PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems

Yu-Tsung Lee
Penn State University

William Enck
North Carolina State University

Haining Chen
Google

Hayawardh Vijayakumar
Samsung

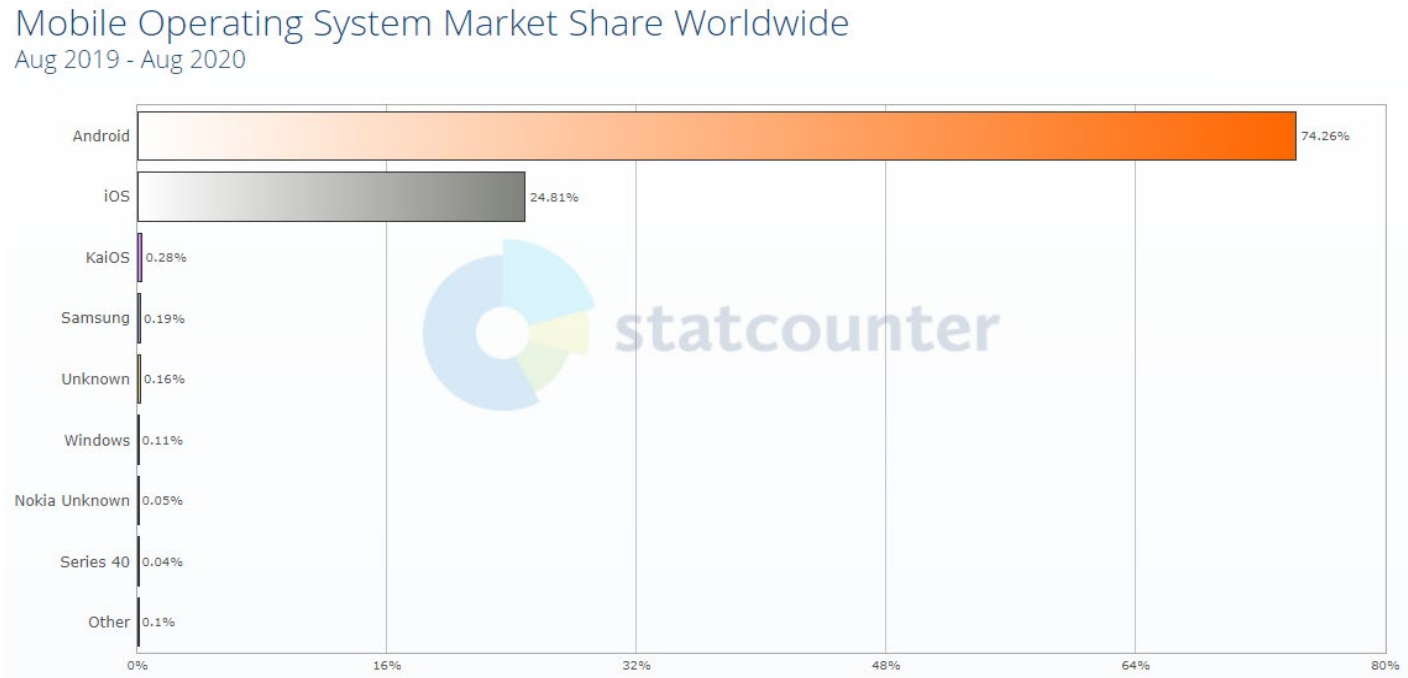
Daimeng Wang, Zhiyun Qian
UC Riverside

Ninghui Li
Purdue University

Giuseppe Petracca
Lyft

Trent Jaeger
Penn State University

Motivations



- Currently the No.1 mobile operating system in the world
- Many different OEMs with different Android implementations

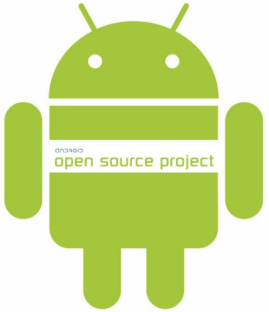
Android Access Control

- 2008: Linux Discretionary Access Control (DAC) only in early days, with Android Permissions
- 2012: SELinux support introduced on Android (4.0)
- 2014: Mandatory Access Control (MAC) fully enforced Android 5.0
- 2015: Upgrade Android permission with runtime permission on Android 6.0
- 2017: Project Treble: privilege separation of system services on Android 8.0
- 2018: MAC Multi-Level Security (MLS) enforcement
- 2020: Scoped storage for external storage control

What can go wrong?

Problems

AOSP



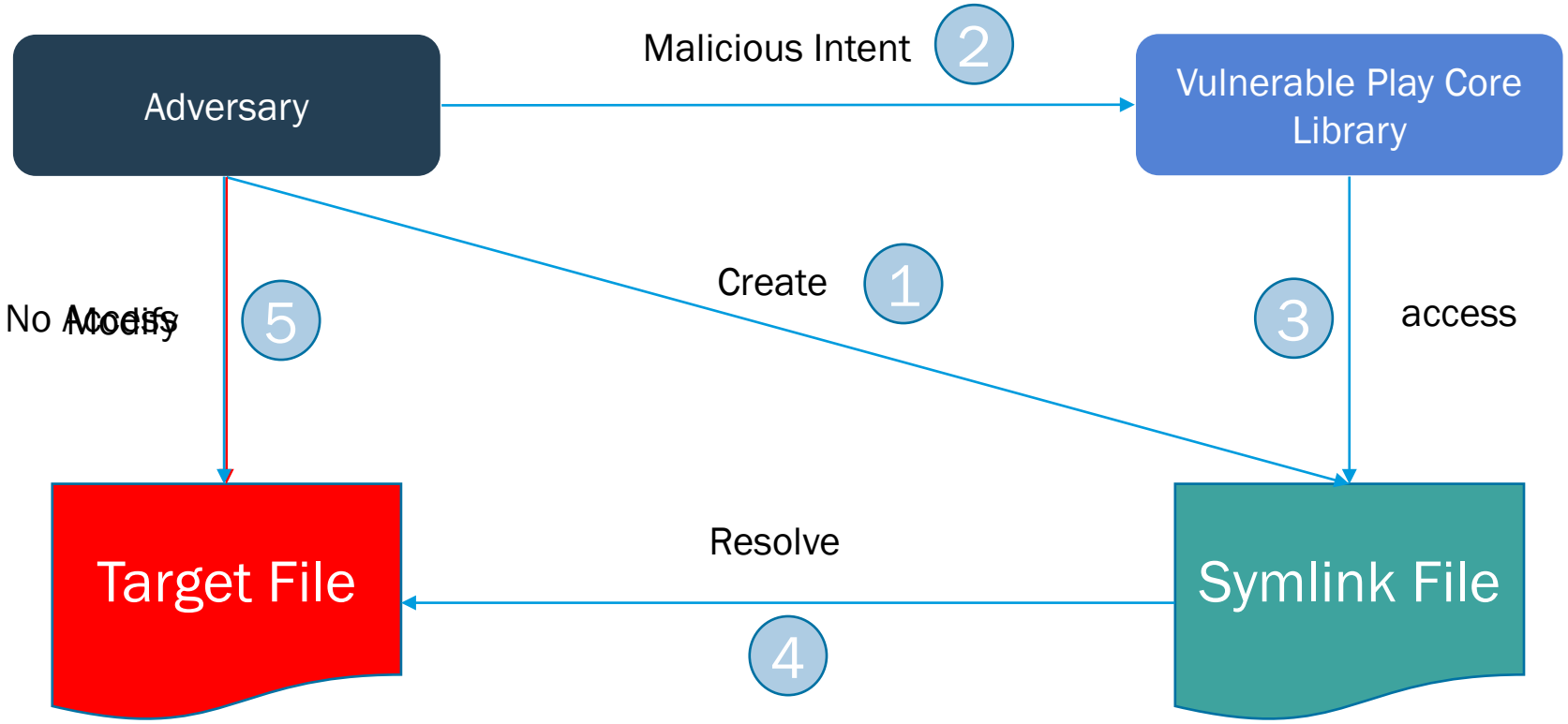
- Very complex access control
- Rapid update and improvements made by Google

Customization

OEM Android

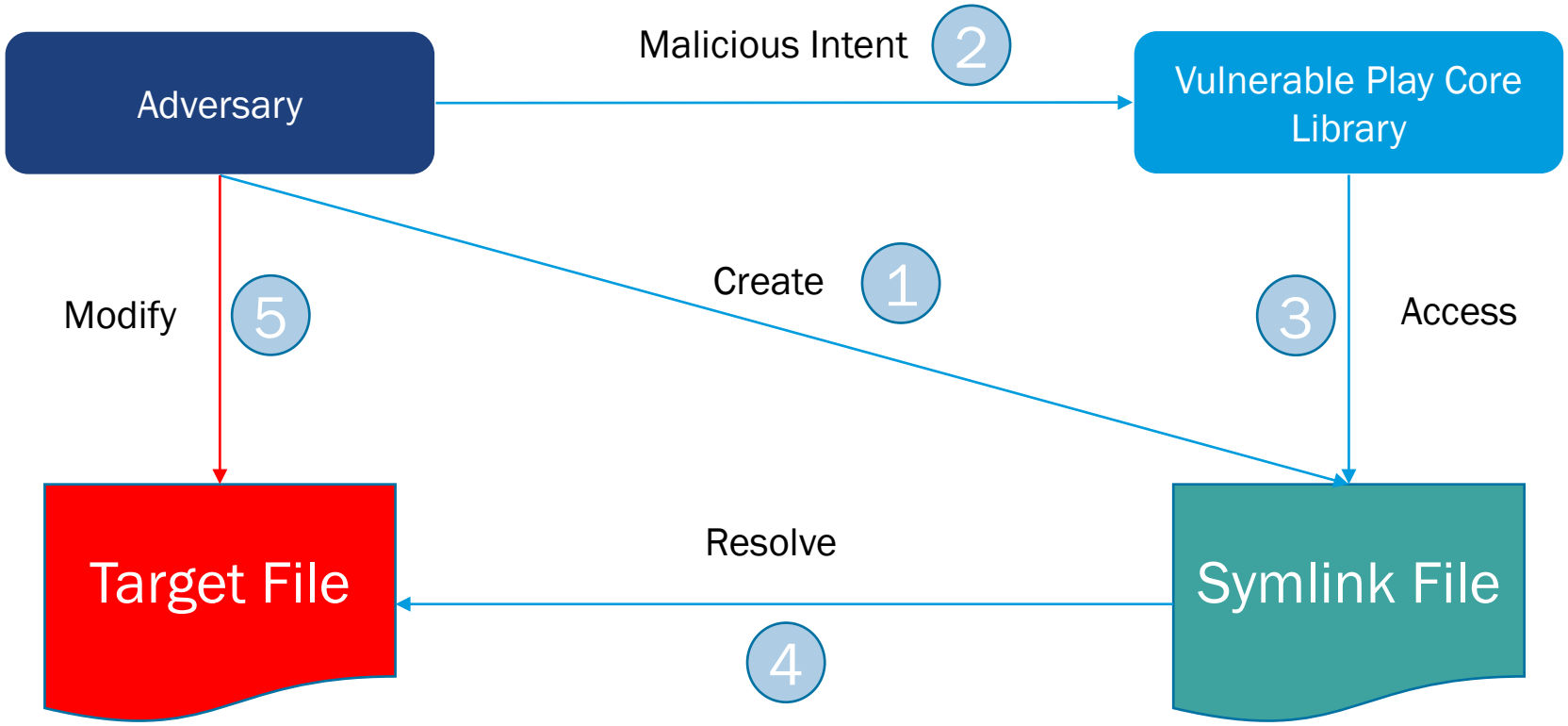
- Extensive access control policy customization for functionality
- Vendor services added to improve value
- Slower to adapt to new security practice

Example Vulnerability – CVE-2020-8931



Link Traversal Attack

Example Vulnerability – CVE-2020-8931



- Access control allows adversary to create symlink
- Victim resolves pathname without sanitization

Automatic Vulnerability Discovery

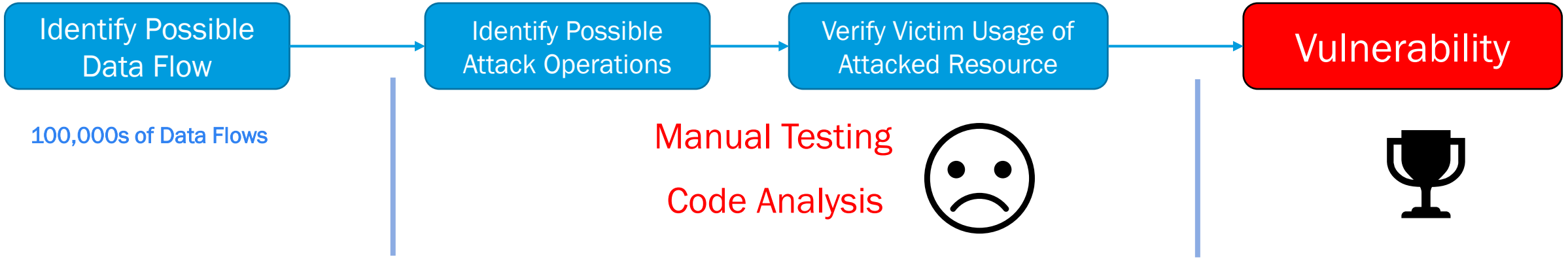
Very hard, why?

- Large number of authorized data flows, and don't know which ones to test
- No systematic way to differentiate adversaries from benign subjects

What can we do?

- **Triage** information flows, highlighting only the risky ones
- Convert information flows into test cases for **attack operations**

Limitation: Too Many Authorized Data Flows



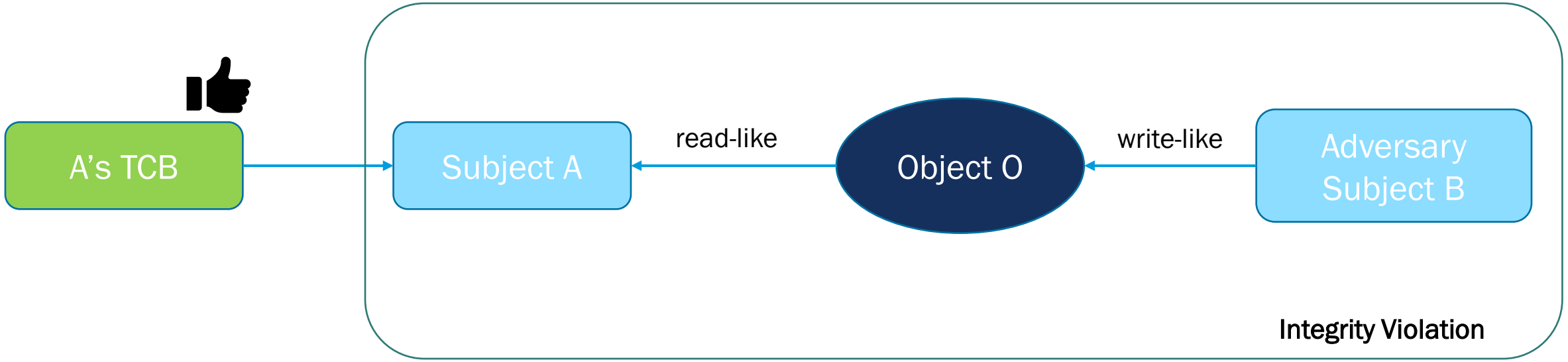
Simply computing information flows is not very useful. Why?

- Simply too much testing needed

What Do We Need To Do?

- Identify meaningful adversaries of each subject
- Only flows that adversaries can impact are relevant

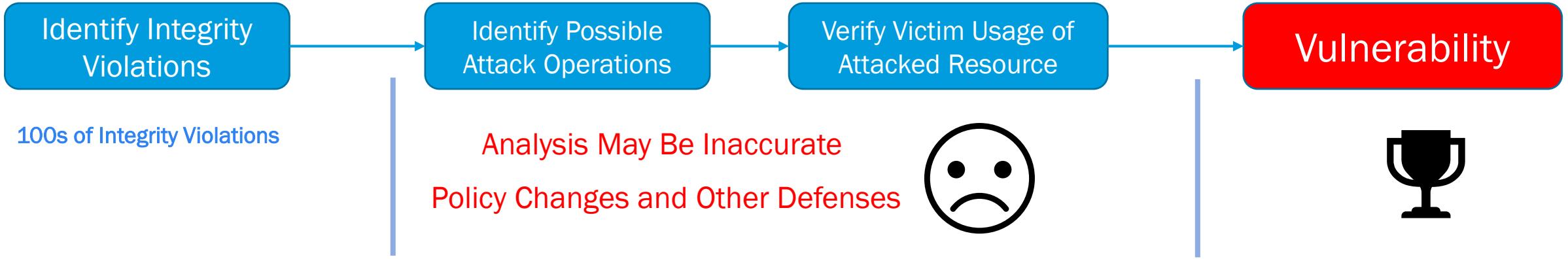
Integrity Violations (IVs)



Integrity Violations

- Adversary modifies a resource that may be used by a potential victim subject
- Adversary Subject B is authorized to perform a write-like operation on object O
- Subject A is authorized to perform a read-like operation on object O
- Object O's use by Subject A is an integrity violation for Subject A

Limitation: Policy Flexibility and Unexploitable Flows



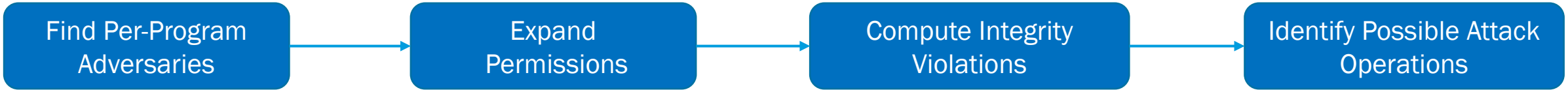
Modifiable Access Control Polices

- Current access control policy only represents a snapshot of the policy
- But some policies can be modified – may miss real attack vectors

Android Configurations May Block Exploitation

- Not all IVs can be exploited
- Program and system configurations may block them

PolyScope Approach



PolyScope Triages Access Control Policies

- Reducing 100,000s policy rules (data flows) to 100s of attack operations
- Which can expose some previously unknown vulnerabilities
- And seed runtime vulnerability testing

PolyScope Solves Key Problems

- Validates per-program adversaries to compute exploitable flows
- Accounts for policy flexibility to compute all “integrity violations”
- Reduces integrity violations to the operations that may be used in attacks

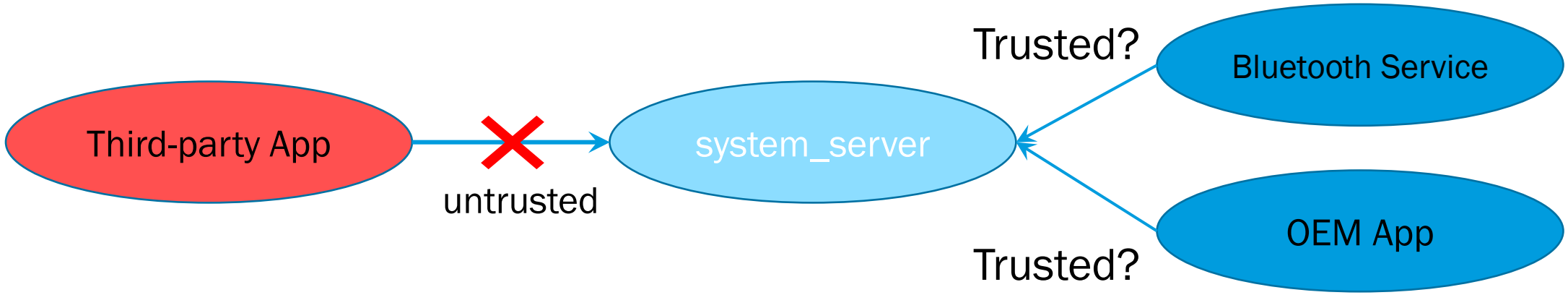
Define Per-Program Adversaries

How previously identified possible adversaries for a subject?

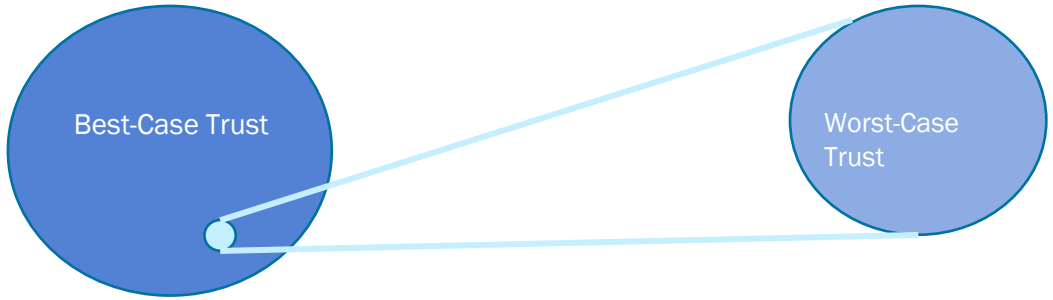
- Guess at trusted subjects: trust core system services (e.g., init) (Jaeger *et al., 2003 USENIX Security*)
- Guess at adversaries: treat Android third-party applications as adversaries (Chen *et al., 2017 ACSAC*)

Above methods are ad-hoc and one-dimensional

- No systematic way to determine whether an adversary is missing or misclassified



Best-Case Trust vs. Worst-Case Trust



PolyScope considers both best-case and worst-case trust

- Worst-case trust: Minimum TCB, only trust subjects that can trivially compromise a subject's program
  Called "Integrity Wall" – Vijayakumar *et al.,* AsiaCCS 2012
- Best-case trust: Based on Google's process privilege level definition

- Worst-case trust must be a **subset** of best-case trust
- Otherwise, Android privilege levels are missing a fundamental trust requirement to prevent trivially compromising the subject
- PolyScope verified both Google and OEMs' access control policy complies with the assumption above

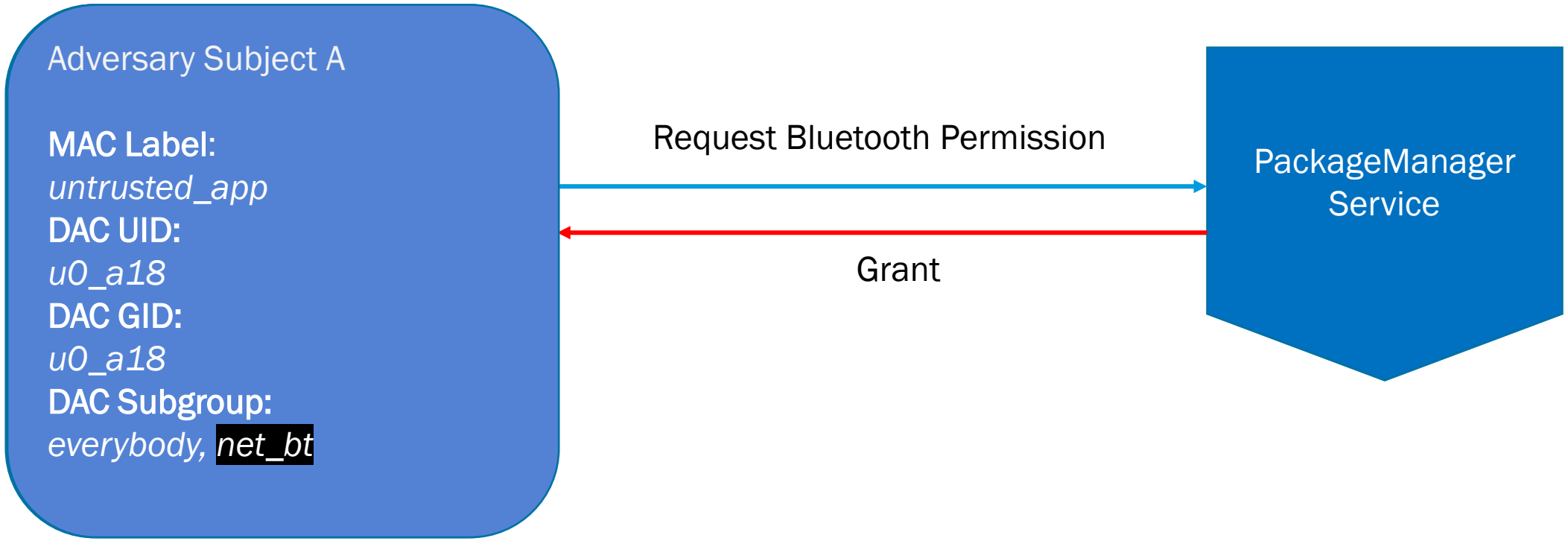
Process Privilege Levels

Process Level[1]	Level Membership Requirements
Root Process (T5)	Process running with UID root (e.g., MAC labels `kernel` and `init`)
System Process (T4)	Process running with UID system (e.g., MAC label `system server`)
Service Process (T3)	AOSP core service providers (e.g., MAC labels `bluetooth` and `mediaserver`)
Trusted Application Process (T2)	AOSP default and vendor apps (e.g., MAC labels `platform_app` and `priv_app`)
Untrusted Application Process (T1)	Third-party applications (e.g., MAC label `untrusted_app`)
Isolated Process (T0)	Processes that are expected to receive adversarial inputs (e.g., MAC label `webview`)

- T5 with highest privilege and T0 with lowest privilege
- Based on functionality and design of Android Framework

Permission Expansion – Adversary Expansion

- Adversaries may obtain Android Permissions that augment their UNIX DAC Permissions



Permission Expansion – Victim Expansion

- Adversaries may delegate DAC permission for objects they own to potential victims
- In most case, victim has MAC permissions to adversary directories but not DAC permission

New Integrity Violation

Accounting for Permission Flexibility

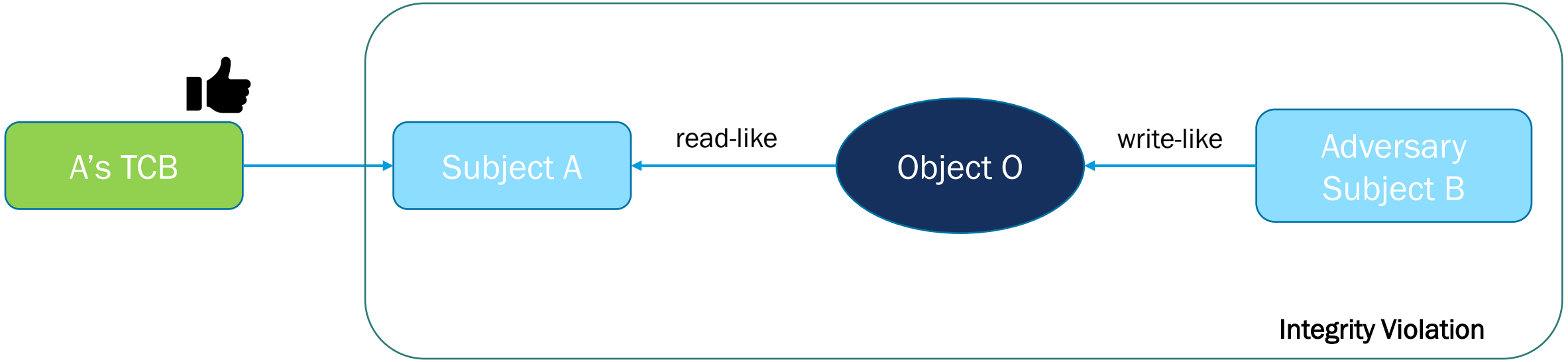
Adversary Expansion

- Assumes users grant all non-signature Android permissions to applications
- Assumes white-listed apps (OEM apps) obtain all signature-level Android permissions

Victim Expansion

- Assumes adversaries will grant read, write, and execute permissions to victim by altering UNIX DAC permission assignments whenever possible
- Victim expansion only possible when adversary owns the corresponding object (file/dir)

PolyScope Integrity Violations



Integrity Violations

- **Rules that account for MAC (SEAndroid and MLS) and DAC (UNIX) with permission expansion**

Types of Integrity Violations

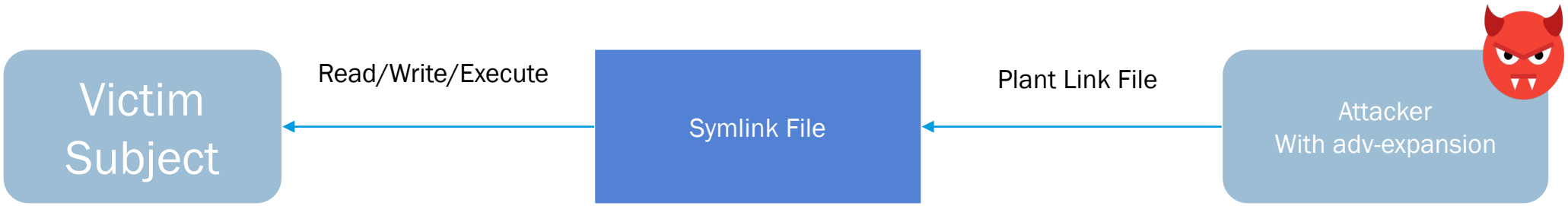
- File-IV
- Binding-IV (directories)
- Pathname-IV

IVs to Attack Operations

Integrity Violations are exploited when an attacker can perform an attack operation

Example of an Attack Operation: Link Traversal

- Create a symbolic link that references a target file the attacker really wants to access
- Victim with permission to the target file accesses the target on behalf of the attacker

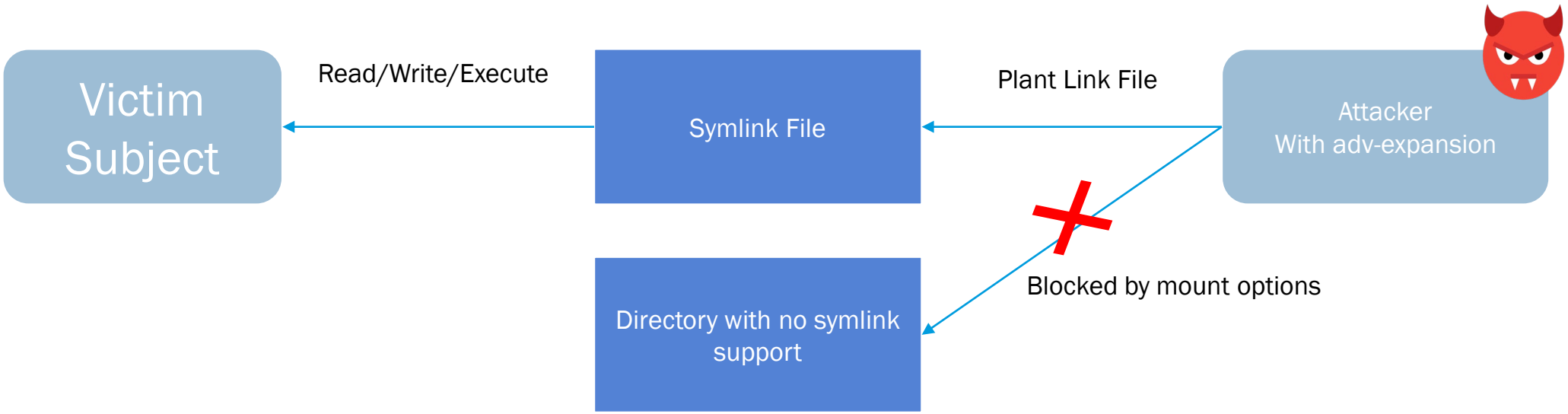


Are Attack Operations Really Exploitable?

Integrity Violations might not be exploitable due to system configurations or additional defenses

Examples of Defenses that Block Link Traversal

- Some Android filesystems do not allow symbolic links
- FileProvider class use by victim subject prevents link traversal



Evaluation Overview

- Did defining per-subject TCB make a difference? (flows vs. IVs)
- Did permission expansion increase the number of IVs?
- Did PolyScope triage attack operations from IVs?
- What's the impact of OEM customization?
- How many integrity violations are found when using the SEAndroid MAC TE policy alone in TE IV Computation?
- How are the number of IVs reduced after TE IV validation from those found in the TE IV Computation?
- How many IVs distributed across Android privilege levels?

Flow vs. IV

Did defining per-subject TCB make a difference?

	Cross Version IVs (Google)			Cross OEM IVs			
	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Mate9 8.0	Mate9 9.0	Mix2 9.0	Galaxy S8 9.0
Authorized Data Flows[1]	204,241	166,027	156,315	240,916	860,508	289,238	259,992
IVs for PolyScope Adversaries[2]	167	80	69	223	166	192	628
PolyScope IVs after Expansion[3]	372	478	1,139	1,682	1,566	2,304	4,377
PolyScope IVs with Operations[4]	297	387	927	1,331	1,327	1,979	3,736
Average Operations per IV[5]	1.18	1.10	1.04	1.62	1.37	1.08	1.36

Yes!

- Computing authorized data flow leave OEMs with hundreds of thousands of cases to assess
- PolyScope's per-subject threat model results in reduction of the number of data flows involve in IV by at least two orders of magnitude
- Runtime of data flow computation greatly improved with PolyScope's threat model

Impact of Permission Expansion

Did permission expansion increase the attack surface?

	Cross Version IVs (Google)			Cross OEM IVs			
	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Mate9 8.0	Mate9 9.0	Mix2 9.0	Galaxy S8 9.0
Authorized Data Flows[1]	204,241	166,027	156,315	240,916	860,508	289,238	259,992
IVs for PolyScope Adversaries[2]	167	80	69	223	166	192	628
PolyScope IVs after Expansion[3]	372	478	1,139	1,682	1,566	2,304	4,377
PolyScope IVs with Operations[4]	297	387	927	1,331	1,327	1,979	3,736
Average Operations per IV[5]	1.18	1.10	1.04	1.62	1.37	1.08	1.36

Yes!

- Victim and adversary permission expansion increases the number of IVs significantly, from 122% to 1550%

Triage Attack Operations

Did PolyScope reduce attack operations from IVs?

	Cross Version IVs (Google)			Cross OEM IVs			
	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Mate9 8.0	Mate9 9.0	Mix2 9.0	Galaxy S8 9.0
Authorized Data Flows[1]	204,241	166,027	156,315	240,916	860,508	289,238	259,992
IVs for PolyScope Adversaries[2]	167	80	69	223	166	192	628
PolyScope IVs after Expansion[3]	372	478	1,139	1,682	1,566	2,304	4,377
PolyScope IVs with Operations[4]	297	387	927	1,331	1,327	1,979	3,736
Average Operations per IV[5]	1.18	1.10	1.04	1.62	1.37	1.08	1.36

Yes!

- By taking Java-level defense and file system configuration into account, around 14% to 21% of the IVs fail to be realized as even a single attack operation

OEM Customization

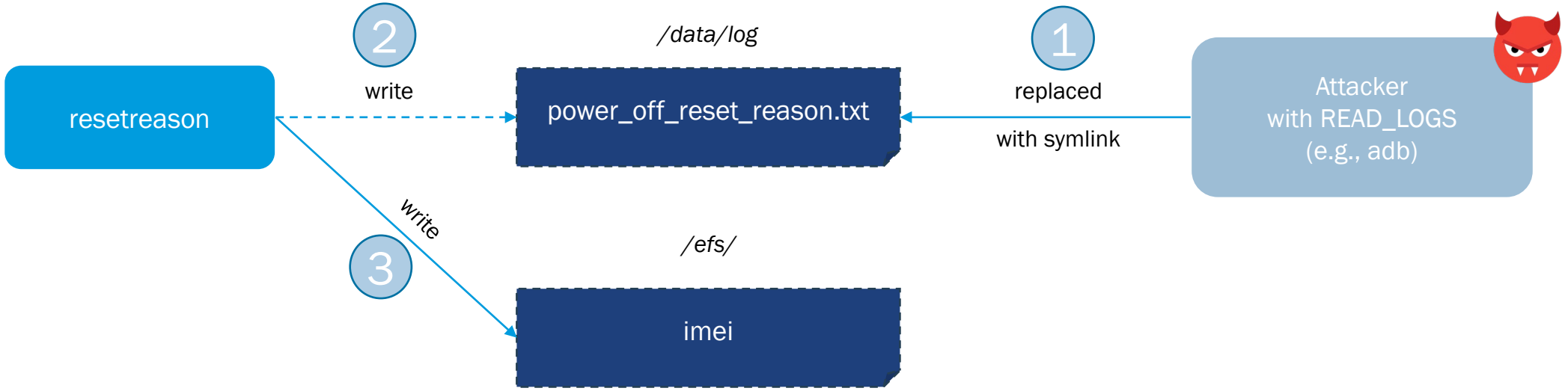
What's the impact of OEM customization?

	Cross Version IVs (Google)			Cross OEM IVs			
	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Mate9 8.0	Mate9 9.0	Mix2 9.0	Galaxy S8 9.0
Authorized Data Flows[1]	204,241	166,027	156,315	240,916	860,508	289,238	259,992
IVs for PolyScope Adversaries[2]	167	80	69	223	166	192	628
PolyScope IVs after Expansion[3]	372	478	1,139	1,682	1,566	2,304	4,377
PolyScope IVs with Operations[4]	297	387	927	1,331	1,327	1,979	3,736
Average Operations per IV[5]	1.18	1.10	1.04	1.62	1.37	1.08	1.36

Making things worse!

- OEMs added large amount of MAC TE allow rules for functionality
- Some OEMs incorrectly use shared memory location for file storage
- OEMs grants many security critical permissions to their own applications/services

Vulnerability Case Study



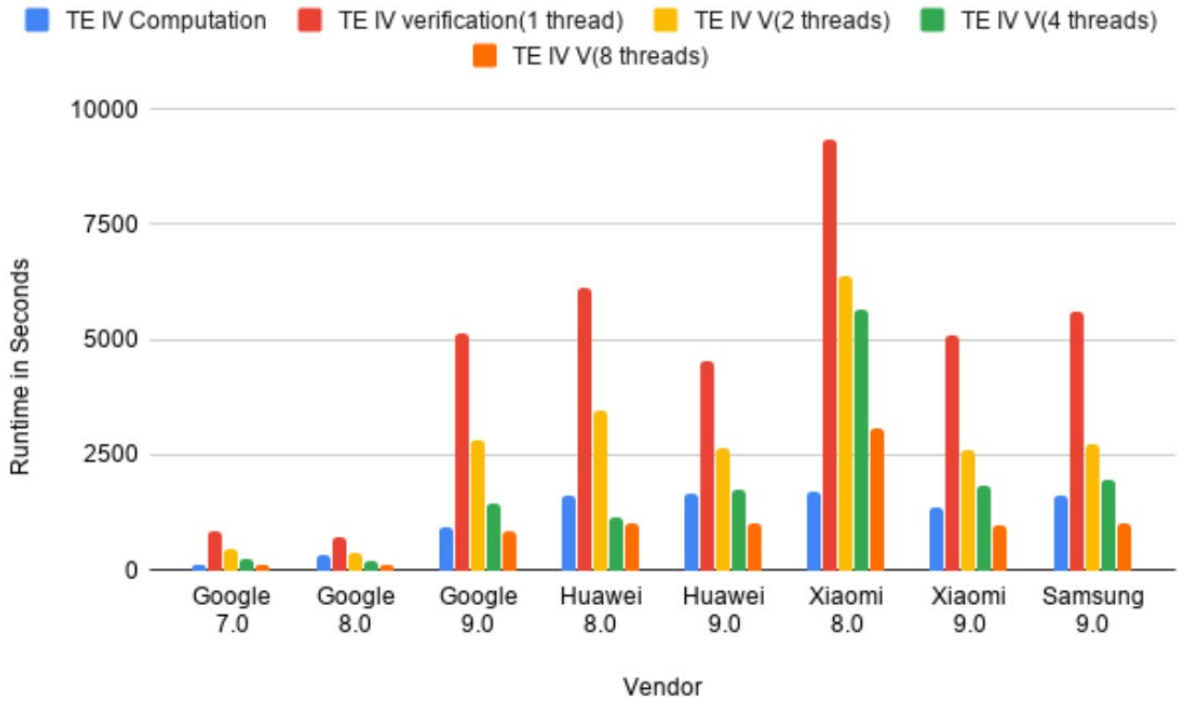
1. Attacker gain DAC GID *log* through Android Permission (adv-expansion)
2. Replaced power_off_reset_reason.txt with symlink to encrypted file system
3. Subject resetreason becomes victim of confused deputy attack when attempting to record reboot logs
4. Could write to encrypted file system and brick the device
5. Assigned CVE-2020-13833

Performance

Benefits from parallelization

- 5127s → 821s (Google 9.0)
- 9347s → 3089s (Xiaomi 8.0)

Finer per-program TCB calculation reduces runtime significantly



Summary and Limitations

Summary

- Novel analysis approach to generate attack operations from a combination of access control policies
- First work to include flexibility of access control into the analysis framework
- Applied PolyScope to multiple Google and OEM Android releases, discovered two vulnerabilities

Limitation

An Attack Operation is **not** necessarily a vulnerability

Thank you!
Question?

PolyScope Analysis Framework:

https://github.com/yxl74/PolyScope.git