西安电子科技大学
XIDIAN UNIVERSITY

LibScan: Towards More Precise Third-Party Library Identification for Android Applications

Yafei Wu
Xidian University

Cong Sun
Xidian University
suncong@xidian.edu.cn

Dongrui Zeng
Palo Alto Networks

Gang Tan
Pennsylvania State University

Siqi Ma
University of New South Wales

Peicheng Wang
Xidian University

Usenix Security 2023

Outline

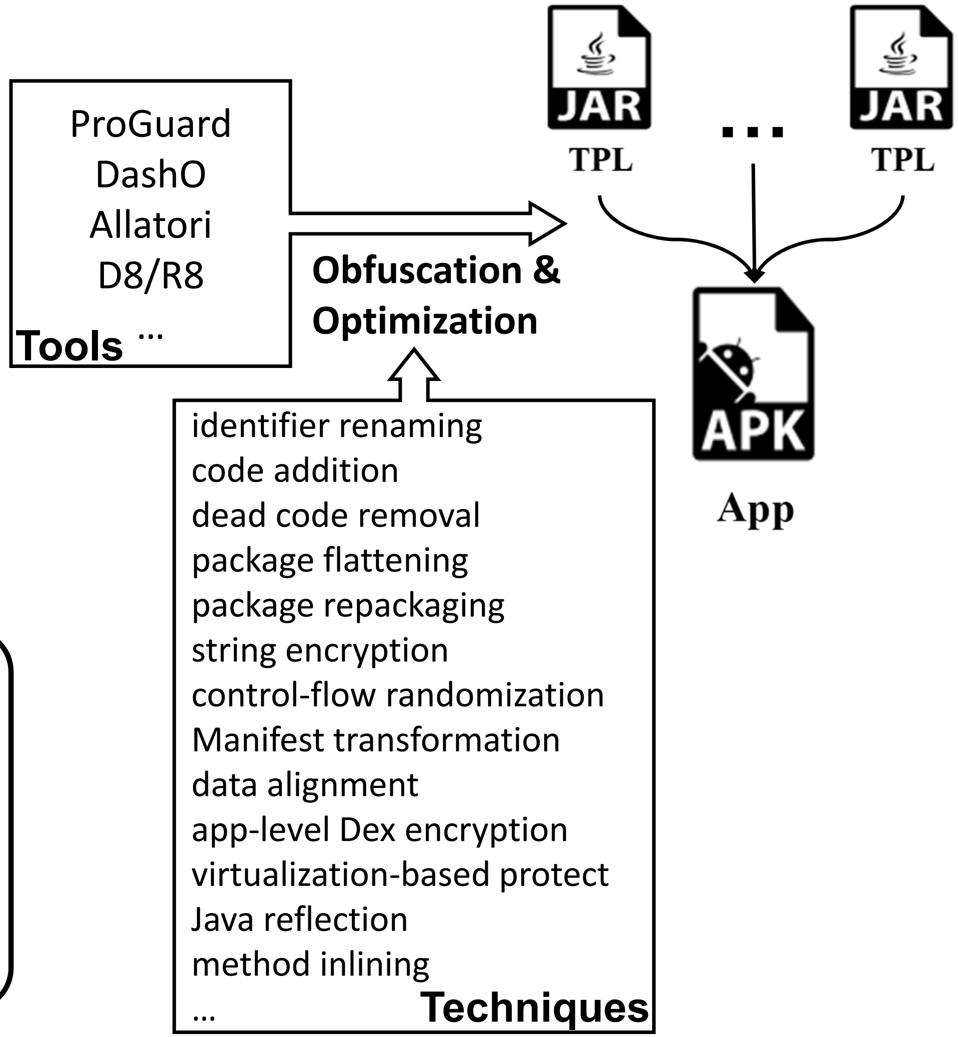
- Background and motivation
- Design
- Evaluation
- Conclusion

Background and Motivation

- Third-party library (TPL) is indispensable for modern apps
  - advertising, social networking, game engine, payment, …
- TPLs account for >60% of the code in Android apps [ISSTA’15]
- Threat of using TPL
  - Delay or no fix of the TPL vulnerabilities in the app
  - Pose threats to the system …
- Urgent requirements for app developers and app-store vetting:
  - Keeping app using up-to-date TPLs.
  - Identifying the used TPLs.
  - Finding potential security vulnerabilities of TPLs.

Background and Motivation

- Potential **obstacles** to identifying TPLs
  - Apps and the in-app TPLs are pervasively obfuscated (24.92% Google Play apps [ACSAC'18]).
  - New development toolchain with new obfuscation techniques (e.g. D8/R8 of Android Studio 3.1+).



Motivation

Implementing more accurate TPL detection, and bridging the gap of prior work's capability in addressing the obfuscation techniques implemented by obfuscators.

Scope of LibScan

Overcome the obfuscation techniques implemented by Allatori, DashO, and ProGuard. Not designed against the D8/R8 compiler, but outperforms other approaches on R8-obfuscated apps in experiments.

Table 1: Obfuscation techniques of android obfuscators (LibScan is robust against techniques marked with (*))

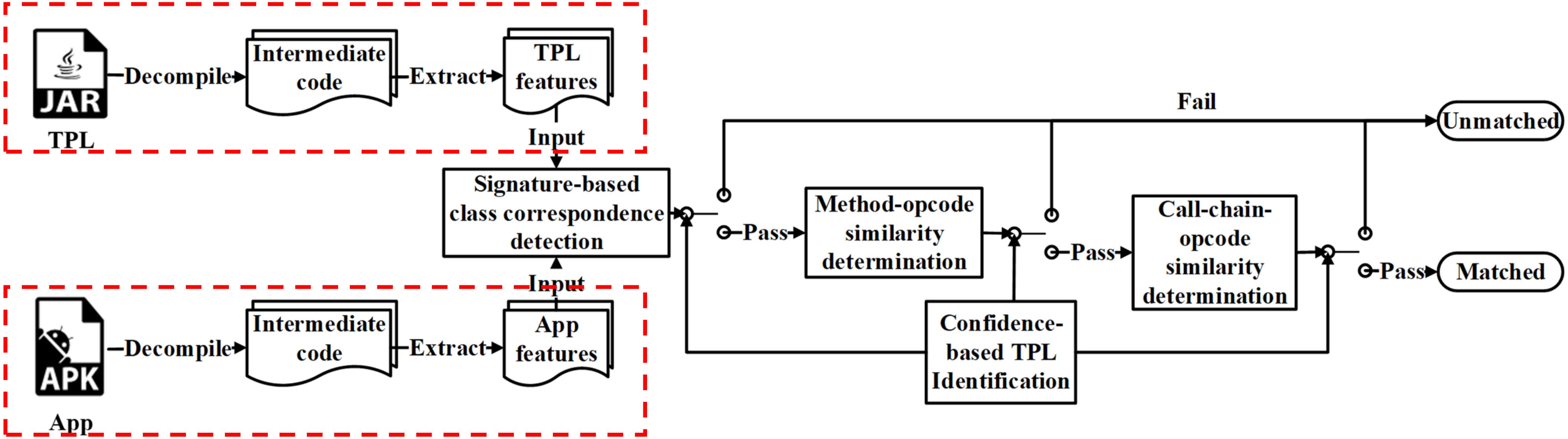
	Allatori	DashO	ProGuard
identifier renaming(*)	✓	✓	✓
code addition(*)	✓	✓	✓
dead code removal(*)	✓	✓	✓
package flattening/repackaging(*)	✓	✓	✓
string encryption(*)	✓	✓	–
control-flow randomization(*)	✓	✓	–
Manifest transformation (*)	–	–	–
data alignment (*)	–	–	–
app-level Dex encryption	–	–	–
virtualization-based protection	–	–	–
Java reflection	–	–	–
method inlining	–	–	–

Outline

- Background and motivation
- Design
- Evaluation
- Conclusion

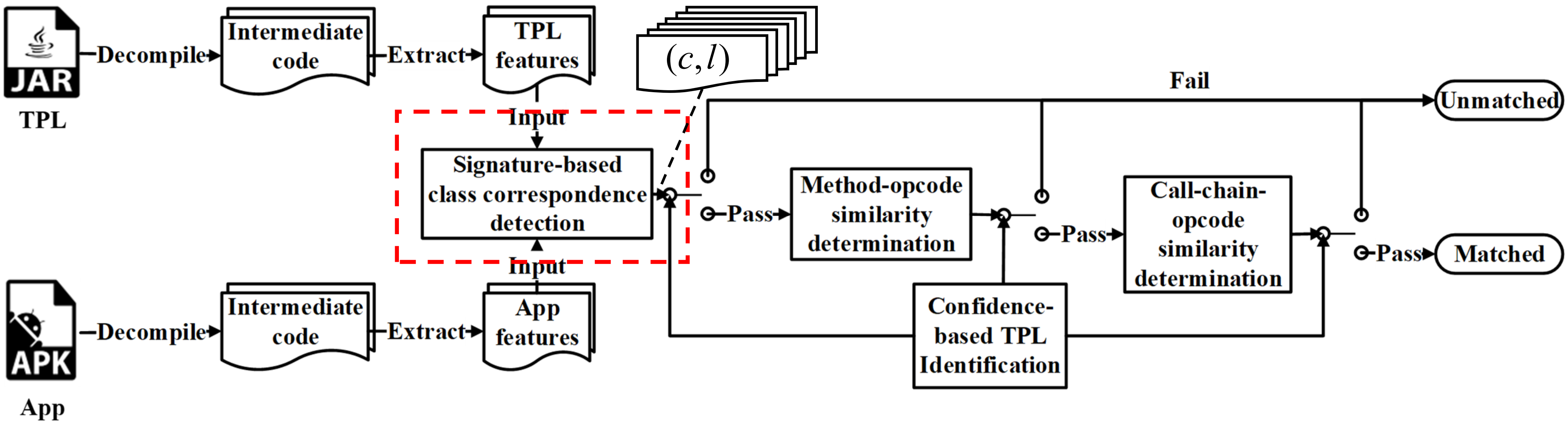
Design of LibScan

- **Initialization: Extract necessary features for every step from app and TPL**



Design of LibScan

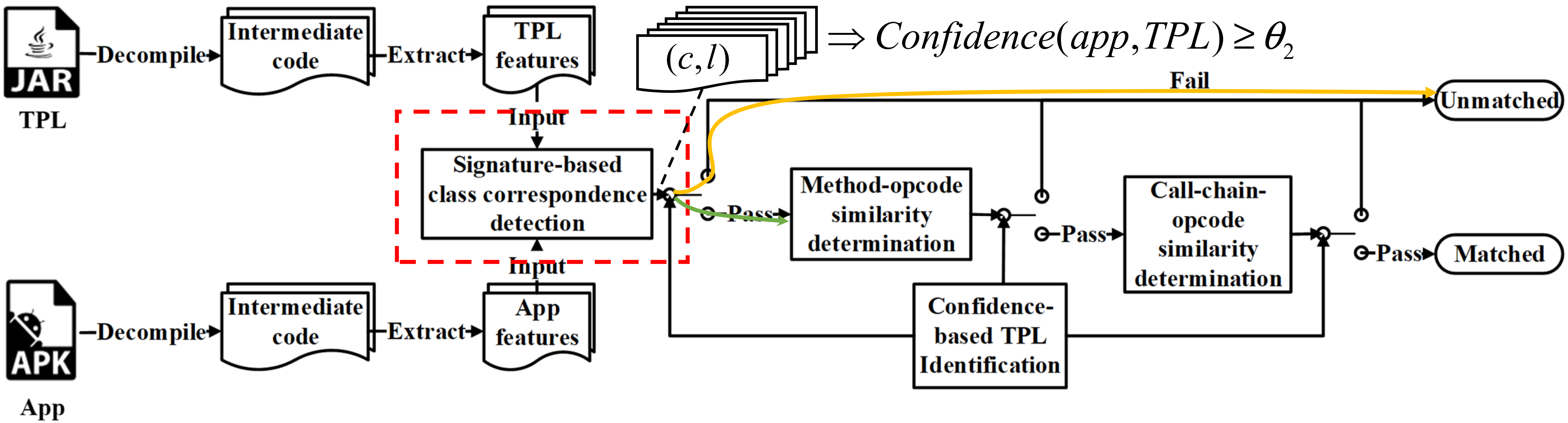
- Initialization: Extract necessary features for every step from app and TPL
- **Step 1: Compare each app class with TPL class, generate a set of pairwise class correspondences**



Design of LibScan

- Initialization: Extract necessary features for every step from app and TPL
- Step 1: Compare each app class with TPL class, generate a set of pairwise class correspondences

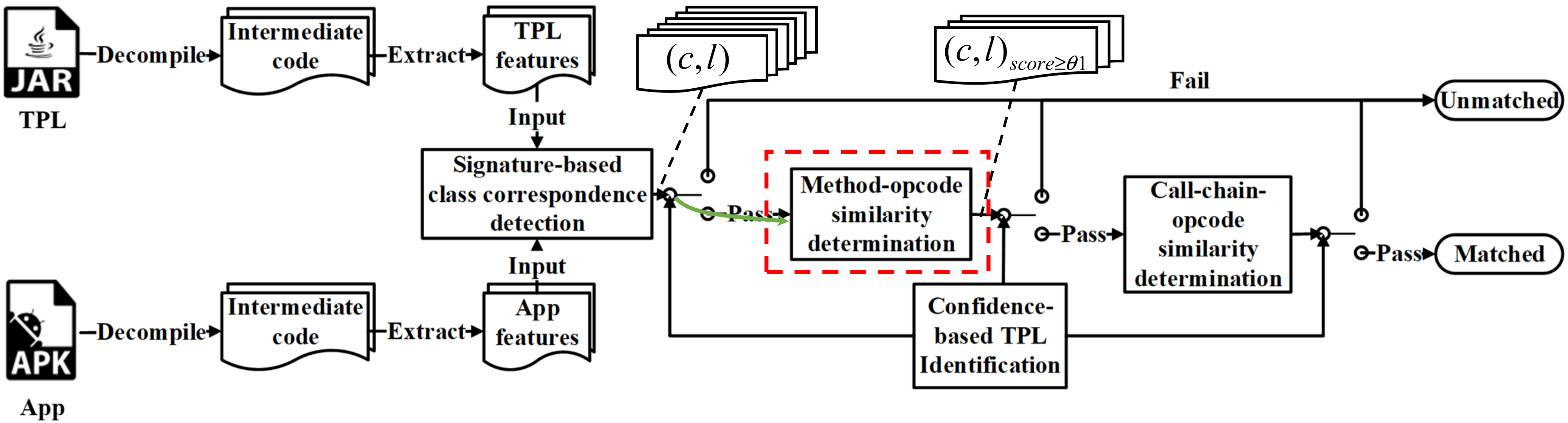
- **After each step, determine a confidence score from the remaining class correspondences to forbid dissimilar TPL from the next step**



Design of LibScan

- Initialization: Extract necessary features for every step from app and TPL
- Step 1: Compare each app class with TPL class, generate a set of pairwise class correspondences
- **Step 2: Compare methods' opcodes similarity of each class correspondence**

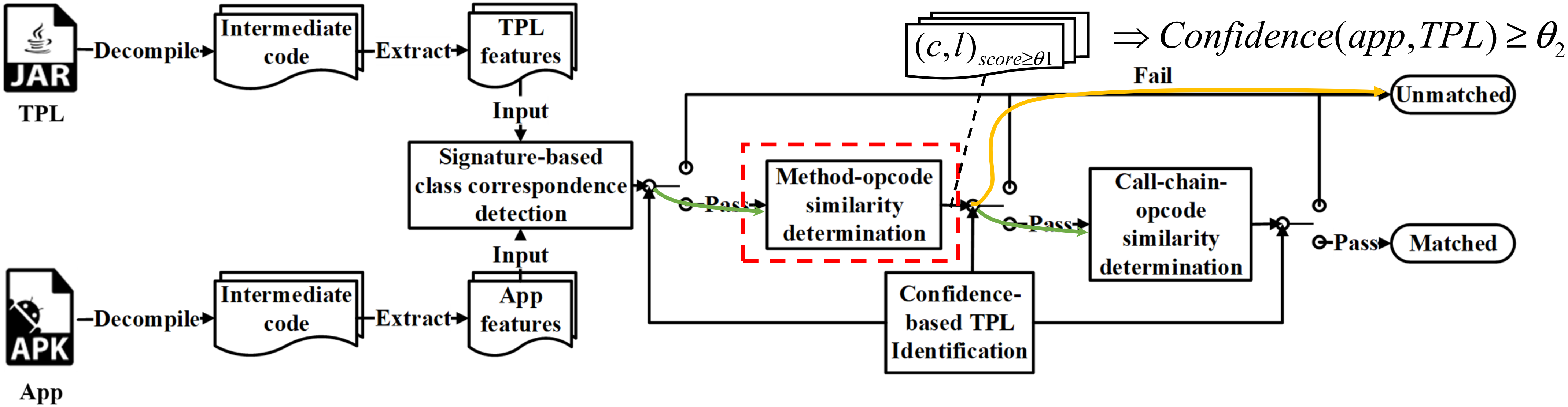
- After each step, determine a confidence score from the remaining class correspondences to forbid dissimilar TPL from the next step.



Design of LibScan

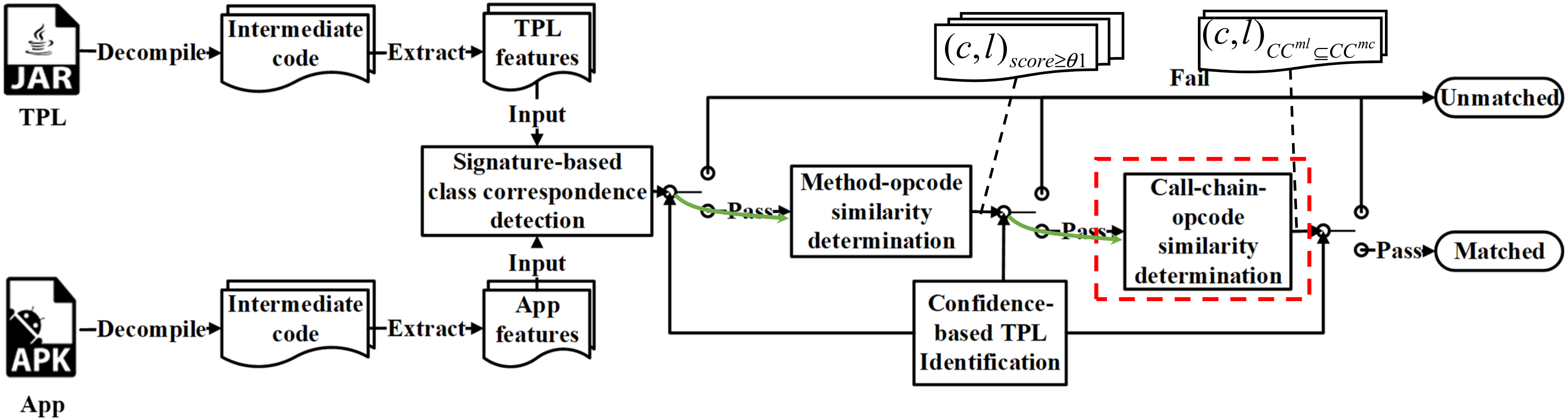
- Initialization: Extract necessary features for every step from app and TPL
- Step 1: Compare each app class with TPL class, generate a set of pairwise class correspondences
- Step 2: Compare methods' opcodes similarity of each class correspondence

- **After each step, determine a confidence score from the remaining class correspondences to forbid dissimilar TPL from the next step.**



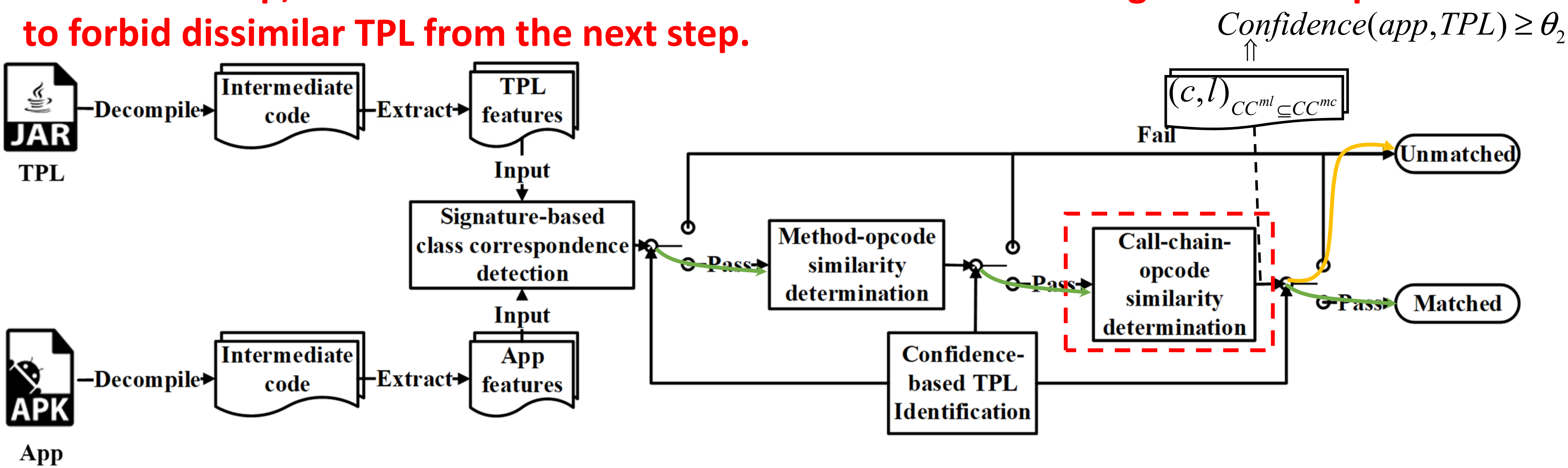
Design of LibScan

- Initialization: Extract necessary features for every step from app and TPL
- Step 1: Compare each app class with TPL class, generate a set of pairwise class correspondences
- Step 2: Compare methods' opcodes similarity of each class correspondence
- **Step 3: Compare method-call-chains' similarity of each class correspondence**
- After each step, determine a confidence score from the remaining class correspondences to forbid dissimilar TPL from the next step.



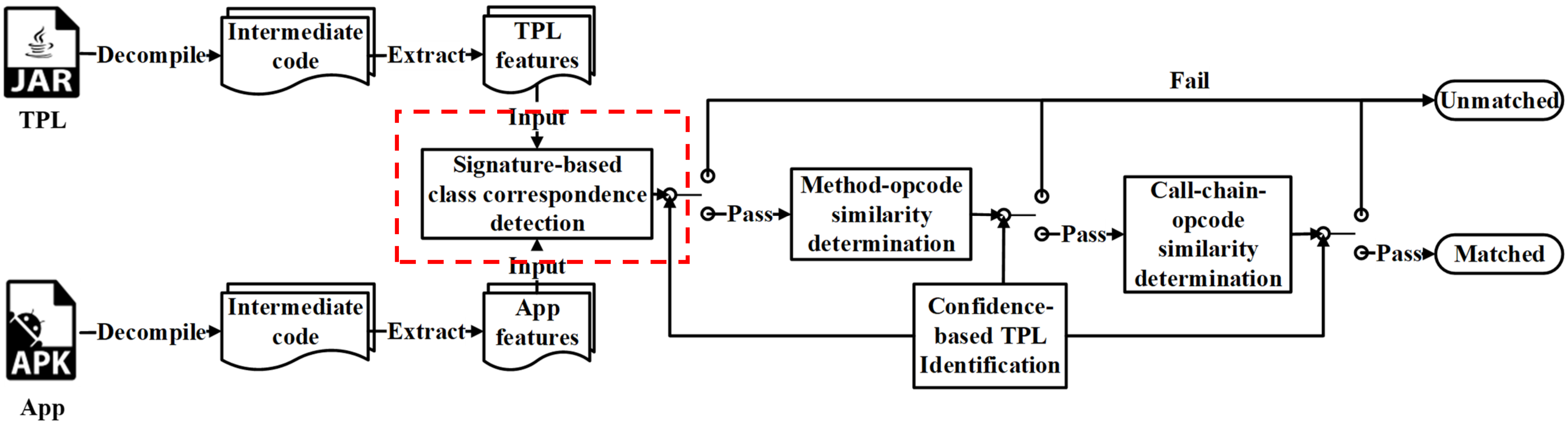
Design of LibScan

- Initialization: Extract necessary features for every step from app and TPL
- Step 1: Compare each app class with TPL class, generate a set of pairwise class correspondences
- Step 2: Compare methods' opcodes similarity of each class correspondence
- Step 3: Compare method-call-chains' similarity of each class correspondence
- **After each step, determine a confidence score from the remaining class correspondences to forbid dissimilar TPL from the next step.**



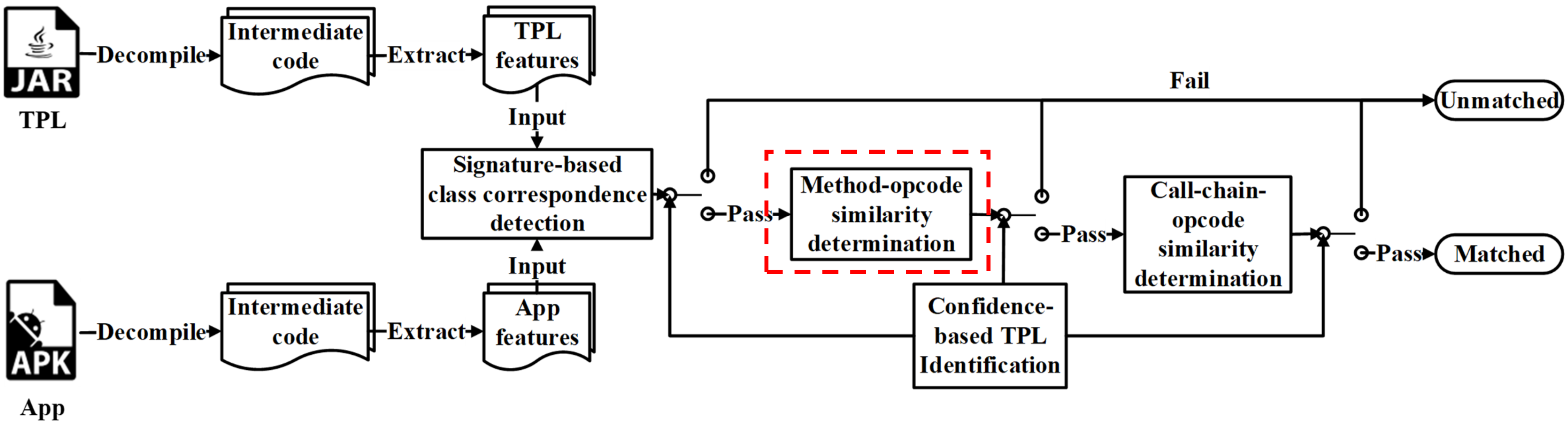
Design of LibScan

- Step 1: Compare each app class with TPL class, generate a set of pairwise class correspondences
  - Focusing on code features that may persist during obfuscation.
  - Signature: 6 class features, 45 field features, and 736 method features (787 in total) for each class
  - Pairwise 787-dimensional Boolean vectors matching to find the class correspondences



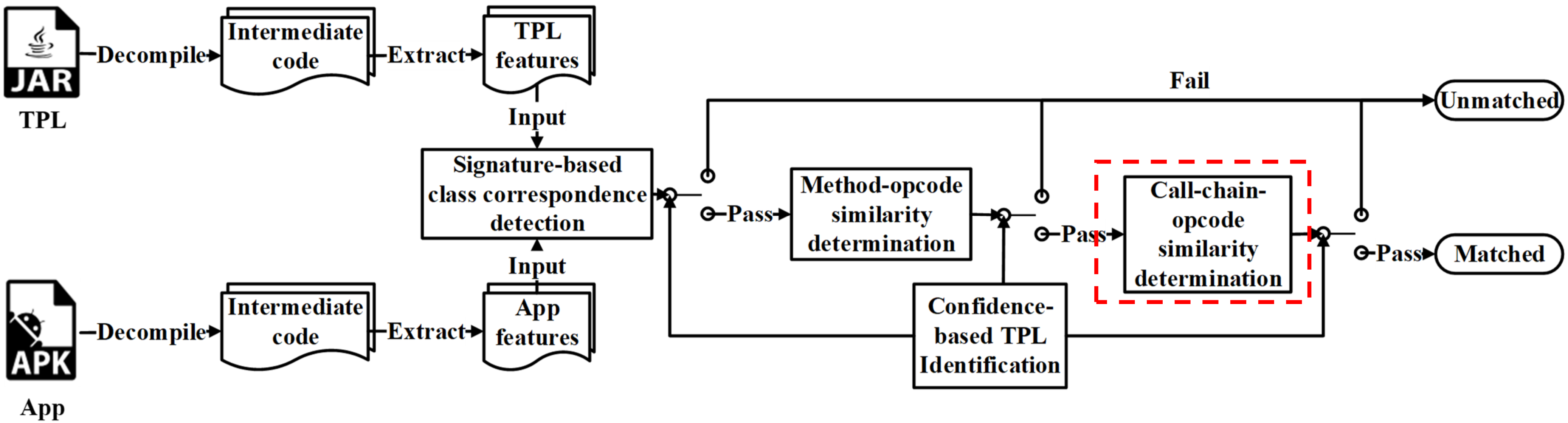
Design of LibScan

- Step 2: Compare methods' opcodes similarity of each class correspondence
  - Make each TPL method match with at most one app method.
    - Selects the best-matched app method with minimal opcode difference compared to the TPL method.
  - A high similarity score ($MOSS(c,l)\geq\theta_1$) indicates that the proportion of best-matched app methods to the TPL class methods dominate the app methods of an app class in size.



Design of LibScan

- Step 3: Compare method-call-chains' similarity of each class correspondence
  - For the best-matched app method and TPL method identified in Step 2, taking them as respective entry method of call chain, the call-chain opcodes of the app method should include the call-chain opcodes of the TPL method.
  - Otherwise, the class correspondence is removed.



Outline

- Background and motivation
- Design
- Evaluation
- Conclusion

Need threshold tuning (θ_1,θ_2)

Grid search on different (θ_1,θ_2) for the optimal F1-score.

On a small ground-truth app dataset (110 apps) and the full TPL dataset (452 TPLs), the tuning procedure takes 21~22 hours to find the optimal (θ_1,θ_2)=(0.7,0.85)

Table 6: Grid Search on F1-scores to Establish Optimal Threshold θ_1 and θ_2

		θ_1						
		0.65	0.7	0.75	0.8	0.85	0.9	0.95
θ_2	0.5	0.891	0.894	0.894	0.895	0.894	0.885	0.880
	0.55	0.893	0.896	0.896	0.894	0.948	0.939	0.947
	0.6	0.894	0.897	0.897	0.894	0.947	0.938	0.944
	0.65	0.894	0.897	0.952	0.949	0.947	0.938	0.942
	0.7	0.901	0.904	0.958	0.955	0.953	0.944	0.942
	0.75	0.899	0.902	0.958	0.955	0.952	0.956	0.933
	0.8	0.954	0.956	0.956	0.953	0.950	0.952	0.910
	0.85	0.964	**0.967**	0.966	0.965	0.961	0.932	0.883
	0.9	0.944	0.947	0.939	0.921	0.912	0.882	0.805
	0.95	0.838	0.832	0.814	0.811	0.808	0.776	0.743

西安电子科技大学
XIDIAN UNIVERSITY

Evaluation

Effectiveness

(compared with state-of-the-art approaches LibScout, Orlis, LibPecker, and LibID)

LibScan outperforms others in most cases (non-obfuscated or obfuscated by DashO, ProGuard, and Allatori), though Orlis has good library-level precision.

Table 7: Effectiveness Comparison of Different Tools on 939 apps of Dataset AS_1 (5,956 Ground-Truth TPL Existences)

Tool	Library-level						Version-level					
	TP_0	FP_0	FN_0	$Precision_0$	$Recall_0$	$F1_0$	TP	FP	FN	Precision	Recall	F1
LibID-S	2,209	1,358	3,747	0.6193	0.3709	0.4639	2,192	1,375	3,764	0.6145	0.3680	0.4604
LibID-A	2,098	622	3,858	0.7713	0.3522	0.4836	2,091	629	3,865	0.7688	0.3511	0.4820
LibPecker	4,563	1,798	1,393	0.7173	0.7661	0.7409	4,243	2,118	1,713	0.6670	0.7124	0.6890
Orlis	1,507	**45**	4,449	**0.9710**	0.2530	0.4014	730	822	5,226	0.4704	0.1226	0.1945
LibScout	2,679	314	3,277	0.8951	0.4498	0.5987	2,664	**329**	3,292	0.8901	0.4473	0.5954
$LibScan^{I}$	5,872	2,211	84	0.7265	0.9859	0.8365	5,846	2,237	110	0.7232	0.9815	0.8328
$LibScan^{I+II}$	5,812	1,199	144	0.8290	0.9758	0.8964	5,685	1,326	271	0.8109	0.9545	0.8768
LibScan	**5,741**	326	**215**	0.9463	**0.9639**	**0.9550**	**5,659**	408	**297**	**0.9328**	**0.9501**	**0.9414**

Effectiveness on different obfuscation levels
(5 DashO obfuscation levels and 4 D8/R8 obfuscation levels)

LibScan outperforms others on each DashO obfuscation level.
On the D8/R8 obfuscation levels, LibScout performs best on D8-built non-obfuscated apps;
LibScan performs best on R8-built apps with code shrinking but disabled optimization;
none tool is effective on R8-built apps with code shrinking and optimization.

Table 8: Effectiveness Comparison of Detecti

Detection Level	Obfuscation Level	LibScan		
		PR$_0$	RC$_0$	F1$_0$
Library-level	Non-obfustated	0.984	1.000	**0.992**
	DashO-cfr	0.984	0.982	**0.983**
	DashO-pf-ir	0.986	0.984	**0.985**
	DashO-dcr	0.997	0.873	**0.931**
	DashO-cfr-pf-ir-dcr	0.986	0.977	**0.981**
		PR	RC	F1
Version-level	Non-obfustated	0.984	1.000	**0.992**
	DashO-cfr	0.954	0.952	**0.953**
	DashO-pf-ir	0.958	0.956	**0.957**
	DashO-dcr	0.963	0.843	**0.899**
	DashO-cfr-pf-ir-dcr	0.956	0.947	**0.951**

Table 9: Effectiveness Comparison of Detection Tools to Different D8/R8 Obfuscation Levels (PR=Precision, RC=Recall)

Detection Level	Obfuscation Level	LibScan			LibScout			Orlis			LibPecker			LibID-A		
		PR$_0$	RC$_0$	F1$_0$	PR$_0$	RC$_0$	F1$_0$	PR$_0$	RC$_0$	F1$_0$	PR$_0$	RC$_0$	F1$_0$	PR$_0$	RC$_0$	F1$_0$
Library-level	D8-non-obfs	0.783	0.981	0.871	0.818	0.969	**0.887**	0.579	0.500	0.536	0.786	0.975	0.871	0.821	0.821	0.821
	R8-shrink	0.904	0.580	**0.707**	0.389	0.272	0.320	0.632	0.457	0.530	0.754	0.568	0.648	0.704	0.352	0.469
	R8-shrink-orlis	0.903	0.574	**0.702**	0.488	0.130	0.205	0.630	0.463	0.534	0.739	0.506	0.601	0.585	0.235	0.335
	R8-shrink-opt	1.000	0.080	0.149	0.258	0.105	**0.149**	0.545	0.037	0.069	0.917	0.068	0.126	1.000	0.068	0.127
		PR	RC	F1	PR	RC	F1	PR	RC	F1	PR	RC	F1	PR	RC	F1
Version-level	D8-non-obfs	0.719	0.901	0.800	0.818	0.969	**0.887**	0.336	0.290	0.311	0.716	0.889	0.793	0.753	0.753	0.753
	R8-shrink	0.808	0.519	**0.632**	0.372	0.259	0.305	0.342	0.247	0.287	0.467	0.352	0.401	0.679	0.340	0.453
	R8-shrink-orlis	0.796	0.506	**0.619**	0.488	0.130	0.205	0.361	0.265	0.306	0.441	0.302	0.359	0.569	0.228	0.326
	R8-shrink-opt	0.769	0.062	0.114	0.197	0.080	0.114	0.273	0.019	0.035	0.917	0.068	0.126	1.000	0.068	**0.127**

Necessity of LibScan's each detection step

The latter steps (Steps 2 and 3) are indispensable for reducing FPs and improving precision.

Ignoring the earlier steps (Step 1 or 2) will drastically increase detection costs.

Table 11: Per-App Efficiency Benefit from Different LibScan Detection Steps

	T_1(s)	T_2(s)	T_3(s)	T_4(s)	T_5(s)	T_{total}(s)
$\text{LibScan}^{\text{III}}$	29.07	–	–	780.20	10.54	819.81
$\text{LibScan}^{\text{II+III}}$	29.07	–	480.10	0.01	10.02	519.20
LibScan	29.07	6.14	0.01	0.01	10.76	45.99

Table 7: Effectiveness Comparison of Different Tools on

Tool	Library-level												
	TP_0	FP_0	FN_0	Precision_0	Recall_0								
LibID-S	2,209	1,358	3,747	0.6193	0.3709	0.4639	2,192	1,373	3,764	0.6143	0.3680	0.4604
LibID-A	2,098	622	3,858	0.7713	0.3522	0.4836	2,091	629	3,865	0.7688	0.3511	0.4820
LibPecker	4,563	1,798	1,393	0.7173	0.7661	0.7409	4,243	2,118	1,713	0.6670	0.7124	0.6890
Orlis	1,507	**45**	4,449	**0.9710**	0.2530	0.4014	730	822	5,226	0.4704	0.1226	0.1945
LibScout	2,679	314	3,277	0.8951	0.4498	0.5987	2,664	**329**	3,292	0.8901	0.4473	0.5954
$\text{LibScan}^{\text{I}}$	5,872	2,211	84	0.7265	0.9859	0.8365	5,846	2,237	110	0.7232	0.9815	0.8328
$\text{LibScan}^{\text{I+II}}$	5,812	1,199	144	0.8290	0.9758	0.8964	5,685	1,326	271	0.8109	0.9545	0.8768
LibScan	**5,741**	326	**215**	0.9463	**0.9639**	**0.9550**	**5,659**	408	**297**	**0.9328**	**0.9501**	**0.9414**

Efficiency

(On both ground-truth apps and most popular Google Play apps)

LibScout is the most efficient.

LibScan is competitive in efficiency.

Table 10: Per-App Detection Efficiency of Different Tools on AS_3

	LibID-S(s)	LibPecker(s)	Orlis(s)	LibScout(s)	LibScan(s)
Q1	47.52	498.23	51.34	3.40	35.12
mean	956.69	797.00	135.66	5.45	45.99
median	151.88	741.01	110.21	5.04	44.10
Q3	654.63	1036.98	219.62	7.14	57.61

Table 15: Per-App Detection Efficiency of Different Tools on AS_1

	LibID-S(s)	LibPecker(s)	Orlis(s)	LibScout(s)	LibScan(s)
Q1	10.08	250.29	39.66	1.17	22.08
mean	72.14	307.75	52.98	1.35	24.18
median	64.92	290.54	51.52	1.30	23.56
Q3	103.69	344.62	64.41	1.49	26.74

Scalability

LibScan detected 3,949 existences of 23 vulnerable TPLs in 3,664 of 100K real-world apps, and the annual existences are investigated.

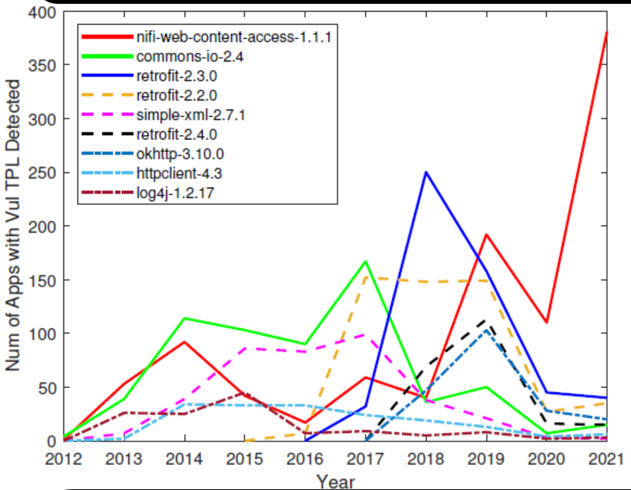


Table 13: AV Vendor Mark Updates of VirusTotal on Different CooTek App Clusters

Cluster ID	0				0’								1		2	3		
#Vendor reported	27	1	25	25	15	0	0	0	0	0	0	0	7	10	28	1	1	1
#Vendor reanalyzed	30	27	26	26	28	24	21	19	19	18	18	17	10	10	20	8	8	8

Facilitating malware detection

Clustering apps based on fuzzy-hash similarity and the same vulnerable TPL usage.

A case study shows 10 correct predictions by propagating LibScan’s verdicts on the clusters of CooTek apps.

When disabling the requirement on using the same vul TPL, predictions become incorrect.

Outline

- Background and motivation
- Design
- Evaluation
- Conclusion

Conclusion

- LibScan is
  - Efficient TPL identification approach for Android apps using static analysis
    - Efficient because the class correspondences reduction procedure can early stop the TPL detection based on the confidence scores
  - Suitable for app-store vetting
    - Caching the code features of apps and TPLs for batch-job TPL identifications
  - More accurate than other approaches
    - Fingerprinting code features and the set-based opcode similarity decision are more tolerable to the state-of-the-art obfuscation techniques

Available: https://github.com/wyf295/LibScan

西安电子科技大学
XIDIAN UNIVERSITY

THANKS

Thanks for listening

Contact: Cong Sun
suncong@xidian.edu.cn