stelftools: Cross-Architecture Static Library Detector for IoT Malware

stelftools: クロスアーキテクチャに対応した静的結合されたライブラリ関数の特定ツール

Shu Akabane[1], Takeshi Okamoto[1], Yuhei Kawakoya[2]

[1]Kanagawa Institute of Technology [2]NTT Security (Japan) KK

2023/11/8

Agenda

1. Background
2. What is *stelftools*?
3. Demos
   - 3.1. Command line
   - 3.2. GUI(IDA)
   - 3.3. Function Tracing
   - 3.4. Unpacking & Identifying
4. Internal of *stelftools*
5. Conclusion
6. Q & A

Background

- Statically-linked and stripped malware is a challenging problme of malware analysis
  - It contains much amount of code of known library functions, meaning that they are not necessarily a to-be analyzed.
- Pattern-matching with pre-defined singatures is often used to identify the statically-linked functions in given malware, resulting in reducing the amount of code for analysts to read.
- However, the architecture of IoT malware is diverse; this diversity becomes a burden to comprehensively prepare such *pre-defined* signatures beforehand.

What is *stelftools*?

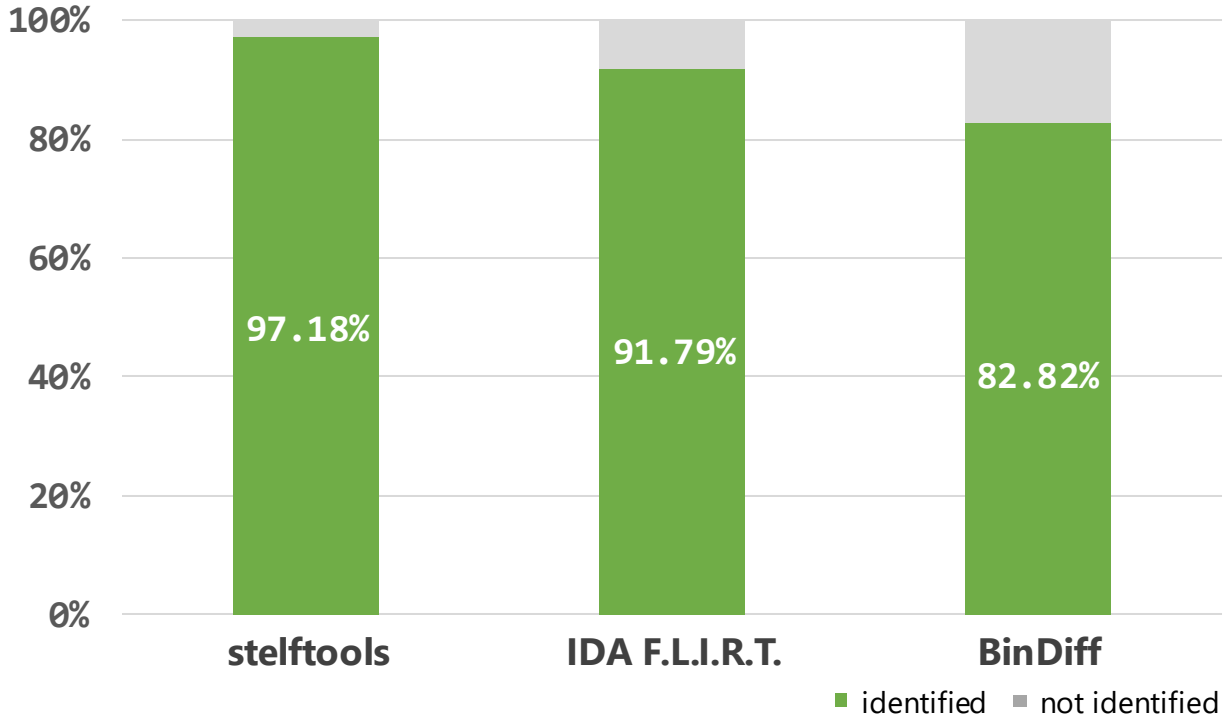
stelftools is a pattern matching tool, enhanced with our-developed heuristics to identify the symbol information of library functions statically-linked to IoT malware

Features

1. **High accuracy**
   - Pattern matching enhanced with call-dependency and linking order
2. **700+ toolchain support**
   - Automatic toolchain identification
3. **17 instruction set architecture support (ISA)**
4. **Integration with external tools**
   - IDA, Ghidra, radare2, Qilling

Feature 1: **High accuracy**

- **Comaparison with IDA F.L.I.R.T and BinDiff**
  - *stelftools* identified statically-linked library functions more accurately than the existing tools through an experiment with 150 IoT malware samples with symbol information, collected by our honeypots.
    - 25,409 of identified library functions / Σ 26,128 of symbols of library functions of samples.



Feature 2: **700+ toolchain support**

- **Support 700+ toolchain patterns**
  - *stelftools* covers most of major toolchains, such as Firmware Linux, Aboriginal Linux, Buildroot, Bootlin, etc
    - One of our past studies found that the diversity of toolchains used in IoT malware is very narrow. With only Firmware Linux, we can cover over 90% malware samples in several IoT malware dataset. [p34]

- **Automatic toolchain identification**
  - Optionally, bruteforce pattern matching is available to automatically identify the toolchain used in IoT malware.

Feature 3: **17 ISA support**

- **Most of major archictecutres of IoT malware is covered.**
  - 17 architectures
    - ARC
    - ARM 32
    - AArch64
    - MIPS(EL)
    - MIPS64(EL)
    - Motorola 68000
    - PowerPC 32
    - PowerPC 64
    - RISC-V 32
    - RISC-V 64
    - SuperH
    - SPARC32
    - SPARC64
    - x86
    - x86_64

Feature 4: **Integration with external tools**

- **Static Analysis**
  - IDA : ida_stelftools.py
  - Ghidra : ghidra_stelftools.py
  - radare2 : r2_stelftools.py *(provided by n01e0)
- **Dynamic Anlaysis**
  - Qilling (experimental) : ql_stelftools.py
- **Any others ?**
  - *stelftools* outputs its results in the plain text format.
    So, it is easy to adopt the results to your analysis tools.

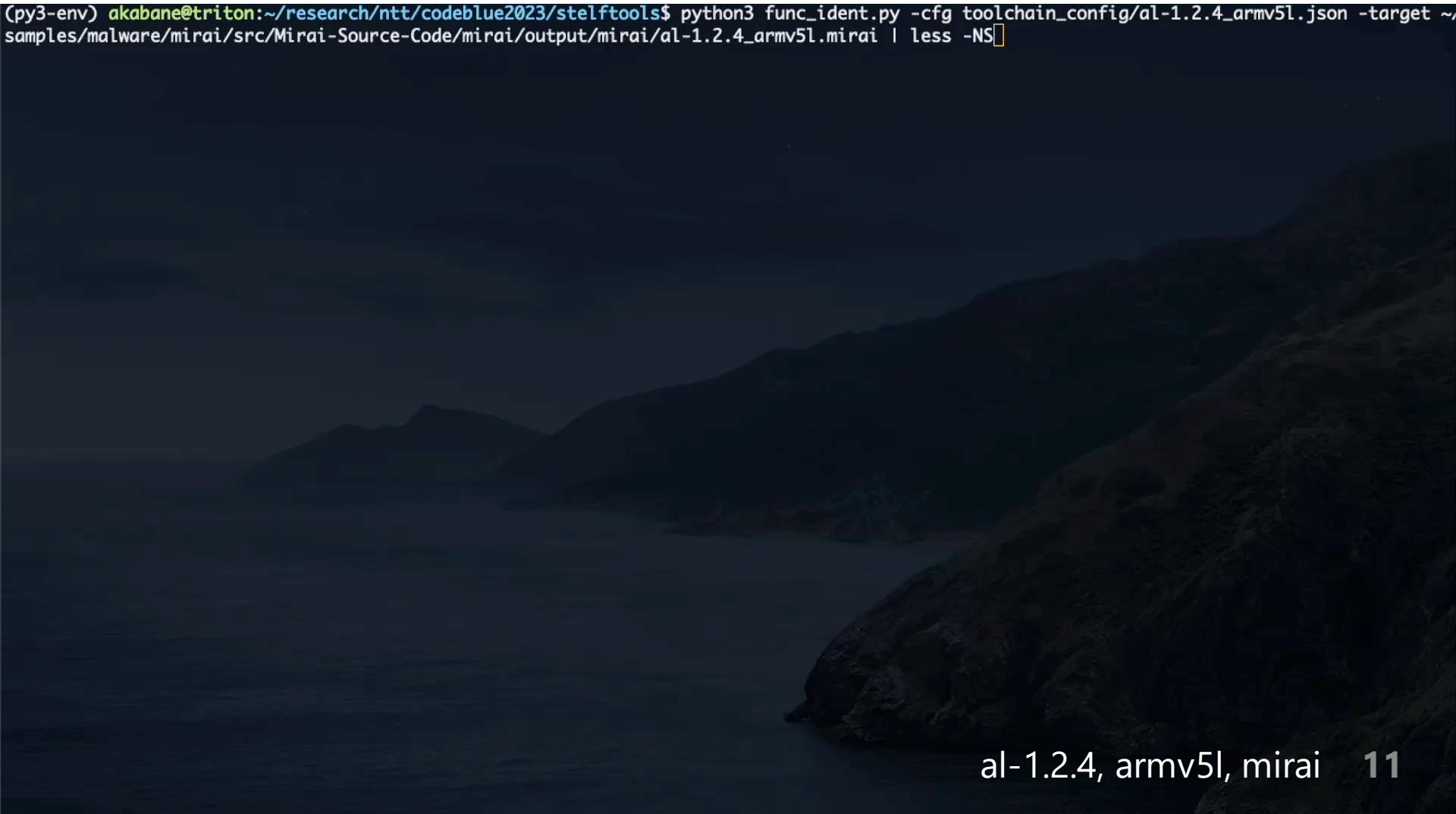
Agenda

1. Background
2. What is *stelftools*?
3. Demos
   - 3.1. Command line
   - 3.2. GUI(IDA)
   - 3.3. Function Tracing
   - 3.4. Unpacking & Identifying
4. Internal of *stelftools*
5. Conclusion
6. Q & A

Demos

- **Command line**
  - Pattern matching (manual)
  - Brute forced toolchain identification(auto)
  - Signature generation
- **GUI (IDA)**
- **Function Tracing**
  - Call tracing in Qilling with stelftools
- **Unpacking & Identifying**
  - Memory dump (user-mode)
  - ELF reconstruction
  - IDA with stelftools

Command line
Pattern matching (manual)



Command line
Usage : Pattern matching

- **Library function identification**

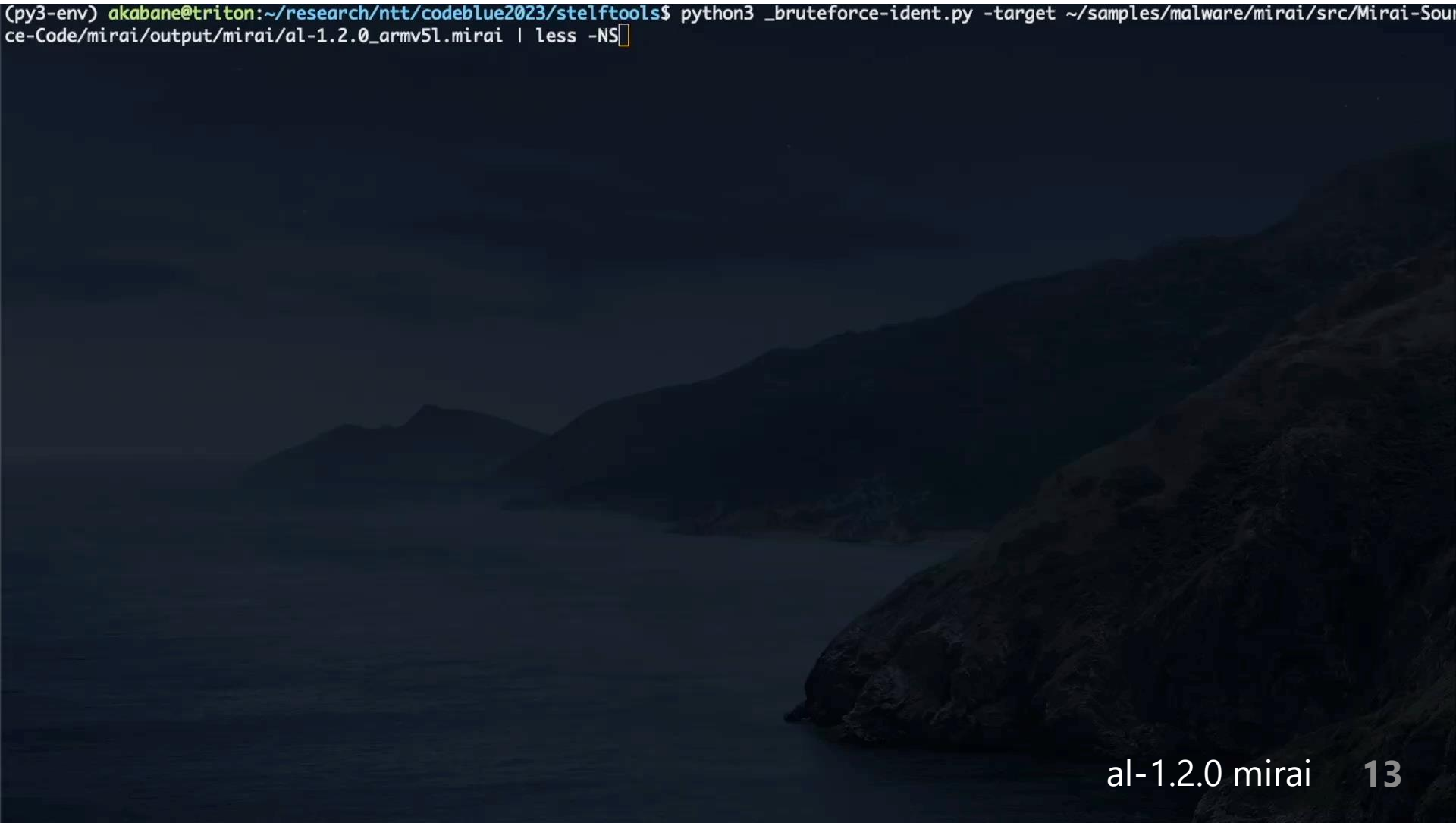
```
python3 ./func_ident.py -cfg ./toolchain_config/{name of toolchain}.json -target {path to target binary}
```

- ex

```
python3 ./func_ident.py -cfg ./toolchain_config/al-1.2.4_armv5l.json -target ./mirai.armv5l
```

Command line

Brute forced toolchain identification (manual)



Command line

Usage : Brute forced toolchain identification

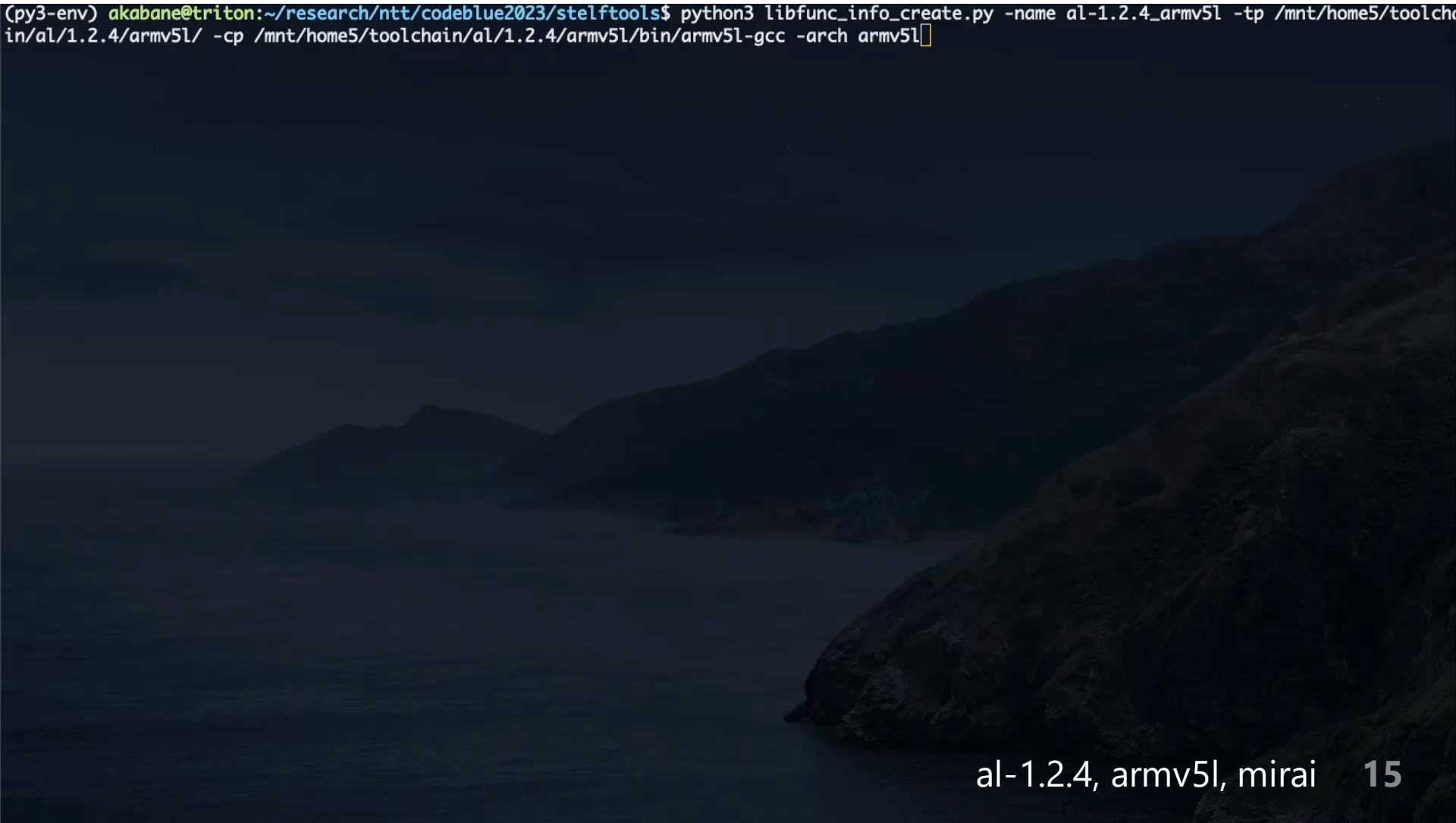
- **Toolchain identification**

```
python3 _bruteforce-ident.py -target {path to target binary}
```

- ex

```
python3 _bruteforce-ident.py -target ./mirai.armv5l
```

Command line
Signature generation



Command line
Usage : Signature generation

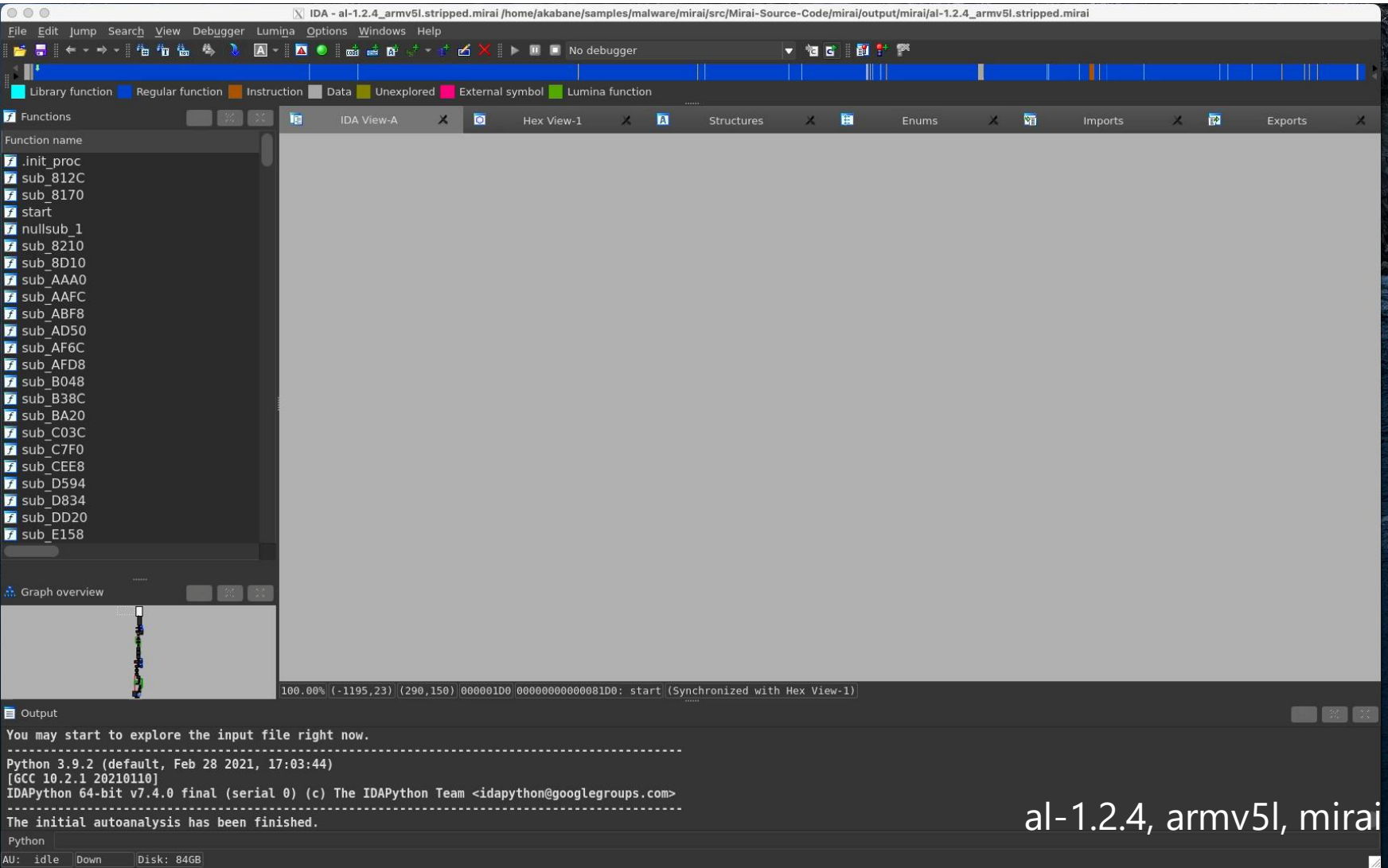
- YARA rules and call-depencency generation

```
python3 ./libfunc_info_create.py ＼
  -name {toolchain name} ＼
  -tp {toolchain path} ＼
  -cp {toolchain compiler path}  ＼
  -arch {toolchain archtecture}
```

- ex

```
python3 ./libfunc_info_create.py ＼
  -name al-1.2.4_armv5l ＼
  -tp /al/cross-compiler-armv5l/ ＼
  -cp /al/cross-compiler-armv5l/bin/armv5l-gcc ＼
  -arch armv5l
```

GUI (IDA)

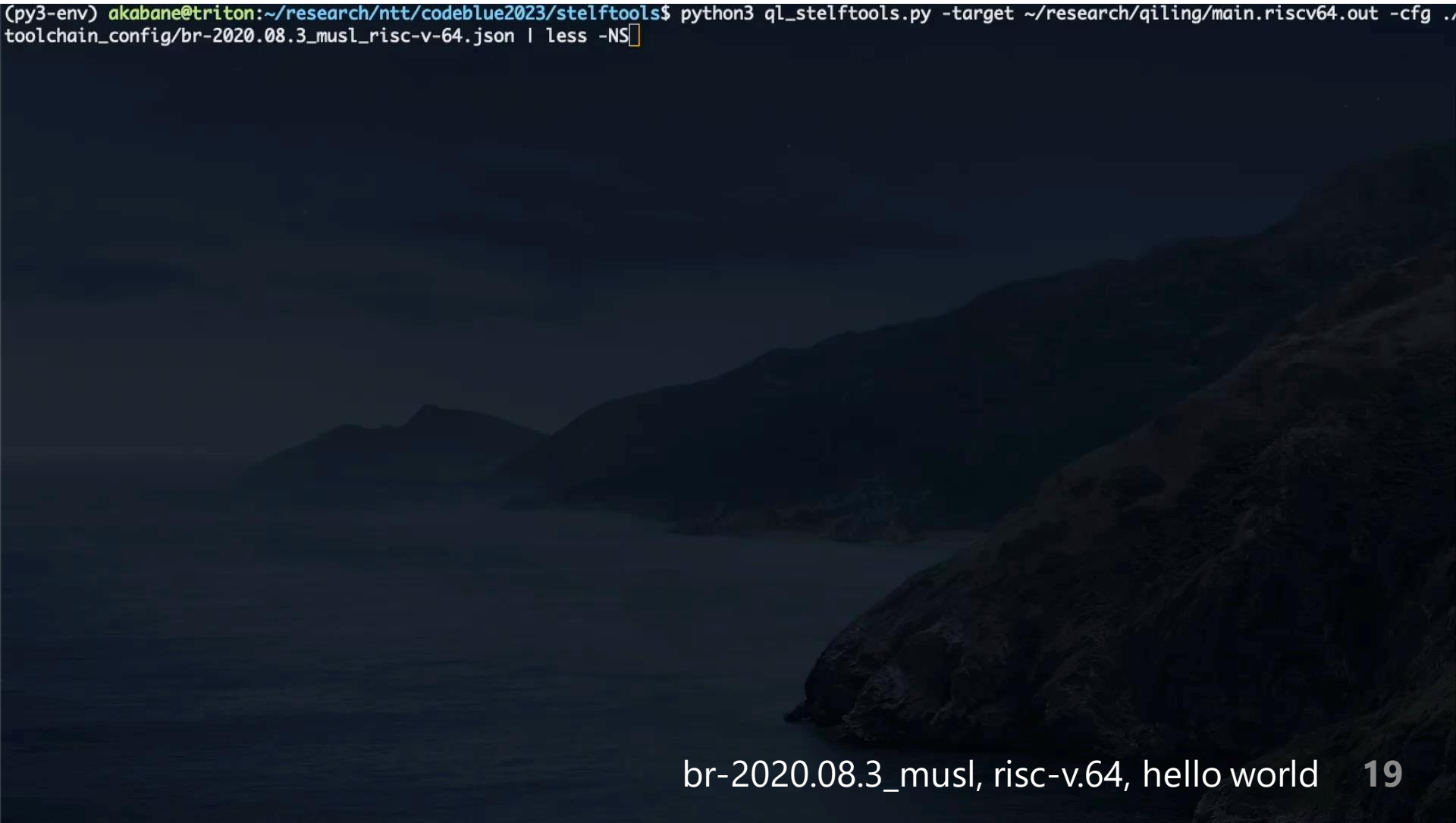


al-1.2.4, armv5l, mirai

GUI (IDA)
Usage : library function identification

- **Identification**
  1. File -> Load file -> stelftools toolchain config file...
  2. open toolchain config file
- **Generation**
  1. File -> Produce file -> stelftools toolchain config file...
     - input toolchain name
     - choose toolchain compiler path
     - input toolchain architecture

Function Tracing
Call tracing in Qilling with stelftools



br-2020.08.3_musl, risc-v.64, hello world

Function Tracing
Unicorn / Qilling with stelftools

- **Tracing library function calls**

```
python3 ql_stelftools.py -cfg ./path/to/{toolchain}.json -target /path/to/bin
  or
python3 ql_stelftools.py -flist /path/to/result_of_libfunc_identification -target /path/to/bin
```

- ex

```
python3 ql_stelftools.py -cfg ./toolchain_config/br-2020.08.3_musl_risc-v-64.json \
  -target ~/samples/main.riscv64.out
```

Our project: Cross-architecture analysis toolset

stelftools

Cross-architecture static library identification

Paper/Presentation

- CSS 2020[1], CSS 2021[2]
- CODE BLUE 2023[3]

Static analysis

Xunpack

Cross-Architecture Unpacking for Linux IoT Malware

Paper/Presentation

- RAID 2023[6] (Tier 2 top conference)

GitHub : https://github.com/ntt-zerolab/Xunpack

Unpack

xltrace

Cross-architecture Tracing for Library Functions

Paper/Presentation

- CSS 2022[4] (Best Paper Award)
- IWSEC 2023[5] (Invited Talk)

Dynamic analysis

Xpack

Cross-Architecture packing for ELF binaries

Presentation

- CSS 2023[7]

Pack

1. Shu Akabane and Takeshi Okamoto, 'Identification of library functions statically linked to stripped IoT malware', CSS 2020
2. Shu Akabane, Yuhei Kawakoya, Makoto Iwamura and Takeshi Okamoto, 'Identifying Library Function Names Based on Function Dependencies and Linking Ordering in IoT Malware', CSS 2021
3. Shu Akabane, Yuhei Kawakoya and Takeshi Okamoto, 'stelftools: cross-architecture static library detector for IoT malware', CODEBLUE 2023
4. Shu Akabane, Yuhei Kawakoya, Makoto Iwamura and Takeshi Okamoto, 'xltrace: Cross-architecture Tracing for Library Functions', CSS 2022
5. Shu Akabane, Yuhei Kawakoya, Makoto Iwamura and Takeshi Okamoto, 'xltrace: Cross-architecture Tracing for Library Functions', IWSEC 2023
6. Yuhei Kawakoya, Shu Akabane, Makoto Iwamura and Takeshi Okamoto, 'Xunpack: Cross-Architecture Unpacking for Linux IoT Malware', RAID 2023
7. Takeshi Okamoto, Shu Akabane, Yuhei Kawakoya and Makoto Iwamura, 'Xpack: Cross-Architecture packing for ELF binaries', CSS2023

Unpacking & Identifying (experimental)

1. Unpacking(Xunpack) -> 2. Library function identification (*stelftools*)

Unpacking & Identifying (experimental)

- **Unpacking** : Xunpack

```
./uunpacker.sh /host/bin/upx_3.96/vanilla/x86_64-buildroot-2020.08.3-uclibc.upx_3.96

python3 elfpack/examples/recon_elf.py ＼
  --arch=x86_64 --output /tmp/dump/7aa1e421-be70-4939-8c1b-e9117bab97c9.elf ＼
  /tmp/dump/7aa1e421-be70-4939-8c1b-e9117bab97c9/0000004c
```

- **Library function identification** : *stelftools*

```
python3 func_ident.py ＼
  -cfg toolchain_config/br-2020.08.3_uclibc_x86_64.json ＼
  -target ~/research/ntt/Xunpack/7aa1e421-be70-4939-8c1b-e9117bab97c9.elf
```

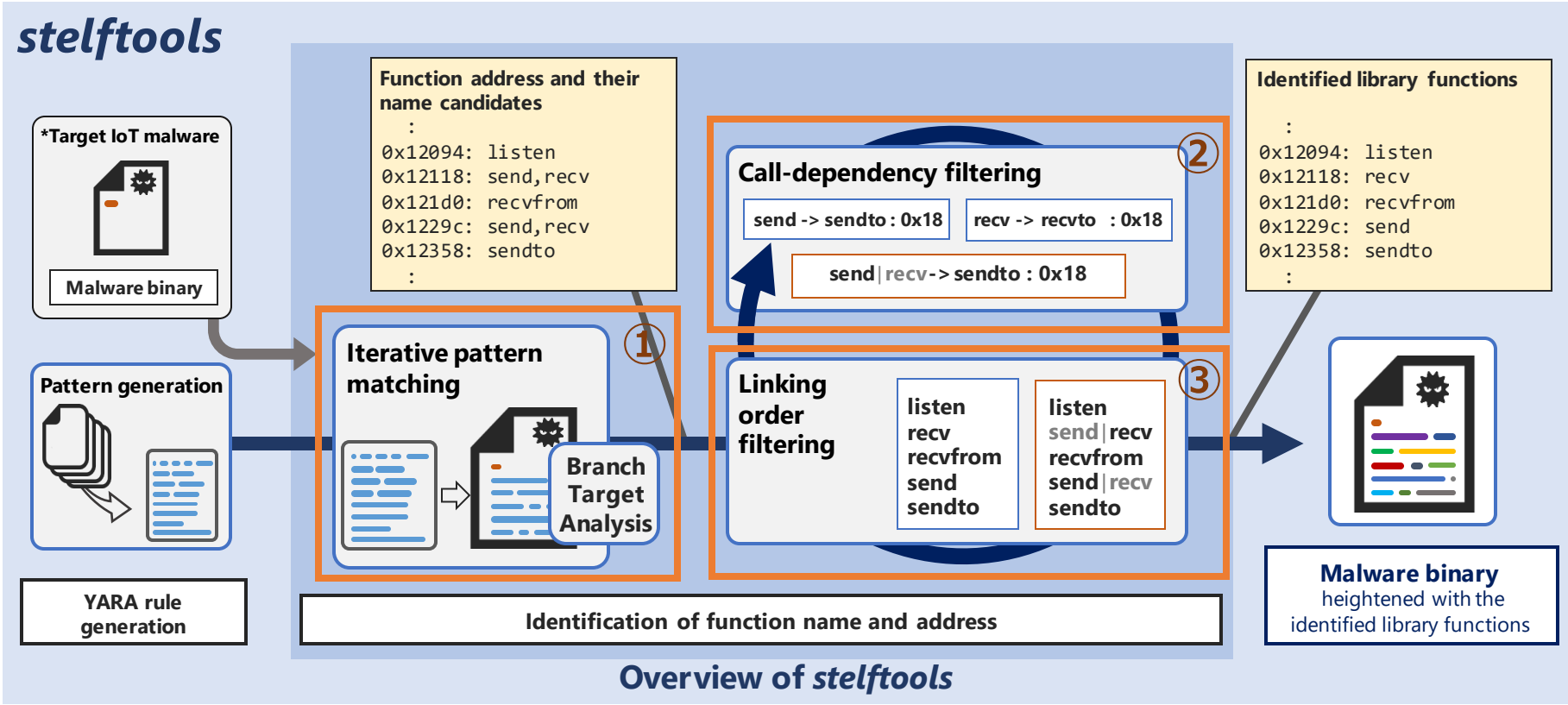
https://github.com/ntt-zerolab/Xunpack

Agenda

1. Background
2. What is *stelftools*?
3. Demos
   - 3.1. Command line
   - 3.2. GUI(IDA)
   - 3.3. Function Tracing
   - 3.4. Unpacking & Identifying
4. Internal of *stelftools*
5. Conclusion
6. Q & A

Internal of *stelftools*

Identify statically-linked library functions through pattern matchings enhanced with branch target analysis(BTA), call-dependency, and linking order.



Overview of *stelftools*

Iterative pattern matching

- Iterative pattern matching involves selecting rules whose pattern length ranges from X to 1.
  - The patterns, defined in YARA rules, are generated from static libraries in advanced.
  - To eliminate the area of a matched function, which is the memory bytes composing the function, from matching memory ranges.
    - Larger functions are matched before smaller ones to avoid false matchings. For example, in the case of a few bytes patterns like `{ E9 ?? ?? ?? ?? }`, which is the pattern of the *random* library function.
- Generate a list of candidates library names for each pattern-matched address.
  - At this point, the generated list inevitably contains false detections. We filter them out using the following heuristics.

Ref: Size X for each architecture

Architecture	Size of X
Motorola 68000	4
SuperH	
ARC	6
ARM 32	8
PowerPC 32	
x86	
x86-64	
AArch64	9
MIPS(EL) 32, 64	
RISC-V 32, 64	
SPARC 32, 64	
PowerPC 64	16

Why does vary between architectures?

This is due to differences of instruction size in the architecture.

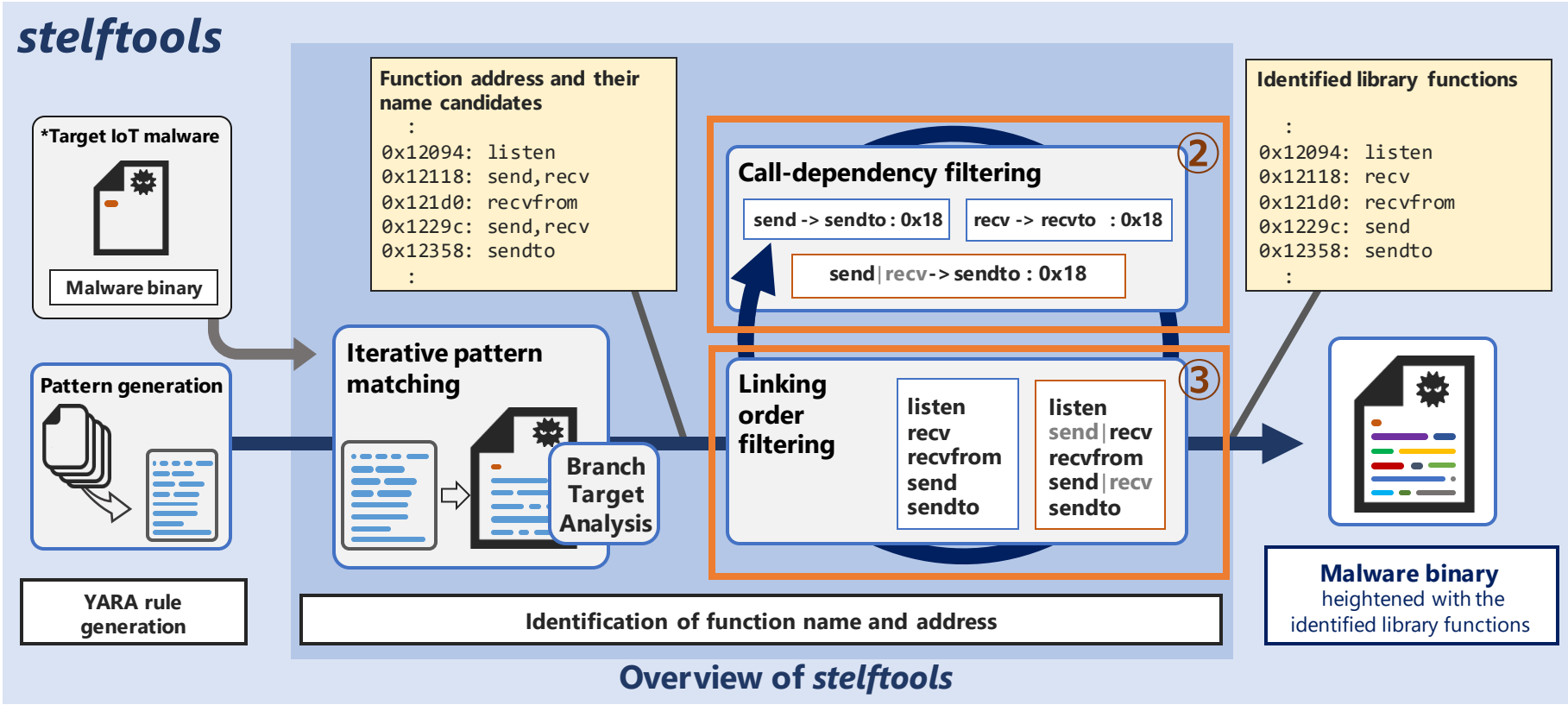
Branch target analysis (BTA)

- We narrow down the address candidates acquired from the iterative pattern matchings.
  - We identify the entry address of library functions by selectively applying one of the following three methods to each architecture, respectively.
    - Method1: identifies the address from the operand value of the instruction calling the function.
    - Method2: identifies the address from the global offset table section.
    - Method3: identifies the address from the data area just after the code area of the function.
  - Then, we remove an address from the candidates if it is not included in any branch target.

Method	Architecture	Target Instructions
1	ARC	operand of **bl, b.d, bl.d** instruction
	ARM 32, AArch64	operand of **b, bl, bx, blls, blne** instruction
	Motorola 68000	operand of **bsr.s, bsr.w, bsr.l, bsr.b** instruction
	PowerPC 32, 64	operand of **b, bl** instruction
	RISC-V 32, 64	operand of **jal** instruction
	SPARC 32, 64	operand of **call, jmp** instruction
	x86	operand of **call, jmp** instruction
	x86-64	operand of **call, jmp** instruction
2	MIPS(EL) 32, 64	
3	SuperH	

Internal of *stelftools*

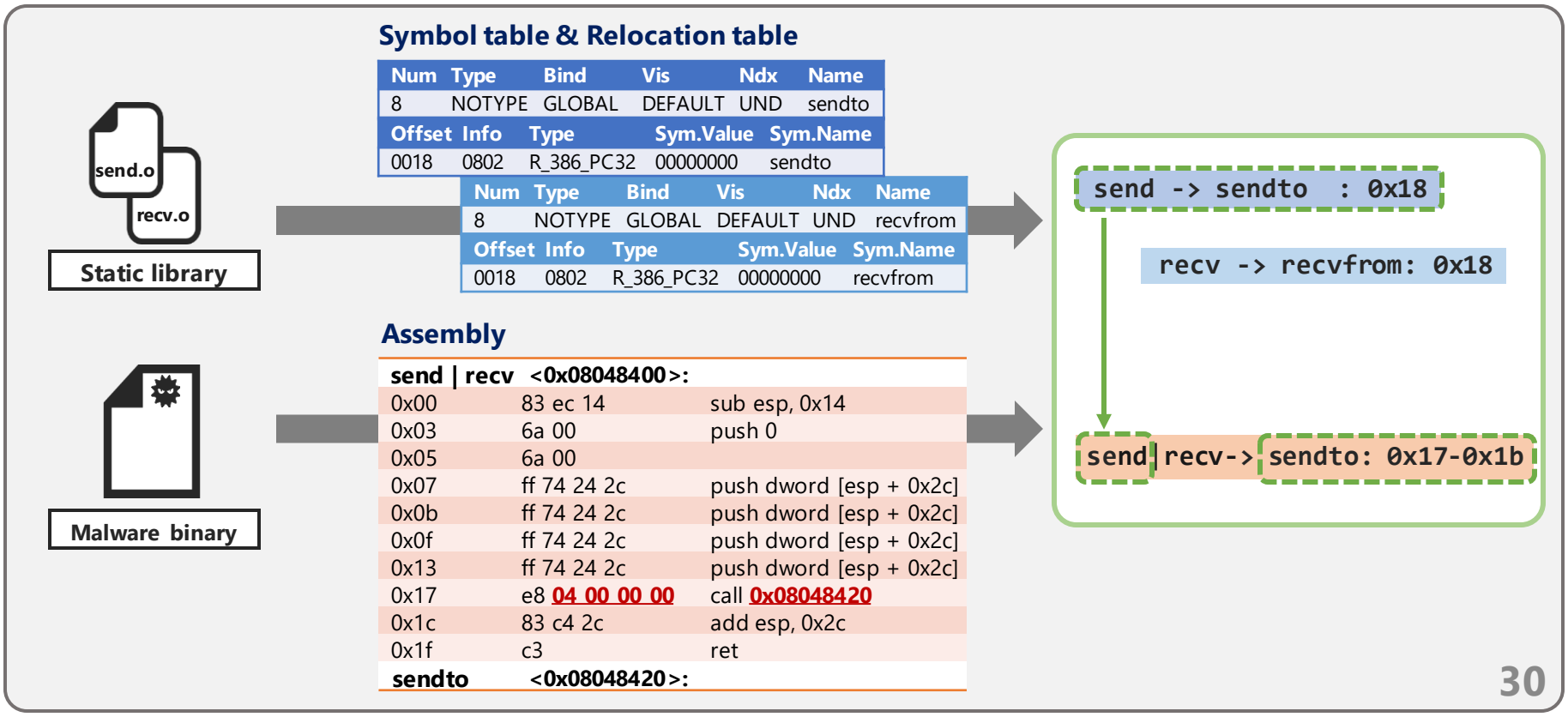
Identify statically-linked library functions through pattern matchings enhanced with branch target analysis(BTA), call-dependency, and linking order.



Overview of *stelftools*

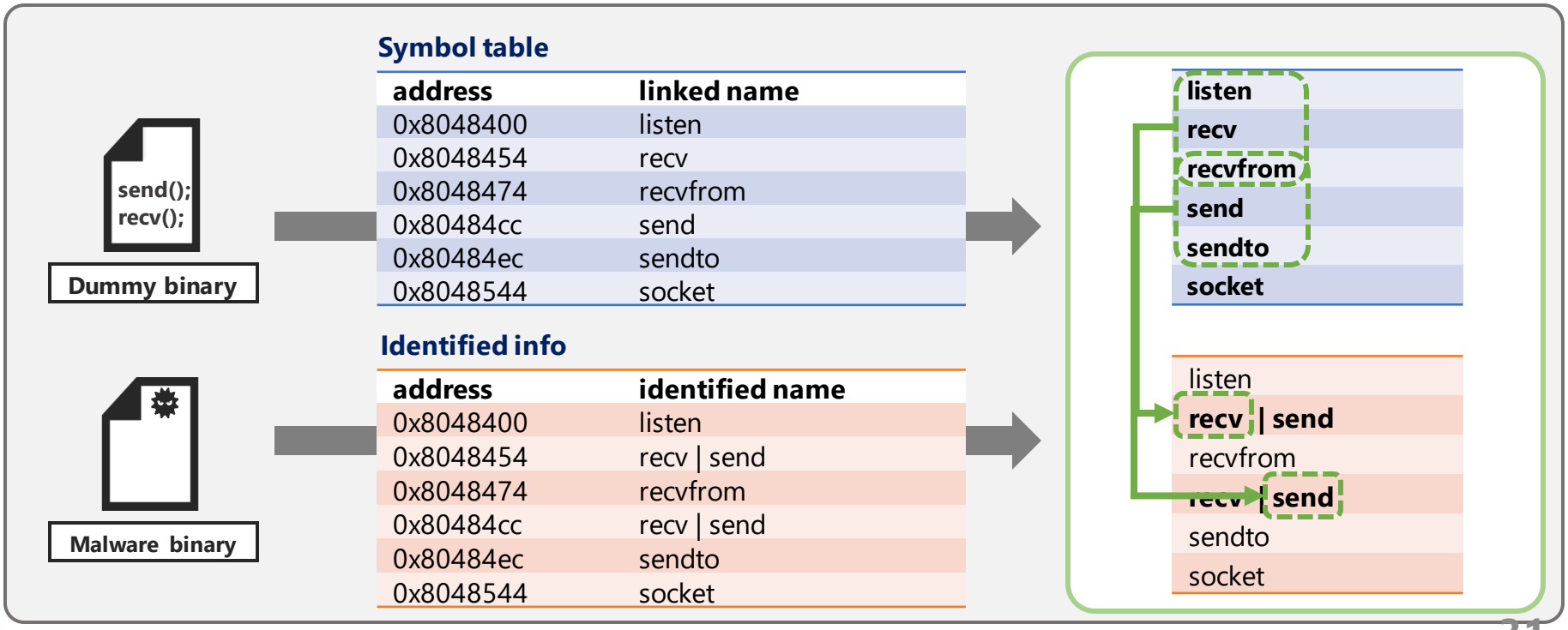
Call-dependency filtering

We narrow down the name candidates for each address using the call-dependency of library functions acquired from static libraries' symbol tables.



Linking order filtering

- **We narrow down the name candidates for each address by referencing dummy binaries' linking order.**
  - Our observation: the order of linked library functions is likely to be unchangeable between dummy binaries and malware.



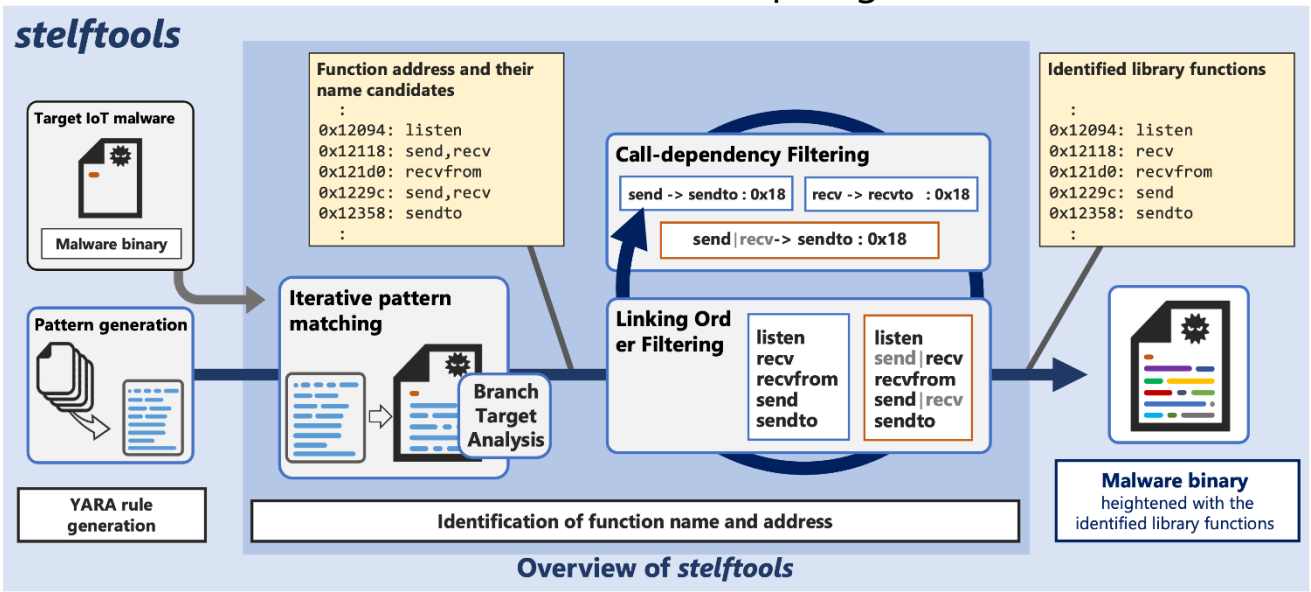
Agenda

1. Background
2. What is *stelftools*?
3. Demos
   - 3.1. Command line
   - 3.2. GUI(IDA)
   - 3.3. Function Tracing
   - 3.4. Unpacking & Identifying
4. Internal of *stelftools*
5. Conclusion
6. Q & A

Conclusion

stelftools is a pattern match tool for statically-linked library functions, enhanced with branch target analysis, call-dependency and linking order.

- **Takeaway**
  - ***stelftools* features high accuracy, 700+ toolchain support, 17 ISA support, and flexible integration with external tools**
  - ***stelftools* are available on GitHub** (https://github.com/shuakabane/stelftools/)

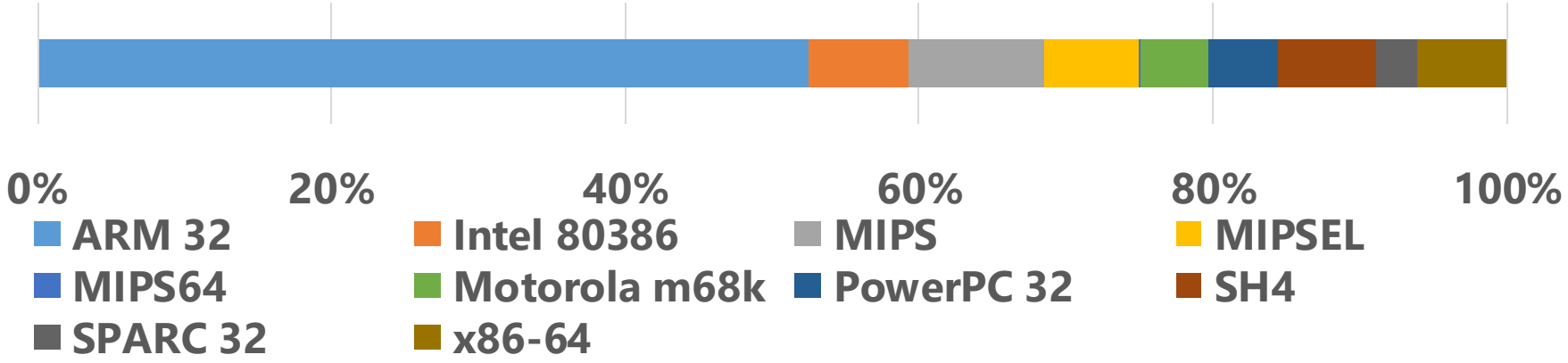


Result of toolchain identification
Our collected dataset

Build tool: Toolchain	Samples
Firmware Linux 0.9.6 : GCC 4.1.2, binutils 2.17, uClibc 0.9.30.1	**3,829**
Aboriginal Linux 1.1.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32	**8**
Aboriginal Linux 1.1.1 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32.1	**6**
Aboriginal Linux 1.2.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.33.2	**17**
Aboriginal Linux 1.2.1 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**11**
Aboriginal Linux 1.2.4 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**27**
Aboriginal Linux 1.2.6 : GCC 4.2.1, binutils 2.17, uClibc .9.33.2	**15**
Aboriginal Linux 1.4.3 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**6**
Aboriginal Linux 1.4.4 : GCC 4.2.1, binutils 2.17, musl 1.1.12	**36**
Buildroot 2018.08 : GCC 7.3.0, binutils 2.29.1, uClibc-ng 1.0.30 (Bootlin prebuilt toolchain)	**22**
Buildroot 2018.08 : GCC 7.3.0, binutils 2.29.1, musl 1.1.19 (Bootlin prebuilt toolchain)	**2**
Buildroot 2018.08 : GCC 7.1.1, binutils 2.29, uClibc-ng 1.0.26 (Synopsys prebuilt toolchain)	**2**
Crosstool-NG 1.24.0-rc1 : GCC 8.2.0, binutils 2.30, musl 1.1.19	**2**

Experiments with YNU dataset

- **We found 7,140 samples in Dataset C that satisfy the following conditions.**
  - ELF format file
  - C language written samples
  - Statically linked to library functions
  - Not stripped of symbol information
- **The architectures of the 7,140 samples**



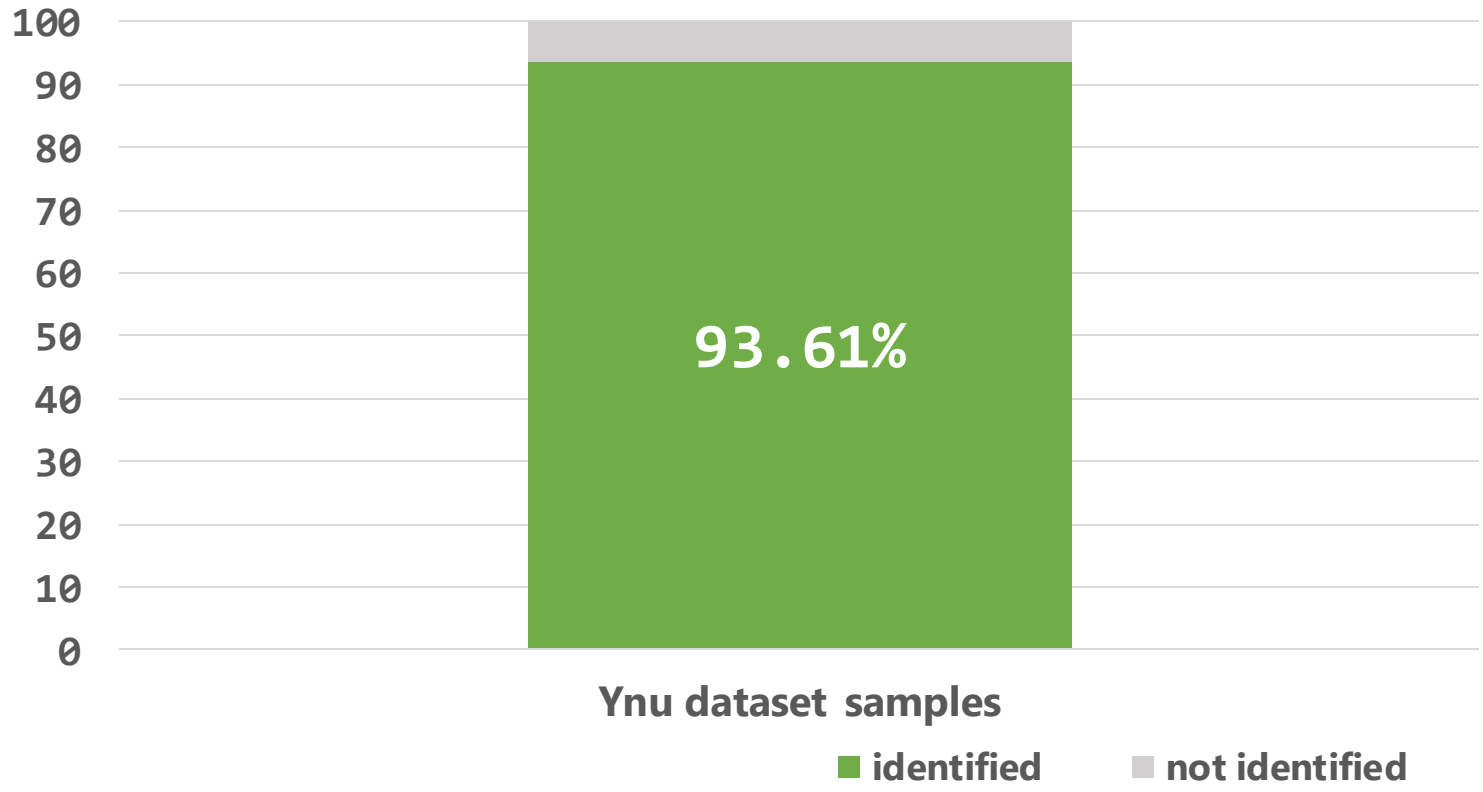
Result of toolchain identification
YNU dataset

Build tool: Toolchain	**Samples** (7,140)
Firmware Linux 0.9.6 : GCC 4.1.2, binutils 2.17, uClibc 0.9.30.1	**3,269**
Aboriginal Linux 1.0.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.31	**2**
Aboriginal Linux 1.0.3 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32	**279**
Aboriginal Linux 1.1.1 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32.1	**1**
Aboriginal Linux 1.2.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.33.2	**3,502**
Aboriginal Linux 1.2.1 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**13**
Aboriginal Linux 1.2.4 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**28**
Aboriginal Linux 1.2.6 : GCC 4.2.1, binutils 2.17, uClibc .9.33.2	**34**
Aboriginal Linux 1.4.3 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**1**
Aboriginal Linux 1.4.4 : GCC 4.2.1, binutils 2.17, musl 1.1.12	**4**
Buildroot 2020.02 : GCC 8.4.0, binutils 2.32, uClibc-ng 1.0.32 (Bootlin prebuilt toolchain)	**1**
Buildroot 2020.08.3 : GCC 9.3.0, binutils 2.33.1, Glibc 2.30	**3**
Buildroot 2020.08.3 : GCC 9.3.0, binutils 2.33.1, uClibc-ng 1.0.34	**1**
Buildroot 2021.11.1 : GCC 10.3.0, binutils 2.36.1, Glibc 2.34 (Bootlin prebuilt toolchain)	**2**

Result of library function identification **YNU dataset**

Accuracy of identification of the library functions in the 7,140 samples with symbol information.

- 1,319,823 of identified library functions / Σ 1,409,929 of symbols of library functions of samples.



Result of toolchain identification
YNU dataset (stripped samples)

Build tool: Toolchain	**Samples** (*41,760/42,215)
Firmware Linux 0.9.6 : GCC 4.1.2, binutils 2.17, uClibc 0.9.30.1	**37,758**
Firmware Linux 0.9.7 : GCC 4.2.1, binutils 2.17, uClibc 0.9.30.1	**2**
Aboriginal Linux 1.0.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.31	**149**
Aboriginal Linux 1.0.2 : GCC 4.2.1, binutils 2.17, uClibc 0.9.31	**21**
Aboriginal Linux 1.0.3 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32	**3,115**
Aboriginal Linux 1.1.1 : GCC 4.2.1, binutils 2.17, uClibc 0.9.32.1	**5**
Aboriginal Linux 1.2.0 : GCC 4.2.1, binutils 2.17, uClibc 0.9.33.2	**233**
Aboriginal Linux 1.2.1 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**30**
Aboriginal Linux 1.2.4 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**297**
Aboriginal Linux 1.2.6 : GCC 4.2.1, binutils 2.17, uClibc .9.33.2	**1**
Aboriginal Linux 1.4.3 : GCC 4.2.1, binutils 397a64b3, uClibc .9.33.2	**4**
Aboriginal Linux 1.4.4 : GCC 4.2.1, binutils 2.17, musl 1.1.12	**465**
Aboriginal Linux 1.4.5 : GCC 4.2.1, binutils 2.17, musl 397a64b3	**68**
Buildroot 2018.02.2 : GCC 6.4.0, binutils 2.29.1, uClibc-ng 1.0.28 (Bootlin prebuilt toolchain)	**1**
Buildroot 2018.11.1 : GCC 7.3.0, binutils 2.29.1, musl 1.1.19 (Bootlin prebuilt toolchain)	**41**
Buildroot 2020.02.1 : GCC 8.4.0, binutils 2.32, uClibc-ng 1.0.32 (Bootlin prebuilt toolchain)	**1**
Buildroot 2020.08.1 : GCC 9.3.0, binutils 2.33.1, musl 1.1.19 (Bootlin prebuilt toolchain)	**3**
Buildroot 2020.08.3 : GCC 9.3.0, binutils 2.33.1, uClibc-ng 1.0.34	**21**