FirmCross: Detecting Taint-Style Vulnerabilities in Modern C-Lua Hybrid Web Services of Linux-based Firmware

Runhao Liu[1], Jiarun Dai[2], Haoyu Xiao[2], Yuan Zhang[2],

Yeqi Mou[1], Lukai Xu[1], Bo Yu[1], Baosheng Wang[1], Min Yang[2]

[1] National University of Defense Technology

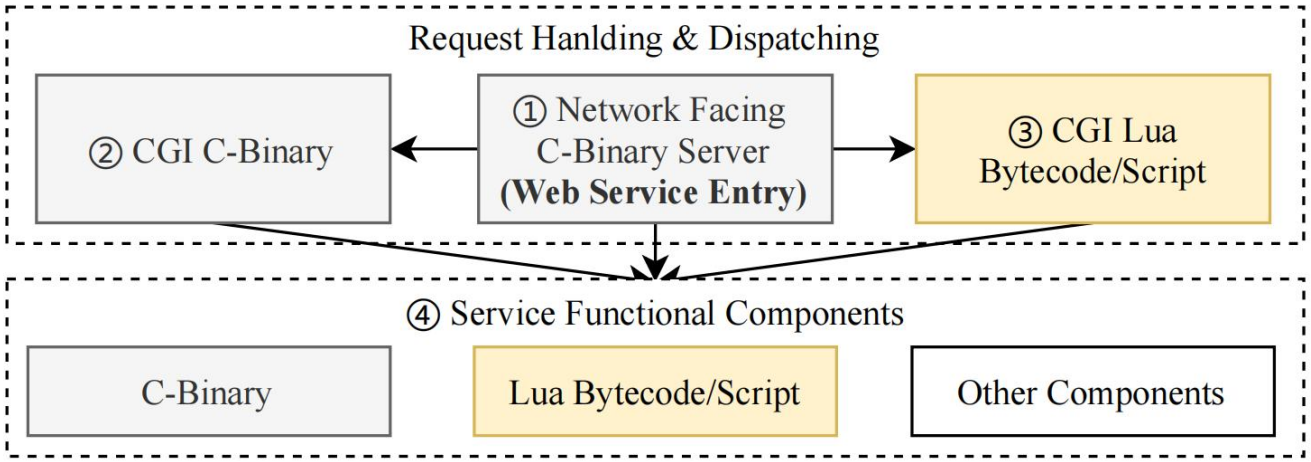
[2] Fudan University

Overview

- The number of IoT devices will reach 40 billion by 2030.
- IoT Vulnerabilities typically arise from web services of Linux-based firmware and can cause severe harm.
- Existing static taint analysis approaches over-simplify the composition of firmware web services (i.e., **long-neglected Lua-involved attack surfaces**)

Empirical Study

- Goal: Understand the security risks of C-Lua firmware web services
- Design: 2461 images across 6 vendors (scraped with firmadyne)
- Findings:
  - **Prevalence** of C-Lua Hybrid Services: 38%(937/2461)
  - Long-neglected **Lua-involved** Attack Surfaces(e.g., ①→③, ①→③→④)



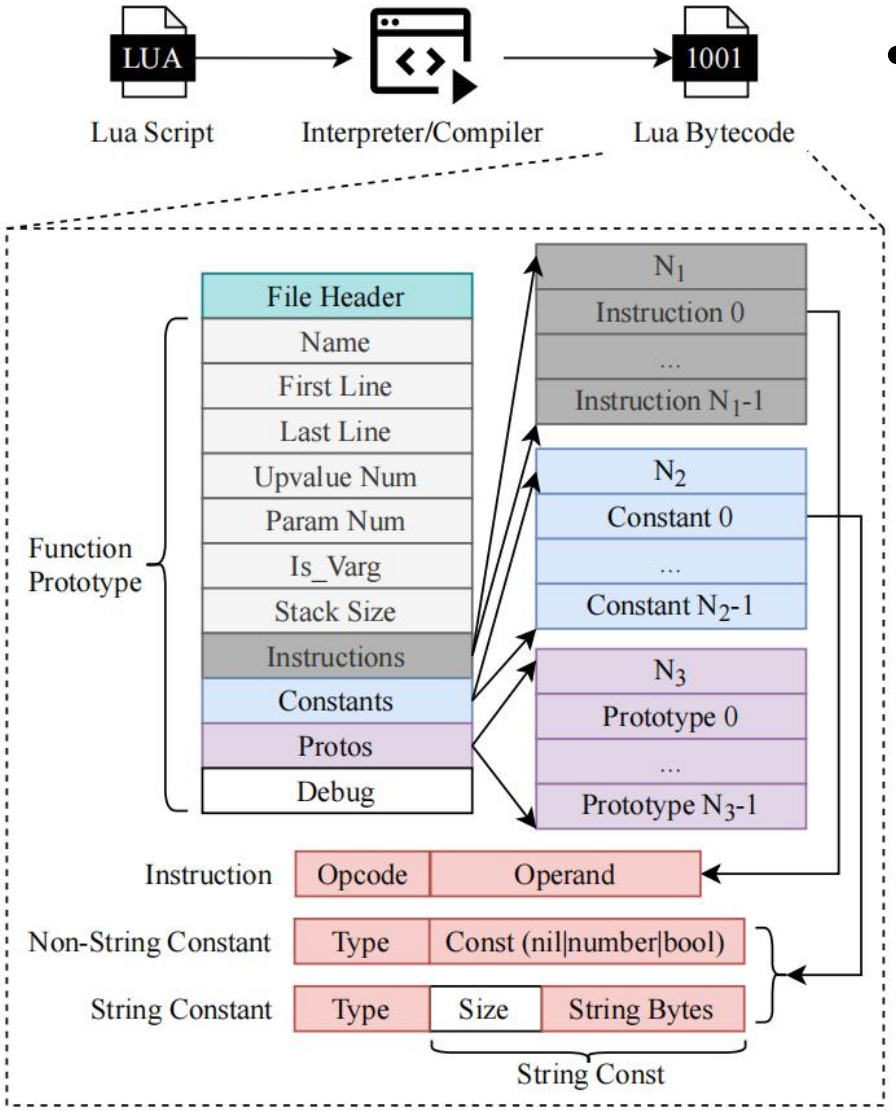
Empirical Study

- Goal: Understand the security risks of C-Lua firmware web services
- Design: 2461 images across 6 vendors (scraped through firmadyne)
- Findings:
  - **Prevalence** of C-Lua Hybrid Services: 38%(937/2461)
  - **Long-neglected** Lua-involved Attack Surfaces (e.g., ①→③, ①→③→④)
  - 34%(316/917) of Lua services are deployed in **obfuscated bytecode**

Limitations of Current Taint-based Methods

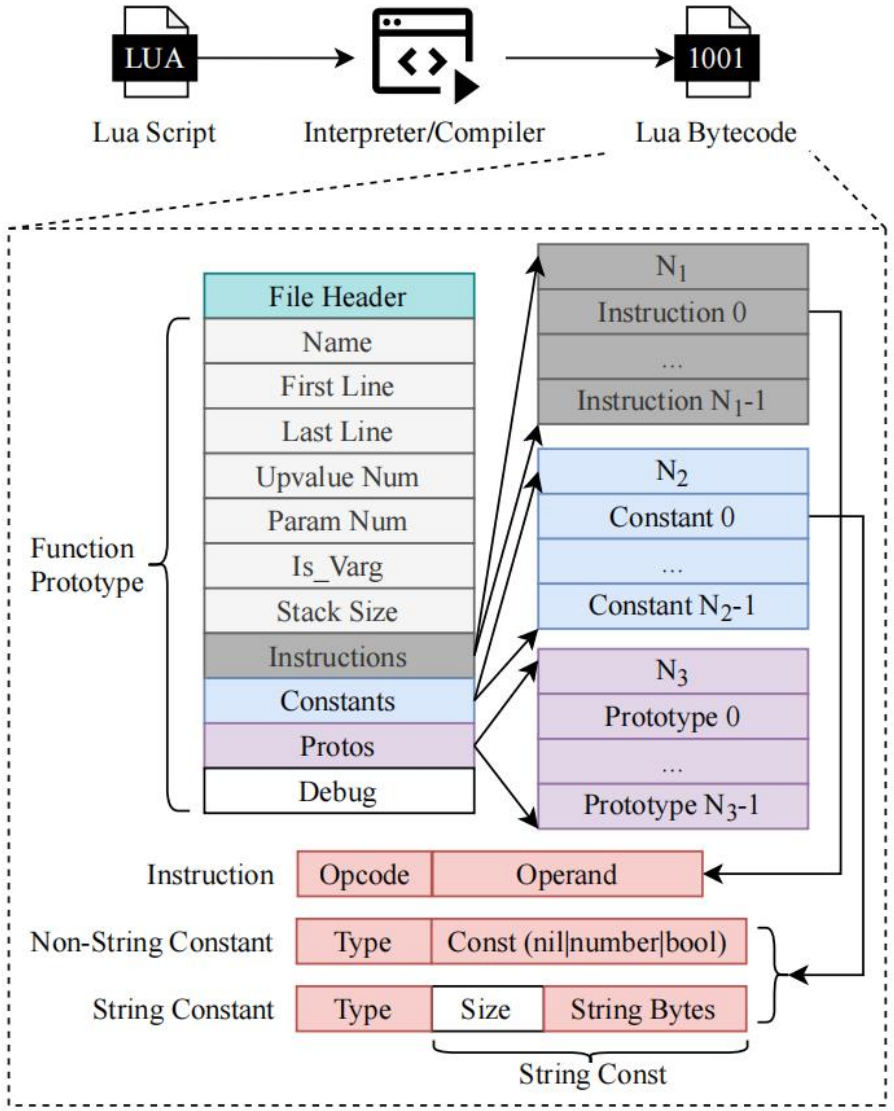
- Main Phase of Taint-based Methods
  - Phase①: Source and Sink Identification
  - Phase②: Taint Propagation from Source to Sink
- Limitations of Existing Methods:
  - Lua Bytecode Deobfuscation (hinders Phase②)
  - Lua-Specific Source Identification (hinders Phase①)
  - C-Lua Communication Modeling (hinders Phase②)

Limitation 1: Drawbacks in Lua Bytecode Deobfuscation



- Bytecode Obfuscation
  - Implementation: Vendor-Customized Lua interpreter
  - Obfuscation Types
    - **Structure Obfuscation**: Structure Field Reordering
    - **Data Obfuscation**: Data Region Modifications (opcode, operand, constant type and content)

Limitation 1: Drawbacks in Lua Bytecode Deobfuscation



- Bytecode Obfuscation (Structure and Data)
- Existing Solutions: Manual analysis (structure) + Mutation-based testing (data)
- Drawback: Labor-intensive and suffers from state space explosion with increasing modified data regions

Problem: How to deal with diverse Lua bytecode obfuscations automatically and efficiently?

Limitation 2: Drawbacks in Source Identification

Dedicated-Function-Targeted Approach

Approach: Using easily recognizable features of dedicated input access functions (e.g., keywords and function signatures)

Drawback: No input access functions in Lua services.

Framework-Specific Approach

Approach: Rely on LuCI-specific file paths or function names

Drawback: Coarse-grained and limited to LuCI framework

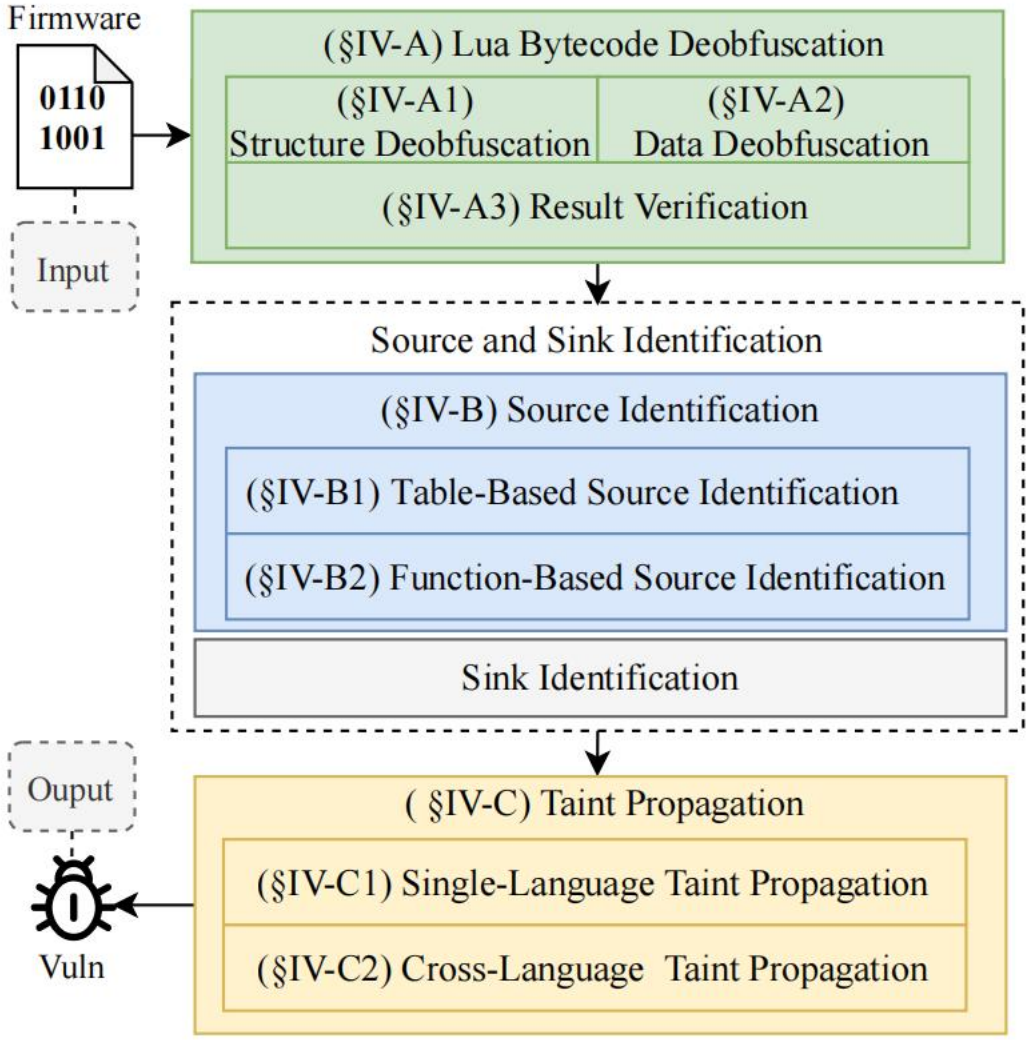
Problem: How to automatically identify Lua-specific sources of firmware services?

Limitation 3: Drawbacks in C-Lua Communication Modeling

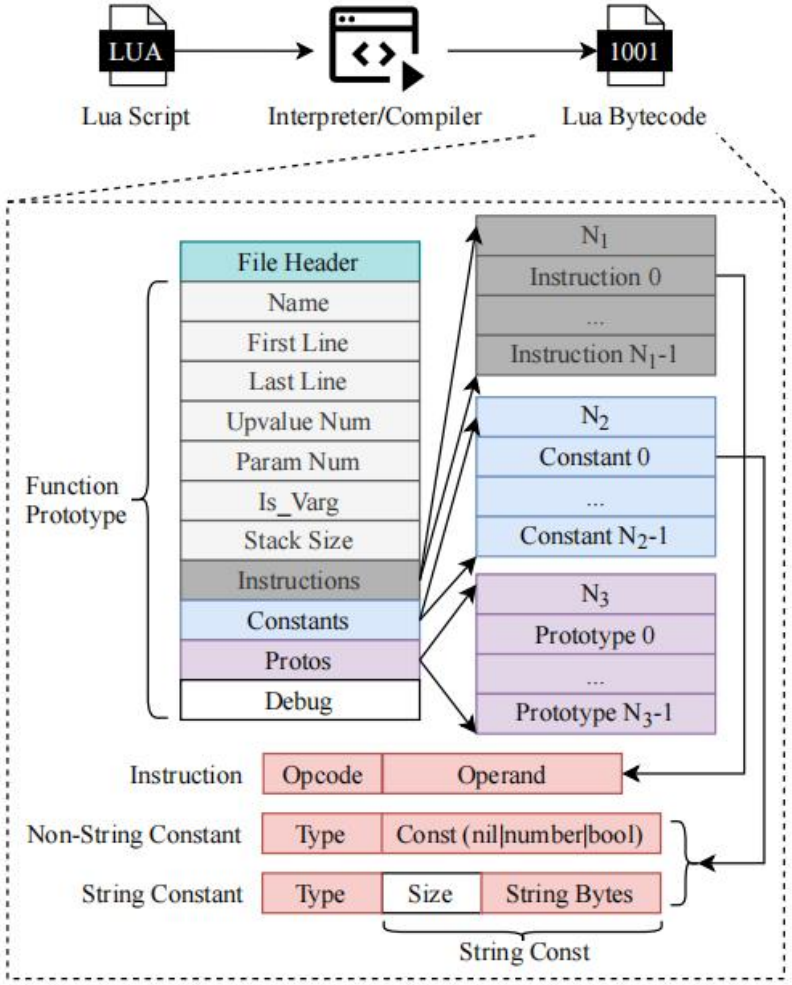
Drawback: There is no work that constructs the communication modeling between C and Lua.

Problem: How to accurately model the communication between C-binaries and Lua scripts/bytecode?

Our Approach: FirmCross



Technique 1: Lua Bytecode Deobfuscation

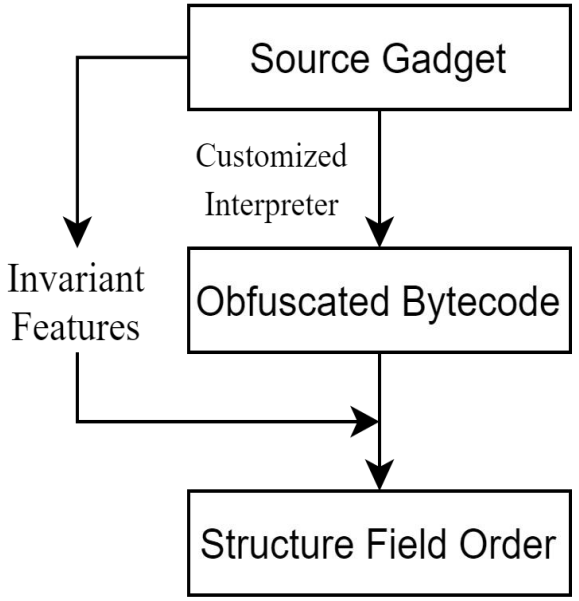


- Observation: **Bytecode Invariants** under Obfuscations
  - Length-header and fixed-termination structure
  - Size-preserving characteristics
  - Prototype Structure Consistency

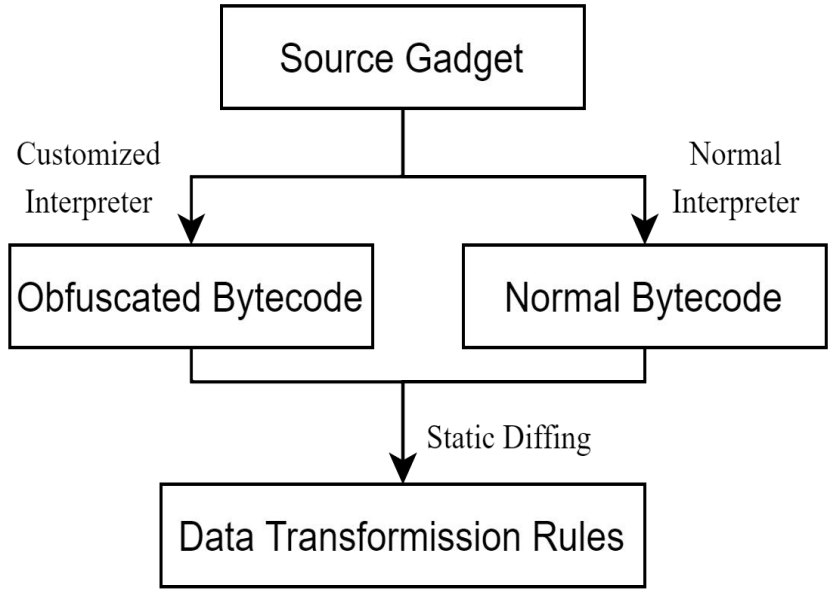
Technique 1: Lua Bytecode Deobfuscation

Problem: How to deal with diverse Lua bytecode obfuscations automatically and efficiently?

Leverage invariant features to solve structure obfuscation



Leverage static-diffing to solve data obfuscation



Technique 2: Lua-Specific Source Identification

Problem: How to automatically identify Lua-specific sources of firmware services?

- Key Insight: Leverage **registration mechanisms of URI handlers** for Lua table-based source identification
  - Sources are usually acquired from **table-structured parameters** of URI handlers
  - URI handlers strictly follow a registration mechanism (through **a nested registry table of a function name string**)

Technique 3: C-Lua Communication Modeling

Problem: How to accurately model the communication between C-binaries and Lua scripts/bytecode?

- Key Insight: Leverage **deterministic patterns** to construct C-Lua communication models.
  - **API-based** Communication: standardized API call chain between C and Lua
  - **IPC-based** Communication: focus on the IPC triggered by command execution

Evaluation Highlights——Bytecode Deobfuscation

- Dataset: 316 images containing obfuscated bytecode collected from our empirical study
- Baseline: The SoTA approach, LuaHunt[TIFS23]

Vendor	#Interp	FirmCross		LuaHunt	
		Pass Rate	AVG time	Pass Rate	AVG time
NetGear	9	100%	19.52	0	-
XIAOMI	40	100%	13.51	0	-
TP-Link	267	100%	21.07	0	-
SUM	316	100%	18.03	0	-

Pass Rate 100% VS Pass Rate 0%

Tool	Ob-Structure	Ob-Data				
		❶	❷	❸	❹	❺
FirmCross	✓	✓	✓	✓	✓	✓
LuaHunt	✓	✗	✗	✗	✓	✗

Ob-Structure: structure obfuscation; **Ob-Data**: data obfuscation; ❶: signed int type addition; ❷: constant type modification; ❸: string xor; ❹: opcode obfuscation; ❺: operand obfuscation;

support 6 types without manual efforts

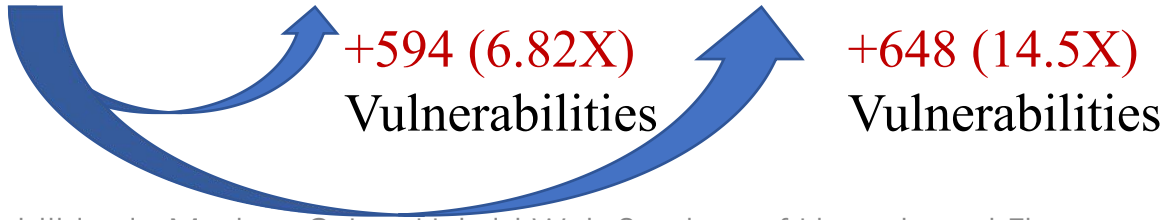
VS

support 2 types with manual efforts

Evaluation Highlights——Vulnerability Detection

- Dataset: 73 images (from existing dataset and new collected) across 11 vendors
- Baseline: MangoDFA[Security24] (Target C Binary), LuaTaint[IOTJ24] (Target Lua Source)

Vendor	#Firmware	FirmCross		MangoDFA		LuaTaint	
		TP	VC	TP	VC	TP	VC
TPLink	11	56	81.16%	8	11.59%	13	18.84%
XIAOMI	10	145	100.00%	0	0.00%	0	0.00%
NetGear	15	117	92.86%	93	73.81%	33	26.19%
Tenda	10	250	100.00%	0	0.00%	0	0.00%
Dlink	10	86	100.00%	0	0.00%	0	0.00%
Ruijie	10	0	-	0	-	0	-
...	...	...	...	...	...	...	...
Total	73	696	96.67%	102	14.17%	48	6.67%



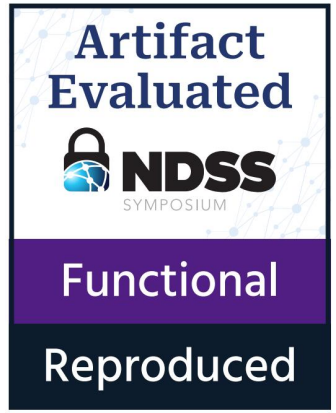
Responsible Disclosure

- 0-Day Vulnerabilities: **610** previously unknown vulnerabilities
- Vendor Feedback: **492** vulnerabilities have been confirmed by vendors, while the remaining are still pending response.
- Acknowledgments from Xiaomi, TP-Link, Tenda, and D-Link.
- Vulnerability Identifiers: **59** official identifiers (CVE/CNVD/CNNVD) to date.

Conclusion

- Main technical contributions
  - Automatic Lua bytecode deobfuscation
  - Lua-specific source identification
  - C-Lua communication modeling
- Key takeaway: long-neglected Lua-involved attack surfaces
- FirmCross: https://github.com/pranksterr009/FirmCross

Github Repo



Thank You

Q & A

Feel free to contact me for follow-up discussions:
runhaoliu@nudt.edu.cn