Electronic Communications of the EASST
Volume 72 (2015)

Proceedings of the 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015)

Verifying Critical Cyber-Physical Systems After Deployment

Colin O'Halloran
15 Pages

Guest Editors: Gudmund Grov, Andrew Ireland
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verifying Critical Cyber-Physical Systems After Deployment

Colin O'Halloran

University of Oxford

Abstract: Cyber-Physical Systems (CPS) are increasingly novel hardware and software compositions creating smart, autonomously acting devices, enabling efficient end-to-end workflows and new forms of user-machine interaction. The heterogeneous, evolving and distributed nature of CPS means that there is little chance of performing a top down development or anticipating all critical requirements such devices will need to satisfy individually and collectively. This paper describes an approach to verifying system requirements, when they become known, by performing an automated refinement check of its composed components abstracted from the actual implementation. This work was sponsored by the Charles Stark Draper Laboratories under the DARPA HACMS project.

Keywords: Autonomy, Cyber-Physical Systems, Internet of Things, Verification, Validation, Security, Safety

1 Introduction

Autonomy does not reside in one particular software system or Electronic Control Unit (ECU). Autonomy arises out of the collection of ECUs responsible for: processing sensor inputs, i.e. **perceiving**; **deciding** what to do; and **acting** upon the decision made. Establishing the safety of an autonomous system, therefore, means assuring the composition, of these Perceive-Decide-Act functions, is safe. There is currently a lot of work on the development of driverless cars as well as other autonomous air and maritime vehicles. In the automotive sector a manufacturer might well develop the decision-making software themselves, but their business model depends upon buying and integrating third-party ECUs from their supply chain with decision-making software. For example, Bosch supplies systems to many car manufacturers and their software design is their protected IP. This means that autonomy relies upon third-party software for which limited assurance is available. To varying degrees the same is true of the aerospace and maritime sectors.

More generally Cyber-Physical Systems (CPS) are increasingly novel hardware and software compositions creating smart, autonomously acting devices, enabling efficient end-to-end workflows and new forms of user-machine interaction. Currently CPS range from large supervisory control and data acquisition (SCADA) systems (which manage physical infrastructure) to medical devices such as pacemakers and insulin pumps, to vehicles such as airplanes and satellites. In the future they will include low power small devices that will make up the Internet of Things.

Clearly resilience, safety, security and privacy are of paramount concern. However, the heterogeneous, evolving and distributed nature of CPS means that there is little chance of performing a top down development or anticipating all critical requirements such devices will need to satisfy individually and collectively.

The looming problem is therefore how to address the verification and validation of such systems when they are critical and rely upon other systems that have limited assurance? Worse still, how can CPS be verified when the requirements will not be known until after they are deployed and composed with other CPS to form unanticipated critical services?

An approach to addressing this problem is to automatically extract the behaviours of such deployed systems that are composed together through well-defined architectures, or protocols. At this point system requirements can be articulated because the overall system of systems has been constructed for some purpose. This means that a system level representation of the relevant behaviours is needed.

The importance of pushing the representation up to the level of system properties is that it is the natural level for people to articulate requirements, it is often difficult to express these as lower level specifications for components unless it is part of a top down development. Such top down development has already been excluded from the problem space being addressed. Verifying components against specified lower level properties that are generally desirable is rather hit and miss. The history of developing secure systems is strewn with examples of systems where low level security properties to hold, but the system is insecure. Conversely it is also possible for the system level to be secure, but low level vulnerabilities to be present – but not exploitable at the system level. There is no reason to believe that the same is not true for systems that are required to be safe.

Another advantage of working with system level requirements is that they can guide what behaviours are relevant to the system level requirements, i.e. they drive relevant abstractions that can be performed. For example a sub-system on a CAN bus might have no relevant critical behaviour except that it does not flood the CAN bus with messages when critical messages are being passed between other sub-systems.

In the past the approach advocated in this paper would have been perceived to have too many obstacles for any chance of success. Advances in various areas of automated reasoning and model checking now, perhaps, make such an approach possible. In the rest of the paper a small example representing encrypted communication to a hobbyist quadcopter is given to illustrate the approach. The software written in C is abstracted to a system level and then verified against a property that encodes the sort of attack a penetration tester might try. Finally future work that employs de-compilation of executable binary into a mathematical representation is outlined along with related work.

2 The System

The example of a system requirement used to illustrate the approach is that of absence from the system's behaviour of one type of penetration tester attack. The simplified example system is based upon an early version of the SMACMMPilot [1] developed under the DARPA High Assurance Cyber Military Systems (HACMS) project [2]. SMACMMPilot is an open-source autopilot software for small unmanned aerial vehicles (UAVs) using new high-assurance software methods, the simple C code used in the example is not from SMACMMPilot.

The relevant part of the system for the requirement of interest is that of encrypted pairwise communication between a ground station and SMACMMPilot and a safety controller and SMACMMPilot, illustrated in Figure 1.

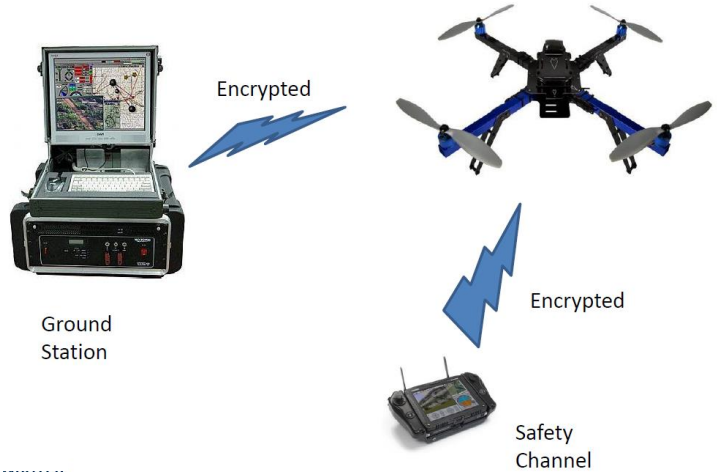

Figure 1: SMACCMPilot communication channels

Within the quadcopter's software there are two channels: Stream1 and Stream2. Stream1 consists of: an internal buffer (for the packets transmitted from the ground station), denoted GCSXMIT; a decryption function; and a process that assembles the message for processing according to the MAVLINK protocol [3]. A diagram representing Steam1 and Stream2 appears in Figure 2.

.

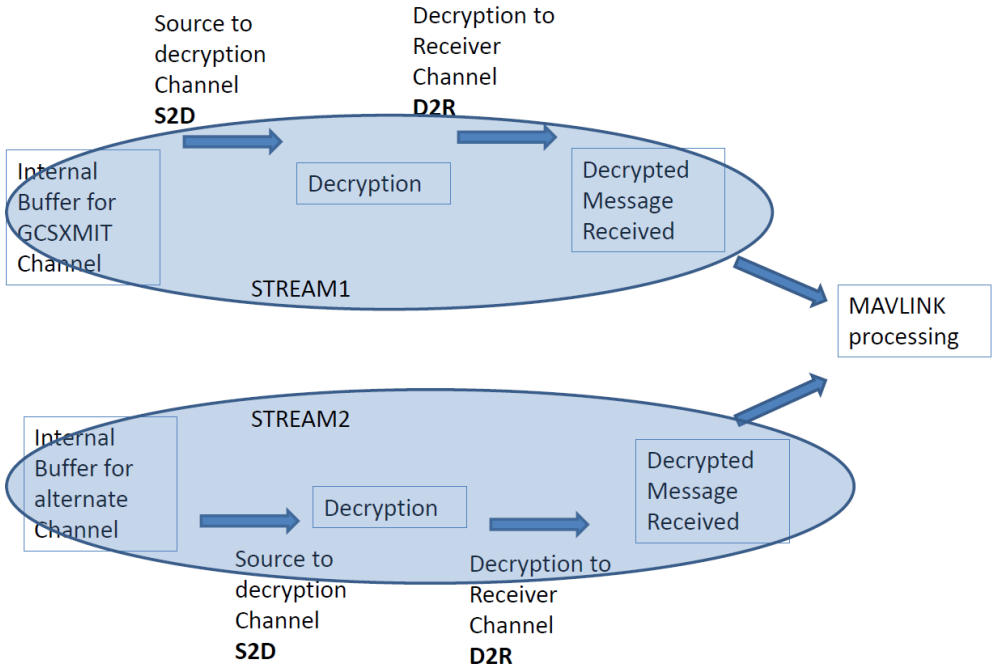

Figure 2: Diagrammatic representation of communication to the quadcopter

Stream2 is a copy of Stream1. The implementation of Stream1 is in terms of 4 reactive processes that use 2 system services (send and receive) to communicate with each other. The 4 processes are GCS, which models the ground station transmitting data; DLR, which takes the data from GCS and sends it onward to be decrypted; DECRYPT, which decrypts the data and sends it onward to be dealt with by RECVR before sending onwards to the MAVLINK protocol software.

In the process algebra of Communicating Sequential Processes [4], CSP, the architectural composition of the streams of Figure 2 is shown in Figure 3.

```
(STREAM1 [| {|GCSXMIT|} |] STREAM2) [| {| R2M |} |] MAVLINK(R2M)
```

Figure 3: CSP composition of communication streams with MAVLINK

STREAM1 and STREAM2 communicate through the messages over the channel GCSXMIT. MAVLINK communicates with STREAM1 and STREAM2 through the messages over R2M. The messages over GCSXMIT consist of symbolic values for a key, a sequence number and data. The messages over R2M consist of a sequence number and data. In CSP the composition of STREAM1 is in terms of the 4 processes GCS, DLR, DECRYPT and RECVR is shown in Figure 4.

```
STREAM1 =
GCS(Sk,GCSXMIT)
[| {|GCSXMIT|} |]
( DLR(GCSXMIT, D2D)
  [D2D <-> D2D]
  ( DECRYPT(Sk, D2D, D2R) [D2R <-> D2R] RECVR(0, D2R, R2M.CMsgStruct.0)  )
)
```

Figure 4: Composition of STREAM1

The messages over D2D (like GCSXMIT) consist of the symbolic values for a key, a sequence number and data. The C code for the function GCS, is shown in Figure 5.

```
void GCS(int Key, chan* out) {
  SMsg GCS_local_message;
  GCS_local_message.Key = Key;
  GCS_local_message.cmsg.SeqNo = 0;
  GCS_local_message.cmsg.msg.val = 0;
  for (;;) {
    GCS_local_message.cmsg.SeqNo = (GCS_local_message.cmsg.SeqNo+1) % 4;
    send(&GCS_local_message, sizeof(GCS_local_message), out);
  }
  return;
}
```

Figure 5: C function GCS

Note that the function GCS does not terminate because the 'for' loop, which calls a service to send a packet onward, does not terminate. The simplifications of functionality do not detract from the points this paper makes about architectural composition, abstraction and verification of system properties.

The composition of STREAM2 is similar to STREAM1 (shown in Figure 4), but a process modelling an intruder's behaviour replaces the process GCS in Figure 6.

```
STREAM2 =
INTRUDER(GCSXMIT, ICPY)
[ICPY<->ICPY]
( DLR(ICPY, D2D)
  [D2D <-> D2D]
 ( DECRYPT(Sk, D2D, D2R)  [D2R <-> D2R]  RECVR(0, D2R, R2M.CMsgStruct.1) )
)
```

Figure 6: Composition of STREAM2

The intruder's behaviour is modelled abstractly as listening to the transmissions from the ground station and replaying them, or non-deterministically ignoring a transmission. The CSP process modelling this behaviour is shown in Figure 7.

```
INTRUDER(in, out) =
in?x -> (out!x -> INTRUDER(in, out))
|~|
in?_ -> INTRUDER(in, out)
```

Figure7: Intruder behaviours

The behaviours of GCS, DLR, DECRYPT, RECVR and MAVLINK are determined by abstracting the code to a predicate representing initial and final states. The abstracted predicate is then translated into CSP enabling the overall system composition, shown in Figure 3, to be verified with respect to a property of robustness against a certain form of external attack.

3 Abstracting the implementation into CSP

MAVLINK is used to illustrate the abstraction of the C code (shown in Figure 8) to a specification and then translated into CSP.

```
int MAVLINK(chan* in) {
  SMsg MAVLINK_local_x;
  do
    { recv((SMsg*) &MAVLINK_local_x, sizeof(MAVLINK_local_x), in);}
  while (1);
  return 1;
}
```

Figure 8: C code for MAVLINK

The code simply acts as a sink for the decrypted messages, the extra functionality for a real implementation would simply be incorporated into the abstracted predicate.

A tool, called FSG and developed by D-RisQ under the DARPA HACMS project, essentially generates the strongest post-condition for a C function. The tool, implemented on top of ProofPower's QCZ [5] tool, generates as output the C compliance notation - a wide spectrum language that includes specifications in the Z language as well as C code. When FSG is run on the C function of Figure 8 it generates the specification statement shown in Figure 9.

C Compliance Notation

$$
\begin{array}{l}
int \\
\ MAVLINK(chan\ \ *in) \\
\ \ \Xi \\
\ \ [((int \\
\qquad)_{\mathbb{Z}}^{\lceil}\mathbb{B}_C\ if \\
\qquad\qquad (DoLoopTest0 \lhd DoLoopBody0)^* \rhd DoLoopTest0 \neq \varnothing\ then \\
\qquad\qquad (in_v,\ MAVLINK\_local\_x_l) = DoLoop0\ (in_v,\ MAVLINK\_local\_x_l)\ else \\
\qquad\qquad true^{\rceil} \\
\ \ ) \ \&\&\ (MAVLINK! == (1))]\ MAVLINK\_1
\end{array}
$$

Figure 9: FSG output for MAVLINK

The specification states that if the "do loop" within MAVLINK doesn't terminate then the result is a relational closure called 'DoLoop0'. Note that neither parameter to 'DoLoop0' is changed by 'DoLoop0'. The point of the predicate is not to reason about it directly to establish some assertion; it is there to record sufficient information in order to translate into CSP.

C Compliance Notation

$$
\begin{array}{l}
MAVLINK\_1 \sqsubseteq \\
\\
\ \ \{\ \Delta\ \ [1, \\
\qquad ((int \\
\qquad\quad)_{\mathbb{Z}}^{\lceil}\mathbb{B}_C\ if \\
\qquad\qquad (DoLoopTest0 \lhd DoLoopBody0)^* \rhd DoLoopTest0 \neq \varnothing\ then \\
\qquad\qquad (in_v,\ MAVLINK\_local\_x_l) \\
\qquad\qquad\quad = DoLoop0\ (in_v,\ MAVLINK\_local\_x_l)\ else \\
\qquad\qquad true^{\rceil} \\
\qquad)]\ MAVLINK\_2 \\
\\
\ \ return\ 1; \\
\ \ \}
\end{array}
$$

Figure 10: First refinement step

The soundness of the abstraction is established by reversing it into refinement steps that generate verification conditions that, if proven by ProofPower, mean that the original C code is

correct with respect to the specification statement. For example, Figure 10 is a refinement of Figure 9; and Figure 11 is a refinement of Figure 10. ProofPower's QCZ tool processes these refinement steps to generate verification conditions that can then be proven by ProofPower.

C Compliance Notation

$$MAVLINK\_2 \sqsubseteq$$

```
do { recv((SMsg *)&MAVLINK_local_x, sizeof(((MAVLINK_local_x))), in); }
   while (1);
```

Figure 11: Second refinement step

Returning to Figure 9, it is translated into CSP as the process MAVLINK defined in Figure 12.

MAVLINK(in) = DoLoop0(in, (| |), SKIP)

Figure 12: Definition of the CSP process MAVLINK

The communication channel 'in' is a parameter of MAVLINK, reflecting the channel 'in' passed as a parameter to the C function MAVLINK in Figure 9.

The definition of DoLoop0 is a relational composition of a function called DoLoopBody0 with its own reflexive transitive closure, i.e. the loop body sequentially composed with a "guarded" set of all finite iterations of the loop body. The guard takes the form of domain restriction. The co-domain anti-restriction characterises the state on loop termination, if the loop does not terminate then this is the empty set.

z

$$DoLoop0 : VALUE_C \times LOC_C \rightarrow VALUE_C \times LOC_C$$

$$\forall\, in_V : VALUE_C;\ MAVLINK\_local\_x_L : LOC_C$$
$$\bullet\ DoLoop0\ (in_V,\ MAVLINK\_local\_x_L)$$
$$= (DoLoopBody0\ \fatsemi\ (DoLoopTest0 \lhd DoLoopBody0)^* \rhd DoLoopTest0)$$
$$(in_V,\ MAVLINK\_local\_x_L)$$

Figure13: FSG generated Z axiomatic definition of DoLoop0

As the name implies, the "do loop" in the C code executes the body of the loop before the loop guard is evaluated. In this example the loop guard is true and the body is executed indefinitely giving the behaviour of a reactive system because of the call of the external function 'recv' used as part of the definition of DoLoopBody0 in Figure 17. The translation of DoLoop0 into machine readable CSP, called CSPm, is shown in Figure 14. It models predicate equalities by use of a let clause and then calls the externally provided function of a receive service that takes the packets of information decrypted by streams 1 and 2.

```
DoLoop0(in_V, Indirections,Q) =
let
  recv__x = MAVLINK_local_x_L
  recv__y = SizeOf(SMsg)
  recv__queue = in_V
within
    EXTERNAL_recv(recv__x,recv__queue,Loop0,in_V, Indirections,Q)
```

Figure 14: CSP translation of DoLoop0

The external function has two arguments that in the CSP are augmented by three more that are used to supply continuation information for subsequent behaviour. The augmented definition of 'EXTERNAL_recv' is shown in Figure 15.

```
EXTERNAL_recv(x_v',queue,P,i,Indirections,Q) =
let
  Input_Channel = queue
within
    Input_Channel?Input_Value ->
    let
      Indirections1 = mapUpdate(Indirections,x_v',Input_Value)
    within
        P(i,Indirections1,Q)
```

Figure 15: Pre-defined definition of the external service recv

The argument 'Indirections' is a map function of the Haskell like functional language that augments the process algebra of CSP that is CSPm. The map function 'Indirections' models simple pointers, such as those used for efficient parameter passing. The process EXTERNAL_recv is defined by a human as part of the model of the CSPm model of the system architecture. It simply takes an input value received and updates the map function that models pointers. It then behaves like the process P supplied as an argument to EXTERNAL_recv with P's required arguments, which also includes another continuation process Q. The actual parameter, Loop0, substituted for P in EXTERNAL_recv is shown in Figure 14.

```
Loop0(in_V, Indirections,Q) =
if true
then
   DoBody0_Star( in_V, Indirections,Loop0,in_V,Indirections,Q)
else
   let
     MAVLINK_output = true
   within
       Q
```

Figure 16: Process supplied as part of a continuation parameter in DoLoop0

The continuation process Loop0, defined in Figure 16, corresponds to the translation of the reflexive transitive closure of the loop body. The loop guard is true therefore the loop never

terminates and hence, in the process definition, the ‘else’ part can never be taken. In general if a loop terminates the process Q takes over the flow of execution. The CSPm definition of DoBody0_Star is translated from the FSG generated axiomatic definition of DoLoopBody0 (used in the definition of DoLoop0 in Figure 14); DoLoopBody0 is shown in Figure 17.

z

$$
\begin{array}{l}
DoLoopBody0 : VALUE_C \times LOC_C \rightarrow VALUE_C \times LOC_C \\
\hline
\forall\ in_V : VALUE_C;\ MAVLINK\_local\_x_L : LOC_C \\
\quad \mid \mathbb{B}_C\ (\forall\ recv\_\_queue,\ recv\_\_recv,\ recv\_\_x,\ recv\_\_y : VALUE_C;\ \sigma, \sigma' : STORE_C \\
\qquad \mid recv\_\_x = PointerVal_C\ \ MAVLINK\_local\_x_L.addr \\
\qquad\quad \land\ recv\_\_y \\
\qquad\qquad = IntVal_C \\
\qquad\qquad\quad (SizeOf_C \\
\qquad\qquad\qquad (StructType_C \\
\qquad\qquad\qquad\quad \langle(\text{"}Key\text{"},\ ArithmeticType_C\ (Signed_C\ Int_C)), \\
\qquad\qquad\qquad\quad\ (\text{"}cmsg\text{"}, \\
\qquad\qquad\qquad\quad\ StructType_C \\
\qquad\qquad\qquad\quad\quad \langle(\text{"}SeqNo\text{"},\ ArithmeticType_C\ (Signed_C\ Int_C)), \\
\qquad\qquad\qquad\quad\quad\ (\text{"}msg\text{"}, \\
\qquad\qquad\qquad\quad\quad\ StructType_C \\
\qquad\qquad\qquad\quad\quad\quad \langle(\text{"}val\text{"}, \\
\qquad\qquad\qquad\quad\quad\quad\ ArithmeticType_C \\
\qquad\qquad\qquad\quad\quad\quad\quad (Signed_C\ Int_C))\rangle\rangle\rangle\rangle\rangle\rangle) \\
\qquad\quad \land\ recv\_\_queue = in_V \\
\qquad \bullet\ (recv_{post} \\
\qquad\quad [recv\_\_queue/queue_v, \\
\qquad\quad recv\_\_recv/recv_v!, \\
\qquad\quad recv\_\_x/x_v, \\
\qquad\quad recv\_\_y/y_v])) \\
\quad\ \neq IntVal_C\ 0 \\
\quad \bullet\ DoLoopBody0\ (in_V,\ MAVLINK\_local\_x_L) = (in_V,\ MAVLINK\_local\_x_L)
\end{array}
$$

Figure 17: FSG generated Z axiomatic definition of the loop body

Although it looks (at first sight) to be complicated, Figure 17 is simply setting the abbreviations for expressions that are substituted for formal parameters of the external function ‘recv’. For example the identifier ‘recv__x’ is an abbreviation for the address of a pointer and ‘recv__y’ is an abbreviation for the size of the structure passed to ‘recv’. The identifier $recv_{post}$ is the schema predicate that is the specification for the external function ‘recv’ and is defined by a human like the corresponding CSPm process.

The translation into CSPm of DoLoopBody0 of Figure 17 is shown in Figure 18.

```
DoBody0_Star(in_V, Indirections,P,i,j,Q) =
let
  recv__x = MAVLINK_local_x_L
  recv__y = SizeOf(SMsg)
  recv__queue = in_V
within
      EXTERNAL_recv(recv__x,recv__queue,P,i,j,Q)
```

Figure 18: Translation of DoLoopBody0 into CSPm

The definition of the process DoBody0_Star is similar to that of DoLoop0 defined in Figure 14 and reflects the axiomatic definition of the loop body in Figure 17. The required recursion is achieved by supplying 'Loop0' as the continuation P (along with Loop0's arguments 'in_v' for 'i' and 'Indirection' for 'j') that is supplied to EXTERNAL_recv in Figure 16.

The translation of the reactive process MAVLINK into CSPm along with the similarly translated C code implementations of the reactive processes, which make up Stream1 and Stream2, can be plugged into CSPm system architecture definition of Figure 3.

4 Verification of a System Property

The encrypted communication over two streams (one for a ground control station and one potentially for a safety controller) is represented in CSPm as

```
SYSTEM = (STREAM1 [| {|GCSXMIT|} |] STREAM2) [| {| R2M |} |] MAVLINK(R2M)
```

Figure 19: CSPm SYSTEM definition

The process MAVLINK is represented by the CSPm process

```
MAVLINK(in) = DoLoop0(in, (| |), SKIP )
```

where the definition of DoLoop is derived from the specification generated from the FSG tool that abstracts the C code implementation, as described in section 3. Thus the behaviour of MAVLINK instantiated with the channel R2M can be plugged into the SYSTEM definition above. Similarly the behaviours of the process that make up STREAM1 and STREAM2 can be derived from their implementation in a similar manner to that described in section 3.

The quadcopter is a Cyber–Physical system that was initially developed without reference to potential penetration tester type attacks. A form of external attack is to interfere with the commands sent from the ground control station by replaying them through the second communication stream. A specification for robustness against this type of attack is that once started with a particular input stream, it must not be possible to receive messages from the other stream. The specification can be specified operationally as the CSPm process in Figure 20.

```
SPEC1 = R2M?CMsgStruct.x.msg -> SPEC1'(x)
SPEC1'(x) = R2M!CMsgStruct.x?_ -> SPEC1'(x)
```

Figure 20

Performing a refinement check using the FDR3 model checker of SYSTEM (shown in Figure 19) against the specification in Figure 20 returns a failure with a counterexample of undesirable system behaviour. The weakness is that because the implementation of Stream2 is a copy of Stream1 it can accept replayed messages from the other stream. The same system representation can be used to check other system security properties specified in CSPm.

5 Related Work

The High Assurance Cyber Military Systems (HACMS) project [2] is creating technology for the construction of high-assurance Cyber-Physical Systems, where high assurance is concerned with functional correctness and satisfaction of appropriate safety and security properties. The work described in this paper is part of the "Red Team" effort that is seeking to use "traditional" penetration testing techniques allied to analysing the system using automated verification. In section 6, ongoing work is discussed that seeks to de-compile executable binary into the C compliance notation from which a specification can be generated and translated into CSPm. Thus the aim is to use the C compliance notation as an intermediate representation generated from a de-compiler.

Recent advances indicate that the time is ripe for verifying properties of systems. A number of projects on systems verification have emerged, most notably, the L4 verified OS Kernel project [6]. The project developed and formally verified a high-performance microkernel on ARM and x86 architectures. Such microkernels are a critical core component of modern embedded systems architectures. They are the piece of software that has the most privileged access to hardware and regulates access to that hardware for the rest of the system. Virtually every modern smart-phone runs a microkernel quite similar to seL4.

In the rest of the HACMS project a clean-slate approach has been adopted using formal methods-based approaches to enable semi-automated code synthesis from executable, formal specifications. In addition to generating code, HACMS seeks a synthesizer capable of producing a machine-checkable proof that the generated code satisfies functional specifications as well as security and safety policies. Work on extending the verified L4 kernel to an RTOS is taking place within HACMS. All the preceding work described involves top-down development of software from scratch where the safety and security requirements are known. The approach advocated in this paper aims to address software that already exists and when requirements only emerge later.

Another notable example is the Verisoft and Verisoft XT projects [7]. They verify special-purpose operating systems and a hypervisor, at the level of source code but without I/O. There have also been systems verifications that focus on safety properties, most notably, Yang and Hawblitzel's type-safe operating system [8]. New supporting technology, such as programming logics for system source code, have been developed, e.g. by Shao's group at the

University of Yale [9]. In all of the above lines of work, verification assumes availability of the source code.

The Research Institute in Automated Program Analysis and Verification [10] in the UK has two related research projects. The first is "Compositional Security Analysis for Binaries" that is translating binaries to C code in order to apply software analysis tools like CMBC [11]. However this approach is focussed on low-level properties, like zeroing of released memory, rather than system-level properties. The second project is "Program Verification Techniques for Understanding Security Properties", focussed on source code rather than binaries, and is therefore not applicable to software that has already been developed.

The VATES project [12] aims at verifying behaviour of embedded software at the level of an LLVM intermediate representation. It uses a top-down approach by inventing an abstract CSP-based specification and then formally relating a model of a concrete system implementation, given in the form of the LLVM intermediate representation. This demonstrated that well-established formal verification techniques for declarative process-algebraic specifications are applicable to low-level software programs. The VATES project developed a formal operational semantics of LLVM and established a bisimulation relation between LLVM and CSP models in order to prove that a given LLVM program is a correct implementation of a given CSP model. Again a top-down development was assumed and the translation from the intermediate representation down to assembly is just assumed to be correct.

The most relevant work has been that of Magnus Myreen who developed an approach to formal verification of machine-code [15] that has been exploited to verify binary code [16]. It has been subsequently extended by Fox to deal with more ISAs [17]. Myreen and Fox's work has so far targeted ARM, Intel x86 and IBM PowerPC processors with a proof-producing de-compiler, which given some machine code for any of the processors mentioned above will automatically, via proof, derive a functional program from the machine code. This functional program is a record of the state change the machine code can perform. Although the approach only provides a low level description in Higher Order Logic it could form the basis for connecting binary executables to system architectures.

6 Ongoing and Future Work

Within the HACMS project, the lead organisation for the Red Team is the Charles Stark Draper Laboratories, or Draper Labs. Draper Labs are working on a de-compiler tool under HACMS. The HACMS de-compiler, called the Fracture tool, converts Executable Object Code for an Instruction Set Architecture (ISA) into an intermediate representation called LLVM [13].

Current work on the HAMCS de-compiler is attempting to emit a representation, in the C compliance notation, from LLVM that is abstracted into a predicate by FSG and then translated into CSPm for the refinement model checker, FDR3 [14], to verify against system security properties. Work is also ongoing on the formal specification of a translator from the specifications, generated by the FSG tool, to CSPm.

Myreen's de-compilation technology provides advantages over the use of Fracture and FSG. For example the Fracture tool does not provide any assurance of correctness against the semantics of the ISA used. The manipulation of LLVM and the translation into the C compliance notation is similarly unassured. Similarly there is no planned verification of the translation from the output of FSG to CSPm. As a consequence any reported violations of system properties may be false positives and satisfaction of the properties cannot be relied upon. Such limitations make its use questionable when safety is required, although its role within the Red Team's assessment of systems is still useful.

If Myreen's de-compilation technology could be allied to a system representation, as illustrated by this paper, then system safety and security requirements could be established with high assurance and automatically. The gap between functional representations of binary executable in Higher Order Logic and CSPm can be addressed through the Unified Theories of Programming (UTP) [18]. An assured translator could be built on top of a current tool [19] that supports UTP and is built on top of Isabelle.

To employ the FDR3 model checker for properties that were sensitive to data values would require a two pronged approach. One is to build data abstraction into the translation from the representation in Higher Order Logic to CSPm. The other is the development of a symbolic model checker for CSPm to assure such data abstractions. The capability of FDR3 to harness cloud resources to check over large state spaces means that it is plausible that embedded systems could be within the reach of automated verification against system properties. For example FDR3 checked a trillion states using Amazon's EC2 web service in a few hours at a cost of approximately $70 [15].

The only part that could not be automated is the human understanding of architectural patterns used for Cyber-Physical Systems. Work on identifying such patterns is ongoing by Draper Labs. The specification of the compositional framework would have to be done by a human, but it is conjectured that there are a relatively small number of such patterns. It is envisioned that a library of re-usable compositions, like the one described in this paper, could be created by human effort and then automatically incorporated into a system representation for verification.

7 Acknowledgements

I am hugely indebted to Neil Brock from Draper Labs for discussions about this problem area and for providing me with the C code and architectural composition used in this paper. This work was sponsored by the Charles Stark Draper Laboratories under the DARPA project on High Assurance Cyber Military Systems (HACMS).

8 References

1. http://smaccmpilot.org/
2. http://www.cyber.umd.edu/sites/default/files/documents/symposium/fisher-HACMS-MD.pdf
3. https://github.com/mavlink

4. A.W. Roscoe. “The Theory and Practice of Concurrency”, Prentice-Hall (1997).
5. http://www.lemma-one.com/ProofPower/index/
6. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish,T. Sewell, H. Tuch and S. Winwood. seL4: Formal verification of an OS kernel, in ACM Symposium on Operating Systems Principles, pp. 207–220, Big Sky, MT, USA, October ‘09.
7. Eyad Alkassar and Mark A. Hillebrand and Wolfgang J. Paul and Elena Petrova. Automated Verification of a Small Hypervisor. In Gary T. Leavens, Peter W. O'Hearn and Sriram K. Rajamani, editors, Verified Software: Theories, Tools, Experiments (VSTTE), Springer, 2010.
8. Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Benjamin G. Zorn and Alexander Aiken, editors, Programming Language Design and Implementation (PLDI), ACM, 2010.
9. Xinyu Feng, Zhong Shao, Yuan Dong and Yu Guo. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. In Rajiv Gupta and Saman P. Amarasinghe, editors, Programming Language Design and Implementation (PLDI), ACM, 2008.
10. https://verificationinstitute.org/
11. http://www.cprover.org/cbmc/
12. B. Bartels, S. Glesner, T. Göthel, Model Transformations to Mitigate the Semantic Gap in Embedded Systems Verification International Colloquium on Graph and Model Transformation. http://journal.ub.tu-berlin.de/eceasst/article/view/418
13. https://github.com/draperlaboratory/fracture
14. T. Gibson-Robinson and A. W. Roscoe, FDR into The Cloud. http://www.cs.ox.ac.uk/files/6642/ClusterFDR.pdf
15. Magnus O. Myreen. Formal verification of machine-code programs. PhD dissertation, University of Cambridge, 2008.
16. 12. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish,T. Sewell, H. Tuch and S. Winwood. seL4: Formal verification of an OS kernel, in ACM Symposium on Operating Systems Principles, pp. 207–220, Big Sky, MT, USA, October ‘09.
17. Anthony Fox. Directions in ISA specification. In Lennart Beringer, Amy P. Felty, editors, Interactive Theorem Proving (ITP), Springer, 2012.
18. C. A. R. Hoare and He Jifeng, “Unifying Theories of Programming”. Prentice Hall College Division. p. 320. ISBN 978-0-13-458761-5.
19. http://www-users.cs.york.ac.uk/~simonf/utp-isabelle/