CONFUSETAINT: Exploiting Vulnerabilities to Bypass Dynamic Taint Analysis

Yufei Wu
Umeå University
yufeiwu@cs.umu.se

Alexandre Bartel
Umeå University
alexandre.bartel@cs.umu.se

***Abstract*—Dynamic taint analysis (DTA) tracks how sensitive data flows through a program at runtime, enabling the detection of security violations such as information leaks and injection attacks. However, most DTA systems assume that memory layouts are type-safe and structurally consistent—an assumption that can be violated by vulnerabilities such as type confusion. While type confusion has been studied in the context of sandbox escape, its ability to silently bypass taint tracking without altering program behavior remains unexplored. In this paper, we present CONFUSETAINT, a technique that leverages type confusion vulnerabilities to corrupt taint metadata without modifying program semantics or the analysis tool. CONFUSETAINT uses wide memory overwrites enabled by type confusion to corrupt taint tags, breaking the assumptions of taint tracking mechanisms that rely on shadow memory.**

We evaluate CONFUSETAINT on two widely used taint tracking frameworks: Phosphor for the JVM and TaintDroid for Android. In both cases, CONFUSETAINT successfully bypasses taint tracking, allowing sensitive data to reach designated sinks without detection. These results reveal a structural weakness in current DTA designs: their reliance on type-safe memory layouts leaves them vulnerable to low-level reinterpretation. Overall, our work reveals that runtime-level memory reinterpretation is an overlooked threat, calling for taint tracking architectures that do not rely on fragile assumptions about type and memory layout.

***Index Terms*—Program Analysis, Dynamic analysis, Taint analysis, Type confusion, Vulnerability**

I. INTRODUCTION

Modern software systems are increasingly complex due to features such as reflection, runtime code generation, and untrusted third-party dependencies. These factors obscure control and data flows, making it difficult to enforce security policies and increasing the risk of logic flaws like insecure deserialization [1]. Dynamic taint analysis (DTA) addresses this challenge by tracking how sensitive data propagates during execution [2]–[4], enabling detection of data leaks and injection-style attacks with fewer false positives than static analysis [4]. Early DTA systems [5], [6] used binary instrumentation, incurring high overhead and limited portability. Later designs introduced shadow memory, which stores metadata alongside program data and was adapted to managed runtimes. For example, Phosphor [2] instruments Java bytecode to perform fine-grained taint tracking without modifying the virtual machine. TaintDroid [3] applies similar techniques to Android. DTA has since been used for debugging [2], verifying program behavior [7], and capturing runtime-dependent data flows that static analysis may miss.

However, dynamic taint analysis relies on fragile memory assumptions—most notably, that object layouts remain reliable and variables are accessed through consistent types. Techniques like shadow memory [8] and label mappings [9] hinge on this structural stability to correctly track data propagation. When these assumptions are violated, such as through type confusion, taint tags may be misapplied, corrupted, or silently discarded. This leads to undetected data flows, even when the control flow remains intact and the program logic appears correct. Such attacks are harder to detect, persist longer in production, and avoid triggering standard defenses, making them especially effective in gatekeeper scenarios like app store vetting [10] or enterprise code review [11].

Despite its potential impact, the use of type confusion as a method to evade dynamic taint analysis has not been systematically explored. In this paper, we present CONFUSETAINT, a technique that exploits type confusion to bypass taint tracking without altering control flow or causing observable runtime errors. Our key insight is that operations such as a wide-field write (e.g., to a `long`) can overwrite both a program variable and its adjacent taint tag in one operation, bypassing layout-based tracking mechanisms such as shadow memory. To achieve this, we coerce the runtime to reinterpret the type of an object—e.g., treating a variable of type `A` as `B`—and issue writes through the forged layout. This enables attackers to remove taint tags without triggering alarms, breaking the assumptions of many DTA systems.

To evaluate the effectiveness of CONFUSETAINT, we applied it to two representative taint tracking systems — Phosphor [2] and TaintDroid [3] — covering both Java and Android runtimes. We first confirmed each tool's baseline ability to track data flows, then introduced type confusion vulnerabilities via custom test cases. These were used to assess whether taint propagation still holds under layout reinterpretation. Our results show that all evaluated tools failed to detect leaks once type confusion was introduced. In each case, sensitive data reached sinks without triggering taint alarms. This highlights a structural weakness in current taint systems: their dependence on type-safe memory layouts can be reliably subverted via runtime reinterpretation.

Our findings expose a structural weakness in current taint tracking mechanisms, where type confusion can be exploited to corrupt taint metadata without detection. This suggests that widely adopted assumptions—such as type safety and layout consistency—may no longer be reliable foundations for secure

dynamic analysis. To address this risk, future systems must rethink how taint metadata is stored and safeguarded. Potential directions include separating metadata from program memory, leveraging hardware-based protection, enforcing runtime type integrity, and mitigating low-level vulnerabilities through platform hardening. To facilitate further research, we release our tool and test cases at https://github.com/software-engineering-and-security/ConfuseTaint.

II. Threat Model and Background

A. Threat Model

In our threat model, the target application runs in a managed runtime (e.g., JVM or Android Runtime) with memory and type safety guarantees such as bytecode verification and runtime type checks. However, field accesses are compiled to static offsets and are not type-checked at runtime.

We assume that the underlying operating system protections (e.g., Address Space Layout Randomization, ASLR [12]) and the DTA tool function correctly under normal conditions: taint flows from sources to sinks are tracked as intended. However, we do not assume that the application or runtime strictly adheres to the assumptions made by taint analysis tools—particularly type consistency and layout stability, which are often not enforced due to performance concerns.

The attacker cannot alter the runtime or the DTA tool, but can modify application-level code. This reflects real-world settings where dynamic analysis is used as a security gatekeeper—for example, Google Play Store employs dynamic analysis [10] to detect privacy leaks in mobile apps. In such contexts, attackers are limited to crafting seemingly benign code that evades detection. Instead of using low-level vulnerabilities to compromise the system or escalate privileges, our attack leverages them to stealthily bypass taint tracking, achieving longer persistence and reduced detection risk.

B. Dynamic Taint Analysis

Dynamic taint analysis (DTA) tracks how sensitive data propagates through a program at runtime. Data from designated sources (e.g., user input, location) is marked as tainted and propagated alongside execution. When data reaches a sink (e.g., file I/O, network), the tool checks whether it remains tainted to detect unauthorized flows. DTA has been applied in Java [2], Android [3], and JavaScript runtimes [13] for security enforcement.

To track taint, tools attach metadata using either shadow memory or tag maps. Shadow memory stores tags near data for low-latency access, while tag maps use separate structures (e.g., hash tables) for more flexible but slower tracking. Fig. 1(a) shows a piece of source code with a data flow leak, where the variable `a.p` receives tainted input from `getUserData()` and carries its tag through execution. When passed to `sendToServer()`, the tool inspects the tag to determine whether sensitive data has reached the sink. Fig. 1(b) shows a typical shadow memory layout.

C. Type Confusion Vulnerability

Type confusion is a memory corruption vulnerability that occurs when an object is accessed under an incompatible type. Although managed runtimes like the JVM enforce type safety, such guarantees can be bypassed, enabling low-level reinterpretation of object layouts without runtime errors [14]. These vulnerabilities are widespread and persistent: they affect 95% of OpenJDK versions (1.6–21.0.4) and 71% of Android versions (2.3–15), with lifespans up to nine years [15].

A common case involves treating memory allocated as one class as another with a conflicting layout. As illustrated in Fig. 1(b), an object of class `A` (with two `int` fields) is reinterpreted as class `B`, which defines a `long` field overlapping both `A.p_tag` and `A.q`. A write to `B.p` modifies both fields in a single operation, potentially corrupting adjacent taint tags stored in shadow memory. Due to JVM alignment rules, padding may shift the offset, but the overlapping region remains vulnerable to misuse.

III. Approach

A. Attack Preconditions

Dynamic taint analysis (DTA) tools associate metadata with program variables using layout-dependent schemes such as shadow memory. These mechanisms assume a stable correspondence between data and tags—a guarantee that breaks under type confusion. Shadow memory, in particular, relies on offset-based addressing schemes and is thus vulnerable to layout reinterpretation.

Managed runtimes like the JVM or Android Runtime enforce type safety via bytecode verification and runtime checks (e.g., `checkcast`, `instanceof`), but these do not validate field-level memory access. Once compiled, field loads become raw pointer arithmetic (e.g., `*(base + offset)`), with no dynamic layout enforcement. This creates a blind spot: if an object is reinterpreted using a type with wider or overlapping fields (e.g., a `long` over two `int`s), a single write can corrupt both data and its taint tag. Since the runtime trusts the declared type, it allows unchecked writes that bypass taint tracking without violating control flow or triggering runtime errors. Our attack hinges on this mismatch: taint tools assume structural integrity, but the runtime does not enforce it at the field level.

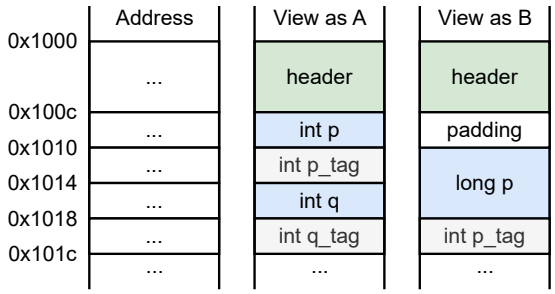
B. ConfuseTaint

Fig. 2 shows the overall workflow of our approach, ConfuseTaint. First, we inject an existing type confusion vulnerability into a known program (❶), and then apply a dynamic taint analysis tool to instrument the program (❷). Next, the instrumented program is executed on a standard runtime environment such as the JVM or Dalvik (❸), which traces data flows from source to sink. During execution, ConfuseTaint rewrites the taint tag, thereby disrupting the propagation of taint metadata. Finally, the DTA tool analyzes the execution and reports all observed tainted paths (❹).

During execution, ConfuseTaint exploits type confusion to reinterpret memory layout and block taint propagation. Specifically, a wide-field write (e.g., to a `long`) targets a

```
1  class A {
2      int p;       int p_tag;
3      int q;       int q_tag;
4  }
5  class B {
6      long p;      int p_tag;
7  }
8  public static void main() {
9      A a = new A();  B b = new B();
10     a.p = getUserData(); // Source
11     typeConfusion(a, b);
12     b.p = 0L;            // Overwrites a.p_tag
13     sendToServer(a.p);   // Sink
14 }
```

(a) Source Code of Taint Analysis Bypassed by Type confusion



(b) Layout reinterpretation under type confusion

Fig. 1: Taint tracking from source to sink: (a) source code, (b) memory layout. The `_tag` fields are inserted by the taint tracking tool as part of its instrumentation and are not present in the original source code.

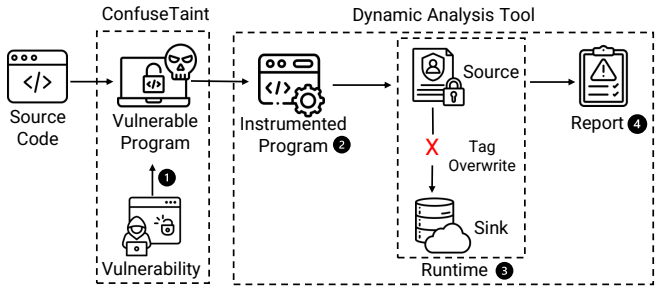


Fig. 2: Overview of the attack process

region expected to hold narrower fields (e.g., two `int`s and their taint tags), corrupting both data and adjacent metadata in one operation.

Fig. 1(a) shows a simplified example. Line 11 demonstrates a type confusion vulnerability where a `B` reference is coerced to point to an `A` instance. As a result, the assignment `b.p = 0L` on line 12 performs an 8-byte write that silently clears the taint tag of `a.p`, without violating type checks or triggering runtime errors. While this example uses integer and long fields, similar attacks apply across other primitive types and references, depending on layout and alignment.

This attack breaks a core assumption in shadow-memory-based DTA systems: that taint tags, placed at fixed offsets near program data, are protected by type-safe access patterns. By exploiting type confusion, an attacker can reinterpret object layouts and issue writes that overwrite both data and taint tags in a single operation. This violates layout-based isolation and bypasses taint tracking without triggering memory or type safety violations.

IV. Evaluation

We conclude our evaluation by answering the following research question:

- **RQ:** What is the effectiveness of our approach in bypassing dynamic taint analyzers?

A. Experimental Setup

We evaluate our approach on two dynamic taint analysis tools: Phosphor [2] and TaintDroid [3] , covering the JVM and Dalvik VM respectively. Experiments for Phosphor were conducted on MacOS 14.6.1 using Oracle JDK 1.8.0_111 and commit `e38e7d6` of the Phosphor repository. TaintDroid was evaluated using a pre-built image compatible with Android 4.3, running on an emulator in Ubuntu 24.04.1. Both tools were used in their latest available configurations without source-level modifications.

B. Experimental Design

We designed two versions of the program to evaluate whether dynamic taint analysis tools can track sensitive data flows and whether our attack can bypass such tracking. The baseline version implements a direct flow from a source to a sink. In TaintDroid, sources and sinks are defined explicitly in the code; in Phosphor, they are configured via a file. A taint flow is considered detected if the tool emits a verifiable propagation report—e.g., a policy violation warning in TaintDroid or a taint trace log in Phosphor.

We then constructed attack variants of each test program by injecting type confusion vulnerabilities while preserving original program semantics. For Phosphor, we exploit CVE-2017-3272 [16], which allows object memory to be interpreted under incompatible types. We allocate two objects to the same memory region and perform a forged field access to overwrite a taint tag. For TaintDroid, we apply a previously reported vulnerability [17] by modifying the smali-level intermediate representation and setting the `IS_CLASS_VERIFIED` flag in `classes.dex` to bypass Dalvik's bytecode verifier. The modified application is then recompiled into an APK and executed in an emulator running TaintDroid.

If the tool detects the baseline but fails on the attack version, this implies that the taint flow was bypassed without disrupting the underlying data flow. This demonstrates that our technique can disrupt dynamic taint tracking without altering the program's observable behavior.

C. Results

In both tools, the baseline programs correctly triggered taint flow detection: each tool identified sensitive data flowing from source to sink and raised the expected security alert. In contrast, the attack programs—constructed using our type confusion technique—preserved the same data flow but did not trigger any alerts. Table I summarizes the results. A checkmark (✓) denotes that the tool failed to detect the taint flow in the presence of our attack, even though the same flow was detected in the corresponding baseline case.

This indicates that our approach effectively removes or corrupts taint labels at runtime, bypassing the core detection mechanisms of dynamic taint analysis. The fundamental reason our attack succeeds is that taint tags stored as inline fields or in shadow memory implicitly rely on type safety and consistent memory interpretation at runtime. When type confusion violates these assumptions, tags become vulnerable to unintended overwrites. This suggests that dynamic taint analysis tools which embed metadata directly into program memory, especially those relying on shadow memory alignment, may be broadly susceptible to similar attacks.

Tool Name	Last Update	Runtime	Bypassed
Phosphor	2024	Oracle JDK 1.8.0_111	✓
TaintDroid	2013	Dalvik VM (Android 4.3)	✓

TABLE I: Results of CONFUSETAINT on Bypassing Dynamic Taint Analysis Tools

V. Discussion and Mitigation

A. Corrupting Taint Metadata: Beyond Type Confusion

While our attack focuses on type confusion, it is not the only way to corrupt taint metadata. Other memory safety issues – such as buffer overflows, format string vulnerabilities, or unsafe operations via JNI or `Unsafe` – can similarly overwrite taint tags stored in shadow memory or on the stack, without altering the associated program data. Even reflective access to taint-related fields (as seen in MirrorTaint [18]) may permit tag manipulation through plugin mechanisms or debug interfaces. These risks extend beyond the JVM and Android Runtime. Dynamic languages such as JavaScript, Python, and .NET CLR, as well as native platforms using binary instrumentation [4], [7], associate metadata with runtime values in similar ways. In all these settings, violations of layout integrity – through union casting, pointer reinterpretation, or reflective access – may silently corrupt taint tracking mechanisms.

B. Mitigation Strategies

To protect taint tracking under such threats, we outline several directions:

Isolated Metadata Storage: Replacing shadow memory with external tag maps decouples metadata from program memory. This separation prevents layout-based overwrites and allows language-level protection (e.g., access modifiers). The trade-off is additional runtime overhead from explicit lookups.

Runtime Validation: Runtimes can enforce type-safe layout access by rechecking class metadata during field loads. While standard JVMs rely on static verification, enforcing dynamic layout checks – as a defense against forged object views – would block many attacks, including ours.

Platform Hardening: Applying security patches (e.g., for CVE-2017-3272) prevents known memory reinterpretation attacks. Safer languages like Rust eliminate many root causes of taint bypass, offering stricter control over memory access and type safety.

Hardware Assistance: Capability-based architectures like CHERI [19] can enforce pointer bounds and object integrity at hardware granularity. These systems prevent reinterpretation of data layouts, but require platform support and nontrivial runtime integration.

VI. Related Work

Phosphor [2] tracks taint in JVM programs using bytecode instrumentation and shadow variables. TaintDroid [3] extends taint tracking to Android by propagating labels across variables, methods, and IPC within the Dalvik VM. MirrorTaint [18] mirrors JVM state in a separate heap for non-intrusive taint tracking without altering program metadata. Other systems such as ViaLin [20], DisTA [21], and TaintART [22] extend taint tracking to distributed or compiler-instrumented contexts. Despite architectural differences, these tools assume type-safe field access and stable object layouts—assumptions our attack explicitly violates.

Prior work on obfuscation has explored techniques to disrupt program analysis or enforce security through misdirection. Early studies [23]–[25] investigated Java bytecode obfuscation via polymorphism, identifier renaming, and opaque predicates. More recent efforts [26], [27] translated bytecode to native C or XOR-masked transformations to obscure type information. In Android, tools like DroidChameleon [28] evaluated how reflection-based obfuscation weakens anti-malware systems, while studies such as [29] empirically compared obfuscation usage in benign and malicious apps.

While prior work focuses on how obfuscation impacts static or dynamic analysis, few explore active attacks on runtime enforcement. We demonstrate that type confusion can subvert taint tracking by breaking type safety assumptions.

VII. Conclusion

In this paper, we present CONFUSETAINT, the first approach to bypass dynamic taint analysis in Java and Android by exploiting type confusion vulnerabilities. By reinterpreting memory layouts, CONFUSETAINT disrupts taint tracking while leaving program execution unchanged. Our evaluation shows that state-of-the-art tools like Phosphor and TaintDroid fail to detect leaks under our attack, exposing a blind spot in how DTA systems associate metadata with program state. We argue that taint tracking must account for adversarial memory reinterpretation. Future research should explore alternative metadata placement, enforce stronger runtime type integrity, and harden language-level abstractions against layout-level inconsistencies.

REFERENCES

[1] "Deserialization of untrusted data — owasp foundation," accessed: 2025-05-11. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data

[2] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. Portland Oregon USA: ACM, Oct. 2014, pp. 83–101.

[3] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, Jun. 2014.

[4] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Journal contribution*, 2005.

[5] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," in *21st Annual Computer Security Applications Conference (ACSAC'05)*. Tucson, AZ, USA: IEEE, 2005, pp. 303–311.

[6] D. Chandra and M. Franz, "Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. Miami Beach, FL, USA: IEEE, Dec. 2007, pp. 463–475.

[7] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. London United Kingdom: ACM, Jul. 2007.

[8] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 116–127.

[9] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Privacy scope: A precise information flow tracking system for finding application leaks," University of California, Berkeley, EECS Department, Tech. Rep. UCB/EECS-2009-145, 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-145.html

[10] Cloud-based protections — play protect. [Online]. Available: https://developers.google.com/android/play-protect/cloud-based-protections

[11] X. Fu and H. Cai, "Scaling application-level dynamic taint analysis to enterprise-scale distributed systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. Seoul South Korea: ACM, Jun. 2020, pp. 270–271.

[12] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 298–307.

[13] R. Karim, F. Tip, A. Sochůrková, and K. Sen, "Platform-Independent Dynamic Taint Analysis for JavaScript," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1364–1379, Dec. 2020.

[14] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. L. Traon, "Confuzzion: A Java Virtual Machine Fuzzer for Type Confusion Vulnerabilities," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2021, pp. 586–597.

[15] "On the presence of java type confusion vulnerabilities," Software Engineering and Security Group (SES), Umeå University, accessed 2025-08-19. [Online]. Available: https://github.com/software-engineering-and-security/TypeConfusionStats

[16] National vulnerability database - cve-2017-3272. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2017-3272

[17] J. Bremer, "Abusing dalvik beyond recognition," Hack.lu, Technical Report Hack.lu 2013, 2013, accessed: 2025-08-18. [Online]. Available: http://archive.hack.lu/2013/AbusingDalvikBeyondRecognition.pdf

[18] Y. Ouyang, K. Shao, K. Chen, R. Shen, C. Chen, M. Xu, Y. Zhang, and L. Zhang, "MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Melbourne, Australia: IEEE, May 2023, pp. 2514–2526.

[19] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 20–37.

[20] K. Ahmed, Y. Wang, M. Lis, and J. Rubin, "ViaLin: Path-Aware Dynamic Taint Analysis for Android," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. San Francisco CA USA: ACM, Nov. 2023, pp. 1598–1610.

[21] D. Wang, Y. Gao, W. Dou, and J. Wei, "DisTA: Generic Dynamic Taint Tracking for Java-Based Distributed Systems," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Baltimore, MD, USA: IEEE, Jun. 2022, pp. 547–558.

[22] M. Sun, T. Wei, and J. C. Lui, "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 331–342.

[23] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Tech. Rep., 1997.

[24] Y. Sakabe, M. Soshi, and A. Miyaji, "Java Obfuscation Approaches to Construct Tamper-Resistant Object-Oriented Programs," *IPSJ Digital Courier*, vol. 1, pp. 349–361, 2005.

[25] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "Towards experimental evaluation of code obfuscation techniques," in *Proceedings of the 4th ACM Workshop on Quality of Protection*. Alexandria Virginia USA: ACM, Oct. 2008, pp. 39–46.

[26] D. Pizzolotto and M. Ceccato, "[Research Paper] Obfuscating Java Programs by Translating Selected Portions of Bytecode to Native Libraries," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2018, pp. 40–49.

[27] D. Pizzolotto, R. Fellin, and M. Ceccato, "OBLIVE: Seamless Code Obfuscation for Java Programs and Android Apps," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Feb. 2019, pp. 629–633.

[28] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: Association for Computing Machinery, May 2013, pp. 329–334.

[29] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild," in *Security and Privacy in Communication Networks*, R. Beyah, B. Chang, Y. Li, and S. Zhu, Eds. Cham: Springer International Publishing, 2018, vol. 254, pp. 172–192.