased security guarantees – thus improving the overall design of RTS in the future.
REFERENCES


APPENDIX

ANALYZING THE IMPACT OF RESTART

For a given system we represent the set of possible exploits with $E = \{e_1, e_2, ..., e_N\}$. Each, $e_j$ is associated with a probability of occurrence $P(e_j)$ which specifies how common the exploit is. From definition we have $\Sigma_j P(e_j) = 1$. Here, $T_R$ is the time for a full complex unit restart, $T_{Com}$ is the time the attacker needs to re-launch the attack (During this time system is not compromised yet), $T_{Detect}$ is the time it takes for the monitoring unit to detect the attack and initiate a restart; $T_{Timer}$ is the fixed time for the periodic timer to restart the system.

For each exploit, $T_{Com}$ and $T_{Detect}$ are defined as random variables associated with a probability density function $P_{Com}(t)$ and $P_{Detect}(t)$. The shape of $T_{Com}(t)$ depends on vulnerability level of the platform and the effectiveness of $e_j$. In a similar fashion, the shape of $P_{Detect}(t)$ depends on the effectiveness of the intrusion detection techniques used in the monitoring unit for detecting $e_j$. Furthermore, a monotonically increasing function called $Damage^2(t)$, is associated with each exploit that defines how much damage $e_j$ causes within $t$ seconds after being effective. The probability density functions and damage function are assumed to be provided by the system designer using the analytic and experimental data and from the knowledge of the system, exploits and detection mechanisms.

In this setting, the only configurable variable is the restart period of the periodic timer, $T_{Timer}$. Very short restart periods can limit the attacker; however, it also reduces the time that the complex unit is available. On the other hand, a long $T_{Timer}$ can increase the risk of damage to the system by the exploits. In what follows, we show how to find the optimal $T_{Timer}$ for a given system.
The expected time that the complex unit is available and not compromised can be computed from the following:

\[ \text{Exp}(T^j_{\text{Com}}) = \int_{t=0}^{T_{\text{Timer}}} t \cdot P^j_{\text{Com}}(t) \cdot dt + T_{\text{Timer}} \cdot \int_{t=T_{\text{Timer}}}^{\infty} P^j_{\text{Com}}(t) \cdot dt. \]  

(2)

Besides, the expected time that it takes for the monitoring unit to detect an exploit is given by

\[ \text{Exp}(T^j_{\text{Detect}}) = \int_{t=0}^{T_{\text{Timer}}} \left( \int_{\tau=0}^{T_{\text{Timer}}-t} \tau \cdot P^j_{\text{Detect}}(\tau) \cdot d\tau + (T_{\text{Timer}}-t) \cdot \int_{\tau=T_{\text{Timer}}-t}^{\infty} P^j_{\text{Detect}}(\tau) \cdot d\tau \right) \cdot P^j_{\text{Com}}(t) \cdot dt. \]  

(3)

The expected restart period for an exploit can be obtained as follows

\[ \text{Exp}(T^j_{\text{Period}}) = T_R + \text{Exp}(T^j_{\text{Com}}) + \text{Exp}(T^j_{\text{Detect}}). \]  

(4)

Restarting the system, increases the unavailable time of the system. The expected percent of the time that the system is not available can be computed as follows

\[ \text{Exp(Unavailability)} = \sum_{e_j \in E} P(e_j) \left(1 - \frac{\text{Exp}(T^j_{\text{Com}})}{\text{Exp}(T_{\text{Period}})}\right). \]

For a fixed \(T_{\text{Timer}}\) value, the expected damage of an exploit can be computed using the following probabilistic distribution of the time to compromise and the time to detect the exploit.

\[ \text{Exp(Damage)}^j = \int_{t=0}^{T_{\text{Timer}}-t} \left( \int_{\tau=0}^{T_{\text{Timer}}-t} \text{Damage}^j(\tau) \cdot P^j_{\text{Detect}}(\tau) \cdot d\tau + \text{Damage}^j(T_{\text{Timer}}-t) \cdot \int_{\tau=T_{\text{Timer}}-t}^{\infty} P^j_{\text{Detect}}(\tau) \cdot d\tau \right) \cdot P^j_{\text{Com}}(t) \cdot dt. \]  

(5)

Therefore, total expected damage from all the exploits on the system is given by

\[ \text{Exp(Total Damage)} = \sum_{e_j \in E} P(e_j) \cdot \text{Exp(Damage)}^j. \]

The expected total damage of exploits and the unavailable time of the system are inversely related. Restarting the system more frequently would decrease the expected damage from security exploits and at the same time will reduce the availability of complex controller. We formulate this as a minimization problem to find a \(T_{\text{Timer}}\) that minimizes the following Cost function:

\[ \text{Cost}(T_{\text{Timer}}) = \text{Exp(Total Damage)} + \alpha \cdot \text{Exp(Unavailability)}. \]

(6)

In the above, parameter \(\alpha\) determines the importance of availability versus the of risk being compromised. This is a design choice based on the applications of the physical systems under control. Note that, more available time for the complex controller can increase system performance. For systems in which security is far more important than performance, a small value for \(\alpha\) is desirable. This leads to a smaller value for \(T_{\text{Timer}}\), thus leading to a higher security but less available time and control performance.

As seen in the plot of the cost functions for two examples, the optimal \(T_{\text{Timer}}\) (the value of \(T_{\text{Timer}}\) that minimizes the Cost function) for Example 1 is smaller than the Example 2. Intuitively, the system in Example 1 needs less frequent restarts. Moreover, the different patterns in the compromise and detection times in the two examples, leads to a larger cost for the system in the second example for the same values of \(\alpha\). For instance, in case of \(\alpha = 160\) (plotted with cyan color), minimum cost is 11.2 for \(T_{\text{Timer}} = 103\) seconds in Example 1 and 65.7 for \(T_{\text{Timer}} = 30\) seconds in Example 2.
Timing Analysis of Secure Communication between Resource Managers in DREAMS

Gautam Gala¹, Thomas Koller², Daniel Gracia Pérez³, Gerhard Fohler¹ and Christoph Ruland²

¹ Chair of Real Time Systems, Technische Universität Kaiserslautern, Kaiserslautern, Germany
{gala, fohler}@eit.uni-kl.de
² Chair for Data Communications Systems, Universität Siegen, Siegen, Germany
{thomas.koller, christoph.ruland}@uni-siegen.de
³ Thales Research & Technology, Palaiseau, France
daniel.gracia-perez@thalesgroup.com

Abstract
The European FP7 project DREAMS provides services for system-wide adaptability of mixed-criticality applications consuming several resources by means of a global resource manager (GRM) in combination with several local resource managers (LRMs). The GRM has an abstract system-wide view and makes global decisions, while the LRM control individual resources in isolation. The LRM s regularly communicate with the GRM in order to deal with unpredictable environment situations, resource fluctuations and the occurrence of faults. Since resource management communication deals with critical system information, it is an promising target for an attacker. This poses an increased risk of malicious attacks on the system. Therefore, security mechanisms have been implemented to ensure an adequate protection of the system’s resource management. A Security and a communication library have been developed as a proof of concept. This paper analyses the overhead taken by resource management communication for secure information exchange. Furthermore, the analysis includes different security algorithms, i.e., two cipher algorithms and two mode of operation algorithms.

pp. 55-60
A Server Model to Integrate Security Tasks into Fixed-Priority Real-Time Systems

Monowar Hasann¹, Sibin Mohan¹, Rakesh B. Bobba² and Rodolfo Pellizzoni³

¹ Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
² School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA
³ Dept. of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada

Email: {mhasan11, sibin}@illinois.edu, rakesh.bobba@oregonstate.edu, rodolfo.pellizzoni@uwaterloo.ca

Abstract
Modern embedded real-time systems (RTS) are increasingly facing security threats than the past. In this paper, we develop a unified framework by using the concept of server and propose a metric to integrate security tasks into RTS that will allow system designers to improve the security posture without affecting temporal constraints of the existing real-time tasks. We demonstrate our framework using a proof-of-concept implementation on an ARM-based embedded platform and realtime Linux OS.

pp. 61-68