Exploiting JDK11+'s Undocumented Behaviour to Boost the Performance of String-Encryption Based Obfuscation

Justus Elias Garbe
Independent
justus.elias.garbe@gmail.com

Shanyu Thibaut Juneja
Manning College of Information and Computer Sciences
University of Massachusetts Amherst
sjuneja@umass.edu

***Abstract*—In the evolving domain of software obfuscation, protecting sensitive information embedded within Java applications remains a paramount challenge. This paper explores the innovative utilization of the ConstantDynamic feature introduced in JDK 11, aiming to enhance the efficiency and invisibility of string encryption methods. By leveraging ConstantDynamic, we propose a novel approach that significantly mitigates the performance overhead typically associated with string encryption. Our method furthermore preserves class mapping whilst maintaining flow context-sensitivity. The empirical evaluation demonstrates a marked improvement in execution speed, with minimal performance degradation compared to traditional string encryption methods. These results not only confirm the effectiveness of our approach but also open new avenues for more secure and efficient string encryption in Java applications.**

1. Introduction

Java Obfuscation comprises of two main component: flow and constant obfuscation. Flow obfuscation seeks to hinder the reversing process and make the bytecode difficult to transpile back into source code using decompilers. Constant obfuscation, on the other hand, seeks to hide application secrets. The latter is the focus of this short study, in which, by exploiting Java's underlying functionality of the $ConstantDynamic$ instruction introduced in JDK 11, we effectively minify the string encryption overhead performance back to $O(1)$.

2. String Encryption

String encryption is the baseline of most secret-protection techniques employed by applications. This allows for URLs, endpoints, keys and most secrets to be stored directly inside an application with some degree of protection. It is undoubtedly recommended to always maintain any sensitive information behind an API of some kind, but in the context where applications are storing DRM and offline licensing, it becomes necessary to store such information directly into the application.

2.1. State of the Art

There have been several approaches to obfuscating and deobfuscating string encryption. Originally, first introduced by Zelix KlassMaster [1], the methodology consisted of simply calling a decryption function such that:

```
public static void main(String[] args) {
    System.out.println("Hello World");
}
```

Became the encrypted output:

```
public static void main(String[] args) {
    System.out.println(
        decrypt("iefzhiuzgiuezf", 123)
    );
}

public static String decrypt(
    String encrypted,
    int key) {
    // ...
}
```

This methodology was then mutated into various different forms, some of which run this routine at the class initialization (this is not recommended as it loses the capacity to use control flow context to harden the encryption), other forms which use direct caching, which is ineffective as Java Deobfuscator VM [2] can simply execute the function and return the cached field. Furthermore, adding additional fields may subsequently impact Reflections outputs and/or impact object serialization.

2.2. Objective

In this scenario, we seek to achieve a string encryption methodology which achieves all of the following criteria:

1) **Performant**: The performance overhead should be minimal and the decryption routine should only be ran once per constant.
2) **Invisible**: The class mapping (fields and methods) should remain unchanged.
3) **Context-sensitive**: To pair with opaque predicates and other obfuscation methodologies, the string encryption should be context sensitive at the control flow level.

3. Exploiting ConstantDynamic

ConstantDynamic, as per documented in the Java EIP, serves the following purpose:

> We seek to reduce the cost and disruption of creating new forms of materializable class-file constants, [...]. We do so by creating a single new constant-pool form that can be parameterized with user-provided behavior, in the form of a bootstrap method with static arguments.
>
> *Brian Goetz, Java JEP 309 [3]*

The behaviour is simple: store constants which are the output of functions directly in memory as opposed to mimicking the behaviour using the ¡clinit¿ method and fields. This however entirely overlooks one main symptom: flow context. We abuse this.

3.1. Integrating ConstantDynamic

Since Java does not evaluate the pure status of functions called by ConstantDynamic, we are able to effectively encrypt all strings, pass them into a common cache in the decryption process such that the encrypted output becomes instead:

```
public static void main(String[] args) {
    System.out.println(
        <constant_dynamic(
            <Handle(
                Main.class,
                "decrypt",
                "(String, Int) -> String"
            )>
            "iefzhiuzgiuezf",
            123
        )
    );
}

public static String decrypt(
    /* Handle args... */
    String encrypted,
    int key) {
    // ...
}
```

3.2. Performance Evaluation

3.2.1. Methodology. To evaluate this, we iterate an obfuscation testing program n times, over 100 different iterations i, such that $n = 100 \times i$. We measure the nanotime timestamp before and after the execution batch and cross-measure across three samples:

- **Control Sample:** Base evaluation jar
- **State of the Art Sample:** Control sample obfuscated with a string encryption routine with no caching
- **ConstantDynamic Sample:** Control sample obfuscated with an identical routine wrapped around the ConstantDynamic instruction.

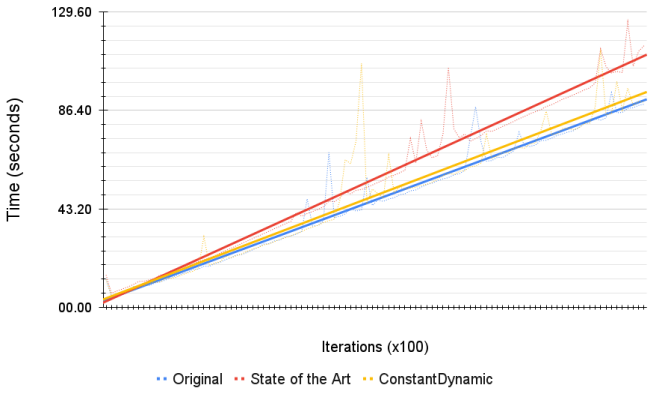

Figure 1. Performance results in seconds adjusted to compensate for Java Garbage Collection

3.2.2. Results. Figure 1 showcases nearly identical performance between the control sample without string encryption and the sample using ConstantDynamic behind a decryption call. On average, adjusted for garbage collection spikes exceeding a 20% difference, the string encryption algorithm, **without** ConstantDynamic, added **15.21%** overhead. However, by wrapping around ConstantDynamic, the overhead dropped to only **1.27%**. This significantly improves the performance of string encryption obfuscation techniques and allows for more controlled and computationally-intensive algorithms to be used safely.

4. Conclusion

The behaviour of ConstantDynamic is of a striking nature: despite having deterministic properties, it does not perform pure-function analysis checks on the bootstrap method. This allows for constants to originate from the same method, be dynamically loaded at class-time during the execution of flow, be context-sensitive and be cached. These properties render it as the perfect addition to be used in obfuscation, allowing for drastic performance increase whilst maintaining an impressive level of security.

Acknowledgments

The authors would like to thank the Recaf community, amongst all other users partaking in the discovery of this, for their extensive support, enthusiasm and kindness.

References

[1] Z. P. Ltd, “Zelix klassmaster,” https://zelix.com.

[2] S. C. Sun, “Java deobfuscator,” https://github.com/java-deobfuscator, 2022.

[3] B. Goetz, “JEP 309: Dynamic Class-File Constants,” OpenJDK JEP, 2018. [Online]. Available: https://openjdk.org/jeps/309