CHARIOT – 3rd Workshop

Wednesday 22 October 2020 (online)

IoT Data Security and Privacy Solutions – Challenges and Opportunities for Airports

Static and Binary Analysis for Software Development

Andrea Battaglia – Aspisec
Franck Vedrine – CEA

The importance of Firmware in IoT

- Smart devices belonging to the Internet of Things ecosystem are increasingly present. However, the technology companies producing these evolved systems tend to underestimate the potential risks to which they are exposed since the latter are connected continuously to the Internet.

- In general, smart devices run bespoke firmware and do not use conventional operating systems and despite their widespread use and importance they are often ‘out of control’.

- It occurs maintaining them updated and considering threats and exploitations that could tamper the integrity of it, compromising the functionality and the security of an entire system.

- However, updates can lead to vulnerable surface potentially exploitable leaving a strong attack vector.

Related Business Cases and Industrial Needs Overview

- Recently there have been a number of well-documented attacks on embedded devices.
- Manufacturer consider security as non payback cost.
- In spite embedded devices include password protected logins and encrypted protocols such as SSH or SSL, this is not enough to make them secure, because they still lack on the design.
- For years embedded devices firmware have promoted the "Whatever it works" philosophy, now we're facing a world where devices are built with software containing well-known issues.
- Need of new methods and tools for more secure and safer IoT software development with static source code analyses to help avoid safety and security defects and risks targeting the software security engineer before the firmware deployment.

Business level Outcomes and Value

- Protect the firmware code and the hardware configuration from tampering, using a self-protecting mechanism
- Defense solution, increased Awareness and Reaction
- Automated mechanism to minimize Cyber Attack surface
- Innovative solution that applies in different sectors (from Industry to Healthcare)
- Innovation to provide incremental guarantees about the absence of vulnerabilities
  - Start with basic verification and add more security checks with reachable & short time objectives
  - Try the open source version and support for industrial deployment

Requirements for formal methods and an extensive security verification framework

- The approach should work without any access to the source code
- The Security Engine has few minutes to accept the firmware
- The security rules to check depend on the industrial end-user (LL)
- The security rules should ensure a continuous improvement of the firmware
- The refusal verdict of the Security Engine should be clear
- The verdict of the Security Engine should be objective
- The Security Engine should provide technical information aimed to help manual investigation

Firmware Security Verification

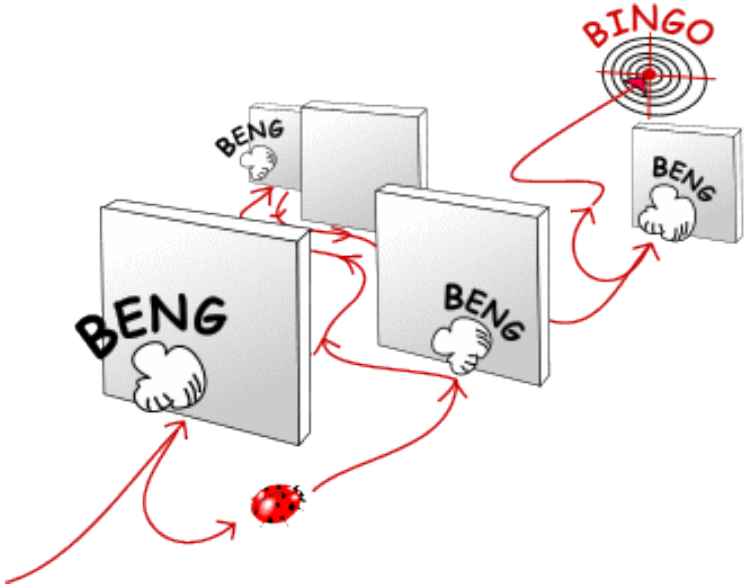
- Heuristic technique and mechanism for IoT Firmware Security verification

  The Firmware Security Verification is achieved by using an HEURISTIC APPROACH

- Innovation aspects

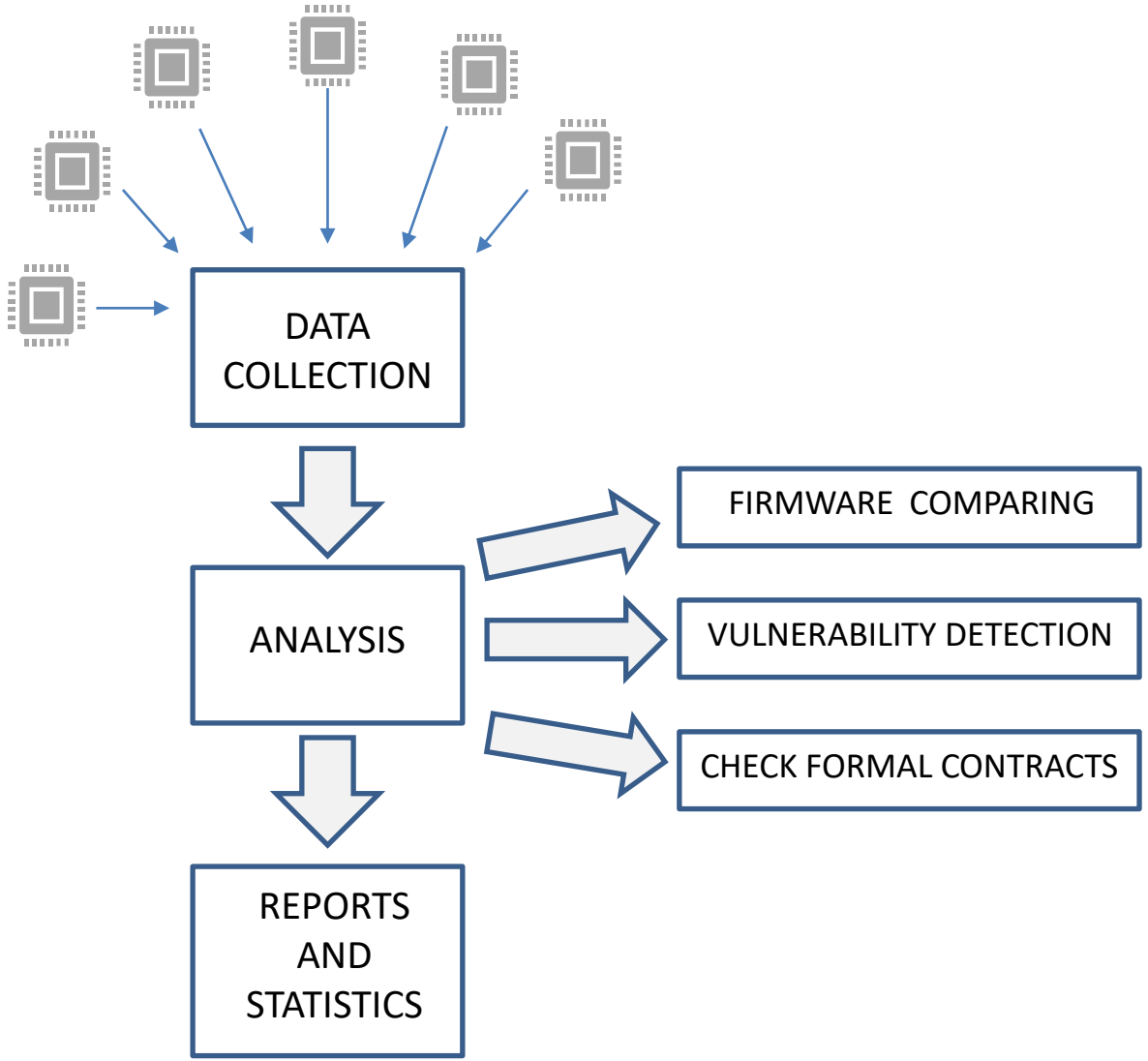
  The vulnerability check is able to ensure that no attack surface could be used in a remote way by a potential hacker.

The main goal of this solution is to strictly protect the firmware code and the hardware configuration from tampering, using a self-protecting mechanisms.



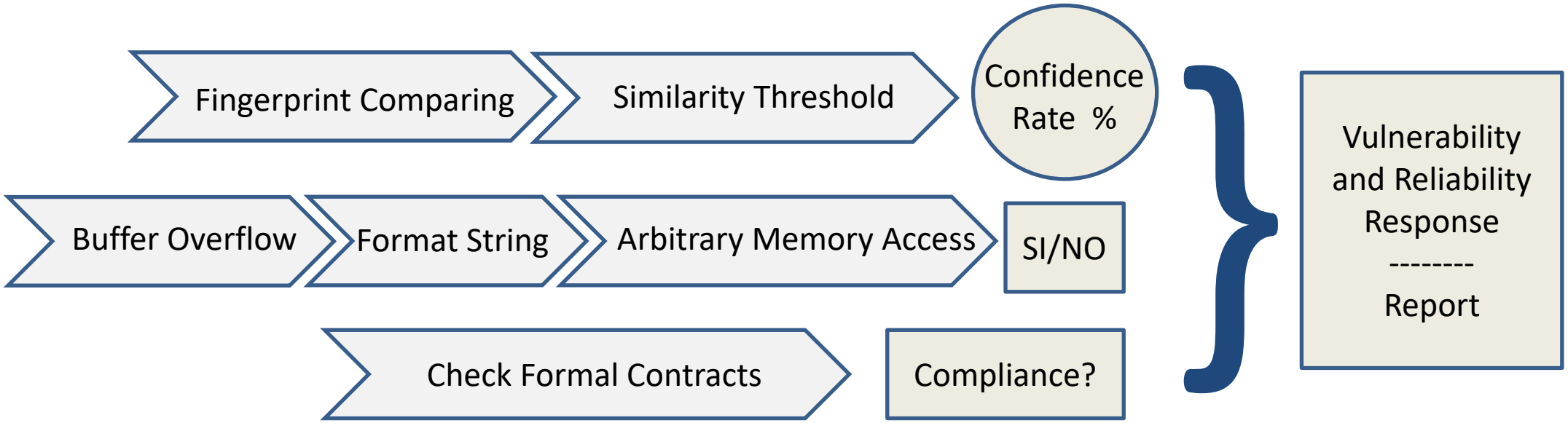
Firmware Security Verification

Firmware Security: Vulnerability checking with Static and Binary Analysis through specialized cross-compiler to help mitigate safety defects and cyber-attacks.



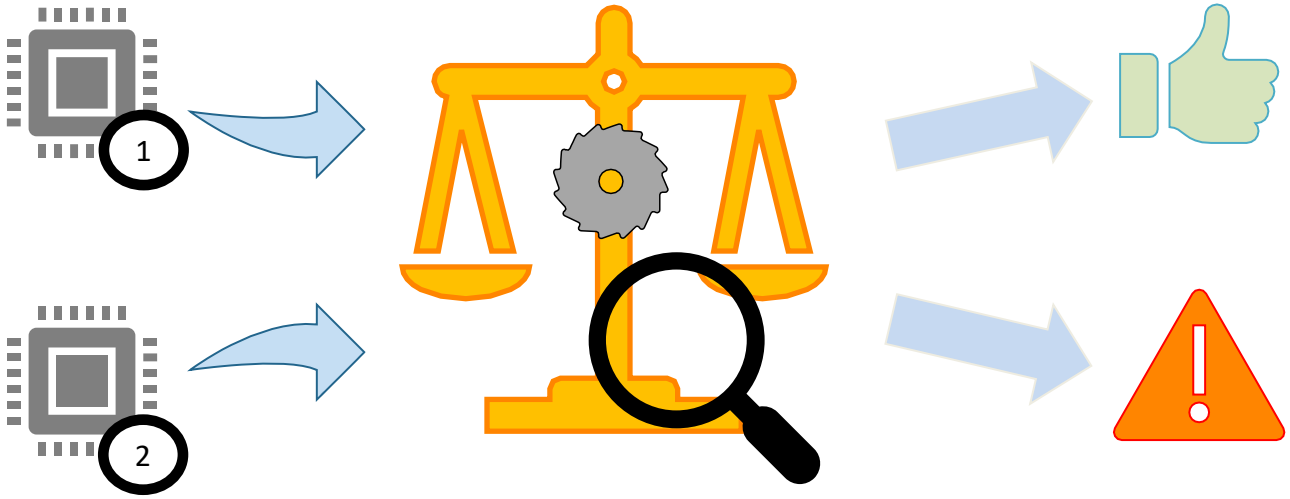
Firmware Security Verification

Heuristics based firmware security checking study has been approached by using similarity, fingerprint and ranking methods, vulnerability detection and check offline analysis contracts.



Firmware Comparing

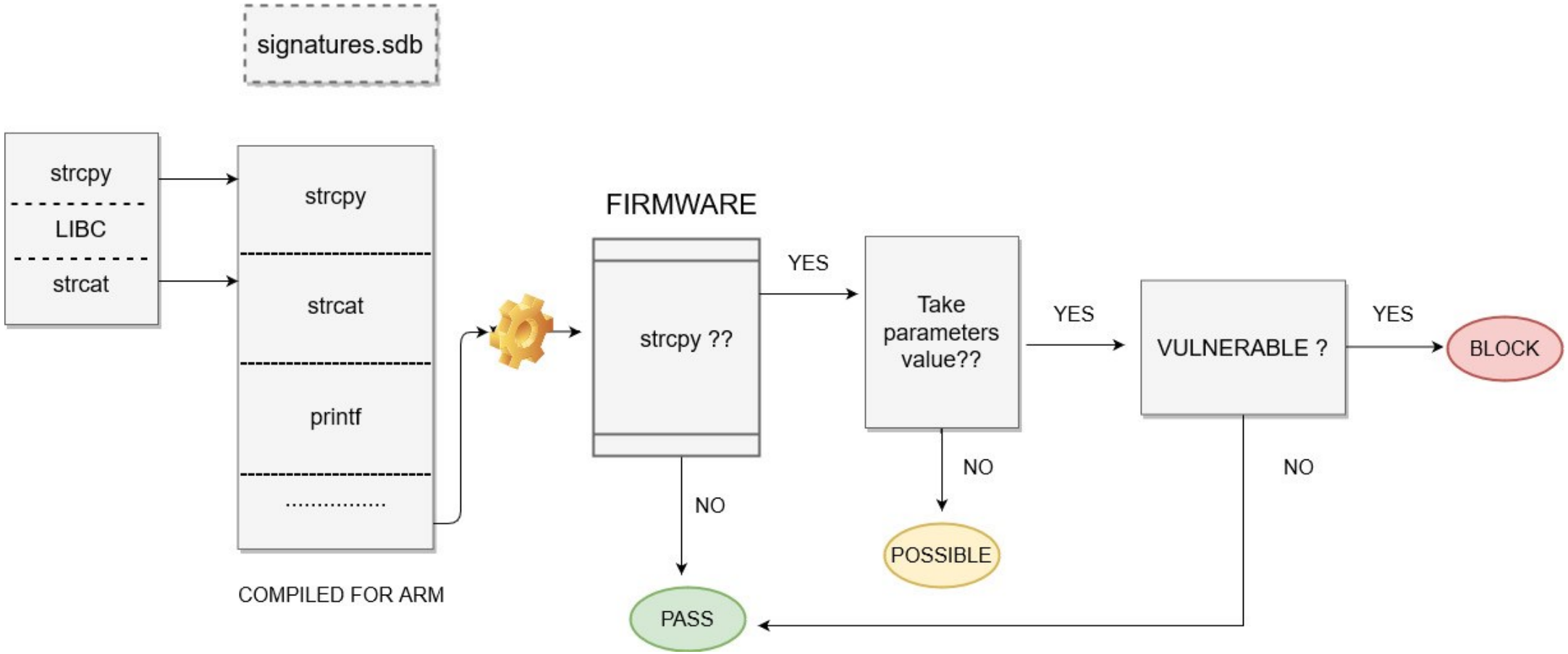
The result of the Analysis is a kind of COMPARISON between the parsed data of two different versions of the same firmware.



- In case of substantial differences in functions and behaviour inside the firmware, the engine indicates that a potential security breach has occurred and firmware is marked as tampered and not secure.
- Otherwise the firmware passes the security check and will be ready to update the devices under Blockchain signature.

Vulnerability Detection

Use a feature that provides compiled examples of signatures of the popular libc functions used during binary exploitation.

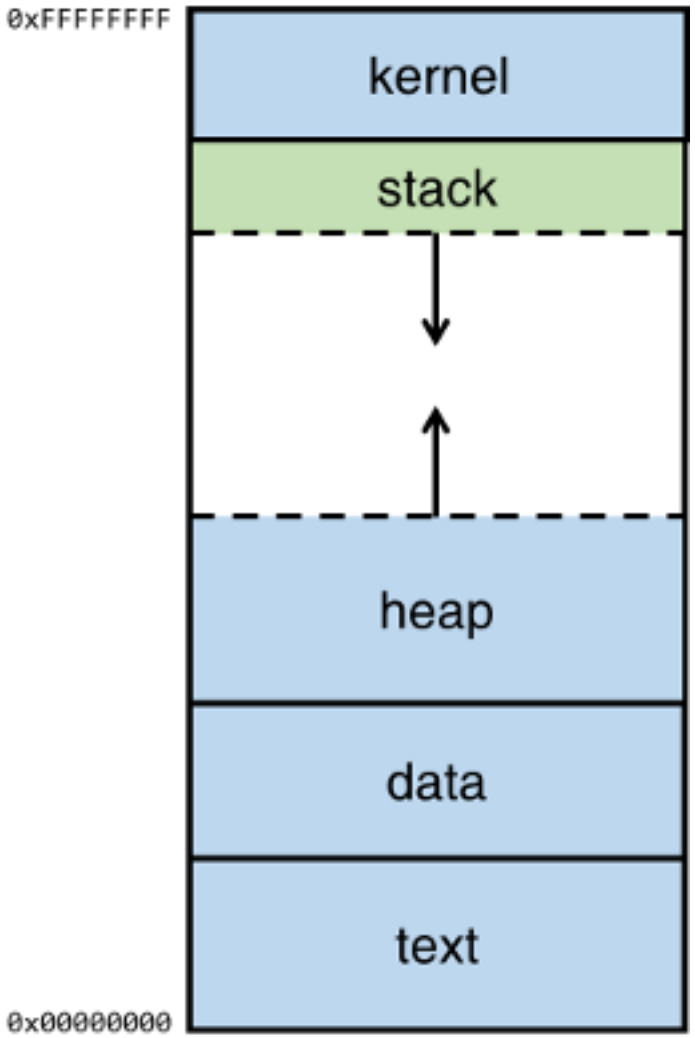


Firmware Analysis and Binary Processing

- class **BufferOverflow**:
  - First load the signature
  - Next apply the signatures and look for functions and look for the parameters with which they are called (r0 and r1) and check them
  - looks for functions that are prone to Buffer Overflow, such as:

    gets() **strcat()** **realpath()**
    strcpy() **getwd()** **(f)scanf()**

  - find the addresses where this functions are called and look for their arguments
  - analyze the actual values of arguments (where possible) and decide whether there is an explicit BOF vulnerability or not



Firmware Analysis and Binary Processing

Format string vulnerabilities are a class of bug that take advantage of an easily avoidable programmer error. If the programmer passes an attacker-controlled buffer as an argument to a printf (or any of the related functions, including sprintf, fprintf, etc), the attacker can perform writes and reads to arbitrary memory addresses.

- class **FormatString**
  - apply signatures that are looking for the printf function in the firmware
  - at the call addresses of this function, we analyze the instructions and look for the parameters with which it is called
  - we analyze the values found and decide whether there is a vulnerability

buffer=some_input

printf(buffer); ⇒ **pot. vuln**

buffer=%s

printf(buffer); ⇒ **read from the stack**

buffer=aa%n

printf(buffer); ⇒ **write on the stack**

printf(“%s”,buffer); ⇒ **not vuln**

Firmware Analysis and Binary Processing

- Class **ArbitraryMemoryAccess**:
  - look for memory access instructions in functions
  - at the addresses of these instructions, we analyze other instructions and look for the parameters by which they are initialized
  - we analyze the values found (if any) and decide whether there is a vulnerability

  - First, we look for memory access instructions for all functions and consider the address at which the memory is accessed.
  - In all types of vulnerabilities, we need to find the values by which different registers are initialized relative to some of the instructions. For this scope we build all possible variations of the function depending on conditional jumps. For this, a separate class was created that deals with this - **CFGBuilder**.
  - Next, based on the results obtained from the control flow builder, another class (**CFGWalker**) passes from the instruction of interest up the graph to the beginning of the function and analyzes the instructions and based on what type of instruction and with which registers and values it works,

Security Formal Verification

- Automated mechanism to minimize Cyber Attack surface.
  CHARIOT provide a way based on Proof Carrying Code to ensure that some Security Properties formally holds.

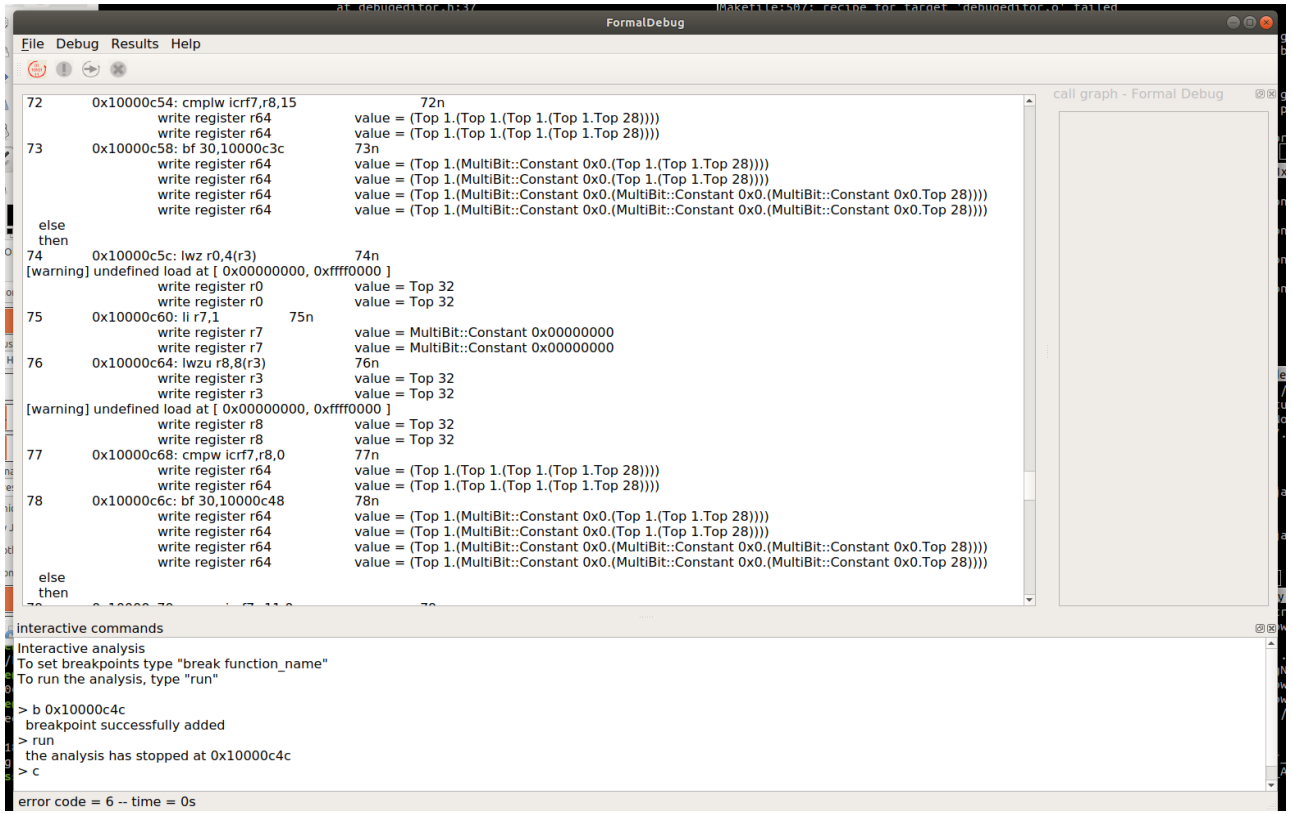
- BISMON-Security-Checker provides an interactive offline static analysis to add formal contracts in the binary firmware
- The Security Engine verifies the validity of the formal contracts and then uses them to check Security Rules.

- Simple rules can set:
  - No buffer overflow in the functions foo, bar
  - No invalid access in the functions foo, bar (Integrity of the dataflow)

- Complex rules may be very elaborate = programming language
  - They explain how to check the Security Property with high-level data-flow & control-flow concepts
  - They are translated into master verification procedures for BISMON-Security-Checker and into contract verification for the Security Engine

BISMON-Security-Checker – interactive offline static analysis

- The interface targets a Security Engineer familiar with tools like IDA-Pro, radare2, ghidra
- Support **armv7**, **mips** Instruction Set
- Decorate the memory with domains = constants, intervals, symbolic values, undefined values



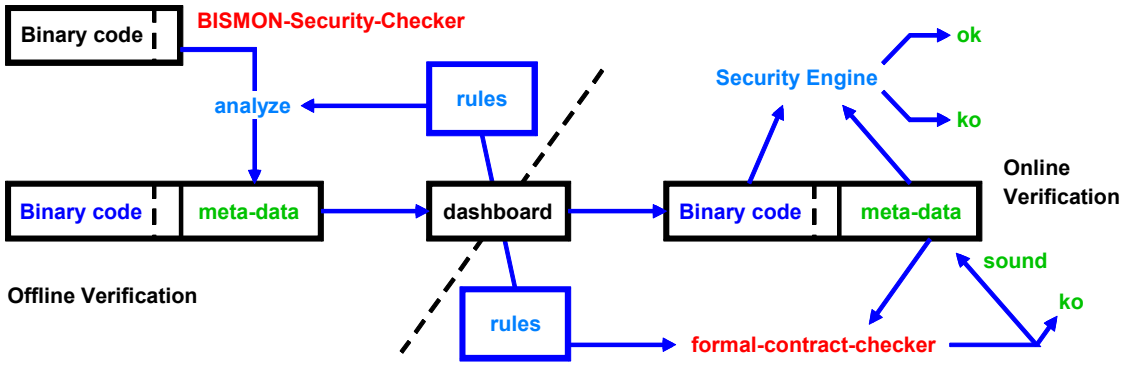
- Internal results: dataflow + control flow + call tree + domains at every address for every instruction + analysis coverage
- External Results
  = **formal contracts** stored before and after the linear blocks of instructions

- Static Analysis = no test
- The analysis can start everywhere
- No false negative:
  security flaws are all detected
  price = false positive
- Once loaded, the Security Rule drives the analysis

Security Engine: quick online formal verification

- The online analysis
  - objectively checks the soundness and the coverage of the contracts – no way to accept unsound contracts
  - formally checks the Security Rules

- The architecture has been designed to scale
  - The **offline** analysis **prepares all intermediate reasoning**
  - The **online** analysis is **quick** (every binary instruction is scanned only once) and it brings **formal guarantees**



- Experimental prototype that will be improved until it answers the research/industrial questions
  - OK: Implementation of a Proof Carrying Code approach with different instruction sets (2-6 p.m to implement an ISA)
  - Open: Does a proof carrying code approach really scale for firmware analysis?
    - inheritance to limit the size of contracts
    - deep learning to mix abstraction & symbolic inference that are two concepts in competition
  - Open: Can the proof carrying code approach benefit from the update process & compare with the annotations of the previous firmware?

Contact Details

CEA

Franck Vedrine

Franck.VEDRINE@cea.fr

Aspisec

Andrea Battaglia

a.battaglia@aspisec.com

The project CHARIOT has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780075