Mining AOSP Dependency Graph for Security

Alexis Challande, Robin David, Guénaël Renault

Quarkslab

Who am I?

Me

Alexis Challande, Ph.D. student (2^{nd} year).
CIFRE between Quarkslab and LiX (Ecole Polytechnique/Inria).

Problem

Problem

Let take a source file $\mathcal{F}$ in a project $\mathcal{P}$.
How to find which **targets** of $\mathcal{P}$ contains $\mathcal{F}$ after the compilation?

What is a **target**?

- Product of a compilation rule;
- Examples: an executable, libraries (shared and static)...

Classical solutions

Handmade process

1. Read the build-file;
2. Find the rules involved to get the final targets;
3. Iterate over every new target using intermediates one.

Building

1. Setup the build environment;
2. Build in debug mode;
3. Read debug information of final targets or parse a `compile-db` file.

Classical solutions

Handmade process

- Time consuming;
- Hard for large systems;
- Error-prone.

Building

- Time consuming;
- Need to have a proper build setup;
- Ressource intensive.

Unified Dependency Graph

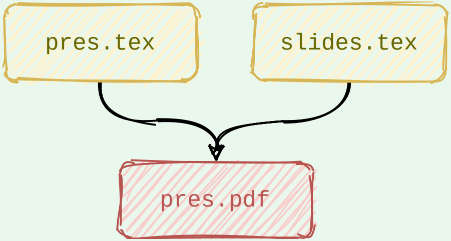
Definition

An **UDG** is a directed graph where:

- Nodes are either source files or compilation targets;
- Edges represent dependency links.

Example

```
# Extract of a Makefile
pres.pdf: pres.tex slides.tex
    lualatex pres.tex
```



pres.tex

slides.tex

pres.pdf

Compilation & Build Systems

GNU autotools (1976)

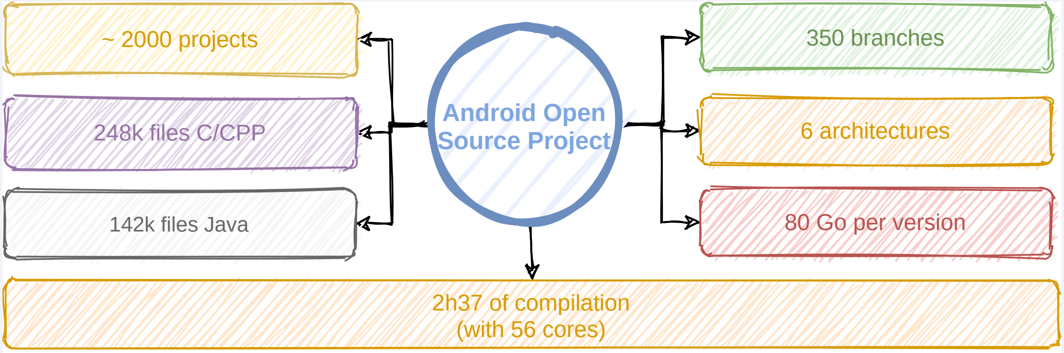
- Defaut build system of the *NIX world;
- Around the `make` command.

(More) Recent challengers

- CMake (2000)
- Ninja (2011)
- Bazel (2015)
- **Soong** (2015)

Android Open Source Project

What is AOSP?



Soong

Soong: a new build system

- Used in AOSP since Android 7;
- Leverage internally Ninja and kati;
- Written in Go;
- Use *blueprint* files for build directives (`Android.bp`).

```
cc_library_shared {
    name: "liblpdump",
    defaults: ["lp_defaults"],
    shared_libs: [ "libbase",
     ↪  "liblog", "liblp",],
    static_libs:
     ↪  ["libjsonpbparse",],
    srcs: ["lpdump.cc",
     ↪  "dynamic.proto",],
}
```

Figure: Extract of an `Android.bp`

From *blueprints* to UDG

Conversion is doable:

- Blueprint are declarative;
- Syntax is explicit;
- Files are easy to parse.

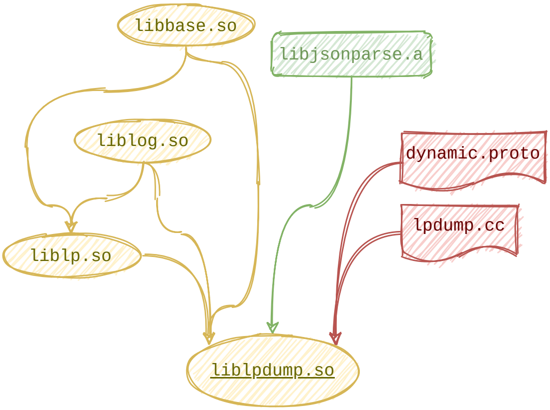


Figure: Extract of the UDG for `liblpdump.so`.

Theoretical grounds

Theorem

A target $\mathcal{A}$ is dependent of $\mathcal{B}$ if and only if a path exists in the UDG from $\mathcal{B}$ to $\mathcal{A}$.

Properties

- The graph induced by a source node represents all its dependencies;
- The intersection of two induced graphs represents common dependencies between two targets.

UDG applied to AOSP

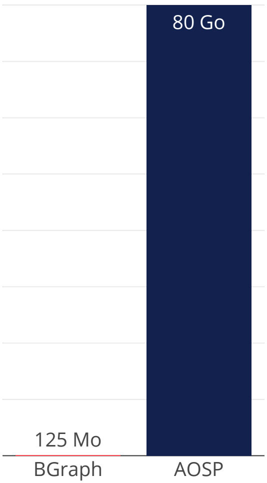
Constructing process for one Android version

1. Checkout all *Android.bp* files;
2. Parse modules;
3. Construct the UDG;
4. Save and use.

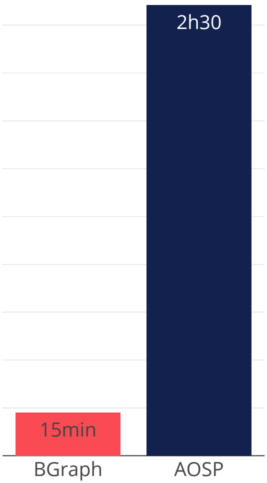
Strengths

- ✓ Fully static: No building time.
- ✓ Sparse: Almost no checkout.
- ✓ Accurate: No guessing.

Figures



Disk usage



Building time

Tool overview

BGraph: Unified Dependency Graphs for AOSP

- Generates and queries *bgraphs*;
- Outputs in multiple formats (`text`, `JSON`, `dot`);
- Works also with a local AOSP mirror;
- Written in Python (Licence Apache 2.0).

Available on GitHub at `https://github.com/quarkslab/bgraph`.[1]

[1] Usually works.

Examples

CVE-2020-0471

CVE-2020-0471

- Fixed in January 2021 in the commit `ca6b0a21`;
- Packet injection in Bluetooth connexions leading to an EoP;
- Patch modified `packet_fragmenter.cc`.

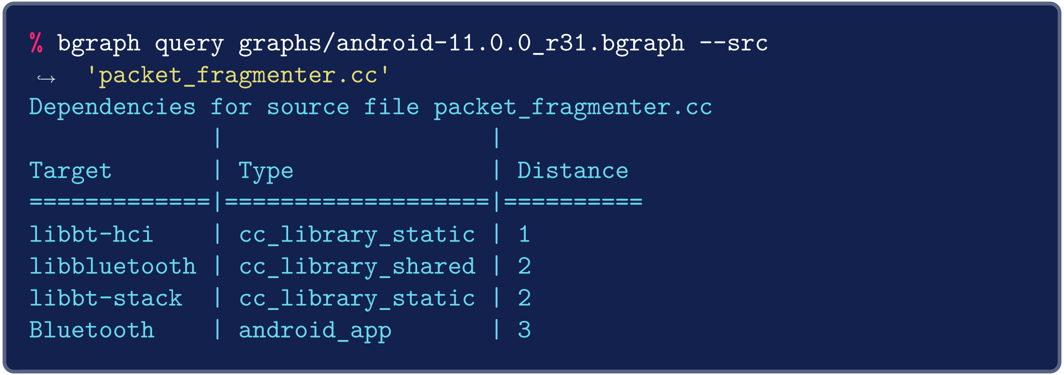
Query

Which entry points in the system that could be impacted by this vulnerability?

CVE-2020-0471

Query

Which entry points in the system that could be impacted by this vulnerability?



```
% bgraph query graphs/android-11.0.0_r31.bgraph --src
↪  'packet_fragmenter.cc'
Dependencies for source file packet_fragmenter.cc
             |                   |
Target       | Type              | Distance
=============|===================|==========
libbt-hci    | cc_library_static | 1
libbluetooth | cc_library_shared | 2
libbt-stack  | cc_library_static | 2
Bluetooth    | android_app       | 3
```

Static vulnerabilities

Definition

A vulnerability affecting a static library is called *static* vulnerability.

Query

What are the *static* vulnerabilities in AOSP (with CVE identifiers)?

Static vulnerabilities

Query

What are the *static* vulnerabilities in AOSP (with CVE identifiers)?

Algorithm

(0.) List vulnerabilities on AOSP.

1. For each vulnerability, list affected files.
2. For each of the affected files, get the first descendent.
3. Accept the CVE if the first descendent is a static library.

Static vulnerabilities

Query

What are the *static* vulnerabilities in AOSP (with CVE identifiers)?

```python
def is_static_lib_vuln(graph: networkx.DiGraph, vuln: Cve) -> bool:
    # Find the (first) target in the graph
    _, targets = bgraph.viewer.find_target(graph, vuln.file, radius=1)
    # Resolve node types
    node_types = set(bgraph.viewer.get_node_type(graph.nodes[targets[0]],
    ↪  all_types=True))
    return 'cc_library_static' in node_types
```

Static vulnerabilities

Query

What are the *static* vulnerabilities in AOSP (with CVE identifiers)?

Results

∼370 vulnerabilities were found, mostly affecting the `Media Framework` and the `System` component.

Artefacts are available in the repository.

Conclusion

BGraph limitations

- ✗ Rely on the exhaustivity on Soong build system;
- ✗ Incomplete parsing/support of blueprint files.

Strengths

- ✓ Resolve the source to target propagation problem.
- ✓ Fast and scalable.

AOSP is an awesome security playground and could bootstrap more security oriented research.

Thank you

Contact information:

✉ achallande@quarkslab.com

☎ +33 1 58 30 81 51

🌐 `https://www.quarkslab.com`

Quarkslab