HACMS Program Overview

Introduction to HACMS

- **Goal**: Transform critical systems' security for both military and civilian applications.
- **Objective**: Develop secure, high-assurance software resistant to cyber-attacks using **formal methods**.
- **Focus Areas**:
  - Drones, helicopters, and automobiles.
  - Eliminating exploitable bugs through **mathematical software verification**.

Key Concepts in HACMS

- **Formal Methods**:
  - Mathematical proofs to verify software correctness.
  - Address vulnerabilities such as memory safety errors and unauthorized access.
- **Secure-by-Design**:
  - Incremental development with built-in security.
  - Emphasis on designing robust systems from the ground up.

HACMS Development Phases

Phase 1:

- Refactored quadcopter software using verified languages and systems.
- **Outcome**: Independent Red Team testing revealed no vulnerabilities.

Phase 2:

- Introduced multi-processor architecture with secure partitions using **seL4 microkernel**.
- **Outcome**: Maintained integrity despite granting root access to less secure partitions.

Phase 3:

- Enhanced features like geofencing and applied methods to military ground robots and networked weapon systems.
- **Outcome**: Demonstrated broad applicability across critical domains.

Validation and Security Testing

- Conducted by independent **Red Teams**.
- Systems consistently resisted extreme attack scenarios.
- Highlighted effectiveness of formal methods in:
  - Eliminating vulnerabilities.
  - Ensuring system integrity under duress.

Impact and Applications

Military Benefits:

- High-assurance systems for drones, helicopters, and ground robots.
- Enhanced security for networked weapon systems.

Civilian Applications:

- **Medical Devices**: Secure pacemakers and insulin pumps.
- **Automobiles**: Protect against remote hacking of critical systems.
- **Industrial Systems**: Safer SCADA and public infrastructure.

Background and Motivation

- **Rising Cybersecurity Threats**:
  - Hacking of insulin pumps, vehicles, and SCADA systems.
  - Modern vehicles: “Computers on wheels” with vulnerabilities in ECUs, ODB-II ports, and CAN bus networks.
- **Real-World Example**:
  - 2015 Jeep Cherokee hack led to recall of 1.4 million vehicles.

Cybersecurity Challenges

- Complexity and connectivity of modern systems.
- Common risks:
  - **Implementation Errors**: Heartbleed, buffer overflows.
  - **Exploit Kits**: Enable attackers to exploit known vulnerabilities.
- **Solution**: Formal methods to proactively eliminate flaws.

Implementation and Results of HACMS

Experimental Platforms

- **SMACCMCopter**: High-assurance quadcopter with memory-safe programming and verified operating systems.
- **Boeing's Unmanned Little Bird (ULB)**: Helicopter used to test formal methods in real-world scenarios.
- Testbeds demonstrated practical applications of secure software design.

Key Techniques in Implementation

- **High-Assurance Code**:
  - Replaced legacy software with memory-safe programming languages.
  - Used verified real-time operating systems.
- **seL4 Microkernel**:
  - Ensured isolation of critical components.
  - Protected systems even with unverified components present.
- **Mathematical Verification**:
  - Designed software to be secure and functional through rigorous proofs.

Phase 1 Results: SMACCMCopter

- **Red Team Testing**:
  - Independent testers given unrestricted access to documentation and source code.
  - **Outcome**: No vulnerabilities exploited in 6 weeks of penetration testing.
- **Success Factors**:
  - Formal methods eliminated common flaws.
  - Memory safety ensured robust functionality.

Phase 2 Results: Multi-Processor Architecture

- **New Architecture**:
  - Verified and unverified partitions managed by **seL4 microkernel**.
  - Red Team granted root access to unverified partitions.
- **Outcome**:
  - Critical functionality remained uncompromised.
  - System integrity preserved despite advanced attack scenarios.
- **Key Takeaway**:
  - seL4 effectively isolated threats, demonstrating the power of formal methods.

Phase 3 Advancements

- **Geofencing**:
  - Added to SMACCMCopter to constrain system behavior.
- **Broader Applications**:
  - Adapted HACMS methods for:
    - Military ground robots.
    - Autonomous vehicles.
    - Networked weapon systems.
- **Impact**:
  - Extended cybersecurity solutions to both military and civilian technologies.

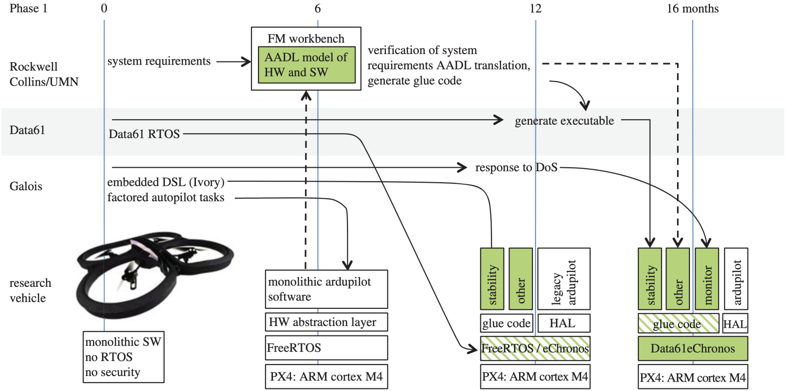

Figure 1: Architecture of the SMACCMCopter. Green boxes is high-assurance components.

Impact and Blueprint for the Future

- **Broad Applicability**:
  - Demonstrated rigorous methods can secure critical systems in diverse domains.
- **High-Assurance Software**:
  - Provided a proven model for addressing modern cybersecurity challenges.
- **Key Achievements**:
  - Developed robust, mathematically validated solutions.
  - Established foundational techniques for secure-by-design software.

Conclusion

- **HACMS Success**:
  - Showcased formal methods' effectiveness in creating resilient software systems.
  - Provided a framework for future development of secure technologies.
- **Vision**:
  - Transform cybersecurity for critical applications in both military and civilian contexts.
- **Legacy**:
  - Pathway for retrofitting legacy systems with advanced security.
  - Transformational impact on both military and civilian cybersecurity.

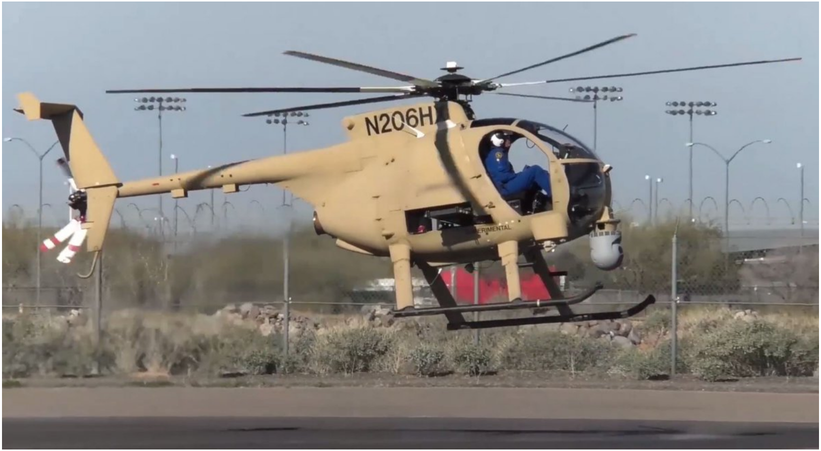

Figure 2: Demonstrated on Boeing's Unmanned Little Bird

Lessons Learned

Avoid Verifying Existing Code

- Verifying existing systems is more challenging than co-developing a system alongside its correctness proof.
- Verification experts must decipher required properties from complex codebases and documentation.
- Systems co-developed with proofs allow developers to make choices that facilitate easier verification.

Focus on Critical Code

- Only verify the parts of the system whose correctness is essential for security.
- Partitioning systems into critical and non-critical code can simplify verification efforts.
- Example: Leveraging the **seL4 microkernel** enabled the HACMS Air Team to sandbox unverified code securely.

Eliminate Obvious Bugs First

- Use low-cost testing tools to remove straightforward bugs before formal verification.
- This reduces the verification effort by focusing on unusual corner cases.
- Verification tools are more efficient when systems are pre-screened for common issues.

Leverage Automation

- Use decision procedures such as **SAT** and **SMT solvers** or tactic libraries to automate portions of the verification process.
- Automation facilitates updating proofs when system changes occur.
- Expanding automation capabilities and improving tactic libraries are ongoing research goals.

Adopt Domain-Specific Languages (DSLs)

- Write code in DSLs designed to support verification, which simplifies proof generation.
- DSLs can simultaneously produce executable code and associated proof scripts.
- These languages reduce complexity and improve the integration of proofs into system development.

Domain-Specific Languate Example

ReWire and FPGAs

ReWire is a purely functional language for creating secure hardware designs that can be implemented on FPGAs. The ReWire language is a subset of Haskell that places some restrictions on the use of recursion. The ReWire compiler `rwc` translates ReWire programs into synthesizable VHDL.

VHDL (VHSIC Hardware Description Language) is a hardware description language that can model the behavior and structure of digital systems at multiple levels of abstraction, ranging from the system level down to that of logic gates, for design entry, documentation, and verification purposes.

ReWire

```
data Bit        = Zero | One
data W8         = W8 Bit Bit Bit Bit Bit Bit Bit Bit

plusW8 :: W8 -> W8 -> W8
{-# INLINE plusW8 #-}
plusW8 = nativeVhdl "plusW8" plusW8

zeroW8 :: W8
zeroW8 = W8 Zero Zero Zero Zero Zero Zero Zero Zero

oneW8 :: W8
oneW8 = W8 Zero Zero Zero Zero Zero Zero Zero One

start :: ReT Bit W8 I ()
start = begin

begin :: ReT Bit W8 I ()
begin = loop zeroW8 oneW8

loop :: W8 -> W8 -> ReT Bit W8 I ()
loop n m = do b <- signal n
              case b of
                  One  -> loop n m
                  Zero -> loop m (plusW8 n m)
```

Formal Methods

Introduction to Formal Methods

- **Definition**: Rigorous mathematical techniques for specifying, designing, and verifying software and systems.
- **Key Features**:
  - Produce **machine-checkable proofs**.
  - Ensure **high confidence** in system correctness.
  - Apply to software, hardware, or higher-level models.

Advancements Enabling Formal Methods

Technological Drivers:

1. **Computing Power**:
   - Growth via Moore's Law: Faster processors, larger memories.
   - Supports intensive computations for verification.
2. **Automation**:
   - SAT solvers improved by two orders of magnitude over a decade.
   - Tools like SMT solvers and tactic libraries enhance automation in theorem proving.

Infrastructure Improvements:

- **Publicly Available Tools**:
  - Examples: Coq, ACL2, Z3, TLA+.
  - Robust and well-documented, enabling broader adoption.
- **Impact**:
  - Reduces reliance on custom-built tools.
  - Expands usability beyond academia.

Importance of Formal Methods

- **Complexity of Critical Systems**:
  - Increased interconnectivity heightens risks.
  - Example: Amazon Web Services uses model-checking to uncover rare design errors.
- **Strengths**:
  - Finds subtle issues overlooked by intuition or conventional testing.
  - Addresses corner cases in large-scale systems.

Spectrum of Formal Methods

Tool Categories:

1. **Type Systems**:
   - Examples:
     - Statically typed: Haskell, C.
     - Dynamically typed: Python, Matlab
   - **Scalable**, offer basic guarantees like type safety.
2. **Model Checkers & Sound Analyzers**:
   - Automate detection of design errors.
3. **Interactive Proof Assistants**:
   - Examples: Coq, Isabelle.
   - Provide **strong guarantees**, like full functional correctness.
   - Require significant expertise and effort.

Examples of Static Type Systems in C

```c
#include <stdio.h>

int main() {
    // Declaring an integer variable
    int number = 10;

    // Declaring a string (character array) variable
    char text[] = "Hello";

    // Attempting to assign a string to an integer variable
    number = "World"; // Error: incompatible types

    printf("Number: %d\n", number);
    printf("Text: %s\n", text);

    return 0;
}
```

Example of Dynamic Type System in Python

```
number = 10          # Variable can hold any type
number = "World"     # No error; type can change at runtime
```

A Proof in F* Proof Assistant

```
let rec fibonacci (n:nat)
  : nat
  = if n <= 1
    then 1
    else fibonacci (n - 1) + fibonacci (n - 2)

val fib_greater_than_arg (n:nat {n >= 2})
  : Lemma (fibonacci n >= n)

let rec fib_greater_than_arg (n:nat {n >= 2})
    : Lemma (fibonacci n >= n)
    = if n = 2 then assert(fibonacci 2 >= 2)
        else fib_greater_than_arg (n-1)
```

output

```
Verified module: Welcome
All verification conditions discharged successfully
```

Application Areas

Critical Software Components:

- **Microkernels**: Ensure reliability and security.
- Example: seL4, verified with Isabelle/HOL.
- **Compilers**: Guarantee correctness in code compilation.
- Example: CompCert, verified with Coq.

seL4 Microkernel:

- Verified for:
  - **Functional Correctness**.
  - **Security Properties**: Authority confinement, non-interference.
- Application: Trusted systems in aviation and military.

CompCert Compiler:

- Verifies C code compilation.
- Combines strong correctness guarantees with competitive performance.

Challenges and Considerations

- **Accuracy of Models**:
  - Incorrect assumptions limit guarantees.
- **Scope and Effort**:
  - Trade-off between scalability and assurance level.
- **Expertise**:
  - Tools like Coq and Isabelle require specialized knowledge.

Conclusion

- **Feasibility and Impact**:
  - Formal methods are increasingly practical and impactful.
  - Examples like seL4 and CompCert demonstrate their success in critical systems.
- **Future Outlook**:
  - Broader adoption as tools and automation continue to evolve.
  - Transformative potential for ensuring reliability in complex, interconnected systems.