PROLEPSIS

Binary Analysis and Instrumentation of IoT Software for Control-Flow Integrity

IWES 2021 – 10/12/2021 – Roma

Gianluca ROASCIO
gianluca.roascio@polito.it

Nicolò MAUNERO
nicolo.maunero@polito.it

CYBERSECURITY
NATIONAL
LABORATORY

SPARTA

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Acknowledgments

The work has been supported by:

CYBERSECURITY NATIONAL LABORATORY

CINI Cybersecurity National Lab

https://cybersecnatlab.it/

Acknowledgments

The work has been supported by:

European Union’s Horizon 2020 research and innovation programme under grant agreement No. 830892, project SPARTA.
https://www.sparta.eu

Blu5 Labs Ltd. (Malta)
https://www.blu5group.com

Outline

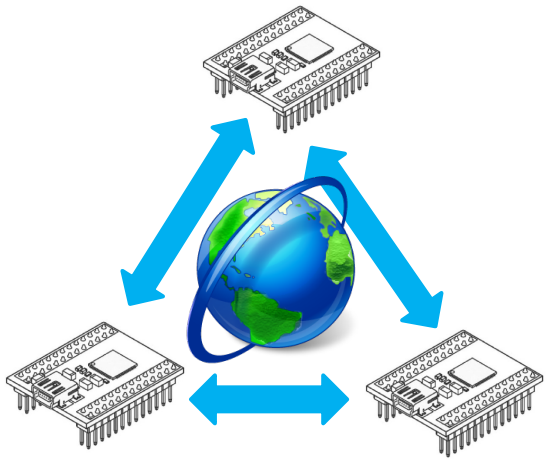
- Introduction
- Our Contribution
- Experimental Results
- Conclusions and Future Work

Outline

- Introduction
- Our Contribution
- Experimental Results
- Conclusions and Future Work

Introduction

- Internet of Things (IoT) is spreading fast
  - 8.74 billion devices in 2020 → 25.4 billions in 2030[1]
- For optimization reasons, C language is widely adopted
  - 2nd most used language in 2020[2]
- Use of C makes devices vulnerable to binary attacks
  - Buffer overflow, control-flow attacks



[1] A.Holst, "Number of Internet of Things (IoT) connected devices world-wide from 2019 to 2030." https://www.statista.com/statistics/1183457/ iot- connected- devices- worldwide/, 2021.
[2] "Interactive: The Top Programming Languages 2020 Spectrum." https://spectrum.ieee.org/static/interactive-the-top-programming- languages- 2020, 2020.

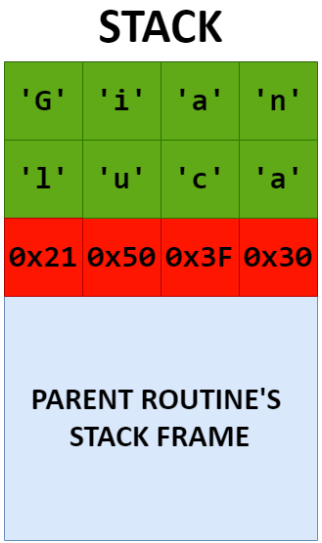
Buffer Overflow

- Boundaries of a data buffer can be overrun and adjacent memory locations can be overwritten

```
void insert_name()

{
char name[8];
...
gets(name);

«Gianluca!P?0»
```

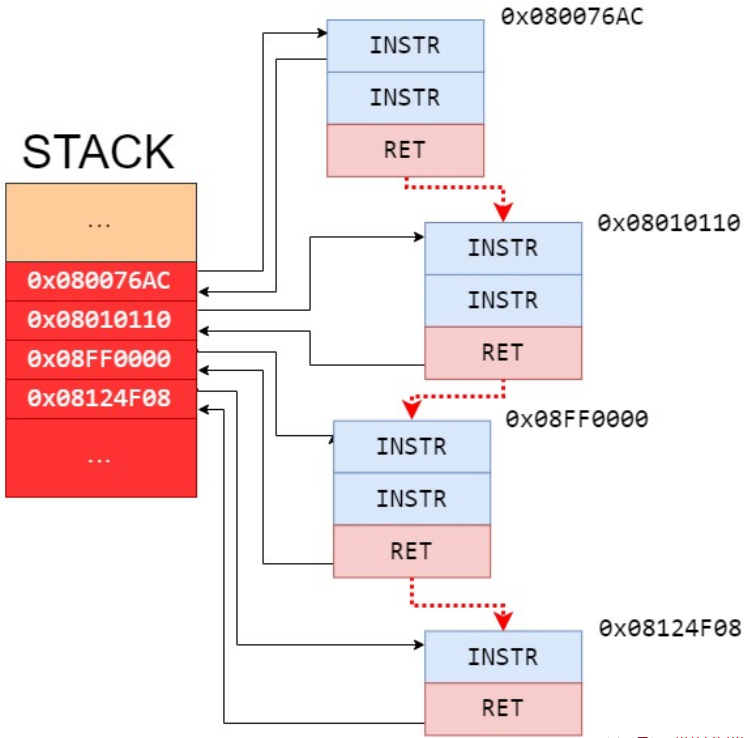


Return address:
0x21503F30

Execution flow is redirected!

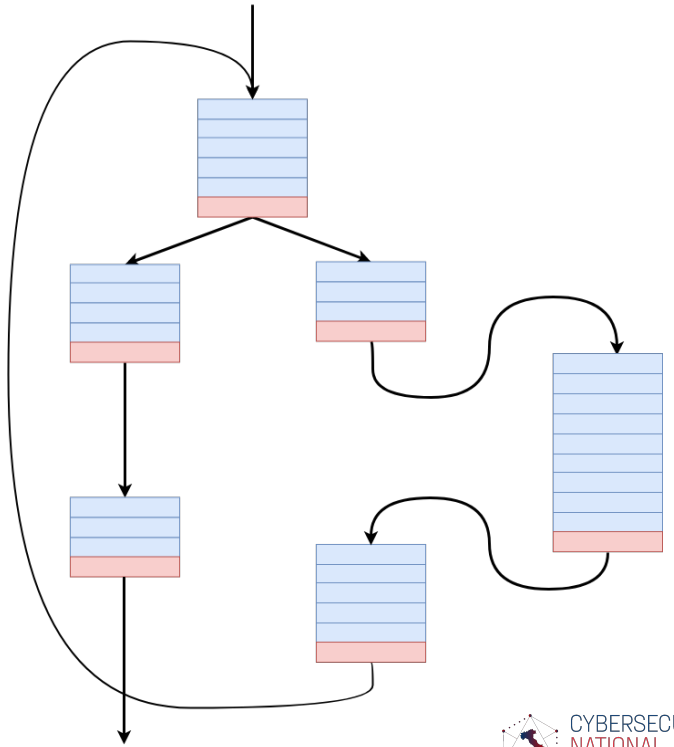
Example: Return-Oriented Programming (ROP)

- Based on gadgets ending with a routine return instruction (RET)
- RET pops the return location from the stack and jumps there
- If the stack data are overflowed, a series of ‘’fake’’ return addresses can be stacked
- Every time a RET is executed, control is passed to the next gadget



Solution: Control-Flow Integrity (CFI)

- The program is allowed to follow predefined paths only, as stated in its Control-Flow Graph (CFG)
  - Vertices: sets of non-jumping instruction (basic blocks)
  - Edges: control-flow transfers (jumps, calls, returns)



CFI: State of the Art

CFG EXTRACTION

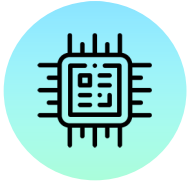
The application is forced to follow only predefined paths. The CFG must be extracted before runtime to correctly instruct the monitor

Software-based solutions

- Static Binary Instrumentation
- Dynamic Binary Instrumentation

ONLINE MONITOR

Piece of hardware or software that is able to ensure that flow transfers are consistent with the CFG



Hardware-based solutions

- Branch target encryption
- Shadow call stack
- Basic-block signature verification
- Instruction Set Architecture modification

Outline

- Introduction
- Our Contribution
- Experimental Results
- Conclusions and Future Work

Problem Statement

- IoT/embedded systems are too resource-constrained to support a complete CFG verification
- Protection must be limited to the points of the program where the risk of control-flow corruption is really concrete
  - i.e., only for transfers whose destination is computed with data that has passed through a data memory area at risk of corruption

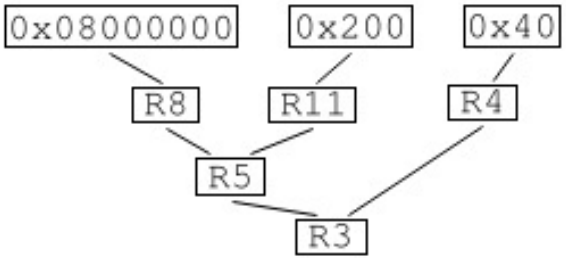
Origin Tree and Protection Rule

- The origin tree Γ_c of a control-flow instruction located at address c represents the computational history of its operand
- Such an instruction is said to be secure iff

$$\nexists x \in \Gamma_c \ : \ x \in X_{nk}$$

- where X_{nk} is the set of non-constant memory locations
- All other branches are considered insecure and need CFG check enforcement

```
MOV R8, #1
LSL R8, R8, #27
MOV R11, 0x200
MOV R4, 0x40
ADD R5, R8, R11
ADD R3, R4, R5
BX R3
```



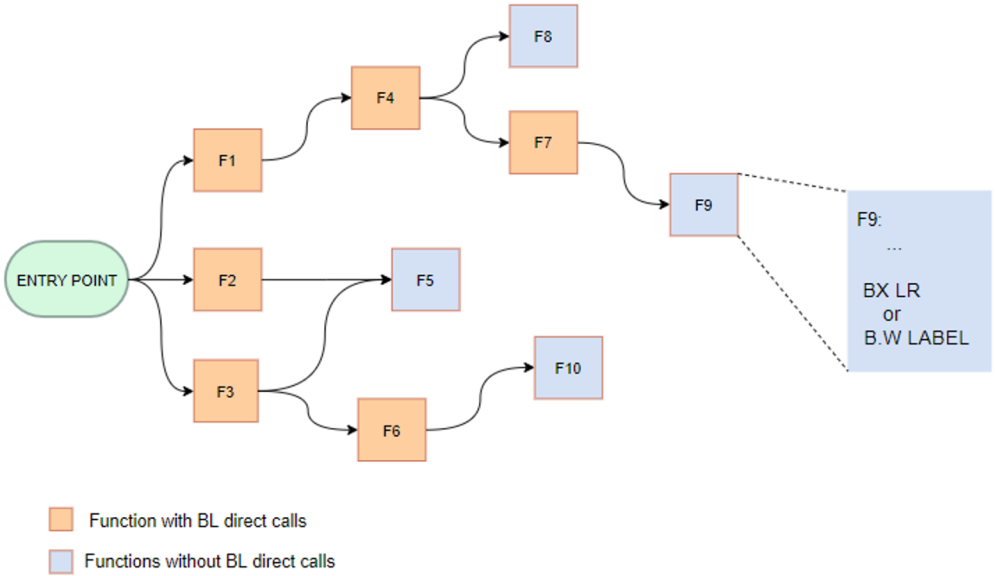
PROLEPSIS: Architecture and Features

- Automatic tool for finding insecure points of binaries executable on ARM architectures
- Prototyped in Python
- Supported by `Radare2` reverse engineering framework
- 5 execution stages:

1. **Parsing**: from binary to disassembly
2. **Extraction**: Call graph outlining
3. **Reconstruction**: backward traversing to outline origin tree of branch targets
4. **Recognition**: classification of branch types
5. **Instrumentation**: insertion of *custom* protecting/monitoring instructions

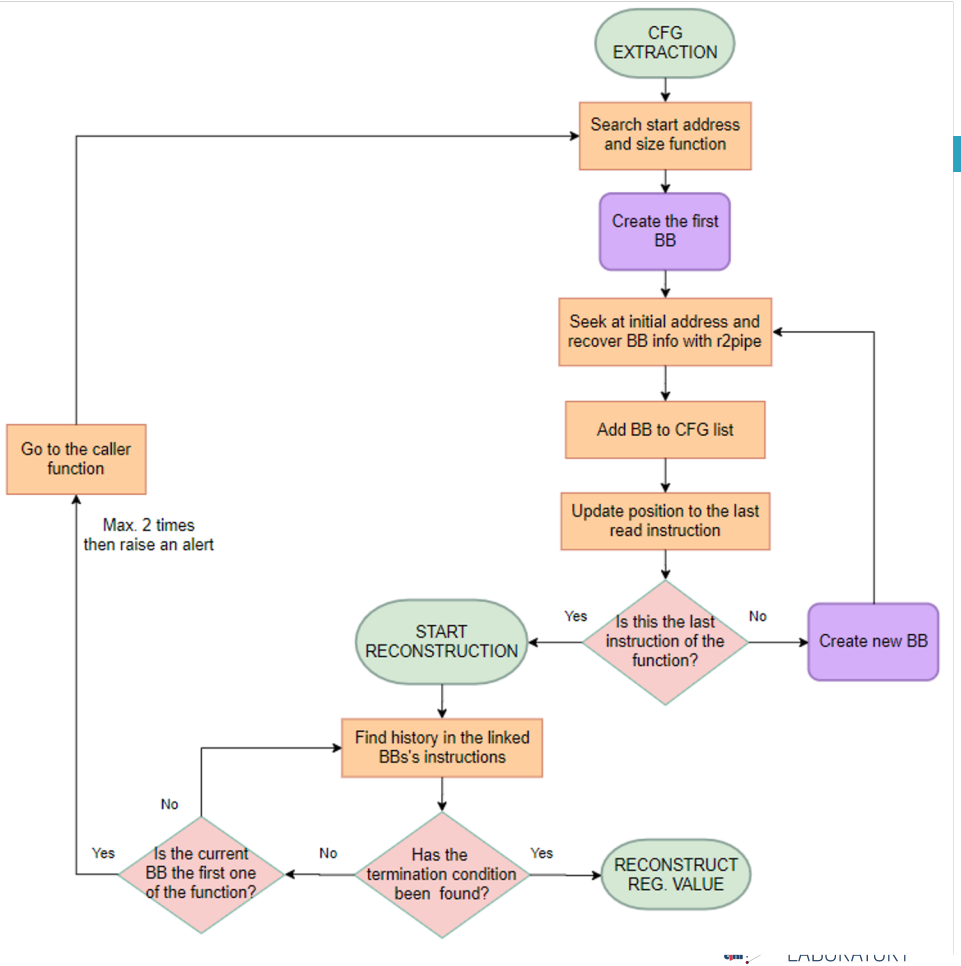
Extraction

- Track the observed software's general flow traversing the so-called Global Call Graph
- Find indirect jumps
- *On-demand* graph: it does not create it for all functions in the file, but only for those that must be processed for the origin tree's production
- It avoids instantiating Basic Block objects that will never be passed as not involved in the origin tree



Reconstruction

- Analyze the indirect jump and trace back its history with a recursive algorithm
- If the termination condition is not found in the BB of the function that contains the jump, it is necessary to generate the CFG of the caller function
- Every statement in the history list “emulates” the ARM statement, storing the result
- If the final address belongs to the data section, it is considered insecure
- The current step and the extraction one are strictly dependent



Experimental Results

TABLE I

PROLEPSIS ANALYSIS AND INSTRUMENTATION STATISTICS PER BENCHMARK.

Application	# instr. (no prot.)	Direct Calls	Insec. Edges	# instr. (prot.)	Instr. overhead
BITCOUNT	20554	177	11	21366	3.99%
DIJKSTRA	20529	186	11	21327	3.89%
SHA	13663	127	14	13959	2.17%
RIJNDAEL	25685	197	11	26494	3.15%
CRC	20320	178	10	21042	3.55%
STRING	12960	127	14	13217	1.98%

TABLE II

PROLEPSIS PERFORMANCES PER BENCHMARK OVER THE 5 ALGORITHM PHASES.

Application	Parsing (s)	Extraction (s)	Reconstruction (s)	Recognition (s)	Instrumentation (s)
BITCOUNT	9.003517	6.134531	2.041636	0.970078	0.117956
DIJKSTRA	8.311881	7.226851	1.928504	0.928324	0.098801
SHA	7.649001	4.681293	1.553550	0.243677	0.122483
RIJNDAEL	9.790732	13.93778	2.420454	1.014930	0.123994
CRC	8.346101	6.336453	1.856236	0.822695	0.096953
STRING	7.273603	4.464079	1.563572	0.143231	0.066789

- Prototype tested on 6 MiBench embedded benchmarks
- 100 run per benchmark
- As a reference protection, used a 6-instructions-per-branch instrumentation
- **Overhead never exceeds 4%**
  - Better than many reference studies[2]

[2]C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables", 2013.
J. Tan, H. J. Tay, U. Drolia, R. Gandhi, and P. Narasimhan, "Pcfire: Towards provable preventative control-flow integrity enforcement for realistic embedded software", 2016.
T. Nyman, J. Ekberg, L. Davi, and N. Asokan, "Cfi care: Hardware- supported call and return enforcement for commercial microcontrollers" 2017.
R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems" 2019.

Conclusions

- Pros:
  - Nominal overhead lower than most of the CFI techniques
  - Possibility to customize the protection technique (hardware or software monitor, custom additional instructions, …)
- Cons:
  - Not a complete defense tool itself (you need a CFI monitoring facility)
  - Only for ARM executable binaries
- Efforts needed to gather more accurate data to measure actual benefits in terms of overhead with respect to real-world applications

Thanks for your attention!

CYBERSECURITY
NATIONAL
LABORATORY

SPARTA