Binary Reversing and Whole Firmware Diffing

Robin David <rdavid@quarkslab.com>
Riccardo Mori <rmori@quarkslab.com>

Version 1.2

BOOTSTRAP

Labs

```
/home/vagrant
├── practicals
│   ├── 01-string-deciphering
│   │   └── 33f46cac84fe0368f33a1e56712add18
│   ├── 02-diffing-cve-patch
│   │   ├── sgdisk-1
│   │   └── sgdisk-2
│   ├── 03-diffing-symbols-porting
│   │   ├── libsensorservice-1.so
│   │   └── libsensorservice-2.so
│   └── 04-firmware-diffing
│       ├── RAX30-V1.0.7.78_1.img
│       └── RAX30-V1.0.9.90_3.img
└── tools
    ├── diffing-documentation
    ├── ghidra_10.3.2_PUBLIC
    ├── idafree-8.3
    └── Sourcetrail
```

Virtual Machine

Ubuntu *22.04*

workshop-bindiff.ova [⬇]

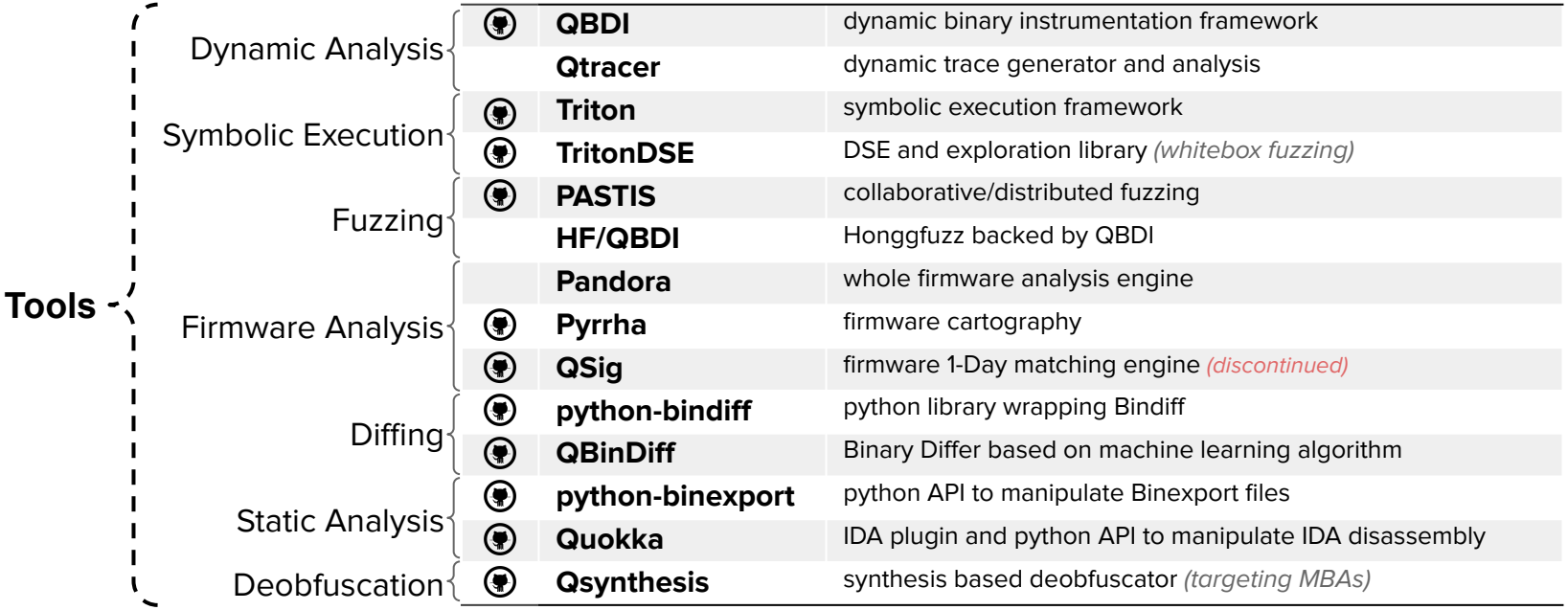
MD5: `456526f45a6f1f319acee7e2c69a1ff3`

Size: 4.0 GB
User: vagrant
Pass: vagrant

The Team

Automated Analysis Team @ Quarkslab

(Reverse wide variety of targets and develop tooling to assists our security assessment)

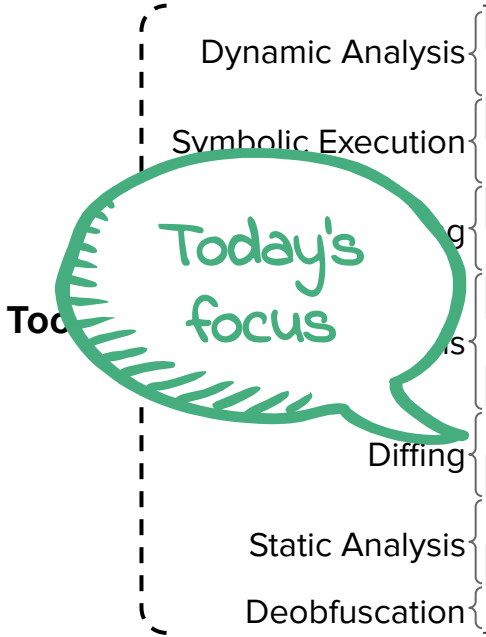


QBDI	dynamic binary instrumentation framework
Qtracer	dynamic trace generator and analysis
Triton	symbolic execution framework
TritonDSE	DSE and exploration library *(whitebox fuzzing)*
PASTIS	collaborative/distributed fuzzing
HF/QBDI	Honggfuzz backed by QBDI
Pandora	whole firmware analysis engine
Pyrrha	firmware cartography
QSig	firmware 1-Day matching engine *(discontinued)*
python-bindiff	python library wrapping Bindiff
QBinDiff	Binary Differ based on machine learning algorithm
python-binexport	python API to manipulate Binexport files
Quokka	IDA plugin and python API to manipulate IDA disassembly
Qsynthesis	synthesis based deobfuscator *(targeting MBAs)*

The Team

Automated Analysis Team @ Quarkslab

(Reverse wide variety of targets and develop tooling to assists our security assessment)



QBDI	dynamic binary instrumentation framework
Qtracer	dynamic trace generator and analysis
Triton	symbolic execution framework
TritonDSE	DSE and exploration library *(whitebox fuzzing)*
PASTIS	collaborative/distributed fuzzing
HF/QBDI	Honggfuzz backed by QBDI
Pandora	whole firmware analysis engine
Pyrrha	firmware cartography
QSig	firmware 1-Day matching engine *(discontinued)*
python-bindiff	python library wrapping Bindiff
QBinDiff	binary differ based on machine learning algorithm
python-binexport	python API to manipulate Binexport files
Quokka	IDA plugin and python API to manipulate IDA disassembly
Qsynthesis	synthesis based deobfuscator *(targeting MBAs)*

Workshop Goal

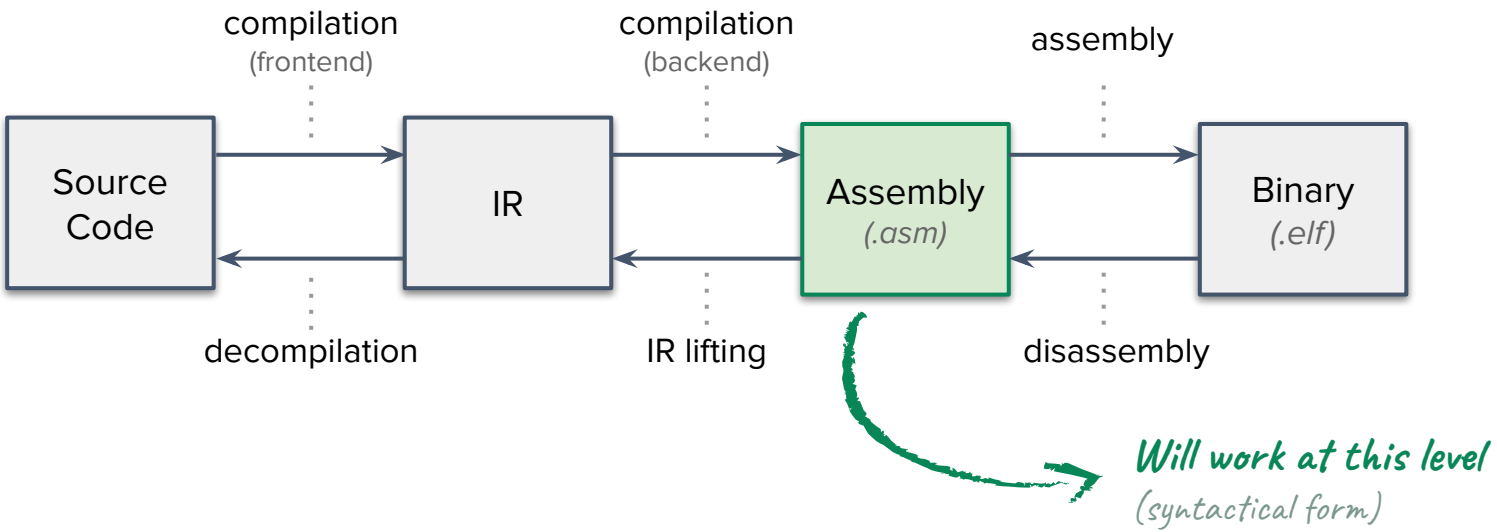
Goal #1

Introducing use-cases and **tools** *(we wrote)* to **speed-up** and to automate reverse & diffing tasks.

Goal #2

Showing how to do **whole** firmware diffing.

Program Representation Levels



Warm-Up: Poll

How familiar are you with

- Assembly (x64, ARM) ?
- Format Analysis (ELF / PE) ?
- IDA / Ghidra  ?
- Python ?

Scripting Reverse Engineering

Quarkslab

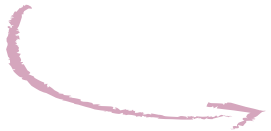
Scripting the Disassembly

Disassembler API
Run the scripting engine within the disassembler context.

- ✓ Usually many features
- ✗ Not portable across disassembler

Exporter
Approach that exports the disassembled program in a file to process it outside of disassembler.

- ✓ API independent from disassembler
- ✓ Can be more compact than disassembler database *(.i64)*
- ✗ Limited features



Study of exporters

Binary Exporters

Binexport

BinExport

Format used by bindiff now maintained by Google

Quokka

Developed by Quarkslab

	Binexport	Quokka
Disassemblers	IDA Pro, Binja, Ghidra	IDA Pro, Ghidra (~)
Format	Protobuf, SQL	Protobuf
Architectures	x86, x64, ARM, Aarch64, DEX, Msil	x86, x64, ARM, Aarch64
Data exhaustiveness	~	+++
Export file size	~	++

Comparison with Quokka

Binary Exporters: Installation

	Binexport	Quokka
Plugins	1. Download the latest release 2. Unpack in the plugin directory 3. Ready to use (more documentation)	1. Download the latest release 2. Unpack in the plugin directory 3. Ready to use (more documentation)
Python API	There is no built-in Python API to manipulate Binexport files! ⇓ *(so we wrote it)* `$ pip install python-binexport`	`$ pip install quokka-project`

https://github.com/quarkslab/python-binexport

Exporting an Executable

	Binexport	Quokka
UI	**IDA:** Edit > Plugins > Binexport **Ghidra**: File > Export Program > Binexport (v2) format	**IDA:** Edit > Plugins > Quokka (Alt+A) **Ghidra**: File > Export Program > Quokka format *(not full)*
Shell	`$ binexporter file.exe` *(wrapper to call idat64 with the good parameters)*	`$ idat64 -OQuokkaAuto:true -A \ hello.exe` *(idat64 not available in IDA Free)*
Python	`from binexport import ProgramBinExport` `p = ProgramBinExport.from_binary_file("file.exe")`	`from quokka import Program` `p = Program.from_binary("file.exe")`

Loading an Export

Binexport

```
from binexport import ProgramBinExport

p = ProgramBinExport("myprogram.BinExport")
for fun_addr, fun in p.items():
  for bb_addr, bb in fun.items():
    for inst_addr, inst in bb.instructions.items():
      for operand in inst.operands:
        for exp in operand.expressions:
          pass  # Do whatever
```

Quokka

```
from quokka import Program

p = Program("prog.quokka", "prog.exe")
for fun_addr, fun in p.items():
  for bb_addr, bb in fun.blocks.items():
    for inst in bb.instructions:
      for operand in inst.operands:
        pass  # Do whatever
```

Quokka Cheatsheet

Accessing functions

```
function = program[0x804F7E0] # address known

function = program.get_function("main") # from name
```

Accessing basic blocks

```
block = function[0x804F7E0] # address known

block = function.get_block(0x804F7E0)
```

Accessing capstone instruction

```
cpst_inst = instr.cs_inst  # capstone object
```

Data access

```
data = program.read_bytes(address, 8)
# Uses file offset
offset = addr - program.base_address
string = program.executable.read_string(offset)
```

Cross References (xrefs)

```
# Call references
call_refs = instr.call_references
address = call_refs[0].address
```

```
# Data references
data_refs = instr.data_references
address = data_refs[0].address
```

Register operations

```
# Find register access (read/write)
from quokka.types import RegAccessMode
instr = quokka.utils.find_register_access(
      "eax", RegAccessMode.WRITE, instructions
)  # Find the instruction that writes into EAX
```

```
# Accessed registers in a instruction
regs_read, regs_write = cpst_inst.regs_access()
```

[documentation]

Practical #0: Exporter usage

Practical #0: Warm-Up

Take any executable on your system and

Tasks:

- Export the binary with Quokka or Binexport
- Load the program using Python API
- Write a script to iterate the content

Practical #01: String Deciphering

Practical #01

The binary is a well-known malware which cipher strings used internally.

Tasks:

- Export the binary with Quokka
- Reverse (manually) to:
  - find the ciphering function
  - understanding the deciphering algorithm
- Write a quokka script to decipher all strings

Link: https://diffing.quarkslab.com/tutorials/ex1_string_decipher.html

Tip

Will need `find_register_access` and `read_bytes` on the executable object.

Solution #01: String Deciphering

⇒ The malware is **mirai** (first seen in 2016)

ciphered strings in .rodata

```
.rodata:08054A70 unk_8054A70     db  5Eh ; ^
.rodata:08054A71                 db  47h ; G
.rodata:08054A72                 db  54h ; T
.rodata:08054A73                 db  56h ; V
.rodata:08054A74                 db  5Ah ; Z
.rodata:08054A75                 db  68h ; h
.rodata:08054A76                 db  45h ; E
.rodata:08054A77                 db  43h ; C
.rodata:08054A78                 db    2
.rodata:08054A79                 db    4
.rodata:08054A7A                 db    2
.rodata:08054A7B                 db    7
.rodata:08054A7C                 db    0
.rodata:08054A7D aOzSGt          db 'OZ_S^GT',0
.rodata:08054A85 aBvycrt         db ']BVYCRT_',0
.rodata:08054A8E unk_8054A8E     db    2
```

⇒

deciphering function calls

```
mov     ecx, 1
mov     edx, offset unk_8054A65
mov     eax, offset aVszY ; "VSZ^Y"
call    sub_804F7E0
mov     ecx, 5
mov     edx, offset unk_8054A70
mov     eax, offset aExxc ; "EXXC"
call    sub_804F7E0
mov     ecx, 1
mov     edx, offset aOzSGt ; "OZ_S^GT"
mov     eax, offset aExxc ; "EXXC"
call    sub_804F7E0
mov     ecx, 1
mov     edx, offset aSrqvbC ; "SRQVB[C"
mov     eax, offset aExxc ; "EXXC"
call    sub_804F7E0
mov     ecx, 1
```

cross ref to data section

⇒

deciphering pseudo-code

```
void sub_804F7E0(char *str1,
                 char *str2) {
    int size = strlen(str1);
    for (int i = 0; i < size;
++i)
        str1[i] = str1[i] ^ 0x37;

    size = strlen(str2);
    for (int i = 0; i < size;
++i)
        str2[i] = str2[i] ^ 0x37;
}
```

Binary Diffing

Quarkslab

Intro Diffing

Introduction

Goal is comparing two *(or more)* binaries to analyze theirs differences. It usually done on functions (1-to-1) mapping computation.
(which can be problematic when functions are merged or split)

Use-cases:

- → malware diffing
- → patch analysis
- → anti-plagiarism
- → statically linked libraries identification
- → symbol porting *(e.g: IDA annotations to a new version of a binary)*
- → backdoor detection *(if a program has been modified)*

Differs

Homemade
https://github.com/quarkslab/qbindiff

		Diaphora	Bindiff	Radiff2	QBindiff	Ghidriff
	Language	Python	Java	C	Python	Python
Disassembler	IDA	✔	✔	✘	✔	✘
	Ghidra	✘	✔	✘	✔	✔
	Binja	✘	✔	✘	✔	✘
	Radare2	✘	✘	✔	✘	✘
	Exporter	SQLite	Binexport	n/c	Binexport Quokka	n/c
	Scripting API	✔	✘	n/c	✔	✔?
	Use decompiler	✔	✘	✘	✘	n/c

Next Bootstrap workshop! You have to attend it ;)

Practical #02: Diffing CVE-2021-0308 patch

Practical #02: Manual Diffing

Diff the two version of the program to understand the CVE patch.

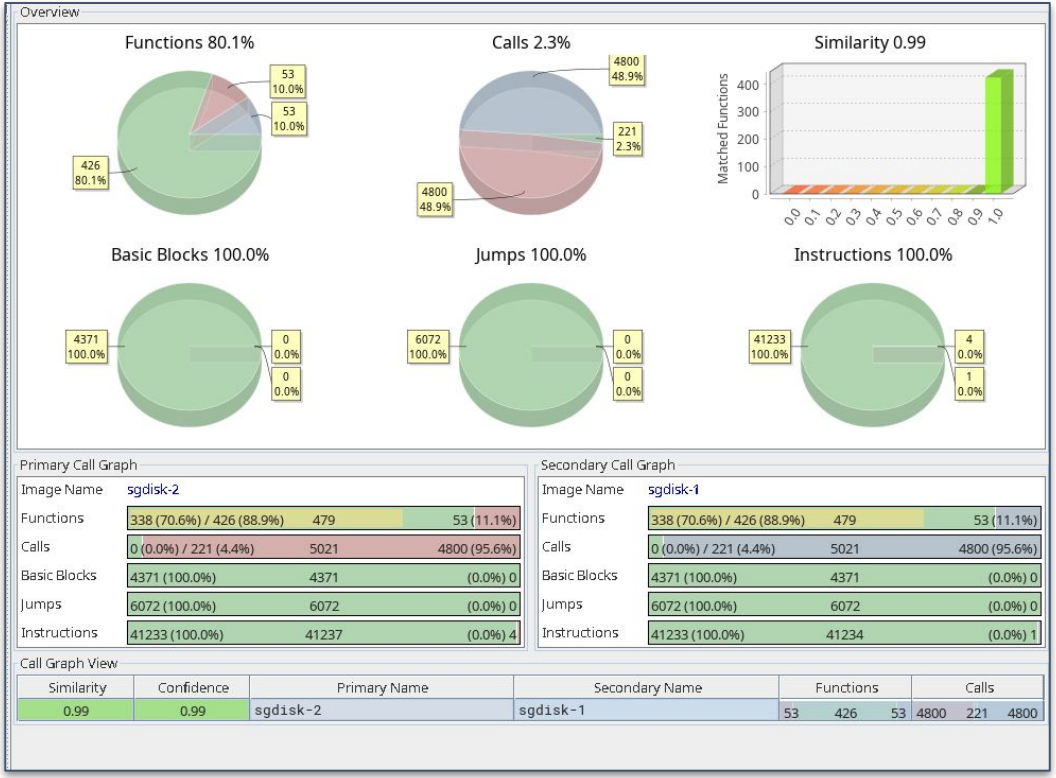
Methodology:

- Export both binaries in BinExport
  - **IDA**: Plugin > BinExport
  - **Ghidra**: Export Progam > BinExport
- Run BinDiff on the exported files
- Open the BinDiff output with: `$ bindiff --ui`
- Identify the code or function affected by the CVE

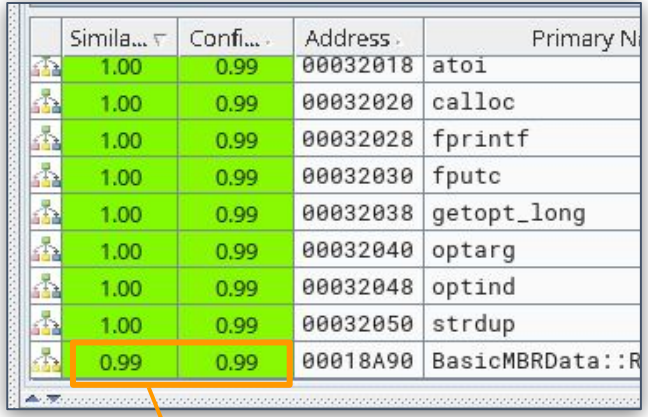
Info

We built the largest dataset of real-world CVEs with both vulns/patched versions. There are **~2000 CVEs**. Its available here: https://github.com/quarkslab/aosp_dataset

Solution #02: Diffing CVE patch



⇒



Function patched!

Scripting Bindiff

Problem

Bindiff made for manual diffing *(with UI)*

⇓

Thus cannot analyze the diff result in a **programmatic** way

Python-bindiff

- Python API to launch Bindiff on two binaries
- Enable scripting the diff result (to analyse it)
- Can automate diffing **whole filesystem**

Python-bindiff

Running a Diff

```python
from bindiff import BinDiff
# Diff two already exported binaries
diff = BinDiff.from_binexport_files(
    "primary.BinExport", "secondary.BinExport", "output.BinDiff"
)
```

```python
# Diff from executable (will call IDA Pro and binexport)
BinDiff.from_binary_files("primary", "secondary", "output.BinDiff")
```

Light-mode

Open diff file (.Bindiff) object and provide an API to manipulate it.

```python
from bindiff import BinDiffFile
# Load a pre-existing BinDiff file
diff = BindiffFile("result.BinDiff")
```

Full-mode

Open diff file and map the result on the two ProgramBinExport objects.

(slower as requires loading the two files)

```python
from bindiff import BinDiff
from binexport import ProgramBinExport
p1 = ProgramBinExport("sample1.BinExport")
p2 = ProgramBinExport("sample2.BinExport")
diff = BinDiff(p1, p2, "output.BinDiff")
```

Practical #03a: Scripting Diffing Result

Practical #03a

There are two binaries which one is stripped. The goal is to automatically port symbols to the stripped binary.

Methodology:

- Generate the diff automatically with python-bindiff
- Find functions changed/added/remove and output a summary
- For matched function add a symbol in the stripped binary

Link: https://diffing.quarkslab.com/tutorials/03a_diffing_porting_symbols.html

Tip: Add symbols in the ELF using LIEF!

LIEF

https://lief.quarkslab.com

```
# List static symbols
binary = lief.parse("./binary")
for symbol in binary.static_symbols:
    pass
```

```
# Add new static symbol
sym = lief.Symbol(...)
binary.add_static_symbol(sym)
```

Solution #03b: Symbol Porting

libsensorservice-2.so

(before symbols porting)

Function name	Segment	Start
start	.text	0000000000016580
nullsub_1	.text	0000000000016590
j_nullsub_1	.text	00000000000165A0
sub_165B0	.text	00000000000165B0
sub_165F0	.text	00000000000165F0
sub_166E0	.text	00000000000166E0
sub_16810	.text	0000000000016810
sub_168F0	.text	00000000000168F0
j__pthread_mutex_destroy	**.text**	**00000000000169C0**
sub_169D0	.text	00000000000169D0
sub_169F0	.text	00000000000169F0
sub_16A20	.text	0000000000016A20
nullsub_2	.text	0000000000016A40

⇒

(after symbols porting)

Function name	Segment	Start
__on_dlclose	**.text**	**0000000000016560**
__emutls_unregister_key	**.text**	**0000000000016570**
__on_dlclose_late	**.text**	**0000000000016580**
android::BatteryService::BatterySer...	**.text**	**0000000000016590**
android::BatteryService::enableSen...	**.text**	**00000000000165D0**
android::BatteryService::checkServi...	**.text**	**00000000000166C0**
android::BatteryService::disableSen...	**.text**	**00000000000167F0**
android::BatteryService::cleanupIm...	**.text**	**00000000000168D0**
android::Mutex::~Mutex()	**.text**	**00000000000169A0**
android::SortedVector<android::Bat...	**.text**	**00000000000169B0**
android::SortedVector<android::Bat...	**.text**	**00000000000169D0**
android::SortedVector<android::Bat...	**.text**	**0000000000016A00**
android::SortedVector<android::Bat...	**.text**	**0000000000016A20**

Automating Firmware Binary Diffing

(batch diffing)

Quarkslab

Batch Binary Diffing

Use-Case

Analyzing a firmware update

Problematic

Diffing the whole filesystem

How

Doing batch diffing

Firmware Diffing

1. Firmware **Extraction**
2. Firmware **Cartography**
3. Firmware **Analysis** & Diffing

Extraction

⇒ Complex tasks, the reference is unblob

```
docker run \
  --rm \
  --pull always \
  -v /path/to/extract-dir/on/host:/data/output \
  -v /path/to/files/on/host:/data/input \
  ghcr.io/onekey-sec/unblob:latest /data/input/path/to/file
```

Cartography

The goal is having a component overview.

⇒ Pyrrha takes filesystem and maps programs and their dependencies

⇒ Mostly a **GUI** to vizualize graphs

```
pyrrha fs ROOT_DIRECTORY
```

Analysis & Diffing

Given two rootfs we can:

- Usual diffing on text files
- Automate bindiff diffing of programs

⇒ **Explore results to understand changes**

Practical #04a: Netgear RAX30

Use-case: Netgear RAX30 Router
⇒ Part of the pwn2own 2022 contest

Versions:

- `V1.0.7.78_1`
- `V1.0.9.90_3` *(released a day before pwn2own submissions!)*

Goal: Identifying what has **been patched**!

Netgear RAX30

Practical #04a: Firmware Extraction

You are given two firmware images for a Netgear RAX30 router. The latter is thus an update.

- Extract the firmware with `unblob`
- Start exploring extracted files

```
docker run \
  --rm \
  --pull always \
  -v /path/to/extract-dir/on/host:/data/output \
  -v /path/to/files/on/host:/data/input \
  ghcr.io/onekey-sec/unblob:latest /data/input/path/to/file
```

Practical #04b: Cartography

Background

⇒ We identified that the router is fetching its firmware updates using libcurl.

⇒ Enabling SSL certificate checks when fetching an URL is done through:

- `CURLOPT_SSL_VERIFYHOST`
- `CURLOPT_SSL_VERIFYPEER`

⇒ These options can be set using `curl_easy_setopt`

Goal: Checking if all binaries using libcurl are properly using SSL *(spoiler they did not..)*

Practical #04b: Firmware Cartography

- Load the first firmware *(1.0.7.78)* rootfs in Pyrrha
- Find the binaries using curl_easy_setopt
- Export executables using this function (with BinExport)

Bonus: Script the check for that flag to identify weak binaries

Pyrrha files:
https://bit.ly/rax30_pyrrha

Practical #04c: Cartography diffing

Background

Before diffing executable at binary level we can:

- diff the two directories with meld etc. (files added/removed)
- diff the two cartography (dependencies added/removed)

Goal ⇒ Having a broad overview of changes.

Practical #04c: Cartography Diffing

➢ Diff the two Pyrrha graph dumps

Can use: https://github.com/quarkslab/pyrrha/blob/main/examples/diffing_pyrrha_exports.py

Practical #04d: Whole firmware diffing

Practical #04d: Firmware Diffing

- Diff all the binaries using: `Bindiff.raw_diffing(p1, p2, out)`
- Load diffs with `BinDiffFile(file)` *(one by one)*
- Print executables that have changed the most!

Link: https://diffing.quarkslab.com/tutorials/04c_firmware_diffing.html

Binexports:
https://bit.ly/rax30-binexports

A Glimpse of QBindiff

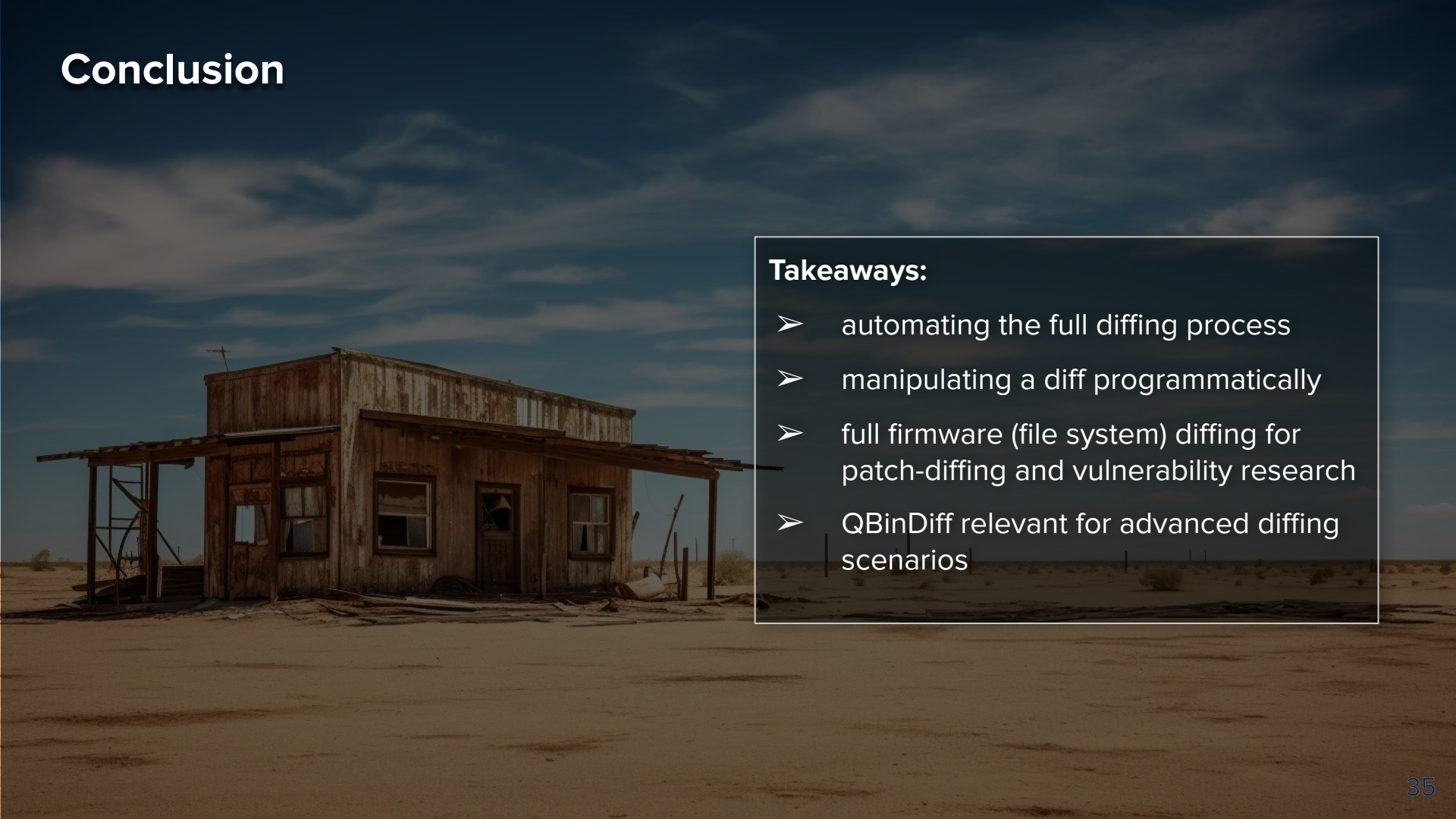
Core Principle:
Diffing essentially made of two components:
Similarity & **Topology** *(and arbitrate the two)*

How: Solve the Network Alignment Quadratic Problem through a Belief propagation algorithm based on **message passing**.

Corollary: Anything that can be encoded as features and a graph can be diffed! *(QBinDiff designed to be highly modular)*

https://blog.quarkslab.com/qbindiff-a-modular-diffing-toolkit.html

Conclusion



Takeaways:

- automating the full diffing process
- manipulating a diff programmatically
- full firmware (file system) diffing for patch-diffing and vulnerability research
- QBinDiff relevant for advanced diffing scenarios