Towards *1-day* Vulnerability Detection using Semantic Patch Signatures

Alexis Challande

Supervisor: Guénaël Renault   Industrial supervisor: Robin David

Motivation

Note **Patching** software is one of the first security measure to protect a device against threats over time.

Extracts from websites on *How to be secure online?*

3.Run the latest security patches.

An important tip among tips to be secure online is updating to the latest security patches which people don't care to notice or do

5. Keep Your Computer Up to Date!

I know it's annoying, but make sure you check your computer for updates!

1. Keep up with system and software security updates

While software and security updates can often seem like an annoyance, it really is important to stay on top of them. Aside from adding extra features, they often cover security holes.

5. Keep your OS, apps and browser up-to-date.

Always install new updates to your operating systems. Most updates include security fixes that prevent hackers from accessing and exploiting your data.

Motivation

A failure to patch a device leaves **end users** at risk.

Patching is hard:

- Patch un-availability *discontinued software, late updates*
- Patch compatibility with the users needs *may break critical features*
- Incomplete patches *incorrectly patches the vulnerability*

Vendors **limit** the support duration of their products.

Motivation

A failure to patch a device leaves **end users** at risk.

Patching is hard:

- Patch un-availability *discontinued software, late updates*
- Patch compatibility with the users needs *may break critical features*
- Incomplete patches *incorrectly patches the vulnerability*

Android Hidden Patch Gap [Sec20]

In 2019, the rate of missed patches was **30%** *per unique firmware build on average*

Vocabulary & Definitions

Definitions required for the remaining of this presentation

- A software **vulnerability** is a defect in a software with security implications
- A **commit** is a modification of a versioned project
- A software **patch** is a set of changes between two versions [Wan+21]

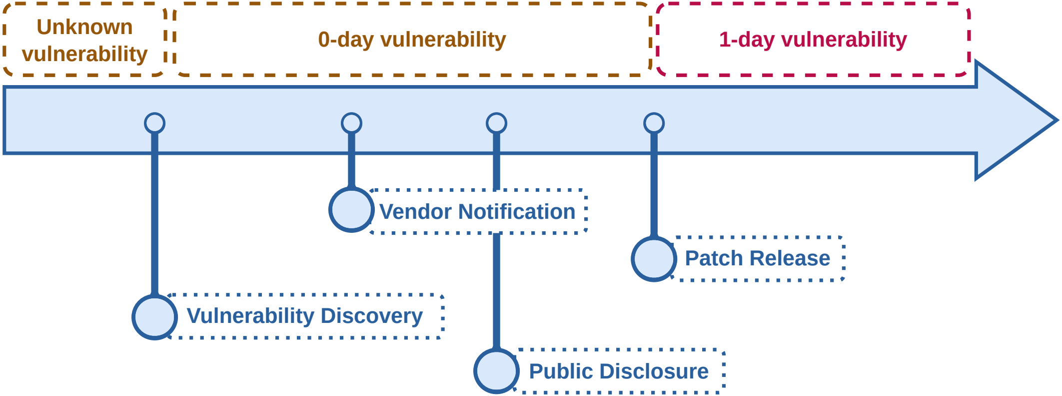
Vocabulary & Definitions

Definitions required for the remaining of this presentation

- A software **vulnerability** is a defect in a software with security implications
- A **commit** is a modification of a versioned project
- A software **patch** is a set of changes between two versions [Wan+21]
- A ***1-day*** is a vulnerability for which a patch has been released *since at least one day*

This work focuses on *1-day* vulnerabilities.

Vulnerability Lifecycle



Problem Statement

How to assert whether a device has been patched against a vulnerability?

Problem Statement

How to assert whether a device has been patched against a vulnerability?

Relying on the reported versions number is insufficient due to patch backport or missing patches.

Patch Presence Test

How to test the presence of a patch in a binary program?

Definition

The **patch presence test** is the capability to accurately check whether a security patch is present inside a software [ZQ18]

The Patch Presence Test is not a generic Bug Search.
Both the **bug** and the **patch** are known.

Binary Analysis

Why considering only binary code?

Focus on **binary only** methods because source code is unavailable

Binary Analysis

Why considering only binary code?

Focus on **binary only** methods because source code is unavailable

WannaCry (2017)

- Massive ransomware attack *attributed to North Korea*
- Propagated using EternalBlue *an exploit stolen from NSA*
- Forced Microsoft to issue a patch for a **deprecated system**

→ The vulnerability affected Samba *a closed source component*

Thesis Contributions

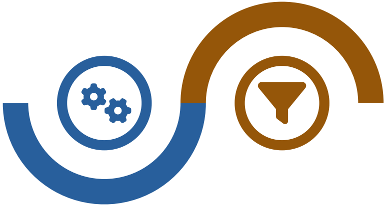
The five main contributions of this work

Formalization of the
Firmware Matching Problem

Thesis Contributions

The five main contributions of this work

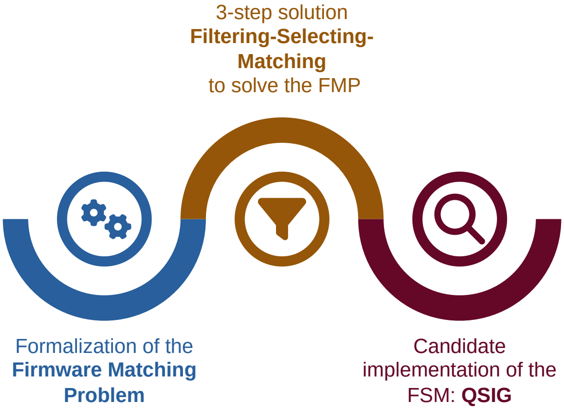
3-step solution
Filtering-Selecting-Matching
to solve the FMP



Formalization of the
Firmware Matching Problem

Thesis Contributions

The five main contributions of this work

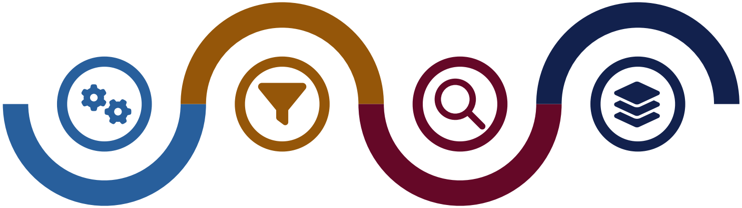


Thesis Contributions

The five main contributions of this work

3-step solution **Filtering-Selecting-Matching** to solve the FMP

Construction of a **commit-level precise dataset** to evaluate QSig



Formalization of the **Firmware Matching Problem**

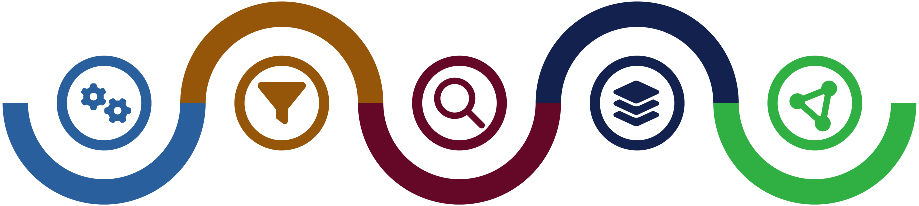
Candidate implementation of the FSM: **QSIG**

Thesis Contributions

The five main contributions of this work

3-step solution **Filtering-Selecting-Matching** to solve the FMP

Construction of a **commit-level precise dataset** to evaluate QSig



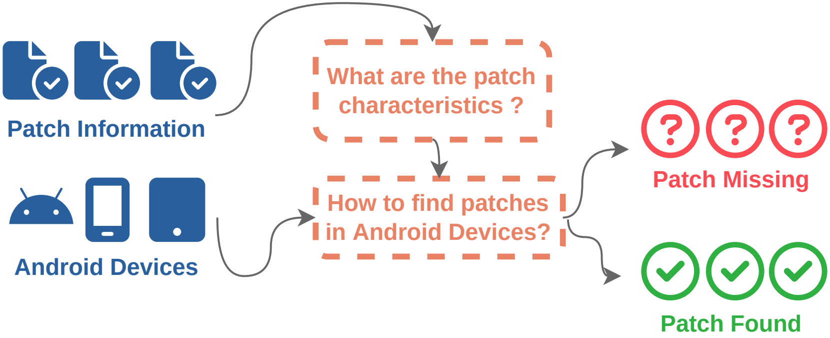
Formalization of the **Firmware Matching Problem**

Candidate implementation of the FSM: **QSIG**

Extending QSig's filtering using **Build Graphs**

Chapter 2:

Firmware Matching Problem



Firmware Matching Problem

Definition

For a given firmware $\mathcal{W}$ and a function specific version f_s find the largest subset $\mathcal{P} \subset \mathcal{W}$ such that $\forall P \in \mathcal{P}, f_s \in P$.
The problem asks to identify the function f_s position in P.

- A $\mathcal{W}$ is a set of programs P *abstract a firmware as a generic filesystem*
- A program P is a set of functions f

Firmware Matching Problem

Definition

For a given firmware $\mathcal{W}$ and a function specific version f_s find the largest subset $\mathcal{P} \subset \mathcal{W}$ such that $\forall P \in \mathcal{P}, f_s \in P$.
The problem asks to identify the function f_s position in P.

The FMP is a systemization of the **Patch Presence Test** on a firmware when the function version is a **patched function**.

State of the Art

Firmware Matching Problem

Numerous approaches have been proposed in the literature to solve the **Firmware Matching Problem**

Selected approaches:

- SPAIN [Xu+17]
- FIBER [ZQ18]
- 1dVul [Pen+19]
- PATCHECKO [Sun+20]
- BINXRAY [Xu+20]
- BScout [Dai+20]
- PDiff [Jia+20]
- VIVA [Xia+21]
- QuickBCC [Jan+21]
- PMatch [Lan+21]
- P1OVD [Li+22]
- ...

State of the Art

Inputs Types

Types of input required by the solution?

- Source code
  - ✅ Precise
  - ✅ Cross-architecture per design
  - ❌ Unapplicable to closed source binaries

- Binary
  - ✅ Generalizable to every target
  - ✅ Precise at the lower level
  - ❌ Single architecture

State of the Art

Inputs Types

Types of input required by the solution?

- Source code
  - ✅ Precise
  - ✅ Cross-architecture per design
  - ❌ Unapplicable to closed source binaries
- Binary
  - ✅ Generalizable to every target
  - ✅ Precise at the lower level
  - ❌ Single architecture

Analysis Types

Does the solution requires a working environment?

- Static
  - ✅ Minimal requirements
  - ✅ Easily scalable
  - ❌ Unable to use runtime values
- Hybrid
  - ✅ Results accuracy
  - ❌ Bootstraping is challenging

State of the Art

Diffing

How to recover the differences between the two inputs?

- General Binary Diffing Tools [Jox21; Zyn21]
  - ✓ Reliable
  - ✓ Offloads parts of the workflow
  - ✗ Not customizable
- Custom solutions
  - ✓ Tailored for a specific problem
  - ✗ Requires additional work in an orthogonal task

State of the Art

Diffing

How to recover the differences between the two inputs?

- General Binary Diffing Tools [Jox21; Zyn21]
  - ✓ Reliable
  - ✓ Offloads parts of the workflow
  - ✗ Not customizable
- Custom solutions
  - ✓ Tailored for a specific problem
  - ✗ Requires additional work in an orthogonal task

Multiple Architectures

How to adapt the solution to multiple binary architectures?

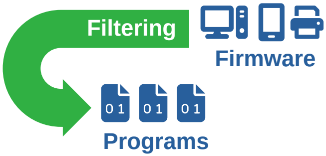
- Intermediate Representation
  - ✓ Write once for every architecture
  - ✓ Easily scalable
  - ✗ Requires an appropriate lifter
- Assembly based
  - ✓ Possibility to use target specific knowledge
  - ✓ No dependency to external tool
  - ✗ Generate a signature per architecture

FSM: Filtering-Selecting-Matching

Solving the FMP using a 3-step solution

Filtering

Identifying inside a firmware the programs containing the target function



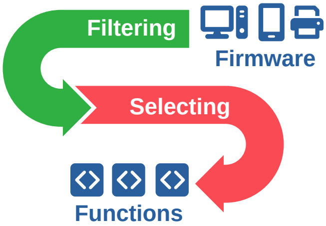
$$\text{Filtering}: \quad \mathbb{W} \times \mathbb{S} \longrightarrow 2^{\mathbb{P}}$$
$$(\mathcal{W}, \mathcal{S}) \longmapsto \mathcal{P} = \{P_0, \ldots, P_n\}$$

FSM: Filtering-Selecting-Matching

Solving the FMP using a 3-step solution

Filtering
Identifying inside a firmware the programs containing the target function

Selecting
Within a program, selecting the appropriate function(s)



$$\text{Selecting}: \quad 2^{\mathbb{P}} \times \mathbb{S} \longrightarrow 2^{\mathbb{F}}$$
$$(\mathcal{P}, \mathcal{S}) \longmapsto \mathcal{F} = \{f_0, \ldots, f_m\}$$

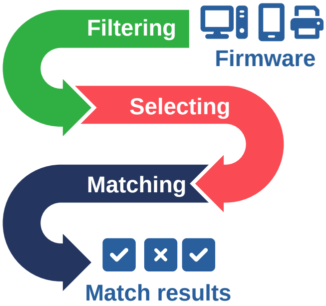
FSM: Filtering-Selecting-Matching

Solving the FMP using a 3-step solution

Filtering
Identifying inside a firmware the programs containing the target function

Selecting
Within a program, selecting the appropriate function(s)

Matching
Determining the selected function specific version



$$\text{Matching}: \begin{array}{ccc} 2^{\mathbb{F}} \times \mathbb{S} & \longrightarrow & \mathbb{R} \\ (\mathcal{F}, \mathcal{S}) & \longmapsto & \mathcal{R} \end{array}$$

FMP in the State of the Art

How is the FMP addressed in the literature?

	Filtering	Selecting	Matching	Multiple Functions
1dVul [Pen+19]	✗	✗	✓	N/A[1]
QuickBCC [Jan+21]	✗	✓	✓	✗
PMatch [Lan+21]	✗	✗	✓	✗
P1OVD [Li+22]	✗	✓	✓	✗
FMP with FSM	✓	✓	✓	✓

None of the previous approaches tackles every aspect of the FMP.

[1] Generates a crashing input

Patch: Fixing Commits Profile

What are the changes induced by a **security** commit on a project?

Objectives

Characterizing patches helps to:

- Design *signatures*
- Search them among other commits *silent fix detection*

Patch: Fixing Commits Profile

Versioned Project

Let us define a project P as a sorted sequence of commits *simplified «git-like» definition*

$$P^i = \{c_0, c_1, \ldots, c_{i-1}, c_i\}$$

- P^i is the project state after the application of c_i

Patch: Fixing Commits Profile

Versioned Project

Let us define a project P as a sorted sequence of commits *simplified «git-like» definition*

$$P^i = \{c_0, c_1, \ldots, c_{i-1}, c_i\}$$

- P^i is the project state after the application of c_i

Code Property Graph

The **CPG** $G = (V, E, \lambda, \mu)$ of a program P is a directed edge-labeled attributed multigraph constructed from its AST *Abstract Syntax Tree*, its CFG *Control Flow Graph* and its PDG *Program Dependency Graph* [Fer87].

PatchAnalysis

Tools

Establish a **fixing-commit** profile by computing the difference between the CPGs of the project in a vulnerable and fixed state.

First define a **labelling** function ψ

$$\begin{aligned}\psi : \quad V &\longrightarrow \Phi \\ v &\longmapsto \{\text{String}, \text{Constant}, \ldots\}\end{aligned}$$

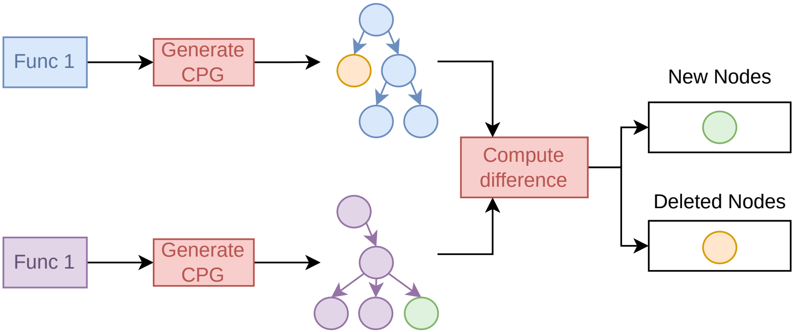
Then, compute the changed nodes between the CPGs for f in **vuln** et **fixed** version

$$\begin{aligned}\mathbb{D}^f = &\left\{(\text{add}, \psi(v)) : v \text{ a vertice in } G^f_{fix} \setminus G^f_{vuln}\right\} \\ &\cup \left\{(\text{del}, \psi(v)) : v \text{ a vertice in } G^f_{vuln} \setminus G^f_{fix}\right\}\end{aligned}$$

PatchAnalysis

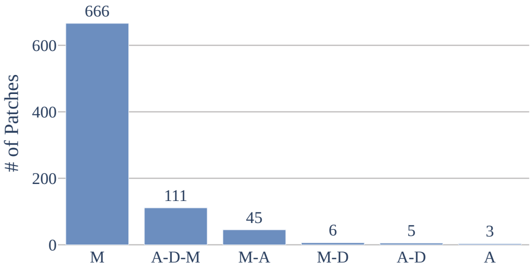
Tools

Establish a **fixing-commit** profile by computing the difference between the CPGs of the project in a vulnerable and fixed state.



Fixing-commit Profiles: Results

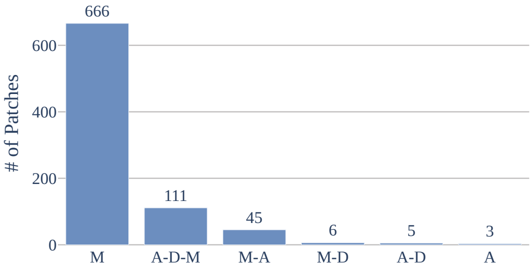
Program Level:



A: Addition
D: Deletion
M: Modification

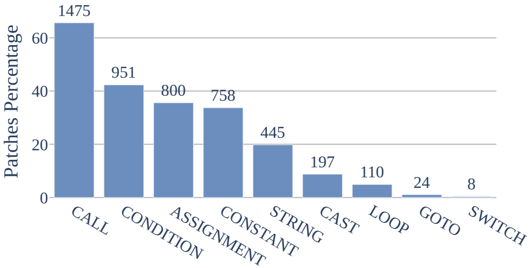
Fixing-commit Profiles: Results

Program Level:



A: Addition
D: Deletion
M: Modification

Function Level:



Patch Signatures

Patch signatures are required to solve the FMP using the FSM

Signatures

Our patch signatures are based on **«semantic invariants»** *portable artifacts*

Signatures Features

Filtering

- Binary Name
- File type
- ...

Selecting

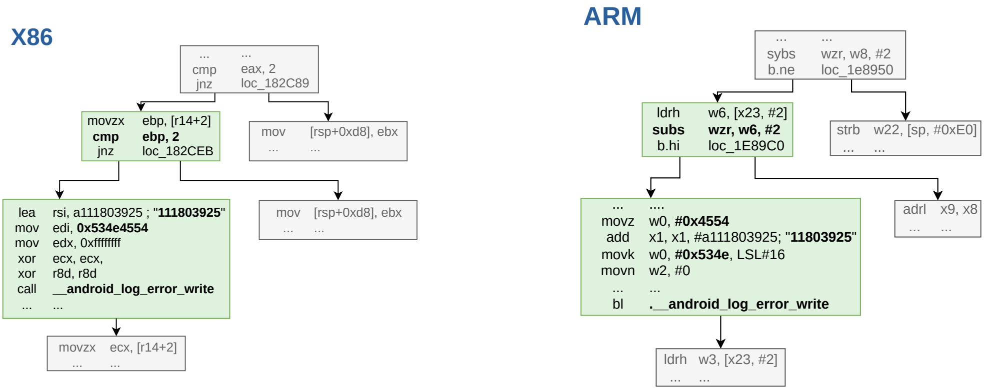
- Function Name
- Index
- Strings
- ...

Matching

- Strings
- Constants
- Calls
- Conditions

Illustration: CVE-2018-9506

CVE-2018-9506 fixes an out-of-bound read in Android's Bluetooth stack



Function Features Detailed

Strings

Common characteristic easily identifiable in binary code

Usage:

- Debug/Log message
- Interface building
- ...

Detection Algorithm

Straightforward from the disassembly

Constants

Immediate values used by the binary code

Usage:

- Computations
- Memory manipulation
- ...

Detection Algorithm

Look at each constant occurrence count in both versions of the function

Function Features Detailed

Strings

Common characteristic easily identifiable in binary code

Usage:

- Debug/Log message
- Interface building
- ...

Detection Algorithm

Straightforward from the disassembly

Constants

Immediate values used by the binary code

Usage:

- Computations
- Memory manipulation
- ...

Detection Algorithm

Look at each constant occurrence count in both versions of the function

Function Features Detailed

Calls

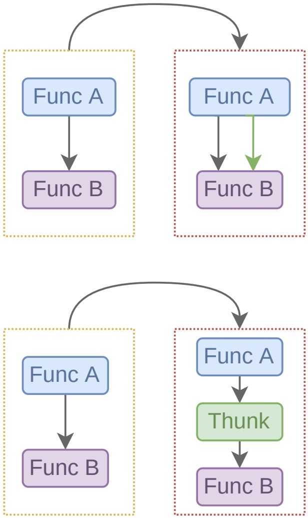
Interfunction flows within the program.

Call Graph Recovery Challenges

- Function boundaries
- Sources / destinations functions
- Inlining

Detection Algorithm

Uses the function degrees and the number of calls within the caller.



Function Features Detailed

New conditions are present in 42% of patches

Conditions

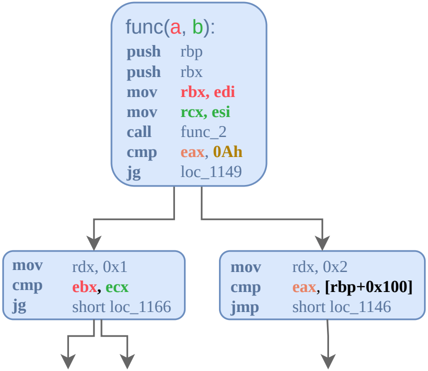
Compare values to determine the control flow

The **origin** of compared terms is a semantic invariant.

Terms origin:

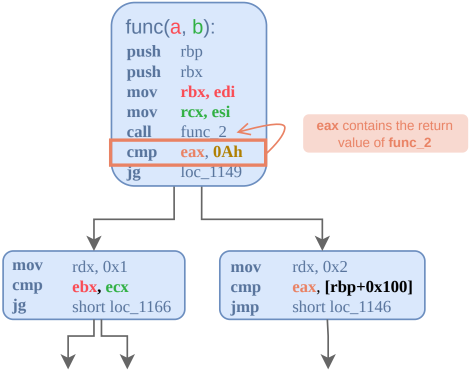
- Constant value *counting the program arguments*
- Call return value *checking the return code*
- Function argument
- Unknown *if no other origin has been identified*

Origin Tracking Tainting Algorithm



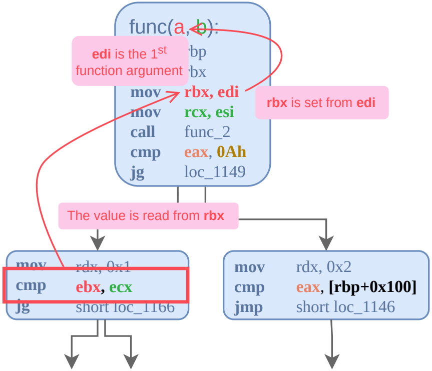
Origin Tracking Tainting Algorithm

- `cmp eax, 0xA`
  Compare the return value of `func_2` with an immediate



Origin Tracking Tainting Algorithm

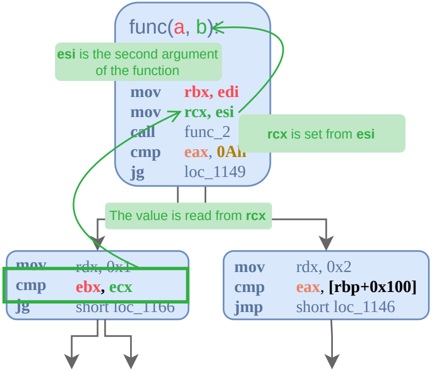
- `cmp eax, 0xA`
  Compare the return value of `func_2` with an immediate



Origin Tracking Tainting Algorithm

- `cmp eax, 0xA`
  Compare the return value of `func_2` with an immediate

- `cmp ebx, ecx`
  Compare the first two arguments of the function

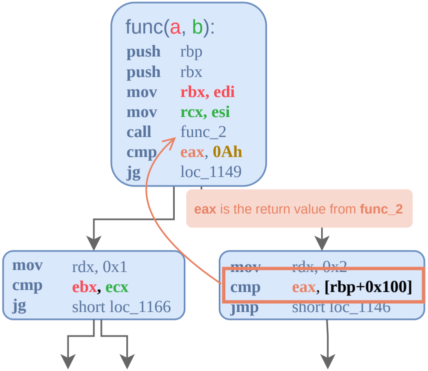


Origin Tracking Tainting Algorithm

- `cmp eax, 0xA`
  Compare the return value of `func_2` with an immediate

- `cmp ebx, ecx`
  Compare the first two arguments of the function

- `cmp eax, [rbp+0x100]`
  Compare a return value with an unknown memory cell



Abstract Interpretation using BinCAT

Abstract Interpretation is an **dataflow** analysis to compute semantic invariants over the program.

To recover the terms origin, use an **abstract interpretation** framework: *BinCAT*.

Tainting Domain *already implemented by BinCAT*

- U is `Untainted`
- S **of** T *set of possible tainting sources*
- $\perp$ is *bottom*
- $\top$ is *top*

Abstract Interpretation using BinCAT

Abstract Interpretation is an **dataflow** analysis to compute semantic invariants over the program.

To recover the terms origin, use an **abstract interpretation** framework:

Relaxations

implemented using BinCAT mechanisms

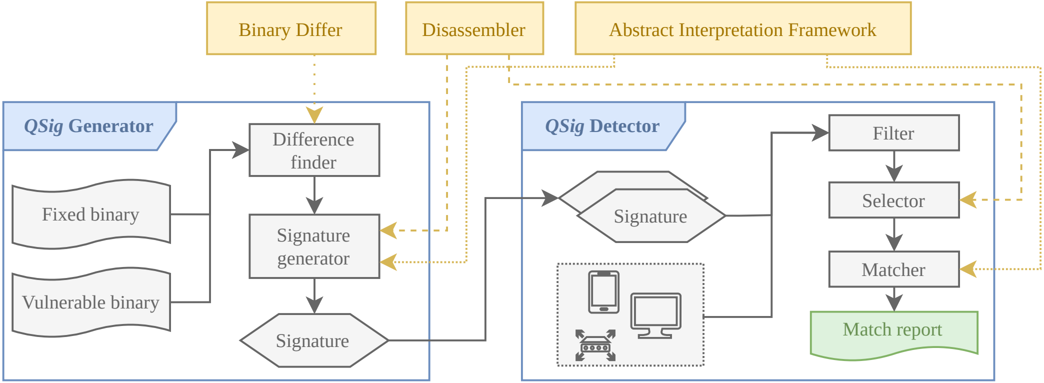
- Skip function calls
- Widen state instead of following backwards edges
- Silently ignore unknown instructions

Tainting Domain a

- U is `Untainted`
- S of T *set of possi*
- $\bot$ is *bottom*
- $\top$ is *top*

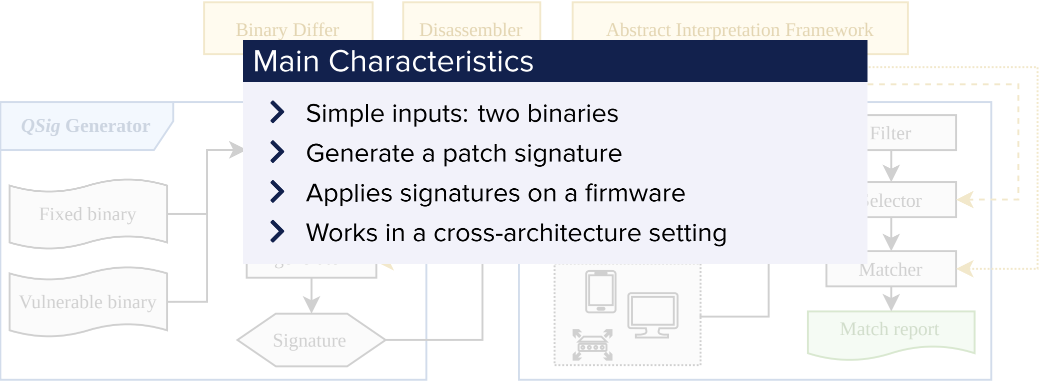
QSig Summary

An implementation solving the FMP



QSig Summary

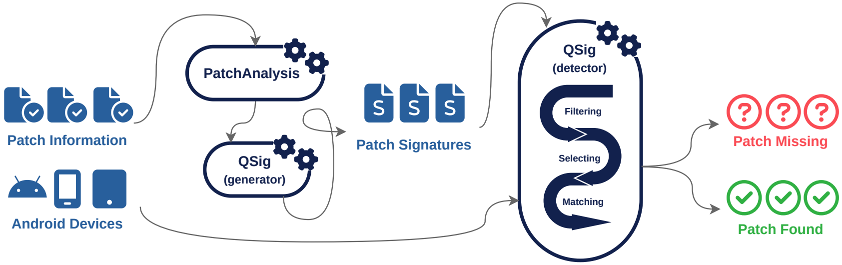
An implementation solving the FMP



Main Characteristics

- Simple inputs: two binaries
- Generate a patch signature
- Applies signatures on a firmware
- Works in a cross-architecture setting

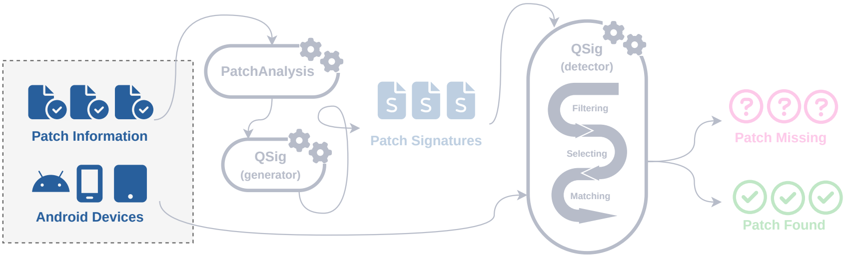
QSig: Summary



Outline

Chapter 3:

Commit-Level Precise Dataset



Research Question

QSig is an implementation of the **FSM**.

? How to test it?

- Compare it against state of the arts approaches
- Evaluate it in real-world scenarios

State of the Art

To evaluate and test new techniques

Standard Test Suites

Test Suites composed of hand-crafted bugs

- ✓ Includes every problem
- ✓ Ground truth known from start
- ✗ Limited by author's knowledge

Example: Juliet [BB12]

Synthetic Datasets

Inject/Craft known bugs in real-world programs

- ✓ Uses legitimate and complex programs
- ✓ Every bug is triggerable
- ✗ Types of bugs are limited

Example: LAVA [Dol+16], MAGMA [HHP20]

From Vulnerabilities

Starts from a list of vulnerabilities

- ✓ Language agnostic
- ✓ Real vulnerabilities in real-programs
- ✗ CVEs data needs to be curated

Example: CVEFIXES [BNM21]

State of the Art

To evaluate and test new techniques

Standard Test Suites

Test Suites composed of hand-crafted bugs

- ✅ Includes every problem
- ✅ Ground truth known from start
- ❌ Limited by author's knowledge

Example: Juliet [BB12]

Synthetic Datasets

Inject/Craft known bugs in real-world programs

- ✅ Uses legitimate and complex programs
- ✅ Every bug is triggerable
- ❌ Types of bugs are limited

Example: LAVA [Dol+16], MAGMA [HHP20]

From Vulnerabilities

Starts from a list of vulnerabilities

- ✅ Language agnostic
- ✅ Real vulnerabilities in real-programs
- ❌ CVEs data needs to be curated

Example: CVEFIXES [BNM21]

State of the Art

To evaluate and test new techniques

Standard Test Suites

Test Suites composed of hand-crafted bugs

- ✓ Includes every problem
- ✓ Ground truth known from start
- ✗ Limited by author's knowledge

Example: Juliet [BB12]

Synthetic Datasets

Inject/Craft known bugs in real-world programs

- ✓ Uses legitimate and complex programs
- ✓ Every bug is triggerable
- ✗ Types of bugs are limited

Example: LAVA [Dol+16], MAGMA [HHP20]

From Vulnerabilities

Starts from a list of vulnerabilities

- ✓ Language agnostic
- ✓ Real vulnerabilities in real-programs
- ✗ CVEs data needs to be curated

Example: CVEFIXES [BNM21]

State of the Art

To evaluate and test new techniques

Standard Test Suites

Test Suites composed of hand-crafted bugs

- ✓ Includes every problem
- ✓ Ground truth know from start
- ✗ Limited by author's knowledge

Example: Juliet [BB12]

Synthetic Datasets

Inject/Craft known bugs in real-world programs

- ✗ Types of bugs are limited

Example: LAVA [Dol+16], MAGMA [HHP20]

From Vulnerabilities

Starts from a list of vulnerabilities

- age agnostic
- ulnerabilities in ograms
- data needs to be curated

Example: CVEFIXES [BNM21]

Ideal Solution

- ✓ Based on a real codebase
- ✓ Composed of real vulnerabilities
- ✓ Maintained over time

Vulnerability Dataset based on AOSP

Rationale of Using AOSP Vulnerabilities for a Dataset

- Heart of a complete Operating System *every vulnerabilities are related*
- Real-word software *billions of users*
- Representative of real problems *found by researchers*
- Always up-to-date *system is actively developped*

Dataset Building

Creating a Dataset from Android Security Bulletins

Android Security Bulletins

- Published monthly
- Contain the list of vulnerabilities fixed by the update
- And a link towards the **fixing commit**

Enables to build a dataset of vulnerabilities precise at the **commit level** *implemented in a tool named Roy*.

Dataset Building

Creating a Dataset from Android Security Bulletins

Android Security Bulletins

- Published monthly
- Contain the list of vulnerabilities fixed by the update
- And a link towards the **fixing commit**

Enables to build a dataset of vulnerabilities precise at the **commit level** *implemented in a tool named Roy*.

Dataset Building

Creating a Dataset from Android Security Bulletins

Android Security Bulletins

- Published monthl
- Contain the list of
- And a link toward

Results

- Huge set of vulnerabilities *$\geq$ 3400 and $\geq$ 1900 with commits*
- Ever increasing *but parser often needs to be updated*

Enables to build a dataset of vulnerabilities precise at the **commit level** *implemented in a tool named Roy*.

Binary Artifacts

To work with **binary only** methods, **binary artifacts** are required.

Binary Artifacts

To work with **binary only** methods, **binary artifacts** are required.

Solution

Using AOSP Build System to compile a project in two versions:

- Vulnerable *before the application of the fixing commit*
- Fixed *after its application*

Binary artifacts obtained differ by **exactly** the patch.

Limitations & Results

Results of **AOSPBuilder**

- $\approx$ 700 vulnerabilities compiled
- From 2012 to 2021
- Targeting 4 architectures (`x86`, `x86_64`, `arm`, `arm64`)

Limitations

- Targets only vulnerabilities on native code
- Build automation is challenging *lot of failures*
- Only vulnerabilities after Android 6

Dataset Usage

How this dataset can be leveraged in various security workflows?

- Silent Fix Detection *detect if a commit fixes a security issue*
- (Cross-architecture) Binary Diffing *uncover the difference between two binaries*
- Decompilation *train algorithms to recover source from binary*

Open-source and available on
`https://github.com/quarkslab/aosp_dataset`

Dataset Usage

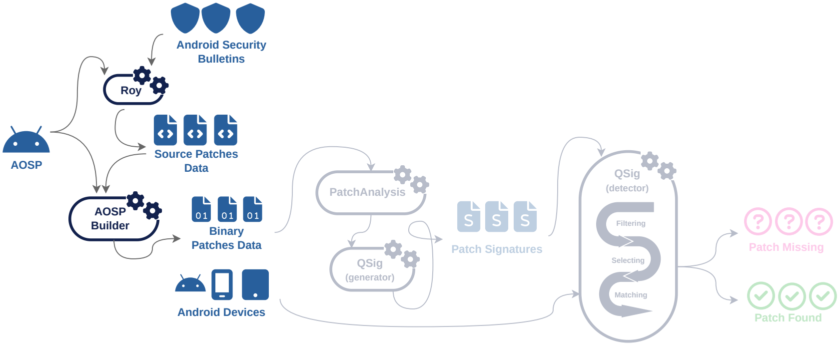
How this dataset can be leveraged in various security workflows?

- Silent Fix Detection *detect if a commit fixes a security issue*
- (Cross-architecture) Binary Diffing *uncover the difference between two binaries*
- Decompilation *train algorithms to recover source from binary*
- **Patch Characterization** *identify patches key components*
- **Patch Detection** *check whether patches have been applied*

Open-source and available on

`https://github.com/quarkslab/aosp_dataset`

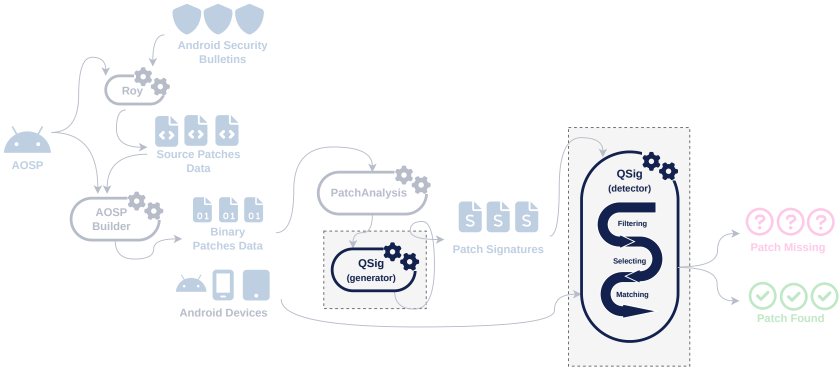
Dataset: Summary



Outline

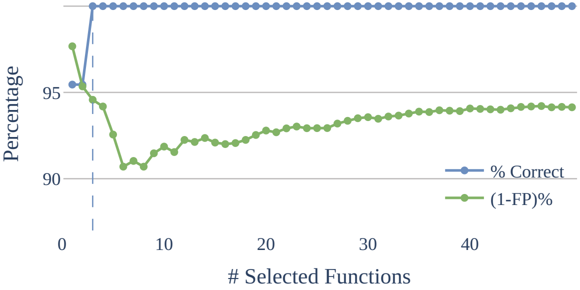
Chapter 4:

Patch Detection Evaluation



Selector Parameters

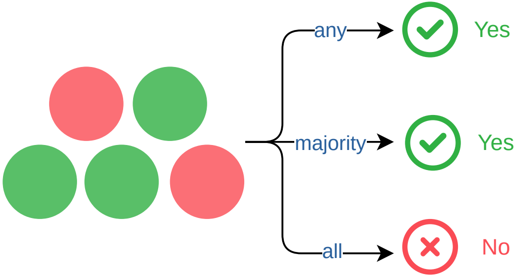
How many functions to select within the binary?



Choose set $n = 3$ for the experiments *because it is the best compromise*

Matcher Parameters

How to arbitrate between **inconsistent** results?



Matcher Parameters

How to arbitrate between **inconsistent** results?

Function	Valid	Invalid	Success Rate
all	7	19	27%
majority	21	5	81%
any	25	1	96%

Matcher Parameters

How to arbitrate between **inconsistent** results?

Function	Valid	Invalid	Success Rate
all	7	19	27%
majority	21	5	81%
any	25	1	96%

Our features are challenging to detect but the presence of at least **one of them** is a sign of the patch presence.

Datasets used for the experiments

*Demonstrating **QSig** versatility using several datasets*

Dataset 1: CGC

Binaries from the DARPA contest and adapted to a regular OS

- A vulnerable binary
- A fixed one
- A Proof of Vulnerability

Dataset 2: Debian 9 ISO

Directly from the official website

- About 5,500 binaries
- 5 CVEs *from QuickBCC [Jan+21]*

Dataset 3: Pixel 4 image

Downloaded from Google's website and flashable

- 3,400 binaries
- 6 CVEs from Oct to March 2019-2020
- 20 CVEs around January 2020

Generating Patch Signatures

On AOSP's compiled CVEs

Architecture	CVE Signatures	Functions	Success Rate
X86	377	1072	61%
X64	371	1273	60%
ARM	401	1069	65%
ARM64	339	938	55%
Union	459	1652	74%

Signature Generation

Generating Patch Signatures

On AOSP's compiled CVEs

Architecture	CVE Signatures	Functions	Success Rate
X86			
X64			
ARM			
ARM64	339	938	55%
Union	459	1652	74%

Signature Generation

Conclusion

- QSig success rate remains stable across architectures
- Our features are sufficient to sign most patches

On Dataset 1 CGC

	Correct	Incorrect	Success Percentage
Patched functions	250	3	99%
Vulnerable functions	212	41	84%
Total	462	44	91%

QSig's Accuracy

Pertinence of a Static-Only Approach

1dVul [Pen+19] uses a hybrid approach

	Total	1dVul	**QSig**	Increase
Changed functions	348	209	253	+21%
Patch detected	348	130	250	+92%

Comparison of **QSig** and 1dVul on Dataset 1 CGC

Configuring a hybrid environment is **challenging** for real-world contexts and does **not yield** to better results.

Matching Results

From x64 signatures to aarch64 binaries from a Pixel 4 image

Feature	TP	TN	FP	FN	Pr.	Rec.	N/A
Strings	21	12	-	1	1	0.95	14
Constants	13	3	-	1	1	0.93	31
Calls	2	6	3	23	0.40	0.08	14
Conditions	2	4	-	4	1	0.33	38
QSig	21	13	5	9	0.81	0.70	-

TP: True Positive TN: True Negative FP: False Positive FN: False Negative Pr: Precision Rec: Recall

Matching Results

From x64 signatures to aarch64 binaries from a Pixel 4 image

Feature	TP	TN	FP	FN	Pr.	Rec.	N/A
Strings	21	12	-	1	1	0.95	14
Constants	13	3	-	1	1	0.93	31
Calls	2	6	3	23	0.40	0.08	14
Conditions	2	4	-	4	1	0.33	38
QSig	21	13	5	9	0.81	0.70	-

TP: True Positive TN: True Negative FP: False Positive FN: False Negative Pr: Precision Rec: Recall

- The call precision/recall is poor

Matching Results

From x64 signatures to aarch64 binaries from a Pixel 4 image

Feature	TP	TN	FP	FN	Pr.	Rec.	N/A
Strings	21	12	-	1	1	0.95	14
Constants	13	3	-	1	1	0.93	31
Calls	2	6	3	23	0.40	0.08	14
Conditions	2	4	-	4	1	0.33	38
QSig	21	13	5	9	0.81	0.70	-

TP: True Positive TN: True Negative FP: False Positive FN: False Negative Pr: Precision Rec: Recall

- The call precision/recall is poor
- But the overall precision / recall is excellent

Matching against PMatch

	TP	TN	FP	FN	Pr.	Rec.
QSig	21	13	5	9	0.81	0.70
PMatch	4	12	0	26	1.0	0.13

On Dataset 3 Pixel Images

PMatch uses a NLP algorithm to generate a binary code semantic representation.

They have a better precision but a limited recall.

Matching against PMatch

	TP	TN	FP	FN	Pr.	Rec.
QSig	21	13	5	9	0.81	0.70
PMatch	4	12	0	26	1.0	0.13

On Dataset 3 Pixel Images

PMatch uses a NLP algorithm to generate a binary code semantic representation.

They have a **better precision** but a limited recall.

Matching against PMatch

	TP	TN	FP	FN	Pr.	Rec.
QSig	21	13	5	9	0.81	0.70
PMatch	4	12	0	26	1.0	0.13

On Dataset 3 Pixel Images

PMatch uses a NLP algorithm to generate a binary code semantic representation.

They have a better precision but a **limited recall**.

Stability Over Time

*Check if **QSig** produces usable results in a real-life scenario*

	2019			2020		
	Oct.	Nov.	Dec.	Jan.	Feb.	Mar.
CVE–2019–2187	✓	✓	✓	✓	✓	✓
CVE–2019–2202	○	✓	✓	✓	✓	✓
CVE–2019–2220	○	○	✓	✓	✓	✓
CVE–2020–0006	○	○	○	✓	✓	✓
CVE–2020–0018	○	○	○	○	✓	✓
CVE–2020–0037	○	○	○	○	○	✓

Stability Over Time

*Check if **QSig** produces usable results in a real-life scenario*

	2019			2020		
	Oct.	Nov.	Dec.	Jan.	Feb.	Mar.
CVE–2019–2187						✓
CVE–2019–2202						✓
CVE–2019–2220						✓
CVE–2020–0006	○	○	○	✓	✓	✓
CVE–2020–0018	○	○	○	○	✓	✓
CVE–2020–0037	○	○	○	○	○	✓

Conclusion

QSig does not find a patch before its release and **always** finds them after.

QSig's Efficiency

Scan a Debian image for 5 CVEs

	Dry Run		Cached	
	QuickBCC	**QSig**	QuickBCC	**QSig**
Run	8h 53m 34s	3m 09s	3m 24s	2m 11s
Preprocessing	8h 53m 19s	1m 9s	1m 34s	8s
Matching (s)	15	108	110	117

On Dataset 2 Debian Live ISO

QSig's Efficiency

Scan a Debian image for 5 CVEs

	Dry Run	Cached
	Q	QSig
Run	8h	2m 11s
Preprocessing	8h	8s
Matching (s)		117

On Dataset 2 Debian Live ISO

Conclusion

- Half the time is taken by the disassembler
- Caching helps tremendously *17,000% improvment*
- **QSig** is **fast** thanks to the FSM

Limitations

- **Adversarial Transformations**
- Tainting Algorithm
- Patch Completeness

Issue

Changes specifically targeted against features used by **QSig** completely defeat the tool

Potential Solution

- Add other features types (I/O behavior)
- Consider this problem *out of scope*

Limitations

- Adversarial Transformations
- **Tainting Algorithm**
- Patch Completeness

Issue

The tainting algorithm does not follow calls

Potential Solution

Create *stub* library to modelize classic function calls

Limitations

- Adversarial Transformations
- Tainting Algorithm
- **Patch Completeness**

Issue

Checking the patch presence is not sufficient to assert the **vulnerable status** of a device

Potential Solution

Combine **QSig** with dynamic approaches using *Proof of Vulnerabilities*

Patch Detection Evaluation Summary

QSig is a versatile solution to search **vulnerability patches** inside complete file systems.

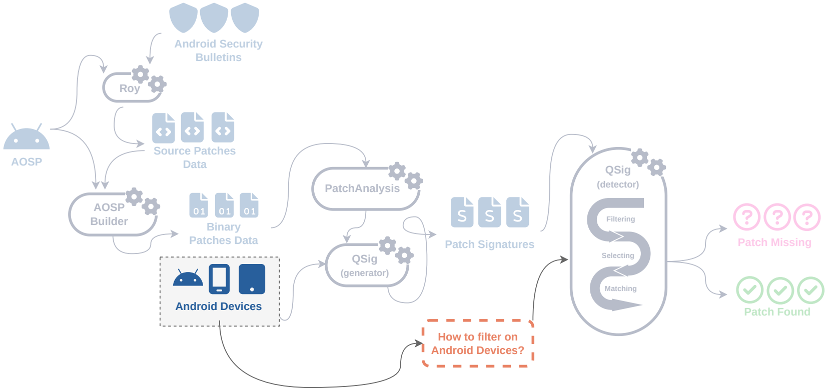
- The **Filtering-Selecting-Matching** strategy is well suited and extensible to solve the Firmware Matching Problem
- **QSig** is fast *successively pruning the search space*
- **QSig** correctly signs the **patch semantic** *manages to do cross-architecture matching*

Open-sourced and available on
`https://github.com/quarkslab/qsig`

Outline

Chapter 5:

Build Dependency Graphs



Static Libraries and Vulnerabilities

Reusing code from other people in binaries is possible using:

- Dynamic linking *resolved at runtime*
- Static linking *resolved at compile time*

Static Libraries and Vulnerabilities

Reusing code from other people in binaries is possible using:

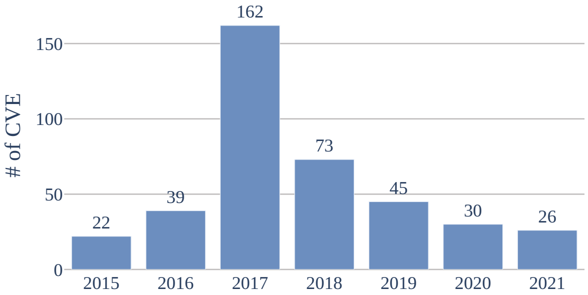
- Dynamic linking *resolved at runtime*
- Static linking *resolved at compile time*

In Android 11, over **52%** of binary targets include a statically linked library.

Static Vulnerabilities

Definition

A **static vulnerability** is a vulnerability affecting a library that will be statically embedded.

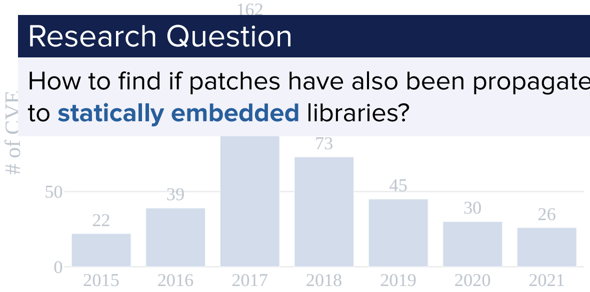


Vulnerabilities affecting static libraries propagate through code bases.

Static Vulnerabilities

Definition

A **static vulnerability** is a vulnerability affecting a library that will be statically embedded.



Research Question

How to find if patches have also been propagated to **statically embedded** libraries?

Vulnerabilities affecting static libraries propagate through code bases.

Unified Dependency Graph

Definition

An UDG is a directed graph $\mathrm{UDG} = (V, E)$ where V is the set of nodes and E is the set of edges.

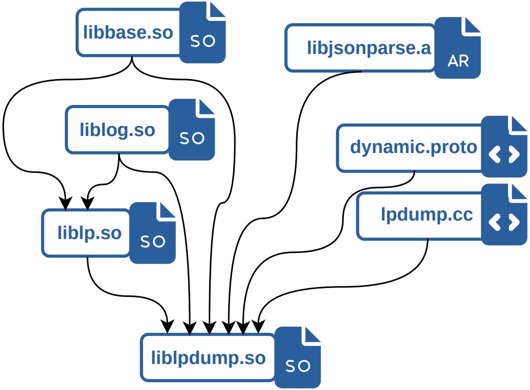
- $V = V_T \sqcup V_F$ with V_T target node set and V_F is the file node set
- The edges represent the different dependency links between nodes

Solution

Create a UDG for AOSP to perform the **filtering**

UDG for Soong

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



Extract of a Soong module and its associated UDG

BGraph: UDG for AOSP

Tools

BGraph generates Build Graphs from AOSP build system

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

Potential Usages

- Given a source file, what are the (build) targets dependent?
- Given a target, what are the source files affected?

Patches in Static Libraries

QSig + BGraph = ♡

Using BGraph, write a new **Filtering** pass for **QSig**.

Key Benefits

- Fast *only a query in a graph*
- Sound *the UDG precisely describes the dependencies*

Results: Static Vulnerabilities

84 vulnerabilities: 35 anterior and 49 posterior

Feature	TP	TN	FP	FN	Pr.	Rec.	NA
Strings	19	60	-	3	1	0.86	48
Constants	25	64	2	8	0.93	0.76	31
Calls	9	61	6	29	0.60	0.24	25
Conditions	6	15	1	-	0.86	1	108
Match	35	70	5	20	0.88	0.64	-

Detection in Static Libraries

Results: Static Vulnerabilities

84 vulnerabilities: 35 anterior and 49 posterior

Feature	TP	TN	FP	FN	Pr.	Rec.	NA
Strings	19	60	-	3	1	0.86	48
Constants	25	64	2	8	0.93	0.76	31
Calls	9	61	6	29	0.60	0.24	25
Conditions	6	15	1	-	0.86	1	108
Match	35	70	5	20	0.88	0.64	-

Detection in Static Libraries

- Few **False Positive**

Results: Static Vulnerabilities

84 vulnerabilities: 35 anterior and 49 posterior

Feature	TP	TN	FP	FN	Pr.	Rec.	NA
Strings	19	60	-	3	1	0.86	48
Constants	25	64	2	8	0.93	0.76	31
Calls	9	61	6	29	0.60	0.24	25
Conditions	6	15	1	-	0.86	1	108
Match	35	70	5	20	0.88	0.64	-

Detection in Static Libraries

- Few **False Positive**
- Good **Precision** and **Recall** overall

Results: Static Vulnerabilities

84 vulnerabilities: 35 anterior and 49 posterior

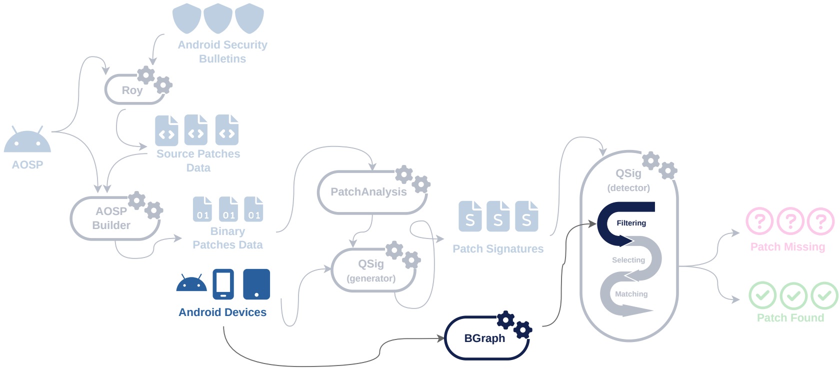
Feature	TP	TN	FP	FN	Pr.	Rec.	NA
Strings	19	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	48
Constants	25	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	31
Calls	9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	25
Conditions	6	15	1	-	0.86	1	108
Match	35	70	5	20	0.88	0.64	-

Detection in Static Libraries

Conclusion

QSig is able to detect **patches** in statically linked libraries.

Build Dependency Graph: Summary



Conclusion: Contributions

General conclusion about contributions provided by this thesis

Practical approaches to detect patches in binary code.

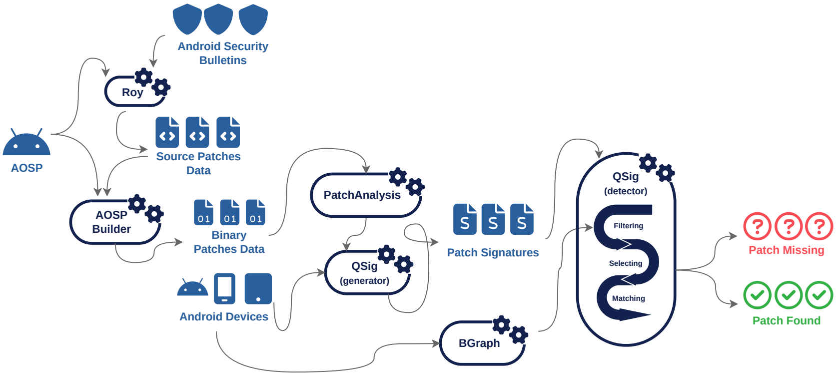
Formalize the **Firmware Matching Problem**

Introduce the **Filtering-Selecting-Matching** strategy

Extensively test its application in **QSig** with a large **dataset**

Extend it by using **Build Graphs** as a filtering step for Android phones

Work: Summary



Research Perspectives

Possible challenges to tackle with a few more years

- Extend to other **contexts**
  Raw Firmwares, Real-Time systems, Windows, ...
- Understand a patch **validity**
  How to be confident that a patch correctly fixes a vulnerability?
- Encode patch presence requests as **semantic queries**
  Using binary-code representation

Industrial Perspectives

How to apply these contributions in industrial contexts?

National Defense Authorization Act (2023) [Ada22]

A certification that each item listed on the submitted bill of materials is free from all known vulnerabilities or defects affecting the security of the end product or service [...]

Industrial Perspectives

How to apply these contributions in industrial contexts?

National Defense Authorization Act (2023) [Ada22]

A certification that each item listed on the submitted bill of materials is free from all known vulnerabilities or defects affecting the security of the end product or service [...]

In other contexts:

- Secure the Supply Chain *SBOM, FBOM*
- Improve audit efficiency
- Gain the knowledge of residual risks in installed fleets

Conclusion: Publications

2021

Exploitation du graphe de dépendance d'AOSP à des fins de sécurité
SSTIC'21

2022

Building a Commit-level Dataset of Real-world Vulnerabilities
CODASPY'22

Quokka: A Fast and Accurate Binary Exporter
GreHack'22

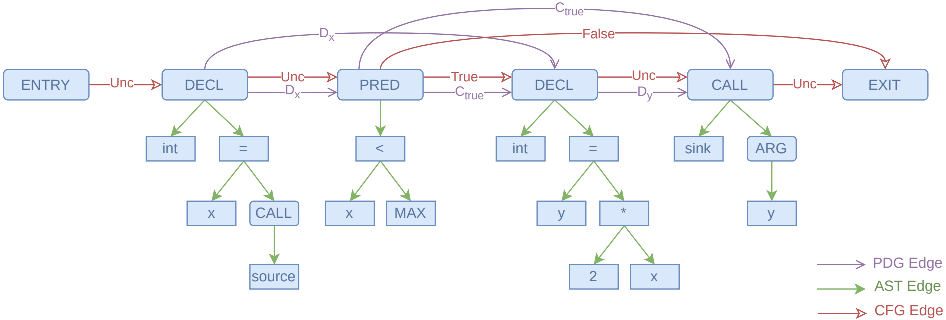
Towards Patch Detection using Binary Only Semantic Signatures
to be submitted

Thank you for your attention

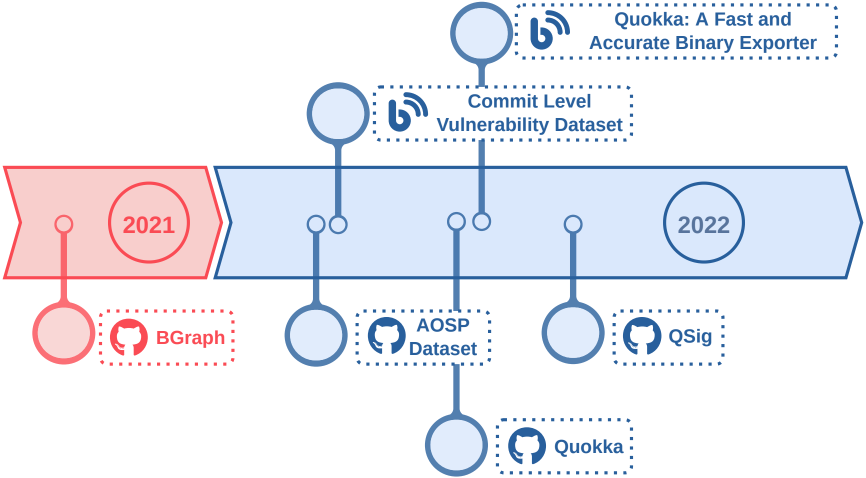
Quarkslab

Code Property Graph

From Yamaguchi et al. [Yam+14]



Tools & Other Publications



Precision / Recall

Some measures used in this presentation

- **Precision**
  Precision is a measure of how many of the positive predictions made are correct

$$Precision = \frac{TP}{TN + FP}$$

- **Recall**
  Recall is a measure of how many of the positive cases the classifier correctly predicted

$$Recall = \frac{TP}{TN + FN}$$

- **F1-Score**
  F1-Score is a measure combining both precision and recall.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Memcpy: Taint Propagation

memcpy Signature

```
void *memcpy(void *dest, const void * src, size_t n)
```

An ideal taint propagation system would also copy the taint of the first **n** bytes of **src** to **dest**.

Limitations

QSig current taint system does propagate the taint.

Abstract Interpretation 101

Abstract interpretation is a theory of sound approximation of the semantics of computer programs.

Problems

How to arbitrate betweeen **decidability** and **tractability**?

Domains

An **abstract domain** is a complete lattice *a set of elements ordonned by a partial order*

Standards domains

- Sign
- Intervals

BinCAT modifications

- Some armv7 and armv8A instructions support
- Added fake sections support to allow dereferencing memory from argument
- Added a `cfgTable` config to resolve dynamic jumps (switch) with IDA information
- Added a `Failed_decoding` exception to continue the execution even if the decoding fails

AOSPBuilder: Compilation figures

Machine

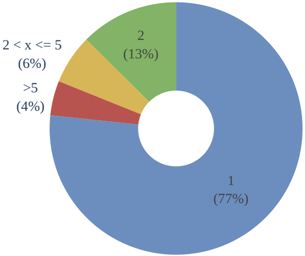
The compilation was performed on a server *thanks INRIA*
AMD Opteron 63xx class CPU - **56 cores** with **120 Gb** of RAM

Key Figures:

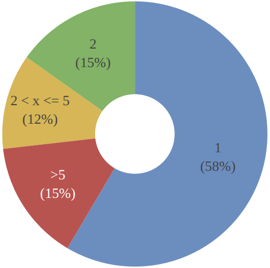
- 750 archives *success*
- About 30 min / compilation / architecture *when it works*
- Assume it takes 5 min when it fails

Rough Estimate: about **1,200** hours of compilation

Patch Analysis Figures



Number of files affected by a patch



Number of functions affected by a patch

CVE Count

Why do we use so few CVEs in our tests?

- When replicating the results of others, we use the same dataset.
- For Pixel image, we need to check **manually** every result, in the binary, which is time consuming.

PDG

Formalized by Ferrante [Fer87]

Definition

The PDG represents a program as a graph in which the nodes are statements and predicate expressions (or operators and operands) and the edges incident to a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends.

To compute the PDG, compute the **post dominator** tree[2]

[2]Some examples on how to compute it in
`https://github.com/cea-sec/miasm/blob/master/miasm/analysis/ssa.py`

Quokka

Quokka is a Fast and Accurate Binary Exporter.

Why creating this tool?

- Untie the dependency of analysis and the disassembler
- Fast and efficient storage capabilities
- May be used in other projects *firmware manipulations, machine learning feature extraction*

Open-source and available on `https://github.com/quarkslab/quokka`

BGraph Limitations

- **Build system exhaustivity**
- Incomplete blueprint support
- Work only for Soong

Issue

BGraph relies on *Soong*'s exhaustivity in AOSP. However, the transition from *Android.mk* is not finished.

Potential Solution

Wait until the migration is completed

BGraph Limitations

- Build system exhaustivity
- **Incomplete blueprint support**
- Work only for Soong

Issue

The parsing of blueprints is incomplete

Potential Solution

- Additional engineering efforts
- Reuse the parser developed by Google directly

BGraph Limitations

- Build system exhaustivity
- Incomplete blueprint support
- **Work only for Soong**

Issue

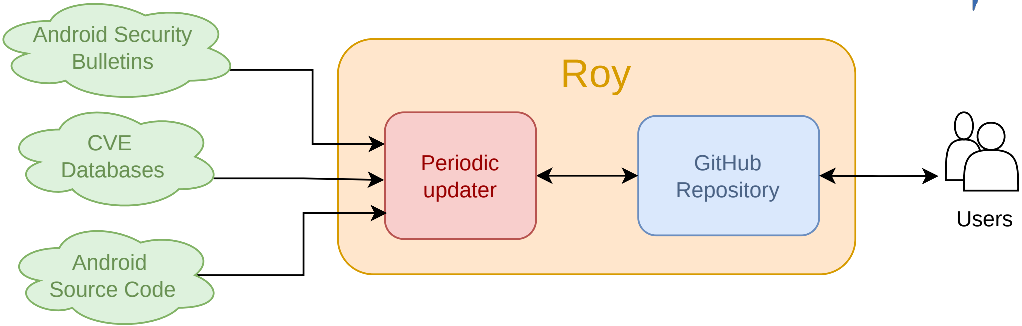
Soong is only used in AOSP, limiting the approach applicability.

Potential Solution

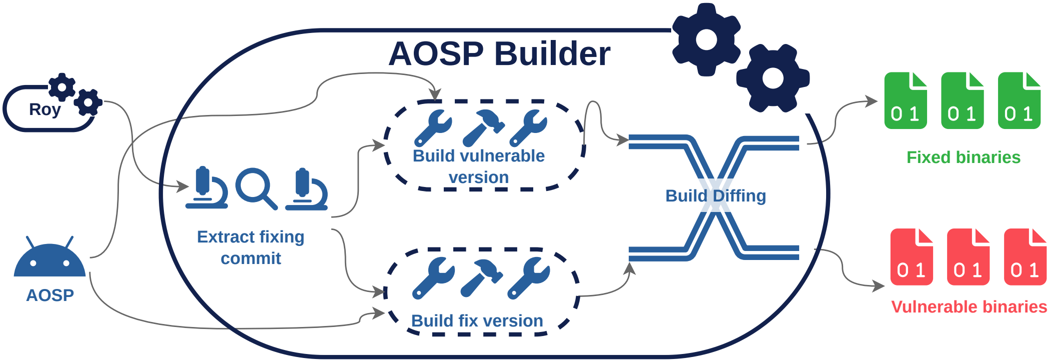
Combining BGraph with other approaches *working for other build systems* but this is challenging as BGraph relies on Soong's particularities.

Roy

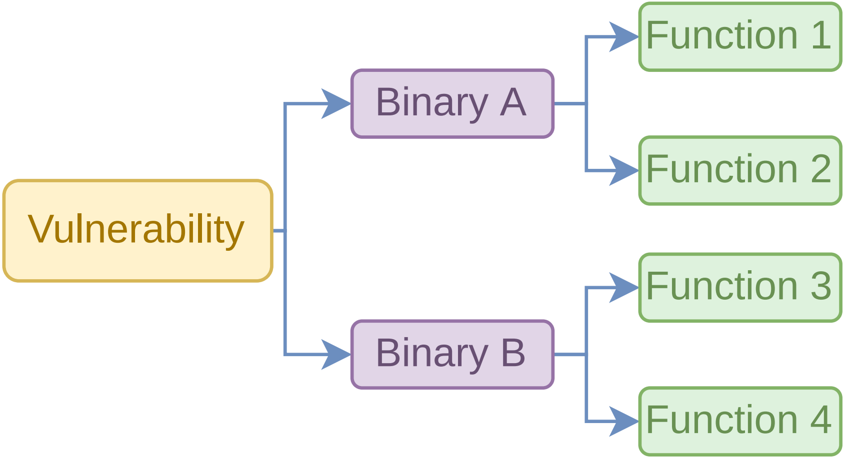
Mining Android Security Bulletins



AOSPBuilder



Patch Anatomy



Binary Diffing

TODO?

Build Graphs

History

Modern softwares and large projects resort using **build systems**.
Driving build systems is done using **build scripts**.

Limitations

Build Scripts are error-prone and most bug stem from **dependency problems**

Build Graphs

Limitations

Build Scripts are error-prone and most bug stem from **dependency problems**

There is a need for solution helping developers to write better build scripts:

Unified Dependency Graph (UDG)

Build Graphs

Limitations

Build Scripts are error-prone and most bug stem from **dependency problems**

There is a need for solution helping developers to write better build scripts:

Unified Dependency Graph (UDG)

Definition

An UDG is a directed graph $\mathrm{UDG} = (V, E)$ where V is the set of nodes and E is the set of edges.

- $V = V_T \sqcup V_F$ with V_T target node set and V_F is the file node set
- The edges represent the different dependency links between nodes

State of the Art

Compilation Database

Types of input required by the solution?

- ✅ Contains the build commands
- ❌ Not an UDG but a JSON file

Example: Clang, GCC

Dynamic Dependency Graph

Instruments the build system operations

- ✅ Usually build-system agnostic
- ❌ Requires a working build system

Example: Licker and Rice [LR19]

State of the Art

Static Dependency Graph

Parses the build scripts to uncover dependencies

- ✓ Works also for incomplete build systems
- ✗ Cannot reason about *missing* dependencies

Example: SYMake [Tam+12]

Hybrid Dependency Graph

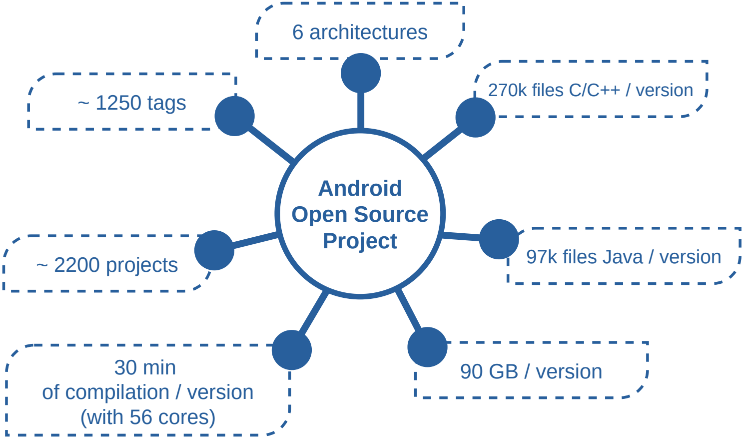
Mixes previous approaches

- ✓ Can reason about discrepencies between actual and declared dependencies.
- ✗ Performs the compilation

Example: VeriBuild [Fan+20]

Android Open Source Project

The heart of Android



Android Build System: Soong

Soong in a nutshell

- Developped by Google for AOSP
- Based on **modules** and **rules**
- Definitions in **Android.bp**

Problem: how to generate an UDG?

- Static approaches do not work for Soong
- Dynamic approaches need to perform the compilation *which takes space and time!*