Mitigating Debugger-based Attacks to Java Applications with Self-debugging

DAVIDE PIZZOLOTTO, Osaka University, Osaka, , Japan
STEFANO BERLATO, DIBRIS, University of Genoa, Genoa, , Italy and Center for Cybersecurity, Fondazione Bruno Kessler, Trento, , Italy
MARIANO CECCATO, University of Verona, Verona, , Italy

Java bytecode is a quite high-level language and, as such, it is fairly easy to analyze and decompile with malicious intents, e.g., to tamper with code and skip license checks. Code obfuscation was a first attempt to mitigate malicious reverse-engineering based on static analysis. However, obfuscated code can still be dynamically analyzed with standard debuggers to perform step-wise execution and to inspect (or change) memory content at important execution points, e.g., to alter the verdict of license validity checks. Although some approaches have been proposed to mitigate debugger-based attacks, they are only applicable to binary compiled code and none address the challenge of protecting Java bytecode.

In this article, we propose a novel approach to protect Java bytecode from malicious debugging. Our approach is based on automated program transformation to manipulate Java bytecode and split it into two binary processes that debug each other (i.e., a self-debugging solution). In fact, when the debugging interface is already engaged, an additional malicious debugger cannot attach. To be resilient against typical attacks, our approach adopts a series of technical solutions, e.g., an encoded channel is shared by the two processes to avoid leaking information, an authentication protocol is established to avoid Man-in-the-middle attacks, and the computation is spread between the two processes to prevent the attacker to replace or terminate either of them.

We test our solution on 18 real-world Java applications, showing that our approach can effectively block the most common debugging tasks (either with the Java debugger or the GNU debugger) while preserving the functional correctness of the protected programs. While the final decision on when to activate this protection is still up to the developers, the observed performance overhead was acceptable for common desktop application domains.

CCS Concepts: • **Security and privacy → Software reverse engineering;**

Additional Key Words and Phrases: Anti-debugging, maliciuos reverse engineering, tampering attacks, man at the end attacks

This work has been partially supported by the MIUR "Dipartimenti di Eccellenza: Informatica per Industria 4.0" 2018–2022 grant. Stefano Berlato has been partially supported by "Futuro & Conoscenza Srl," jointly created by FBK and the Italian National Mint and Printing House (IPZS). Davide Pizzolotto has been supported by JSPS KAKENHI grant number 18H04094.

Authors' addresses: D. Pizzolotto, Osaka University, Osaka, Japan; e-mail: davidepi@ist.osaka-u.ac.jp; S. Berlato, DIBRIS, University of Genoa, Genoa, Italy and Center for Cybersecurity, Fondazione Bruno Kessler, Trento, Italy; e-mails: stefano.berlato@edu.unige.it, sberlato@fbk.eu; M. Ceccato, University of Verona, Verona, Italy; e-mail: mariano.ceccato@univr.it.

ACM 1049-331X/2024/04-ART107
https://doi.org/10.1145/3631971

ACM Reference Format:
Davide Pizzolotto, Stefano Berlato, and Mariano Ceccato. 2024. Mitigating Debugger-based Attacks to Java Applications with Self-debugging. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 107 (April 2024), 38 pages. https://doi.org/10.1145/3631971

1 INTRODUCTION

Software might contain valuable assets that an attacker could be interested in stealing, such as secret keys to decode paid live media streams. A program might also enforce certain usage constraints that an attacker may want to subvert or skip, such as a routine to check a license validity. All these attacks are known as Man-at-the-End attacks, because they aim at tampering with a piece of software that runs on a client that can be observed, analyzed, and eventually tampered with by the attacker.

Code splitting [6, 7] was proposed to mitigate malicious reverse-engineering and code tampering attacks by splitting a program into two components. A component is run locally on the end-user machine, while a second component, which contains the security-sensitive parts, is run on a protected server (e.g., in the cloud) where it can not be inspected nor tampered with. The overall correct program execution is guaranteed as long as the server part runs in parallel with the client part.

While moving (a portion of) the computation to the cloud might represent a definitive solution to hide a program from a potential client-side attacker, this solution involves some constraints. First, the application owner should sustain the additional cost for the cloud infrastructure, which would probably scale with the number of connected clients. Second, code-splitting requires the client-side application to be always connected and constantly communicating with its server-side counterpart. This might not be totally accepted in some contexts, especially when causing usability or latency problems and scalability issues. For instance, the gaming industry has shown a concrete example of problems with code-splitting, due to the always-online constraint and because it resulted in Denial of Service to legitimate users [18, 25, 29]. When the cloud cost is not compatible with the application business model or when the mandatory online constraint is not met, pure-offline protections are an option to mitigate code tampering attacks.

Existing studies [8, 9] with professional hackers and practitioners suggested that any tampering attack is preceded by malicious reverse-engineering activities to let an attacker understand the code, formulate assumptions, and plan for the actual attack. The same studies also highlight that, especially when code obfuscation [13, 14] is used to turn a program harder to statically understand and analyze, attackers prefer to start with dynamic analysis by concretely executing the program. Indeed, since the effort to undo obfuscation is considered not worth it, attackers typically consider it more effective to use a standard debugger to follow the program execution and to inspect its memory at certain points that they deem interesting. The adoption of protections against malicious debugging is also recommended by specialized organizations, for instance, by OWASP [27].

For these reasons, several *anti-debugging* approaches have been proposed [1, 2] to limit the possibility of using a debugger, usually by disrupting and messing up the interface that a running program offers to the debugger.

While such anti-debugging approaches have shown to be effective to protect compiled binary code, to the best of our knowledge, no approach is available to effectively protect Java programs. In fact, Java compiles to high-level bytecode that is quite easy to analyze and decompile [26], so anti-debugging protections would be easier to spot in the Java bytecode and, thus, to bypass (e.g., by removing or tampering with them).

Nonetheless, Java is a mainstream programming language, largely used in both open source and industrial projects [5]. Additionally, Java is often used to implement commercial software that includes license-check routines or intellectual-property-sensitive components, which could be the target of malicious reverse-engineering. Popular examples of this kind of license-protected Java software include design and modeling tools (e.g., IntelliJ IDEA Ultimate Edition,[1] JRebel[2]), enterprise and productivity programs (e.g., Camunda BPM,[3] Alfresco Process Service,[4] RapidMiner[5]) and scientific applications (e.g., MATLAB Compiler,[6] COMSOL Multiphysics Geneious[7]).

In this article, we propose a novel approach, namely, `conceal.it`, to protect Java programs from malicious debugging. Since Java code is quite challenging to harden due to its easily reversible nature, in our approach, we first translate (portions of) a Java application into C code that compiles to binary, and then we apply our anti-debugging protection both to the remaining Java code and to the binary executable. Our protection is inspired by the intuitive notion of self-debugging initially suggested by Abrath et al. [2], i.e., a program that debugs itself. Concretely, self-debugging consists in splitting the program into two processes, where each of the two attaches as a debugger to the other one. Since the most commonly used operating systems (e.g., Windows and UNIX-based) allow a process to have just one debugger attached, an attacker would not be able to attach her/his own debugger to perform malicious reverse-engineering.

The solution by Abrath et al. exploits the UNIX `fork` system call to create the second process (called *child*) from the first process (called *parent*). While their approach works on compiled code, it does not apply to our context. In fact, compiled C code is integrated in Java as JNI code. It runs within the Java Virtual Machine that, as a multithreaded process, is not fully compatible with the solution proposed by Abrath et al. based on UNIX `fork`.[8]

This article presents a novel approach to self-debugging with a more general applicability that overcomes the constraints of the forking mechanism. Instead of creating a copy of the first process with the `fork` call, we simply start a new, generic, external process. In this way, our approach can be applied to and protect a wider set of programs, including also JNI code in Java programs. However, larger applicability comes with the drawback of a larger set of potential vulnerabilities to mitigate because of a larger attack surface that could be exploited by a potential attacker.

For instance, an attacker could stop the creation of the second process to keep the debugging slot of the main program available for malicious debugging. Alternatively, an attacker could detach the second process without stopping it to free the debugging slot. The attacker could also intercept the point where the external process is started and the communication channel between the two processes is established to mount a sophisticated Man-in-the-middle attack, with the final objective of reading and tampering with the communication between the two processes to hide his/her malicious debugger.

Delivering a secure solution for anti-debugging of Java code, as well as any self-debugging mechanism that does not rely on the UNIX `fork`, requires addressing compelling research challenges. This article identifies and proposes a novel solution to address these two research challenges: (i)

[1] https://www.jetbrains.com/idea
[2] https://www.jrebel.com/blog/what-is-jrebel
[3] https://camunda.com
[4] https://hub.alfresco.com/t5/alfresco-process-services/getting-started-with-alfresco-process-services/ba-p/290016
[5] https://rapidminer.com
[6] https://www.mathworks.com/products/matlab-compiler-sdk.html
[7] https://help.geneious.com/hc/en-us/sections/360009220612-General
[8] According to the `fork` documentation, "after a `fork()` in a multithreaded program, the child can safely call only async-signal-safe functions" https://man7.org/linux/man-pages/man2/fork.2.html

ensuring integrity of the self-debugging protection and (ii) protecting the communication channel between the two self-debugging processes. As discussed in more detail in Section 2.2, previous research addresses the first challenge by splitting the code between the parent and the child—an approach that makes more difficult to break the self-debugging mechanism but may still allow an attacker to *statically* reconstruct the original code—while the second challenge is addressed by relying on (the peculiarity of) the UNIX `fork` mechanism and the special parent-child relationship between the self-debugging processes. A child process, in fact, inherits file descriptors (of, e.g., pipes and sockets) from its parent, and these descriptors can be used for establishing a dedicated communication channel. Moreover, parent-child processes can share memory segments—which are ideally not accessible by unrelated processes—over which they can communicate. Nonetheless, attackers with administrator privileges may still intercept and read (cleartext) data exchanged between the two processes.

In any case, the UNIX `fork` system call cannot be used in the more general case addressed in this article. Therefore (as we later show in Section 4.3), we design and propose novel strategies specifically addressing these two challenges in a more general context. More in detail, the novel contributions of this article are:

- A novel anti-debugging approach that is able to protect Java programs from both the Java debugger (e.g., JDB) and the binary debugger (e.g., GDB) at the same time;
- A completely open-source implementation of our approach[9] available to researchers and practitioners;
- Even if our anti-debugging approach is based on the known intuition of self-debugging, a series of novel contributions have been proposed to make it resilient against a number of attacks and satisfy the aforementioned research challenges. They are:
  - A novel program transformation from a Java function to a **Java Native Interface (JNI)** C program that is then split in two halves. These two halves must be run in parallel as two distinct processes to keep the original program semantics, so the attacker can not simply stop one of them to attach his/her debugger. In particular, while in existing approaches the protected code is moved either into the debugger or the debuggee, in ours, each original Java statement is transformed in an equivalent series of C instructions scattered between both debugger and debuggee. This prevents terminating any of the two processes even if the original unprotected code is composed by a single statement.
  - A novel communication scheme between these two processes that integrates the debugger system calls to make message exchange dependent on debugging through message masking. Thus, detaching the debugger by patching its system calls would make a process unable to communicate with the other one and cause the program to fail. Additionally, the mask used to encode messages changes after each message in a way that is hard to predict, allowing the two processes to safely use an external, unsafe communication channel.
  - A novel authentication mechanism between the two processes that uses process identifiers in a way that is hard for the attacker to replicate, thus preventing an attacker from mounting a Man-in-the-middle attack between the two processes.
- The empirical validation of the proposed novel approach on a set of 18 open-source Java programs with more than 1,000 stars on GitHub and at least 18 test cases each. It studies to what extent our approach prevents common debugging tasks in mainstream debuggers and measures what is the runtime overhead.

[9] https://github.com/davidepi/oblive

Our approach is implemented in an open source tool named `conceal.it`, which is fully automated and annotation-driven, thus it can be added as a transparent post-compilation step by editing existing build scripts (e.g., Maven[10] or Gradle[11]).

Despite the focus of this article being on mitigating the attacks based on malicious debugging, as emerged from the analysis on reverse-engineering activities [8, 9], one protection alone is too weak and should always be complemented with other protections, since an attacker is assumed to always follow the easiest path and attack the weakest point first. For instance, anti-debugging should be paired with code obfuscation [13, 14] and anti-tampering [35] to make protections protect each other. However, we highlight that these kinds of reverse-engineering protections are out of the scope of this article, which focuses on anti-debugging only.

The article is organized as follows: In Section 2, we discuss the novelty of our approach with respect to related work. After recalling some relevant background on debugging and automated translation from Java to C in Section 3, Section 4 and Section 5 illustrate our approach against binary-level and Java-level debuggers, respectively. Then, Section 6 presents our experimental framework and the empirical results, which are discussed in Section 7. Section 8 closes the article with final remarks and future work.

2 RELATED WORK

Given their actionable intrinsic nature, software protection techniques have a direct real-world impact, especially on programs that contain valuable assets to protect. For this reason, many protections are designed and developed by practitioners (e.g., published in GitHub[12],[13]), organizations (e.g., OWASP[14]), and enterprises (e.g., into commercial products like Guardsquare's ProGuard[15]). However, besides the industry, also academic researchers proposed a few original software protection techniques. Below, we present the most relevant work related to software protection available in the literature, with a specific focus on anti-debugging techniques at Java (Section 2.1) and native (Section 2.2) level. Thereafter (in Section 2.3), we present relevant work on code-splitting, an alternative approach to anti-debugging to hide the sensitive part of a program on a secure server where it can not be analyzed.

2.1 Java-level Anti-debugging Protections

The first set of anti-debugging approaches at Java-level is limited to determining the presence of a debugger (i.e., debugger detection). Most commonly, these approaches query existing **Application Programming Interfaces (APIs)** to detect a debugger, such as `IsDebuggerPresent` in Windows and `Debug.isDebuggerConnected` in Android. Other anti-debugging protections notice the presence of a debugger by detecting the typical debugging-related flags, e.g., the flags `-Xdebug`, `-Xrunjdwp`, or `-agentlib:jdwp`[16] in the command line arguments of the program under execution.

Lim et al. [22] propose an anti-debugging technique for Android coupled with root and method hooking detection. In detail, the authors base their protection on the `android:debuggable` flag in the manifest of the Android application.[17] However, the same authors noted that, as their protection is implemented in Java, it can be easily subverted and bypassed, e.g., by reversing and patching

[10] https://maven.apache.org/
[11] https://gradle.org/
[12] https://github.com/DimaKoz/stunning-signature
[13] https://github.com/CalebFenton/AndroidEmulatorDetect
[14] https://owasp.org/www-project-mobile-security-testing-guide/
[15] https://www.guardsquare.com/proguard
[16] https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html
[17] https://developer.android.com/guide/topics/manifest/manifest-intro

the code. For this reason, they extended their initial work by adding a native module written in C++ as a shared library [21]. The module checks the integrity of the protection implemented in Java, eventually detecting method hooking and code modification attacks by examining the call stack trace. Unfortunately, even if this follow-up work is implemented in native code, it still suffers major limitations. As the authors themselves acknowledge, an attacker may replace the entire native module and bypass the protection.

Another anti-debugging protection based on debugger-detection for Android was proposed by Wan et al. [37]. Using checkpoints for integrity verification, the authors analyze open-source tools for hookings methods and APIs (e.g., XPosed[18]). Their protection consists in asserting whether one of these tools is currently being used for debugging by ensuring that the value of the related checkpoint was unaltered.

These protections are generally simple to implement but, at the same time, easy to locate, bypass, or patch [21, 22]. Indeed, these APIs and flags are known and well documented, as they are originally conceived to be used by developers. Anyway, the effectiveness of these protections does not depend on the protection tool (which just detects the activity of a debugger). A major responsibility is left as a manual task to the developers, who are supposed to implement the reaction when a debugger is detected. For instance, after the detection of a debugger, the developers may immediately terminate the application (e.g., like in Reference [22]). Our approach is fundamentally different, because it is not just limited to *detecting* the presence of a debugger, but it actively *prevents* a debugger to attach to the protected program. As an example, consider the previously mentioned protection from Lim et al. [21]. While the authors themselves acknowledged that replacing the native module would bypass the protection, this happens because the protection module is used merely to detect the presence of a debugger. In our approach, instead, the protection module contains also program logic and needs to run to ensure a correct execution of the protected code.

Indeed, our approach belongs to the second set of Java-level anti-debugging protections, which aim at directly hindering the correct functioning of a debugger. Another approach belonging to this second set of protections was proposed by Matenaar et al. [24] against the **Java Debug Wire Protocol (JDWP)** for Java in Android applications. The authors exploit the fact that in the Dalvik Virtual Machine[19] there exists a global variable *gDvm* pointing to the debugger memory structure. Essentially, the proposed protection uses this variable to manipulate the low-level JDWP interface, blocking all debugging capabilities. Also Cho et al. [11] proposed a protection targeting Android, although considering native-level anti-debugging as well. The authors aggregate several debugger detection methods such as hash value comparison (i.e., to detect the insertion of breakpoints opcodes), time check, and dedicated flags (i.e., the `TracerPID` flag, which reports the process ID value of an eventual debugger). Again, the Java-level anti-debugging protection is based on the *gDvm* structure of the Dalvik Virtual Machine, as in the approach by Matenaar et al.

While these two protections are limited to a specific and obsolete version of Android (the Dalvik Virtual Machine was replaced by the Android RunTime starting from version 5.0 of Android in late 2014[20]), our approach is more generally applicable. In fact, our protection targets the Java debugger intending to completely block any debugging-related capability, as it will be shown in Section 6. Moreover, our anti-debugging protection is neither limited to the Android ecosystem nor relies on specific dependencies (e.g., the *gDvm* variable). Instead, as later described in Section 5, we rely on the consolidated **Java Platform Debugger Architecture (JPDA)** and on commonly used Unix system calls (e.g., `mprotect`).

[18] https://repo.xposed.info/module/de.robv.android.xposed.installer
[19] https://source.android.com/devices/tech/dalvik
[20] https://www.android.com/versions/lollipop-5-0/

2.2 Native-level Anti-debugging Protections

The idea of exploiting the limit of a single debugger for a given process was already considered both by practitioners[21][22][23],[24] and researchers [1, 2, 28, 38].

A self-debugging protection requires at least two processes, one of which (i.e., the mock debugger) debugs the other (i.e., the debuggee). However, the robustness of the protection is given by the degree of interdependence between the involved processes [2]. Indeed, a strong binding is needed between debugger and debuggee, so attackers cannot trivially replace the mock debugger with their own.

Xu et al. [38] first survey traditional protection schemes for Android apps to then propose their own software protection schemes. The anti-debugging techniques presented are similar to those considered in References [11, 22, 24] (i.e., flag-based). However, the authors also introduce the use of the `ptrace` system call for self-debugging. In their approach, the process forks and the child process debugs the parent process. However, an attacker can easily kill the child process and then start debugging the parent process.

The first proposal for a tightly coupled self-debugging protection was discussed by Abrath et al. [2] and later patented [33]. The authors discuss a self-debugging protection, which targets native code of Android applications only, in which fragments of code are moved from the original process to their mock debugger. This approach makes it harder for malicious reverse-engineers to replace the mock debugger and reconstruct the original unprotected program. The debugger and the debuggee processes communicate through exceptions (e.g., raised by breakpoints) to further hinder reverse-engineering activities. However, the use of exceptions is a major giveaway of the protection strategy. Moreover, migrated code fragments might be easy to identify. Finally, the mock debugger process is left unprotected. Hence, attackers could easily attach their own debugger to the mock debugger process to reverse-engineer it.

These shortcomings were addressed in consequent work [1, 28] that built its contributions directly on top of previous protection [2]. First, the authors replace exceptions with `sigsegv` signals thrown by illegal memory accesses. Then, they hide migrated code fragments using complex xor-masking. Finally, they allow reciprocal debugging between the debuggee and the mock debugger, thus creating a circular dependency between the two processes to further harden the protection.

Similarly to these approaches, we also propose a self-debugging solution to block malicious debugging. However, in the prior art, the mock debugger was created as a child process of the debuggee. This allowed previous protections to exploit a tight connection between parent/child processes and create a safe communication channel. Conversely, when working in a Java environment, an alternative, more general, process-creation strategy must be used that involves a more tenuous link between the two processes. This required us to elaborate a totally different protection to strengthen the bonding between the mock debugger and the debuggee. To achieve this objective, we proposed a novel automated transformation procedure to split the program into two halves, one with the computation and the other with a stack of data, so neither of them can be disconnected. Moreover, we proposed a custom messaging system among the two processes with a language that can change, whose messages are exchanged over an encoded communication channel and based on an authentication system. This novel approach allows mitigating several attacks, including the Man-in-the-middle attack.

[21] https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software
[22] https://blog.malwarebytes.com/threat-analysis/2013/07/zeroaccess-anti-debug-uses-debugger/
[23] https://anti-debug.checkpoint.com/techniques/interactive.html
[24] https://www.virusbulletin.com/virusbulletin/2008/12/anti-unpacker-tricks-part-one

2.3 Client-server Code-splitting

A complementary approach is represented by code-splitting. Instead of trying to protect a program from malicious static/dynamic analysis, the program is *split* into two components. A first component is still run on the end-user premises, where it could be subject to analysis and tampering. However, the security-sensitive parts of the program should stay on the second component that is run on a secure server (e.g., on the cloud), where no tampering and no detailed code inspection could happen.

Ceccato et al. [6] presented a fully automated approach to apply code-splitting protection to any arbitrary Java program. The developer is only supposed to decorate the code with custom annotations, to specify the program security requirements, in terms of what are the program variables that contain sensitive and non-sensitive data. Then, static program slicing is used to identify those program statements that are involved in the computation of sensitive data, because they could be subject to malicious analysis or tampering. These statements represent the sensitive portion of the program to be moved and executed in the cloud. Subsequently, automated code rewrite performs the actual split and changes the original client-only program into the two client-side and server-side components, with new statements and data structures to manage client-server synchronization and remote execution.

The portion of the computation that is moved to the server is secured against man-in-the-end attacks, but it requires constant connectivity to the internet (the protected client program would not run offline), and it involves the additional cost of the cloud infrastructure. In a subsequent work [7], a tradeoff between security and performance is investigated by changing the amount of code that is moved to the cloud. In fact, cloud cost could be reduced by moving part of the sensitive code back to the client, at the cost of a less secure solution. However, the sensitive part left on the client could be periodically replaced [10] to minimize the impact of attacks. This code replacement strategy fits very well with code-splitting, because a new version of the client component could be delivered that requires a new version of the corresponding server component. By stopping the expired versions of the server components, the application owner can easily disconnect all the clients that refused updates that might be under attack.

The level of security of different splitting configurations has been investigated by Viticchié et al. [36]. A user study has been conducted in a controlled environment to measure the correctness and the amount of time spent by participants to complete attack tasks on different client versions, analyzing which attack strategies were adopted depending on the split configuration to tamper with.

Remote attestation [3, 12] allows server-side verification of the integrity of client-side programs (e.g., the client program has not been tampered with). Code-splitting can be used to take actions when client integrity is not met. Remote attestation has been integrated with code-splitting in a novel approach called *reactive attestation* [34], i.e., the server component is stopped for those client components that are under attack, preventing the correct execution of tampered programs.

3 BACKGROUND

Debugging is one of the most prominent techniques used by developers and reverse-engineers to understand the functioning of and extract information from a piece of software [9]. Through debugging, a process (called “debugger” or “tracer”) can inspect the status and observe and control the execution flow of another process (called “debuggee” or “tracee”). Intuitively, debugging is widely used by software developers to identify defects in a program. However, debugging can also be used by malicious reverse-engineers and attackers to, e.g., extract valuable information, nullify security measures, and inject malware.

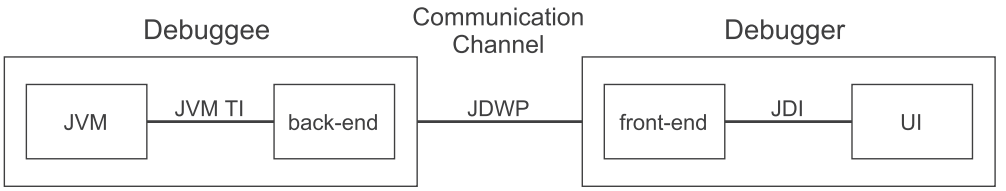

Fig. 1. The Java platform debugger architecture.

In this section, we illustrate how debugging occurs in Java (Section 3.1) and natively on Linux-like Operative Systems (OSs) (Section 3.2). Then, we give an overview of the *Java2C* tool [30] that we later employ (Section 3.3).

3.1 Debugging in Java

The JPDA defines the software modules needed for debugging Java applications. Figure 1 illustrates the JPDA for Java 11[25], which consists of two main modules—a debuggee and a debugger—split into two submodules each, two interfaces—**Java Virtual Machine Tool Interface (JVM TI)** and **Java Debug Interface (JDI)**—and the JDWP communication protocol:

- *debuggee* - It consists of the **Java Virtual Machine (JVM)** running the Java application being debugged together with the back-end submodule. The JVM TI is the native interface defining the expected services that a JVM should provide toward the back-end to allow the debugging of Java applications. In other words, the JVM TI is the interface through which the back-end submodule obtains debugging-related information from the JVM (e.g., current stack frames, breakpoint hit notifications). The back-end is usually written in native code and supplied as a native shared library (i.e., `.so` files in Linux-based OSs and `.dll` files in Windows-based OSs), like in the reference implementation by Oracle.[26]
- *debugger* - It consists of the front-end and the **User Interface (UI)** submodules. Through different connectors, the front-end can either wait for incoming connections or attach to an already running back-end. The JDI is the high-level (usually Java-based) interface offered by the front-end and used by the UI (e.g., an **Integrated Development Environment (IDE)**) to allow users to send commands (e.g., set breakpoint) and receive debugging information.

The debuggee and the debugger exchange messages over an abstract communication channel typically implemented either through sockets (for remote debugging) or shared memory (for local debugging). In both cases, the JDWP defines the semantic, syntax, and temporization of the messages exchanged between the back-end and the front-end. The JDWP is asynchronous, stateless, and it expects two basic packet types, i.e., command and reply packets. Command packets are used by the front-end to request debugging specific information (e.g., show the current stack trace, variables, or trace output) or control the execution of the Java application (e.g., through stepping and breakpoints). The back-end can also send command packets to notify the front-end of events (e.g., breakpoint hit). Commands sent by the front-end expect a reply packet, while commands sent by the back-end do not require a response packet.[27] Last, we highlight that the JPDA is highly modular. As such, developers and malicious attackers may provide custom submodules instead of using the Oracle reference implementation at any level.

[25] https://docs.oracle.com/en/java/javase/11/docs/specs/jpda/architecture.html
[26] https://docs.oracle.com/en/java/javase/11/docs/specs/jpda/jpda.html
[27] https://docs.oracle.com/en/java/javase/11/docs/specs/jdwp/jdwp-spec.html

3.2 Debugging Binary Code

The actual implementation of a native-level debugger heavily depends on the underlying OS (and kernel). In Linux-like OSs, debugging usually happens through the `PTRACE`[28] system call, which expects as arguments (at least) a command to execute and the **Process ID (PID)** of the tracee.

To start debugging, the tracer has first to attach to the tracee through either the `PTRACE_SEIZE` or the `PTRACE_ATTACH` commands. The former simply creates an attachment between the tracer and the tracee, while the latter also sends a `SIGSTOP` signal to the tracee, causing it to stop its execution. While the tracee is stopped, the tracer can use several debugging commands to, e.g., inspect the memory and influence the execution of the tracee. All commands are sent through the `PTRACE` system call. The tracer can then resume the execution of the tracee, usually after having defined further stopping conditions (e.g., a breakpoint or a watchpoint). Finally, the tracer can use the `PTRACE_DETACH` command to detach from the tracee. Another command worth mentioning is `PTRACE_TRACEME`, which is often used for self-debugging. In detail, a process can fork its execution and have the resulting child invoke the `PTRACE_TRACEME` command, which signals that the child is to be traced by its parent. Then, the parent can attach to the child using either `PTRACE_ATTACH` or `PTRACE_SEIZE`.

It is fundamental to highlight that, in Linux-like OSs, debugging works per thread and not per process. Moreover, the Linux kernel allows the presence of **one and only one** tracer, i.e., there cannot be two different threads that simultaneously debug the same thread.

There exist several debuggers targeting C and C++ executables in Linux-like OSs, one of the most commonly used being the GNU Debugger (GDB).[29] GDB offers a command-line interface through which it is possible to eventually start and then debug a running thread. Various UIs (e.g., gdbgui[30]) were developed to facilitate the use of GDB.

3.3 Java2c

The Java language, unlike languages such as C or C++, is not based on native CPU instructions depending on the CPU architecture. Instead, Java code compiles to an architecture-independent intermediate representation called Java bytecode. To run a Java program, its bytecode is interpreted and executed by the JVM. This interpreter is compiled for the specific CPU architecture of the system on which the Java program is meant to run.

In addition to this intermediate representation, the JVM allows calling C and C++ functions from within a Java program through the JNI framework. The purpose of this framework is to allow the execution of platform-specific code that can not be written in Java, either because it comes from an external library written in a different language or because it interfaces directly with the underlying operating system.

Java2C [30] automatically generates a C function that is functionally equivalent to the segment of the Java bytecode to replace, which is removed and changed into a call (via JNI) to the freshly generated C. Practically, this transformation works by parsing the Java bytecode and, for each bytecode instruction, a C fragment of code is emitted with one or few C statements that call function(s) provided as a library by *Java2C*. Of course, a different fragment is available for each Java opcode, with the objective that each fragment implements the same behavior as the Java opcode to replace. For instance, the Java bytecode opcode that pushes an integer to the JVM stack is replaced by a call to the *Java2C* library function that pushes the same integer value to a C local stack. This library and all the fragments have been manually written when developing the *Java2C* tool. After

[28] https://linux.die.net/man/2/ptrace
[29] https://www.gnu.org/software/gdb/
[30] https://github.com/cs01/gdbgui/

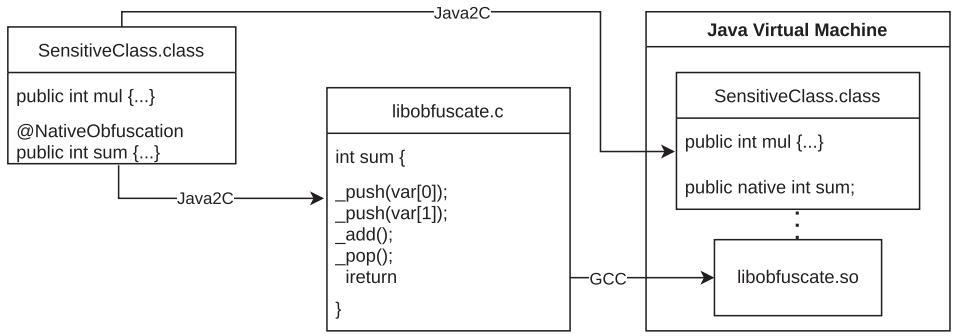

Fig. 2. Overview of the *Java2C* transformation setup: Sensitive methods are removed from the original class file and inserted into a C file. For each JVM opcode, a C function implementing similar behavior is created. The file is converted into a library and loaded into the JVM along with the modified `.class` file.

that, they are available as part of the tool, such that the translation can be applied automatically to a specified method with no need to manually write any line of C code.

Translating a portion of Java code to C would allow exploiting software protections that only make sense in low-level programs (e.g., binary code) and that would be of limited effectiveness [26] on a high-level program (e.g., Java bytecode) such as the anti-debugging techniques we are presenting in this article.

An overview of the transformation is shown in Figure 2. In particular, *Java2C* scans the compiled Java bytecode and looks for the particular annotation that signals the developer's intention to transform a method from Java to C. When such annotation is found, the tool removes the body of the method from the Java bytecode and adds the "`native`" modifier, meaning that the implementation of this method will be provided as a JNI library. *Java2C*, then, reads the opcodes in the method body and, for each statement, an equivalent C implementation is emitted. Finally, when the C source code is completely generated, it is compiled as a shared library. At runtime, when the transformed Java method is called, the JVM will load this C library that contains the transformed method body.

An example of this transformation is shown in Figure 3. The original Java code (left-hand side) just returns the sum of the two formal parameters, the method is annotated as `@NativeObfuscation` so it will be transformed by *Java2C*. It compiles to the Java bytecode in the center of the figure. The JVM adopts a stack-based architecture and each operation removes or adds an element from/to the stack. In this example, the `LOAD` opcode pushes the operand values to the stack, and the `ADD` opcode pops the two topmost elements from the stack, adds them, and pushes the results to the stack. The top of the stack is then used as the return value.

The translated C code (right-hand side) replicates this stack-based behavior by using a local C stack with the functions `_push` and `_pop`. The fact that the original Java bytecode and the generated C code are both stack-based will be the foundation of our anti-debugging approach, which will be presented in Section 4.

4 ANTI-DEBUGGING AT C LEVEL

The main objective of `conceal.it` is to provide anti-debugging for Java programs. However, this goal is very hard to achieve, because Java code compiles to Java bytecode, a high-level representation that is interpreted by the JVM. Unfortunately, the Java bytecode can be easily parsed and analyzed, and it would be quite simple for an attacker to locate and circumvent possible instructions used to block debugging [26].

A:12

```
class Calculator {
  @NativeObfuscation
  int sum(int x, int y){
      z=x+y;
      return z;
  }
}
```

(a) Original Java code

```
@NativeObfuscation
sum(II)I
ILOAD 1
ILOAD 2
IADD
IRETURN
```

(b) Compiled Java bytecode

```
#include <jni.h>
JNIEXPORT jint JNICALL Java_Calculator_sum(
JNIEnv *env, jobject thisObj, jint x, jint y) {
  jvalue vars[3]; //function inputs
  _push(vars[1]); //ILOAD 1
  _push(vars[2]); //ILOAD 2
  _push( (int)_pop() + (int)_pop()); //IADD
  return (int)_pop(); //IRETURN
}
```

Fig. 3. Transformation of a Java method used to perform an addition into a C method by means of *Java2C*.

For this reason, we adopt a different approach. Instead of preventing debugging at Java level,[31] we use *Java2C* to translate sensitive Java parts into C code. Then, we apply advanced anti-debugging techniques at C level, where they are harder to detect and circumvent.

We propose to apply two anti-debugging techniques at C level: Time-check and Self-debugging. Time-check is simpler, and it comes with a very low-performance impact. Self-debugging is more sophisticated, and it is expected to be more resilient at the cost of higher price in terms of execution time. These techniques are described in detail in Section 4.2 and Section 4.3, respectively.

To offer some protection also at Java level, a third anti-debugging technique is proposed and described in detail in Section 5.

4.1 Overview and Configuration

We can see an example of the protected code in Figure 4. The configuration of `conceal.it` is annotation driven, similarly to *Java2C*. Methods to be protected against malicious debugging should be annotated with `@AntidebugTime` or `@AntidebugSelf` to specify which anti-debugging technique to apply. These annotations have the `@NativeObfuscation` annotation as a dependency, because *Java2C* has to run before `conceal.it`.

After running the *Java2C* transformation (as seen in Section 3.3) additional steps are taken by `conceal.it`:

— **Preamble**: Some code is added at the beginning of the translated C code to activate the anti-debugging protection at C level, either Time-check or Self-debugging, and to activate anti-debugging at Java level (as explained later in Section 5).

[31] Note that our tool offers protection against debugging from the Java layer, but the entry point for the debugging protection is in the C layer.

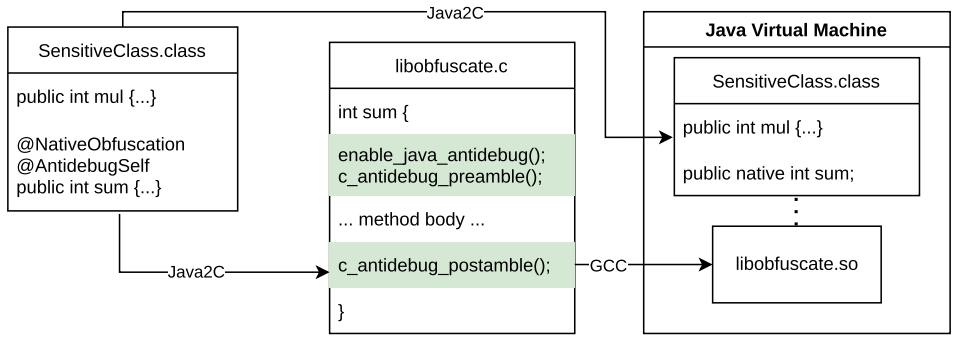

Fig. 4. Additional code produced by the `@AntidebugTime` and `@AntidebugSelf` annotations compared to the original *Java2C* structure presented in Section 3.3.

— **Postamble**: Some code is appended at the end of the generated C code to deactivate the anti-debugging protection at C level and to deal with recursion before exiting the C scope and returning to Java.

The preamble and postamble code differ between Time-check or Self-debugging, as we explain in the following in their respective sections.

The decision on what methods to annotate, which means what code to protect, is not automated and is delegated to the developers. Indeed, this decision should be based on the security requirements specific to each program that are known only to the developers. However, general considerations can help and guide in making this decision. In general, we suggest deploying anti-debugging protections on those portions of a program that manage valuable assets that regulate payments or implement critical parts of business logic that an attacker might be interested in inspecting or changing. For instance, the following functionalities may reasonably be deemed to be worth protecting against malicious debugging:

— Checks on commercial licenses to enable or disable paid features;
— Decisions on the activation of premium features or features that should be available only in specific environments;
— Handling of sensitive data (e.g., credit card numbers) and personal information to be protected according to the applicable regulations and laws, like the General Data Protection Regulation[32] in the European Union;
— Execution of cryptographic functions and management of secret keys and digital certificates;
— Unlocking of achievements and paid features in games (e.g., levels, goals, special items);
— Other complementary protections, such as anti-tampering, that should be used in combination with anti-debugging.

We remark that, once the annotations are manually added, all the remaining steps to apply anti-debugging are totally automated.

4.2 Time-check

The Time-check anti-debugging technique is based on the idea that, at debugging time, an attacker performing step-wise execution across instructions is expected to cause a noticeable execution time delay. Detecting this delay is an evidence to reveal debugging attempts, so reactions may be taken to prevent malicious analysis, such as program termination or execution of alternative code to protect sensitive computation.

[32] https://gdpr-info.eu/

In particular, our approach is the following: In the preamble of the generated C code, we insert a call to record the current time. Then, whenever we want to check if the program is being debugged, we collect the new current time and compare it to the time recorded in the preamble. If the elapsed time is longer than a threshold, then we assume that the program is under debugging and it should react accordingly. By default, this check is performed at least once in the postamble. Another option is to add time checks more frequently, after each opcode translated by *Java2C*.

This approach is quite simple and comes with a small performance overhead. However, it is expected to be not very resilient. In fact, most time functions are calls interfacing with the underlying hardware clock, and these functions might be statically detected or dynamically intercepted by an attacker and then subverted. For this reason and its overall simplicity, we use this method as a baseline for resilience and performance comparison.

4.3 Self-debugging

The self-debugging is an anti-debugging technique based on the constraint that only a single debugger can attach to the debuggee program. This constraint holds in all Unix-like and Windows operating systems. The underlying idea is that the debuggee attaches to and debugs itself, so no other additional debugger can attach. Thus, an attacker would not be able to attach his/her own debugger and perform malicious analysis. This intuition was first proposed by Abrath et al. [1, 2], which attached a custom debugger to the program to protect to prevent additional debuggers. Moreover, they moved a part of the original program to this custom debugger to make the protected program fail in case the custom debugger was detached in the attempt to attach the attacker's debugger.

Their approach, however, is generally aimed at C code and cannot work for Java applications. While a combination of existing tools may seem a trivial solution, i.e., *Java2C* to translate Java to C and then Abrath et al. anti-debugging to protect the translated C code, this would be a quite weak solution that would be vulnerable both to static-analysis attacks and to dynamic-analysis attacks.

In fact, the anti-debugging approach proposed by Abrath et al. relies on `fork` to start the second debugger process, and this is not compatible in general with JVM. While this method has been used successfully in Android (Dalvik virtual machine), we could not replicate the same result generically, e.g., on desktop JVM. The alternative technical solution involves a more tenuous link between processes, because it consists in spawning the companion debugger process using the Java ProcessBuilder API and an external communication channel requires to be established that is very insecure and could be the target of multiple attacks.

To deliver a novel solution that overcomes these serious limitations, two major research challenges need to be addressed, respectively, ensuring integrity of the self-debugging protection and the security of their communication channel.

Research challenge 1 - At any moment during the execution of the protected code, the two self-debugging processes must be debugging each other. An attacker must not be able to compromise the integrity of the self-debugging protection by detaching or tampering with one of the two processes.

An attacker must not be able to compromise the integrity of the self-debugging mechanics by detaching or swapping the debugger or the debuggee with a malicious process. Moreover, although the main objective of anti-debugging is mitigating debugger-based malicious analysis, we note that an anti-debugging protection should not be trivially circumvented by an attack that statically identifies where the protection operates and patches the code to remove or disable such protection. To address this research challenge, we propose these novel strategies:

— Mitigate an attack that aims at terminating our debugger process by splitting the program code between the debugger and the debuggee and by making them actively communicate to run in sync.
— Mitigate an attack that aims at detaching the debugger by using the debugger interface as an active communication channel, required to run the protected code.

Similarly to Abrath et al., we also split the original computation between the debuggee and the debugger, such that they are both mandatory for the overall computation to remain correct. In fact, in case the debugger process is not running, the computation will be incorrect. However, the way the program is split is completely different than the approach proposed by Abrath et al., because we intend to mitigate various termination attack scenarios. Additionally, different attacks might aim at tampering with protected code (e.g., remove system calls related to debugging) to still have the debugger and debuggee run, but detached from each other, and make the debugger slot available for the attacker. To mitigate these attacks, we use the debugger interface to exchange the data needed to complete the computation. Thus, to preserve the correctness of the execution, the debugger and the debuggee not only need to run, but they need to run and engage each other's debugging interface; otherwise, they would work on wrong data.

Research challenge 2 - Dynamic analysis should not easily decode or forge messages in the communication between debugger and debuggee.

When static analysis and static tampering are hard, attackers typically move to dynamic analysis [8]. Assuming that program execution can not be traced due to anti-debugging, an attacker is left with the option to study the communication between the debugger and the debuggee processes. For this reason, the second research challenge is about mitigating attacks on this communication protocol. To address this research challenge, we adopt the following strategies:

— Message replay attacks are mitigated by using a one-time pad approach;
— Man-in-the-middle attacks are mitigated by mutual debugger-debuggee authentication.

An attacker might run the protected program a first time to record those messages that are exchanged between the debugger and the debuggee. Then, the attacker might perform a second execution in a tampered configuration (e.g., the original debugger is detached) but replay the previous messages to hide the attack. To prevent this attack, messages are encoded with a mask that is different for each message and hard to predict, so old messages can not be reused, because their mask will expire immediately after they are captured by the attacker. If an attacker is able to observe the sequence of masks used to encode a sequence of messages, then she/he might try and guess the next mask and adapt an expired message with the guessed mask. To mitigate this attack, we turn the mask sequence hard to predict by skipping some masks in the sequence.

Eventually, the attacker might deploy a complex attack setting in which two new processes pair, respectively, with the original debugger and debugee, to preserve the self-debugging pattern but allow dynamic analysis. To mitigate this attack, our debugger and debuggee authenticate each other before attaching to the other process.

To summarize, with our novel approach, we are tailoring the self-debugging intuition to exploit the specificity of JVM and the peculiarities of C code translated by *Java2C* to overcome these vulnerabilities and deliver a solid and effective anti-debugging solution. In fact, given that JVM code is structurally different from typical C source code, because *Java2C* outputs C code that resembles the stack-based nature of Java bytecode, we can exploit this fact to adapt anti-debugging specifically to a stack-based C program.

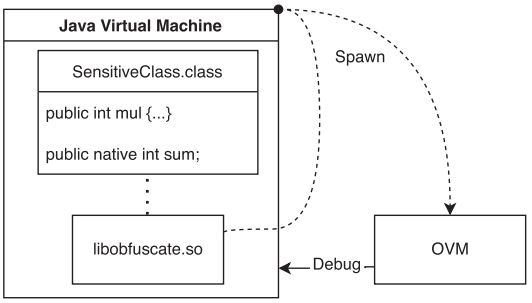

Fig. 5. Basic idea of the self-debugging technique. The protected code, `libobfuscate.so`, requests the JVM to spawn an additional process, **Obfuscation Virtual Machine (OVM)**, that will attach to the JVM itself.

In the following, first, we present the base idea, and then we gradually refine it by explaining, one-by-one, all the technical solutions that are needed to prevent those problems and vulnerabilities that the trivial concatenation of *Java2C* and C anti-debugging would involve.

4.4 Overview

The intuition of self-debugging by Abrath et al. [2] consists of running an additional process whose purpose is to attach as a debugger to the program to protect. We use *Java2C* as starting point to translate Java code to C, so the entry point of the code to protect is always a JNI function that is generated by *Java2C*. Our objective is to spawn the additional process and attach it as a debugger to the process of the program to be protected using `ptrace`, which requires the PID of the debuggee process. The standard procedure in POSIX systems to spawn a new process would require calling `fork` to create the new process and then call `exec` to load an alternative executable code in the fresh process. However, due to the multithreaded nature of the JVM and the lack of thread-safety in `fork`, we must opt for a different approach. Instead, we use the Java `ProcessBuilder` class, called by the C code, to spawn a new process and execute what we call the OVM, an additional program, generated as part of the *Java2C* transformation, acting as a debugger. Additionally, we pass as a parameter the PID of the C code running in JNI, obtained with `getpid`. The OVM, in turn, will use this PID and attach to it using `ptrace`, so no additional debuggers can attach.

Figure 5 shows an overview of the protection. The anti-debugging protection at C level is at the beginning of the translated C code (`c_antidebug_preamble()` function in Figure 4), whereas the OVM code is a separate component. Thus, after running `conceal.it`, we obtain three files: the modified Java bytecode, the compiled C code as a JNI library, and the compiled OVM executable.

However, this simple protection scheme would be vulnerable to several attacks to circumvent it, either based on static tampering or dynamic analysis (e.g., man-in-the-middle between the two processes). We elaborated a novel solution to address these two research challenges, summarized in the following and described in detail in the rest of this section:

First, an attack could just terminate the OVM to free the debugging slot of the program under attack. To prevent this attack, we establish a strong link between the JNI and the OVM by moving a part of the original program from the JNI part to the OVM. In this way, disconnecting the OVM would cause an incorrect execution of the program.

To limit potential static analysis attacks to the OVM, the representation of the OVM language is not constant, but it changes randomly. However, we note that this does not remove the need for a strong obfuscation component, as we describe in Section 7.2.

Considering that the OVM cannot be removed, an alternative attack to free a debugging slot could be just removing the `ptrace` system calls. To mitigate this attack, we made sure that the

`ptrace` call cannot be removed by assigning it the responsibility to share a decoding mask between the OVM and the JNI part, needed to decode their messages. In case the `ptrace` is removed, the two processes would miss the decoding mask and they would not be able to communicate and complete the overall execution.

An attacker might intercept some messages between the OVM and the JNI and conduct a plain-text attack to guess the decoding mask. To prevent this attack, we decided to change the mask at each message. In this way, even if a mask can be guessed, it would be useless to attack the rest of the communication.

Additionally, to prevent the attacker from guessing the sequence of masks, some masks in the sequence are skipped, adding non-linearity to the sequence value generation.

Eventually, an attacker might not change the code but just position herself in the middle and intercept and tamper with messages. To prevent this attack, we enforced that the OVM and the JNI authenticate each other in a way that the attacker cannot mimic.

4.5 Research Challenge 1: Ensuring Protection Integrity

Addressing the first challenge consists in mitigating those attacks aiming at freeing the debugging slot during the execution of the protected code, which either terminate the debugging process or patch the debugging system call. In the following subsections, we explain how we mitigate an attack that terminates the debugging process by splitting the necessary execution code between the two processes, JNI and OVM, in our case. Subsequently, we mitigate an attack that patches the debugging system call by exploiting the debugging-dependent interface between these two processes to exchange execution data. In fact, this interface works as communication channel as long as the two processes are still debugging-attached.

4.5.1 Overview of the Computational Split to Mitigate OVM Termination Attack. There is no stable communication between the JNI part and the OVM, only the initial PID is exchanged. Thus, the protected process cannot tell if the OVM has been successfully spawned and attached to the JNI. This means that, even if an attacker cannot attach a debugger *after* the invocation of the protected method, because a debugger would be already attached, the attacker may still circumvent this protection by attaching a debugger *before* the invocation of such method. In that case, the OVM would fail to protect the JNI and the JNI would continue anyway. More simply, due to the lack of communication, an attacker can simply kill the OVM process without any side effect on the JNI part and commence debugging.

To strengthen the bonding between JNI and OVM, we want to ensure the correct activation of the OVM debugging before executing the protected method. This means that passing the JNI PID to the OVM is not sufficient, but we must wait for an answer from the OVM to acknowledge the correct initialization of the `ptrace` call. For this reason, we need a communication channel between the two processes. In our implementation, we used a single IPC socket, but any other bidirectional communication channel is compatible with our scheme.

Figure 6 shows the initial setup between the JNI and the OVM. Initially, the JNI creates an IPC socket with a random name,[33] then it spawns the OVM process, passing the socket name as a parameter. The OVM immediately connects to the socket and replies with its PID. The JNI can now ptrace the OVM process and, if successful, send its PID. The OVM will then ptrace the JNI and send an ACK message in case of success. At this point, both processes are debugging each other. If any ptrace call fails, then the failing process will stop sending responses, leaving the other

[33]This was done to prevent clash of names if multiple obfuscations run at the same time. If the OS supports it, then a better approach would be to use abstract sockets that do not exist on the filesystem and are thus harder to attack.

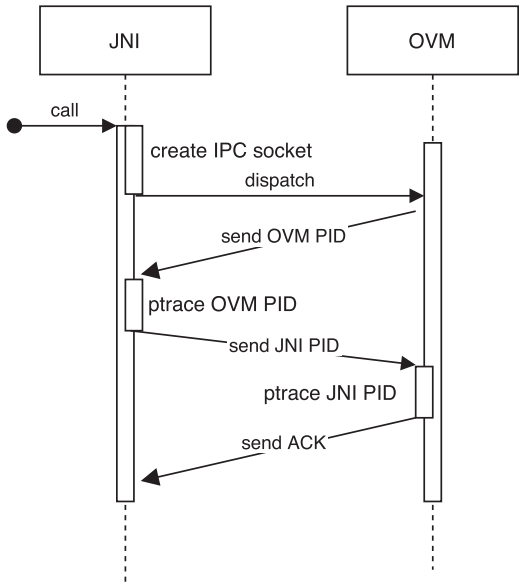

Fig. 6. Message exchange when spawning the OVM.

process to wait indefinitely on the socket. Note that in our implementation IPC sockets are always reliable, as defined in the manual.[34] However, if this is not the case, then additional engineering effort is required to handle lost messages.

Despite this setup, the OVM is not essential for a correct execution: An attacker can either kill the OVM process after the setup and send random data to the socket to let the JNI side continue its execution. This problem can be solved only by having an OVM that is mandatory for the correct execution of the program. To achieve this, we exploit the fact that the C code translated by *Java2C* closely follows the stack-based JVM style. In particular, in the JVM there are no CPU registers: A stack is used to store and retrieve data across statements [23]. Each statement consumes zero or more values from the stack and pushes zero or more values to the same stack, effectively using it as temporary memory. We split the translated C code such that opcode execution is separated from data access: We move the stack to the OVM side while keeping the computation in the JNI part.

To better visualize how the splitting transformation is applied, we consider again Figure 3, which shows the code after applying *Java2C*. We can notice that the addition opcode (`IADD`) pops two values from the stack and pushes the sum result back onto the same stack. The two popped values correspond to the addition operands (`ILOAD1` and `ILOAD2`), and the pushed value corresponds to the result, to be later popped by the function return (`IRETURN`). Every opcode thus needs to pop some input or push some output to/from the stack. According to Java specification, this holds for every single non-reserved opcode.[35]

To effectively split this (translated) C code, we separate the stack and the *Java2C* opcodes implementation in two different processes. Then, we replace every `push` and `pop` call with `send` and `recv` to communicate data between the two processes. This is done by modifying all the *Java2C* library

[34] https://man7.org/linux/man-pages/man7/unix.7.html

[35] The only exception to this is the `BREAKPOINT` opcode. However, being used only for debugging, *Java2C* does not support this opcode.

Table 1. OVM Instruction Set

Command	Description
STACK	Creates a stack with the given size. Can be nested in case of recursive calls.
KILL	Destroys the stack and, if no other stacks are left, terminates the OVM.
ACK	Returns a result. The OVM will never reply with anything but ACK
CLR	Resets the stack (set pointer to 0). This command is particularly useful when raising an exception.
FRONT	Returns the top value of the stack, without removing it
PUSH	Same as the JVM PUSH
PUSH2	Same as the JVM PUSH2
POP	Same as the JVM POP
POP2	Same as the JVM POP2
DUP	Same as the JVM DUP
DUP2	Same as the JVM DUP2
DUPX1	Same as the JVM DUPX1
DUPX2	Same as the JVM DUPX2
DUP2X1	Same as the JVM DUP2X1
DUP2X2	Same as the JVM DUP2X2
SWAP	Same as the JVM SWAP

functions that mimic the JVM opcodes. Communication between the two sides is done through messages passing over the IPC socket: The JNI side sends a command (1 byte) and a parameter (8 bytes) and waits until the OVM responds with the ACK opcode and the result (8 bytes).[36] These messages require the creation of a new Command Set for the OVM, shown in Table 1. The implementation of the OVM now involves repeatedly waiting for a message in the IPC socket, decoding the command and parameter, executing it, and returning the answer.

The example of code in Figure 3 is eventually split in the code in Figure 7. On the right-hand side of Figure 7, we can see how the OVM is essentially operating as a virtual machine. Data is fetched by the `_recv` call, decoded into instruction and payload and executed by the `switch` statement. Then, the result (if any) is encoded and sent back. On the left-hand side, the JNI performs a series of send and recv to get and set the intermediate results of every opcode and execute the opcode itself (i.e., performing the addition after getting the operands from the OVM or calling a JVM function after getting its parameters). The initial call `STACK` and the final call `KILL` are used, respectively, to set up and tear down the OVM. We can note how each `_send` and `_recv` call in the JNI part of Figure 7 corresponds to a `_push` or `_pop` in the transformed code of Figure 3.

As an additional hardening measure, the binary value assigned to each opcode is randomly decided at compile time. This randomization is meant to prevent an attacker from building a potential C2Java tool to reverse the transformation. In fact, this split execution adopts a certain opcode/payload mapping, i.e., each *Java2C* opcode is transformed in a call to `send()` with peculiar data in its payload. Randomizing the opcode/payload map means that the same operation in different builds will use different payloads. This would prevent an attacker from reusing the opcode/payload mapping guessed in a program to automatically attack another program (or a different build of the same program), thus mitigating the risk of a generic automated C2Java translator. This possible reverse translation attack is explained in detail in Section 7.2 under "Reversing the protection."

[36]The JVM stack can store only values up to 4 bytes, but some opcodes (i.e., PUSH2, POP2) can push or pop up to two values from the stack, hence why the 8 bytes parameter.

```
#include <jni.h>
JNIEXPORT jint JNICALL ... {
  jvalue vars[3]; //function inputs
  _send(_encode(STACK, 2));

  /* ILOAD 1 */
  _send(_encode(PUSH, vars[1]));
  _recv(); //wait for ACK
  /* ILOAD 2 */
  _send(_encode(PUSH, vars[2]));
  _recv(); //wait for ACK
  /* IADD */
  _send(_encode(POP));
  int val1 = _recv();
  _send(_encode(POP));
  int val2 = _recv();
  int res = val1 + val2;
  _send(_encode(PUSH, res));
  _recv(); //wait for ACK
  /* IRETURN */
  _send(_encode(POP));
  int retval = recv();

  _send(_encode(KILL));
  return retval;
}
```

(a) JNI side

```
void ovm_main() {
  jvalue* stack;

  while(true) {
    OvmOpcode opcode;
    uint64_t payload;
    uint64_t retval = 0;
    RecvData data = _recv();
    _decode(&opcode, &data);
    if(opcode == KILL) {
      free(stack);
      return;
    }
    switch(opcode) {
      case STACK:
        stack = malloc(payload);
        break;
      case PUSH:
        _push(stack, payload);
        break;
      case POP:
        retval = _pop(stack);
        break;
      ...
    }
    _send(_encode(ACK, retval));
  }
}
```

(b) OVM side

Fig. 7. Pseudocode showing the *Transformed code* of Figure 3 after the splitting transformation of `conceal.it`. The STACK and KILL system should be multidimensional to support recursive calls, but the code here is simplified.

In conclusion, this split execution mitigates those attacks aiming at terminating the OVM, as the JNI requires it to store intermediate results until the very last operation.

4.5.2 Exchanging Data with Debugger to Mitigate Code-patching Attacks. Considering that the debugger makes use of `ptrace` which is a system call, calls to `ptrace` could be spotted by an attacker and patched away. In fact, an attacker may simply replace the `ptrace` invocation with NOOP (i.e., null operation) and attach his/her own debugger to the JNI layer. Removing the OVM is not possible due to the split execution, however, it is still possible to patch just the `ptrace` and have JNI and OVM communicate without the need of debugging each other, thus circumventing our protection. For this reason, we are going to introduce a way to turn the `ptrace` mandatory for a correct execution, such that an attacker patching this system call will face crashes or wrong executions, i.e., use the `ptrace` to exchange data between the two processes.

To fulfill this requirement, we exploit the `PTRACE_POKEDATA` that allows a debugger to modify the value of a variable in the tracee process. Our approach consists of making the JNI and OVM exchange encoded data using the IPC socket (e.g., using XOR masking) with the data being decoded using the mask generated at runtime and shared via `PTRACE_POKEDATA`.

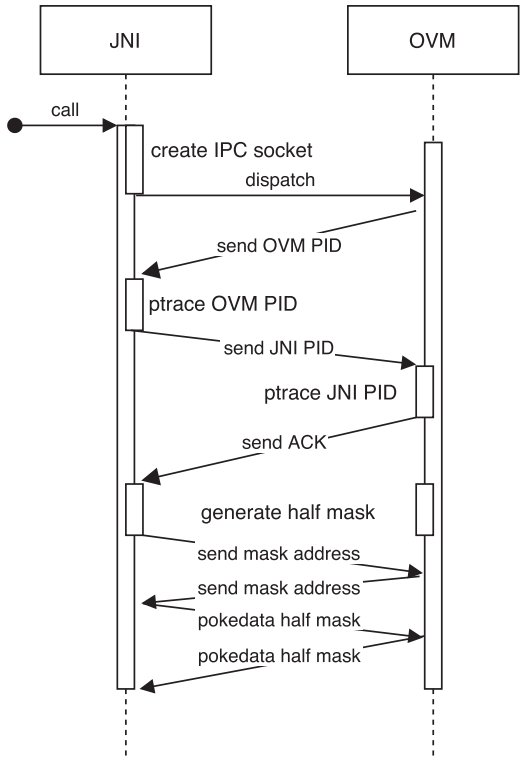

Fig. 8. Message exchange when spawning the OVM. Including mask generation for mandatory ptrace.

In case an attacker replaces the `ptrace` call with NOOP, the decoding mask will not be shared correctly because the `PTRACE_POKEDATA` call would fail, so the encoded exchanged data would be useless and the computation of the attacked program would fail. In addition, to ensure the two processes are debugging each other, the overall mask is computed by both processes: The first half mask is randomly generated by the JNI, and the second half mask is randomly generated by the OVM.

Figure 8 shows the updated message exchanged when spawning the OVM process. We can notice that, after the `ptrace` call, both processes exchange the mask address via the IPC socket, but they communicate the mask value via `PTRACE_POKEDATA`. After this initial step, every message exchanged on the IPC socket can be encoded, and they will be decoded on the other side. Thus, in case a wrong mask is used, because of a missing `ptrace`/`PTRACE_POKEDATA` removed by an attacker, exchanged messages would be incorrect and the program cannot execute.

4.6 Research Challenge 2: Preventing Attack to the Communication Channel

The second challenge is to secure the communication channel between the two self-debugging processes. First, to prevent message replay, we decided to encode each message with a different mask, so each message is valid only once and can not be reused by an attacker. Second, the two self-debugging processes mutually authenticate to prevent a man-in-the-middle attack, in which the attacker analysis process pretends to be one of our two self-debugging processes.

4.6.1 One-time Pad to Mitigate Known Plaintext Attacks. The first message sent by the JNI to the OVM always contains the `STACK` opcode with a small operand value (i.e., the stack size). This

highly predictable message could be starting point of an attack. In fact, an attacker could guess the decoded version of (most of the) first message and intercept the encoded version of it. This information allows the attacker to attempt a **Known-plaintext attack (KPA)**. With this attack model, an attacker may guess the mask used to encode/decode messages and to make the `ptrace` mandatory. If the mask is known to her/him, then the attacker could unravel the protection by patching the `ptrace`. This is mainly because, as explained in Section 4.5.2, the mask is generated once when spawning the OVM and then reused for every opcode.

To overcome this problem, we need to use a different mask for each message. However, the `PTRACE_POKEDATA` is a system call and using it for each instruction execution would result in significant overhead. For this reason, instead of using different `PTRACE_POKEDATA` values directly as different masks for every message, we use a single one to seed a **Pseudo-random number generator (PRNG)** that in turn will generate the masks, reusing the same data exchange of Figure 8. In particular, we use *Xoroshiro256*** as PRNG due to its speed [4].

We create a PRNG for both JNI and OVM, seeded by the same value. This seed is generated half on the JNI and half on the OVM and exchanged with `PTRACE_POKEDATA` to keep the `ptrace` mandatory, as explained in Section 4.5.2. Then, for each message, we generate a new random value with the PRNG and use it as a mask to encode/decode the next message, effectively creating a **One-time pad (OTP)** system. Each message exchange will now involve generating at least two values from each PRNG: the first one to mask/unmask the JNI request, the second one to mask/unmask the OVM response. With this enhancement, even in the case of a KPA, if the attacker would recover the mask used by a message, then the attacker could not recover the seed used to generate the full sequence of masks. A different mask will be used in the subsequent message(s) that cannot be decoded by the attacker.

Furthermore, to make the PRNG harder to attack, we implemented a system to prevent state compromise extension attacks [19]. A state compromise extension attack attempts at recovering the sequence of generated numbers upon knowing a single state of the PRNG. In our implementation, however, the PRNG skips some states upon fulfilling some conditions decided by both JNI and OVM at runtime, adding some non-linearity to the value generation. These conditions may be a particular number being generated or a particular response sent or received and are decided based on the portion of mask exchanged with `PTRACE_POKEDATA` but not used in the PRNG seed.

4.6.2 Mutual Authentication to Mitigate Man-in-the-middle Attacks. The protection described so far involves two processes debugging each other, where none can be killed and the `ptrace` cannot be removed. However, a Man-in-the-middle attack would still be possible, a third process between the JNI and OVM. A typical solution to this attack is based on cryptography to hide the clear text from the attacker who can observe the message exchanged by the two processes [15]. The typical solution involves a secure way to establish a secure channel, e.g., using a **Certificate Authority (CA)**. Our problem is slightly different, because the attacker has additional capabilities, i.e., the attacker could perform static or dynamic analysis on the two processes. Here, we present a scenario that enables this kind of attack and our solution to protect against it.

As shown in Figure 9, an attacker may replace the executable of the original OVM with a fake one that will just invoke a second fake OVM that in turn will invoke the original OVM.[37] With some care in setting up the message passing, in particular by sending the `PTRACE_POKEDATA` variables to the real OVM and by ensuring a correct message forwarding from the JNI/OVM through the fake OVMs, an attacker may be able to circumvent the protection. This attack does not require to access the decoded (clear text) messages, but forwarding their encoded form is enough. So, encryption

[37] In self-debugging implementations using `fork`/`exec` the attacker can replace the `exec` part of the call.

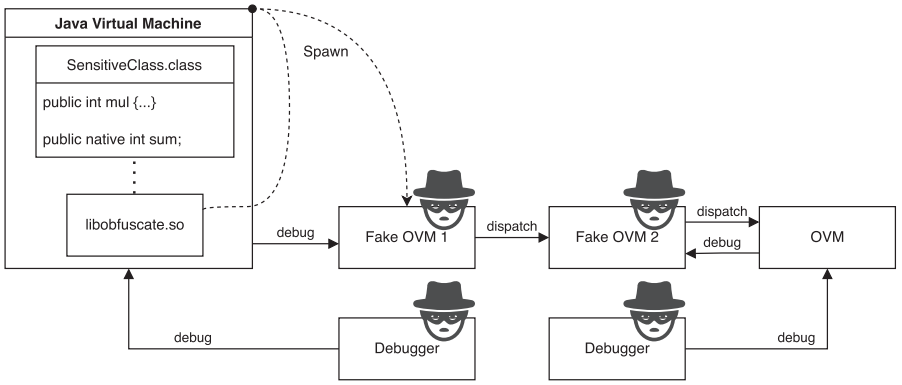

Fig. 9. Man-in-the-middle attack to the Self-debugging.

does not mitigate the kind of attack. Figure 9 shows an example of an attack scenario where both JNI and OVM debug and are debugged by malicious processes.

To mitigate this attack scenario, the only possible solution is to verify the authenticity of the target process. Typically, this attack is mitigated by authentication, e.g., based on CAs for network communication. We also apply authentication, but in a different way, because CAs are not available in our offline scenario [15]. In our approach, JNI and OVM are assigned the same AES key, which is randomly generated at compile-time. At runtime, before exchanging `PTRACE_POKEDATA` variables that contain the OTP seed, both JNI and OVM alter the seed by summing the PID of the other process. Then, they encrypt the seed with the AES key and write it back via the `pokedata` exchange. The receiving process has to subtract the PID from the seed before using it.

In this way, if a malicious process exists in the middle, then the JNI and OVM will not add/subtract the same PID from the key and will use two differently seeded PRNGs while using the OTP encryption. The attacker, in turn, cannot tamper with the seed and correct this mismatch, because it is encrypted with the AES key she does not know.

However, if an attacker is able to retrieve either the AES key or the OTP key, then she/he will still be able to (partially or completely) subvert the protection. Even if this attack is in principle feasible, it is expected to be quite hard and time-consuming. We aim to delay and slow down as much as possible this potential attack. This and other attacks are discussed later in Section 7.2.

5 ANTI-DEBUGGING AT JAVA LEVEL

While the anti-debugging protection presented in Section 4 hampers native-level debugging, malicious reverse-engineers may still infer valuable information by debugging Java code. For instance, an attacker could still observe and tamper with input data of the JNI toward the protected method and then observe the corresponding output through the JDWP. Even though this attack would not allow an attacker to directly observe the execution of the most sensitive code (that is translated to C), it may still leak some valuable information on its input-output logic.

To complement the anti-debugging at the C level presented in Section 4, we now propose a novel defense against debugging attempts at Java level, which we call "Back-end Damaging." Below, we first discuss the high-level design (Section 5.1) to then describe the novel Java anti-debugging protection we proposed in detail (Section 5.2).

5.1 High-level Design

As discussed in Section 2.1, the most effective approach to anti-debugging aims at completely blocking the capabilities of the debugger. Therefore, we base our anti-debugging protection on the

```
r--p 0000a000 08:01 7476236                  /lib/x86_64-linux-gnu/libnss_nis-2.27.so
rw-p 0000b000 08:01 7476236                  /lib/x86_64-linux-gnu/libnss_nis-2.27.so
r-xp 00000000 08:01 7476222                  /lib/x86_64-linux-gnu/libnss_compat-2.27.so
---p 00008000 08:01 7476222                  /lib/x86_64-linux-gnu/libnss_compat-2.27.so
r--p 00008000 08:01 7476222                  /lib/x86_64-linux-gnu/libnss_compat-2.27.so
rw-p 00009000 08:01 7476222                  /lib/x86_64-linux-gnu/libnss_compat-2.27.so
r-xp 00000000 08:04 13764593                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjdwp.so
---p 0003c000 08:04 13764593                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjdwp.so
r--p 0003b000 08:04 13764593                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjdwp.so
rw-p 0003c000 08:04 13764593                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjdwp.so
r-xp 00000000 08:04 13764586                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjava.so
---p 00029000 08:04 13764586                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjava.so
r--p 00028000 08:04 13764586                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjava.so
rw-p 00029000 08:04 13764586                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libjava.so
r-xp 00000000 08:04 13764616                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libverify.so
---p 0000d000 08:04 13764616                 /home/ardasy/Downloads/idea-IC-193.6494.35/jbr/lib/libverify.so
```

Fig. 10. The partial content of the `/proc/pid/maps` file of a Java application being debugged.

idea of damaging the debugging infrastructure that the JVM exposes. By relying on Java artifacts only, our anti-debugging protection results to be applicable to any Java application, both mobile (i.e., Android) and desktop. Moreover, our protection blocks the debugger activities while raising no errors, making it more difficult to be located by the attacker.

Then, we remind that Java bytecode can be easily decompiled into a very close representation of the original Java code, facilitating malicious reverse-engineering and static analysis [22]. Instead, decompilation is more difficult for compiled programs. For this reason, we choose to implement our Java-level anti-debugging protection in native code. The protection is activated in the preamble of the protected code, together with the Time-check or the Self-debugging protection.

Finally, it is worth highlighting that our Java anti-debugging protection is not mutually exclusive with those already existing, e.g., obfuscation and tampering detection. Instead, different protections can be used complementarily and synergize to increase the overall defenses of the application, as discussed in Reference [21].

5.2 Damaging the Java Debugger Back-end

As explained in Section 3.1, the back-end module of the JPDA is typically implemented at native level with shared libraries. Therefore, the first step for detecting a debugging attempt is to look for artifacts in memory that may reveal the presence of a debugger.

In our prototype implementation, we consider Linux with the OpenJDK[38] JVM (Java 11), however, other systems are not conceptually different. By scanning the native libraries mapped in memory of a Java application under debugging (i.e., by checking the `/proc/pid/maps` file), we observe the presence of the `libjdwp.so` and `libdt_socket.so` libraries, as shown in Figure 10. The first library loads the back-end, while the second library is related to the communication channel (sockets, in this case) with the front-end.

Each library is mapped in regions of contiguous virtual memory. These regions may have read (r), write (w), and execute (x) permissions, expressed by the first three flags at the beginning of the rows. If a process attempts to access a location in memory in a way that is not permitted, then a segmentation fault error is generated. The fourth flag (p) stands for "private," meaning that the memory region is not shared and is instead a copy of the original code (i.e., the original shared library). In particular, the `libjdwp.so` library is mapped in four different regions:

— The first region has read and execute permissions, and it contains the actual code of the back-end.

[38] https://openjdk.java.net/

— The second region does not have any permission. Mainly, this region is created as a guard against buffer overflows and to protect any possible gaps between memory regions.
— The third region has read-only permission, and it contains read-only data like constant values.
— The fourth region has read/write permissions, and it contains dynamically allocated data.

To prevent the correct functioning of the Java debugger, we propose to act on the virtual memory regions of the back-end submodule. We note that randomly tampering with read-only data or dynamic variables may disrupt and crash the debugger, revealing the presence of the anti-debugging protection. Therefore, our approach consists in corrupting the memory region of the `libjdwp.so` library containing the executable code. In particular, we first obtain write permissions on the first memory region by using the `mprotect`[39] system call. Then, we overwrite specific bytes of the library code segments with the `0xC3` opcode, which corresponds to a `return` instruction in the x86 **Instruction Set Architecture (ISA)**. In particular, we replace the first instruction of each function with the `0XC3` opcode. In this way, the back-end will immediately return without actually executing any code. While this approach works for x86-based processors only, it is very easy to include support for other ISA (i.e., change the specific return opcode), which only requires engineering effort.

When running a Java program protected with Back-end Damaging anti-debugging protection, the debugger front-end seems to operate fine without raising any error. However, commands sent by the debugger front-end are not actually executed by the debuggee back-end. For instance, while a breakpoint seems to have been successfully set by the front-end, the execution of the application is not suspended when the breakpoint is reached, causing the debugging infrastructure to be useless for an attacker.

6 EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation aimed at assessing our anti-debugging protections, which protect either the C level with Time-check and Self-debugging (see Section 4) and the Java level with Back-end Damaging (see Section 5). Below, we formulate three research questions to guide the definition of our experimental settings (Section 6.1). Then, we identify nine debugging tasks (Section 6.2) encompassing malicious reverse-engineering activities (e.g., set breakpoints, modify variables, and registers) along with a set of metrics to measure experimental data (Section 6.3). We present our case studies (Section 6.4) and give an overview of the procedure we followed during the evaluation (Section 6.5). Finally, we conclude by answering each of the research questions based on the data collected from the experimental evaluation (Sections 6.6 to 6.8). We publish the complete replication package[40] to allow replicating the experimental evaluation with our (or, maybe, with different) anti-debugging protections. In detail, the replication package contains all the case-study applications with their source code, test cases, and build scripts (and also the already compiled `.jar` files for ease of use); the completely automated experiment including annotation, compilation, testing, protection, and execution of debugging tasks on case studies with the collection of all the experimental results.

It is worth noting that, in our experimental evaluation, there is no direct comparison with the previous work done by Abrath et al. [1, 2] due to the different applicability domains of the two tools. Despite both tools target compiled C code, their tool requires plain C code, while ours exploits the structure of the JVM and thus can work only in JNI code generated by *Java2C*. Conversely, their tool is based on the `fork` primitive to start the debugging process, which, according to our preliminary investigation, is not working in a JNI environment.

[39] https://www.man7.org/linux/man-pages/man2/mprotect.2.html
[40] https://github.com/stfbk/Mitigating-Debugger-based-Attacks-to-Java-Applications-with-Self-Debugging

Table 2. Java- and Native-level Debugging Tasks

Task Name	Level	
	Java	Native
Attach	Attach to the program prior runtime	Attach to the program prior runtime
Set breakpoint	Reach a breakpoint after the protected code has been invoked (i.e., in the `java.lang.System.exit` function)	Reach a breakpoint at the beginning of the protected code
Stepping	Step among instructions after the protected code has been invoked	Step among instructions of the protected code
Show call stack	Print the Java call stack after the protected code has been invoked	Print the call stack when inside the protected method
Show variables	Print the values of all local variables after the protected code has been invoked	-
Trace output	Trace method entries and exits before and after the protected code have been invoked	-
Show registers	-	Displays the contents of all processor registers
Set watchpoint	-	Set a watchpoint for an expression (i.e., the debugger stops the execution whenever the value of the expression changes)
Set register	-	Set the value of an expression or a register

Similarly, our Java-level anti-debugging protection cannot be compared with the debugger detection approaches discussed in Section 2.1, as they just detect debuggers, but they do not actually block any debugging-related capability. In other words, all debugging tasks we identified would succeed. Moreover, the two proactive anti-debugging protections proposed in References [11, 24] are not relevant, as they rely on obsolete technologies and techniques (i.e., manipulating the *gDvm* global variable exposed by the Dalvik Virtual Machine) and are thus not deployable anymore.

6.1 Research Questions

We formulate three research questions to guide our experimental evaluation:

— RQ_1: What is the resiliency of the anti-debugging protections?
— RQ_2: What is the robustness of the anti-debugging protections?
— RQ_3: What is the overhead of the anti-debugging protections?

The first research question aims at measuring the extent to which our anti-debugging protections ward the protected code against automated debugging tools. The second research question investigates the impact of the protections on the functionality of the protected code. In other words, we assess whether the protections compromise the correct execution of the protected code. The third research question measures the computational overhead of the protections in terms of execution time.

6.2 Debugging Tasks

To evaluate our anti-debugging protections, we automate the execution of several debugging tasks (see Table 2) that are supported by common debuggers. These concrete debugging tasks can be seen as the building blocks for the abstract strategies and activities that attackers adopt while performing malicious reverse-engineering [9]. In detail, we identify nine debugging tasks that map to commands available in Java Debugger (JDB)[41] and **GDB**.[42] Although executed differently, some Java and native debugging tasks are similar in objectives. In particular, we define four

[41] https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html
[42] https://visualgdb.com/gdbreference/commands/

common debugging tasks, two tasks at Java-level only, and three tasks at native-level only, for a total of nine distinct tasks. With the only exception of the *Attach* debugging task, which is executed when launching the application, all the Java-level tasks are executed after the Java anti-debugging protection has been activated, i.e., after the protected code has been invoked. Conversely, native-level tasks are executed on the protected code. We automate the execution of debugging tasks by implementing a wrapper sending commands to the GDB and JDB tools and observing the resulting output. Hence, a debugging task consists of a list of commands to send to the (either Java or native) debugger along with the debugger's expected output. For instance, the JDB command "`stop in com.example.ClassName.methodname`" expects as output "`Set breakpoint com.example.ClassName.methodname`," while any other output (e.g., "`Deferring breakpoint com.example.ClassName.methodname`") is considered as a failure.

6.3 Metrics

To answer the research questions, we define the following metrics to apply to the data resulting from the evaluation:

- *Protection Resiliency* - The number of debugging tasks that succeed when an anti-debugging protection is applied;
- *Functional Correctness* - The number of JUnit tests that pass when an anti-debugging protection is applied;
- *Running Time* - The average duration and standard deviation of JUnit test cases on the original code (i.e., unprotected) and when an anti-debugging protection is applied.

6.4 Case studies

For our empirical validation, we collected open-source Java projects from GitHub.[43] In detail, we consider only projects that were starred by more than 1,000 GitHub users to choose popular and widely adopted projects, which, even though not for sure, we expect to be more thoroughly and well-tested and of higher quality. Here, by high-quality projects, we mean those that compile and run out-of-the-box, with no custom compilation or configuration process, which would have caused expensive manual overhead in our experiment.

For each project, we downloaded the source code and the test cases. Then, we kept only projects with at least 30 JUnit tests to exclude projects with poor test coverage that would not fit our experimental settings, which rely on test cases to collect experimental data. We collected these projects during the first three months of 2021, as that was the time during which we ran the experimental evaluation.

In the end, our data set consists of 18 case studies comprising 18,618 successfully running JUnit tests. We report the case studies along with relevant information (e.g., link, name, description, number of stars, number of JUnit tests) in Table 3.

6.5 Experimental Procedure

The experimental evaluation consists of two phases (i.e., configuration and experiments) that we describe in detail below. The two phases have been scripted and the scripts are available, together with the results of the experimental evaluation and the code of the anti-debugging protections, in our GitHub repository as replication packages.

Configuration. The configuration is run on each case study according to these steps:

- *Analysis*: The JUnit test cases are executed on the original unprotected application code. Tests that failed to execute (due to, e.g., missing dependencies or external modules) are filtered out

[43] https://github.com/

Table 3. Case Studies

Name	Description	Stars	Test Classes	Successful Tests	Failed Tests
Apache Avro™	A data serialization system	1,607	81	3,159	40
CommaFeed	Google Reader inspired self-hosted RSS reader	1710	10	21	30
Apache Commons Lang	Utilities for classes in java.lang's hierarchy	1,925	180	5,868	58
docker-maven-plugin	Maven plugin for managing Docker images	1319	82	228	139
google-java-format	Ensure compliance with Google Java Style	3325	19	699	374
Jimfs	An in-memory file system for Java 7 and above	1787	50	5518	177
jsoup	Library for working with real-world HTML	7,941	53	726	21
Uber JVM Profiler	Java Agent to collect metrics and stacktraces	1315	24	64	0
logback	A successor to the popular log4j project	1,875	194	452	59
PF4J	Make monolithic applications modular	1123	45	65	21
poi-tl	Generate word(docx) documents with template	1,325	84	53	39
Pushy	Library for sending APNs push notifications	1185	28	286	52
ScribeJava	The simple OAuth client Java lib	5,040	24	95	5
Soot	A Java optimization framework	1441	102	40	122
Traccar	An open-source GPS tracking system	2,390	328	30	318
Truth	Make test assertions and messages readable	2081	328	1278	8
Webcam Capture API	Use built-in or external webcams in Java	1,707	15	13	26
Web Magic	A scalable crawler framework	9001	22	23	23

(i.e., removed from the case-study source code). Then, only successful tests are run again with JaCoCo[44] to collect code coverage at the method level. In other words, for each method, we collect the percentage of instructions executed while running the (successful) tests.

— *Annotation*: Among the case-study methods whose code coverage is at least 70%, we pick the one with the longest body, i.e., the method with the highest number of statements. This method is annotated as requiring anti-debugging protection. In this way, we protect the longest method, and we skip trivially short ones whose code is largely covered by tests.

— *Transformation*: The chosen method is translated from Java to native code by Java2C [31] and protected with either the Self-debugging or the Time-check anti-debugging native-level protection. The Back-end Damaging anti-debugging Java-level protection is always added to the code and invoked at the beginning of the protected method. This implies that the tool allows evaluating two pairs of anti-debugging protections, i.e., Back-end Damaging-time-check and Back-end Damaging-self-debugging.

Experiments. The experiment includes the evaluation of the robustness, resiliency, and computational overhead of our anti-debugging protections:

— *Protections Resiliency* - Debugging tasks are applied on at most 10 JUnit tests among the successful JUnit tests that execute the protected method. In other words, we launch at most 10 tests trying then to complete each of the debugging tasks defined in Table 2. Debugging tasks are attempted both on the original unprotected application and the new protected application code. During this phase, the outcome of the debugging task is collected as either success or failure. Only 10 test cases are executed to limit the amount of time taken by this step, because some debugging tasks (e.g., stepping execution) take up to several hours to complete, depending on the length of the protected method.

[44] https://www.eclemma.org/jacoco/

Table 4. RQ1 - Java-level Anti-debugging - JDB Debugging Tasks on Original and Protected Code

Case Study		Original Code						Protected Code					
	Number of Tests	Attach	Set Breakpoint	Stepping	Show Call Stack	Show Variables	Trace Output	Attach	Set Breakpoint	Stepping	Show Call Stack	Show Variables	Trace Output
Apache Avro™	10	10	10	10	10	10	10	10	0	0	0	0	0
CommaFeed	4	4	4	4	4	4	4	4	0	0	0	0	0
Apache Commons Lang	10	10	10	10	10	10	10	10	0	0	0	0	0
docker-maven-plugin	10	10	10	10	10	10	10	10	0	0	0	0	0
google-java-format	8	8	8	8	8	0	8	8	0	0	0	0	0
Jimfs	6	6	6	6	6	6	6	6	0	0	0	0	0
jsoup	10	10	10	10	10	10	10	10	0	0	0	0	0
Uber JVM Profiler	10	10	10	10	10	10	10	10	0	0	0	0	0
logback	9	9	9	9	9	9	9	9	0	0	0	0	0
PF4J	10	10	10	10	10	10	10	10	0	0	0	0	0
poi-tl	10	10	10	10	10	10	10	10	0	0	0	0	0
Pushy	10	10	10	10	10	10	10	10	0	0	0	0	0
ScribeJava	5	5	5	5	5	5	5	5	0	0	0	0	0
Soot	2	2	2	2	2	2	2	2	0	0	0	0	0
Traccar	4	4	4	4	4	4	4	4	0	0	0	0	0
Truth	10	10	10	10	10	10	10	10	0	0	0	0	0
Webcam Capture API	3	3	3	3	3	3	3	3	0	0	0	0	0
Web Magic	1	1	1	1	1	1	1	1	0	0	0	0	0
Sum of Successful Tasks	-	132	132	132	132	132	132	132	0	0	0	0	0
% of Successful Tasks	-	100%	100%	100%	100%	100%	100%	100%	0%	0%	0%	0%	0%

— *Functional Correctness* - We identify the subset of successful JUnit tests that execute the method to protect. Then, we run these tests on both the original unprotected application code and the protected application code to check if their functional correctness still holds.
— *Running Time* - The running time of the JUnit tests is extracted from the JUnit reports to measure the computational overhead caused by the anti-debugging protections. To reduce measurements errors, tests are run 100 times.

6.6 RQ1 - Analysis of Resiliency

Java-level Anti-debugging Protection Resiliency. Table 4 reports the results of the execution of Java-level debugging tasks on both the original code and the code protected with the Back-end Damaging anti-debugging protection. From the table, we note that all debugging tasks execute successfully on the original code. Instead, the only debugging task that executes successfully on the protected code is the "Attach" task. Indeed, as explained in Section 6.2, the Java debugger attaches to the application before the anti-debugging protection is applied. However, when the protection is applied, the commands sent by the front-end debugger are not executed by the back-end.

Native-level Anti-debugging Protections Resiliency. Table 5 reports the results of the execution of native-level debugging tasks on the code protected with the Time-check and Self-debugging anti-debugging protections. We recall that the tool applies only one of these two protections at a time, i.e., we test each native-level anti-debugging protection separately.

Table 5. RQ1 - Native-level Anti-debugging - GDB Debugging Tasks on Code Protected with Time-check and Self-debugging

Case Study		Code Protected with Time-check							Code Protected with Self-debugging						
	Number of Tests	Attach	Set Breakpoint	Show Call Stack	Stepping	Show Registers	Set Watchpoint	Set Register	Attach	Set Breakpoint	Show Call Stack	Stepping	Show Registers	Set Watchpoint	Set Register
Apache Avro™	10	10	10	9	0	10	10	10	10	0	0	0	0	10	10
CommaFeed	4	4	4	4	0	4	4	4	4	2	0	0	0	4	4
Apache Commons Lang	10	10	10	10	1	10	10	10	10	1	0	0	0	10	10
docker-maven-plugin	10	10	10	10	0	10	10	10	10	6	0	0	0	10	10
google-java-format	8	8	0	0	0	0	8	8	8	0	0	0	0	8	8
Jimfs	6	6	6	6	2	6	6	6	6	0	0	0	0	6	6
jsoup	10	10	10	10	0	10	10	10	10	0	0	0	0	10	10
Uber JVM Profiler	10	10	9	10	1	10	10	10	10	3	0	0	0	10	10
logback	9	9	9	9	0	8	9	9	9	0	0	0	0	9	9
PF4J	10	10	10	10	7	10	10	10	10	0	0	0	0	10	10
poi-tl	10	10	10	9	0	10	10	10	10	0	0	0	0	10	10
Pushy	10	10	10	10	0	10	10	10	10	0	0	0	0	10	10
ScribeJava	5	5	5	5	0	5	5	5	5	0	0	0	0	5	5
Soot	2	2	2	2	0	2	2	2	2	1	0	0	0	2	2
Traccar	4	4	4	4	0	4	4	4	4	0	0	0	0	4	4
Truth	10	10	10	9	1	10	10	10	10	0	0	0	0	10	10
Webcam Capture API	3	3	3	3	1	3	3	3	3	0	0	0	0	3	3
Web Magic	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1
# of Successful Tasks	-	132	123	121	13	123	132	132	132	14[a]	0	0	0	132[a]	132[a]
% of Successful Tasks	-	100%	93.18%	91.66%	9.84%	93.18%	100%	100%	100%	10.60%	0%	0%	0%	100%	100%

[a]False positives

Regarding the Time-check protection, from Table 5, we note how almost all debugging tasks still have a success rate of more than 90%. The only notable exception is the "Stepping" task (success rate of 9.84%), which, since stepping through the protected code, takes more time to execute than the other tasks and is more likely to trigger the Time-check protection.

Regarding the Self-debugging protection, we note the success of some "Set Breakpoint" debugging tasks (i.e., 14 on 132) and the success of all "Set Watchpoint" and "Set Register" debugging tasks. We manually investigate these results, concluding that all the successful "Set Breakpoint" debugging tasks are actually false positives. These false positives are due to the subtle approach of the Self-debugging protection described in Section 4. In detail, when debugging the protected method, the Self-debugging protection blocks its execution and simply returns 0. However, we found that in some JUnit tests (i.e., those corresponding to the false positives "Set Breakpoint" debugging tasks), the protected method incidentally returns the same value as expected by the JUnit test to succeed. In other words, the JUnit test succeeds even if the protected method does not really execute. The successes on the "Set Watchpoint" and "Set Register" debugging tasks are due to the fact that these tasks are executed immediately before the protected method is invoked (i.e., before the protection is applied). Afterward, the protection detects the debugging attempt and stops the execution of the protected method, making the watchpoints and the set register commands pointless. For this reason, we consider these successes to be false positives as well.

RQ1 answer - The resiliency of the Back-end Damaging Java-level anti-debugging protection is 100%, while the resiliency of the Time-check and Self-debugging Native-level anti-debugging protections is (on average) of 19% and 100%, respectively.

Table 6. RQ2 - Successful Tests after an Anti-debugging Protection Was Applied

Case Study	Number of Tests	No Protection	Back-end Damaging	Time-check	Self-debugging
Apache Avro™	97	97	97	83	97
CommaFeed	4	4	4	4	4
Apache Commons Lang	25	25	25	24	25
docker-maven-plugin	11	11	11	11	11
google-java-format	8	8	8	6	8
Jimfs	6	6	6	6	6
jsoup	14	14	14	13	14
Uber JVM Profiler	10	10	10	10	10
logback	16	16	16	13	16
PF4J	12	12	12	10	12
poi-tl	37	37	37	0	37
Pushy	24	24	24	23	24
ScribeJava	5	5	5	5	5
Soot	11	11	11	11	11
Traccar	4	4	4	4	4
Truth	505	505	505	497	505
Webcam Capture API	3	3	3	3	3
Web Magic	1	1	1	0	1
Sum of Successful Tests	-	793	793	723	793
% of Successful Tests	-	100%	100%	91.17%	100%

6.7 RQ2 - Analysis of Robustness

Table 6 reports the results of the execution of JUnit tests on the original (unprotected) application code and the protected code. From the table, we see that the Back-end Damaging and the Self-debugging anti-debugging protections do not impact the functional correctness of the protected method. Instead, the Time-check protection causes almost 10% of JUnit tests to fail. After inspecting the logs that are generated during the executions of the JUnit tests, we note that the failures are due to the Time-check protection. Indeed, the logs reveal that the Time-check protection detected a debugging attempt, even though no debugging activity was being performed. The main reason behind this problem is that the time taken by the protected method to execute (without any debugging attempt) is longer than the threshold set for the Time-check protection, so the default threshold is not appropriate. A different threshold value should be set by the developer that can better estimate the expected execution time.

RQ2 answer - The robustness of the Back-end Damaging Java-level anti-debugging protection is 100%, while the robustness of the Time-check and Self-debugging Native-level anti-debugging protections is 91% and 100%, respectively.

Table 7. RQ3 - Average Increase in Time after Anti-debugging Protections Were Applied (in Seconds)

Case Study	Back-end Damaging Time-Check		Back-end Damaging Self-debugging	
	Avg. Time Increase	% Avg. Time Increase	Avg. Time Increase	% Avg. Time Increase
Apache Avro™	0.00348	850.07151	1.47115	656,529.84206
CommaFeed	0.00073	1.34592	0.25817	111.14308
Apache Commons Lang	0.00319	265.21143	9.76244	667,808.49008
docker-maven-plugin	0.00011	0.99954	0.09404	99.95265
google-java-format	0.05062	78.61165	13.80167	25,224.10695
Jimfs	0.00182	129.99639	3.70568	282,501.45419
jsoup	0.00059	30.97813	1.39990	55,391.69714
Uber JVM Profiler	0.00092	866.03615	1.84341	1,760,604.30882
logback	0.00046	15.06699	0.57712	17,425.56170
PF4J	0.00072	9.11208	1.52946	16,934.09078
poi-tl	0.00017	3.12915	0.61584	493.51491
Pushy	0.00045	322.65856	1.60390	215,014.51724
ScribeJava	0.00073	660.14571	1.02596	303,335.36074
Soot	0.00019	25.68404	0.18854	11,874.65119
Traccar	0.00087	3,022.93563	1.03786	2,041,853.45912
Truth	0.00074	28.25344	2.70142	93,664.23941
Webcam Capture API	0.00040	3.80386	2.01817	34,979.51507
Web Magic	0.03859	30.68055	7.59732	5,574.37816
Average Increase	0.00582	352.13680	2.84622	343,856.68240

6.8 RQ3 - Analysis of Overhead

Table 7 reports the increase in the running time of JUnit tests between the original and protected code. Note that we measure the overhead of the Back-end Damaging anti-debugging protection together with either the Time-check or the Self-debugging anti-debugging protection. In fact, Java-level and (one of the two) Native-level protections should always be used together to guarantee an adequate level of protection against malicious debugging attempts.

From the table, we see that the Back-end Damaging Time-check anti-debugging protections pair causes an average overhead of a few milliseconds at worst. The average increase of time in percentage indicates that protected code executes 3 times slower than unprotected code. This overhead includes both the time taken by the protections and the fact that the method is translated from Java to Native code. Regarding the Back-end Damaging Self-debugging anti-debugging protections pair, the average overhead is almost three seconds, with a peak of more than 10 seconds. The average increase of time in percentage indicates that protected code executes hundreds of thousands of times slower than unprotected code.

RQ3 Answer - The overhead of the Back-end Damaging Time-check anti-debugging protections pair is of 5.82 milliseconds, on average (an increase of around 350% with respect to the unprotected code), with a maximum of 5.06 milliseconds and a minimum of 0.01 milliseconds. The overhead of the Back-end Damaging Self-debugging anti-debugging protections pair is of 2.85 seconds, on average (an increase of around 340,000% with respect to the unprotected code), with a maximum of 13.8 seconds and a minimum of 9.4 milliseconds.

7 DISCUSSION

In this section, we present a discussion on the results of the empirical assessment, limitations of the proposed protections (Section 7.1), possible attacks and mitigation strategies (Section 7.2), and threats to validity (Section 7.3).

Our approach to limit malicious debugging proved to achieve high resiliency in blocking most of the debugging tasks that a common debugger is supposed to support. At the same time, no malfunction nor undesired side effect is added as a consequence of anti-debugging (provided that a reliable estimation is available for code execution time), which suggests the large applicability of our approach.

7.1 Maximizing Resiliency

Despite automating most of the transformation to the point that the user is required to use a single Java annotation, some considerations can be made about which methods to protect. In fact, like any other protection, ours do not come for free, and carefully placing the required Java annotation may increase the speed or the resilience of the protection.

Non-negligible performance overhead The execution time of an application might increase, depending on the deployed protection and the amount of protected code. However, whether the execution overhead is acceptable for end-users depends on the specific context of the application at hand. Indeed, it is up to the developers to strike the best possible balance between security and performance, evaluating on a case-by-case basis which is the most appropriate protection (and its configuration) for the requirements of their application; this is a cost-benefit analysis that the developers should carry out using our quantitative evaluation as a guideline.

When to activate Java-level protection The Back-end Damaging Java-level anti-debugging protection described in Section 5 has 100% resiliency only after the protection is applied, i.e., after the protected method is executed. Indeed, as shown in Table 4, the "Attach" debugging task always succeeds. This limitation implies that a malicious reverse-engineer has control over the application until the protected method is invoked and can thus manipulate the inputs of such a method. Nonetheless, we note that the attacker cannot observe the output of the method. A possible solution to this limitation is to apply the Back-end Damaging protection at the startup of the application, at the cost of making the protection less subtle. The most appropriate point where to activate this protection should be decided by the developers based on the security requirements of the Java application to protect.

Time-threshold calibration is crucial The results in Table 5 show that the Time-check anti-debugging protection has limited resiliency (less than 10%). Although these results are also due to the fact that all debugging tasks were automated (i.e., they often executed too quickly to be detected by the protection), we note that the resiliency of the Time-check protection largely depends on the debugging activity being performed and the time threshold triggering the protection. Therefore, the threshold should be fine-tuned according to the length and complexity of the protected method. For instance, a developer might compute an accurate value of this threshold by measuring the execution time of his/her code in worst-case scenarios.

Input data vs. output data protection The "Set watchpoint" and the "Set register" debugging tasks always succeed despite the Self-debugging anti-debugging protection. Indeed, these tasks were performed at the beginning of the protected method, i.e., before the protection was actually applied. The results in Table 5 prove that, while not being able to modify registers during the execution of the protected method, a malicious reverse-engineer could still modify the inputs of the protected method. However, as for the Back-end Damaging protection, the attacker is not able to inspect the corresponding output.

Default return value As discussed in Section 6.6, the "Set breakpoint" debugging task for the Self-debugging anti-debugging protection has 14 false positives. This is due to the Self-debugging protection returning the default value of 0 when detecting a debugging attempt. However, as the value 0 represents a successful execution, it may be desirable to adjust the return value, depending on the method chosen to be protected.

Code injection for anti-debugging Currently, the Back-end Damaging protection just overwrites the first opcode of the back-end submodule library code. A possible improvement is to elaborate on this concept by injecting snippets of more meaningful code, aiming at disrupting the Java debugger even more, e.g., by sending reply packets containing wrong information.

7.2 Possible Attacks

Although the scope of this article is debugger-based attacks, other attacks at different levels (e.g., at the operating system level) could represent a threat. Even if anti-debugging is not meant to mitigate these attacks—hence, they are clearly out of the scope of anti-debugging—it is worth discussing them to explain why different complementary protections (such as remote attestation, anti-tampering, and obfuscation) need to be deployed together with anti-debugging to protect each other.

Malicious JPDA module As explained in Section 3.1, the JPDA is highly modular. Therefore, a malicious reverse-engineer could replace the modules of the Oracle reference implementation with custom modules at any interface level. Although the main idea of the Back-end Damaging Java-level anti-debugging protection (i.e., replace the first opcode of every function with a return opcode) should theoretically be resilient with custom implementations of the back-end submodule as well, further investigations and experiments should be conducted. Moreover, we note that it is not difficult to look for revelatory strings (e.g., "maps," "proc") and system calls (e.g., `mprotect`) through static analysis. Despite these calls being protected from patching attacks in self-debugging, as shown in Section 4.5.2, a malicious reverse-engineer could still try and circumvent some of them and bypass the application of the Back-end Damaging protection that is applied before self-debugging, thus exposing at least the Java part to debugging. This possible attack highlights the need to couple anti-debugging protections with other techniques such as obfuscation, anti-tampering, or remote attestation to certify that the original JPDA modules are running and not those tampered with by the attacker.

Attacking process authentication Despite protecting both the JNI and the OVM processes against debugging, as described in Section 4, some attacks against the native protection are still possible. In particular, we showed in Section 4.6.2 a possible Man-in-the-middle attack that is not particularly difficult to set up and has the potential of completely deactivating the native protection. We protected against this type of attack by using an AES key, effectively moving the problem from protecting the JNI and the OVM processes to protecting this encryption key.

Although meant to avoid scenarios where an attacker could simply patch away portions of the anti-debugging protection by replacing the relevant instructions with NOOP operations, our protection is meant to be integrated with other protection techniques that are capable of statically obfuscating the code. In fact, if the attacker retrieves the AES key used to authenticate the message exchange between JNI and OVM, then the attacker can impersonate the other process and spoof all the messages exchanged between the two processes, with the attack shown in Section 4.6.2.

Another potential attack point is the OTP key generation. As described in Section 4.6.1, the entire communication between JNI and OVM is encrypted with a key generated half by one process and half by the other process and written directly to the companion process via debug calls. An attacker could dump the memory of the processes, store the entire communication, and later retrieve the

key and decrypt the messages. The challenges of this attack are finding the correct decryption key in the dumped memory and capturing all the communication if the domain socket is abstract (abstract sockets do not exist in the filesystem). Moreover, after decrypting the messages, their semantics still needs to be inferred by reversing the OVM ISA.

Reversing the protection Given the open source nature of *Java2C* and `conceal.it`, an attacker may potentially write an "unconceal.it" tool that attempts to automatically revert back the protected code to the original Java. To make such a tool automated (it is a sort of decompiler), it is necessary to guess the OVM ISA that is never reused, because it is randomly generated when our protection is applied. Opcode guessing is a task in attacking virtual-machine-based obfuscations, and to solve it is necessary to reverse-engineer the virtual-machine itself [32]. This could be done either dynamically using debugging [16], which is turned harder by our protection, or statically by static analysis. The latter case is the reason why our approach should be integrated with a strong static code obfuscation technique. In addition, this decompiler should deal with the strongly optimized JNI and OVM binaries.

Malicious Java Virtual Machine In our study, we are assuming that the JVM used in the system is genuine. However, a particularly determined attacker may re-implement the JVM and debug the JVM itself, as opposed to the JNI layer, to extract some information about the obfuscated method. This is possible because, despite the communication between JNI and OVM being obfuscated, the JNI layer still needs to exchange data with the JVM layer. In particular, all the tasks involving objects allocation, fields access, and methods invocation are performed by the JVM layer.

An attacker using a malicious JVM implementation may leak and obtain information about these method calls and allocations, effectively lowering the resilience of the native obfuscation. However, moving these allocation and method invocation tasks to the JNI layer would require essentially moving most (or all of) the application from Java to C. Another option would be to use a modified JVM implementation encrypting the channel between the JNI and the JVM in the same way the channel between the JNI and the OVM is encrypted.

As a final remark, although professional hackers may reuse reverse-engineering tools they developed (e.g., custom re-implementations of the JVM) to reduce the overall cost of attacking (Java) applications, it is worth noting that implementing such tools—and adapting them for use in different environments—is not a trivial task and may still require considerable effort.

Malicious Kernel The core idea of our anti-debugging obfuscation relies on the fact that only a single process can debug another process. However, if this is not the case, then an attacker would be able to attach a second debugger to the protected processes. Given that, to the best of our knowledge, every operating system allows a single process to debug another process, an attacker needs to write a custom kernel to attack the protection by exploiting the ptrace. Also in this case, the cost that a malicious reverse-engineer would sustain to write and debug its own kernel may be even higher than the value of a successful attack.

7.3 Threats to Validity

Here, we summarize those threats that limit the validity of our results, divided into threats to internal and external validity.

Threats to Internal Validity: Considering that the translation from Java bytecode to C source code is addressed using an existing tool, namely, *Java2C*, verifying the semantic equivalence between original and translated code is out of the scope of the present article and delegated to the *Java2C* paper [30]. Nonetheless, we acknowledge that any error in the translation would result in a defective anti-debugging protection. To mitigate this threat, we resorted to software testing to spot semantic deviations in the protected code. However, some defects could still be present in the translated code that might be overlooked by test cases.

In the empirical validation, we decide which method to annotate and protect by choosing methods with high test coverage (< 70%) and far from trivial (with the highest number of statements). However, these methods might not be the most relevant from a security viewpoint. A more appropriate annotation should be based on the security requirements of each case-study that were, however, not available to us. Anyway, considering the objective of our experiment, our annotation strategy is still sound, because it allows us to measure protection impact on code correctness and on the resiliency with respect to common debuggers debugging tasks.

While the objective of translating part of the Java code to C code is to implement a barrier to malicious reverse-engineering with anti-debugging, this introduces an interface that might be an evident starting point for an attacker. However, even if an interface is there, still it would be hard for an attacker to analyze it. In fact, (*i*) malicious dynamic analysis is mitigated by anti-debugging and (*ii*) static analysis is known to be much harder [17, 20] on multi-language programs (e.g., Java and C) than single language programs (e.g., Java only) even if source code is provided. In our context, source code would not be available. Moreover, our anti-debugging solutions are meant to be complemented by code obfuscation as an additional barrier to malicious reverse-engineering.

Threats to External Validity: Despite considering a set of 18 open-source Java projects with more than 1,000 stars on GitHub and at least 18 test cases each to assess our approach from different domains, they are all open-source projects and they might not be representative of all the potential software systems that might benefit from anti-debugging. Only replications with different software, e.g., commercial closed-source programs, can prove or disprove our observations. Assessing anti-debugging with Android apps is indeed on our research agenda.

Our experimental validation has been conducted on a limited list of debugging tasks. Even if we filled this list by considering all the commands available in common debugging tools (i.e., JDB and GDB), this list might be incomplete and attackers might consider other tasks, possibly implementing their own custom debugger. However, new debugging features would still require a debugger to attach to the process under attack, so it is unlikely that they would succeed when our Self-debugging protection is deployed to actively engage the process debugging slot.

8 CONCLUSION

Previous empirical investigation has shown that, when sensitive code is obfuscated to prevent malicious reverse-engineering, attackers do not attempt to statically undo obfuscation, but typically opt for dynamic analysis based on debuggers. This article presented a novel approach to protect Java programs from malicious debugging. First, the compiled Java bytecode is automatically translated to C code. Then, based on the expected style and structure of this translated C code, automated code transformation is applied to split data access from computation into two distinct processes that, to guarantee correct program execution, must be executed simultaneously. Additionally, each of these two processes attaches as a debugger to the other process to engage its debugging interface, which is no longer available to the attacker to attach her/his own malicious debugger. The two processes share an encoded communication channel and adopt a custom authentication protocol to prevent the attacker to replace one of them or to intercept their communication in plaintext.

While our empirical assessment shows our protection to be resilient against standard (either Java or C) debuggers, as future work, we plan to experiment with custom debuggers, e.g., based on QEMU or virtualization infrastructures, such as hypervisors. We plan to extend our approach to address the potential new threats coming with these additional contexts and technologies. Moreover, as already established in the literature, we plan to involve professional attackers and run public challenges to study to what extent our protection is effective in practice in the field.

REFERENCES

[1] Bert Abrath, Bart Coppens, Ilja Nevolin, and Bjorn De Sutter. 2020. Resilient self-debugging software protection. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW'20)*. IEEE, 606–615.

[2] Bert Abrath, Bart Coppens, Stijn Volckaert, Joris Wijnant, and Bjorn De Sutter. 2016. Tightly-coupled self-debugging software protection. In *6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'16)*. ACM Press, 1–10. DOI: https://doi.org/10.1145/3015135.3015142

[3] Alexander Sprogø Banks, Marek Kisiel, and Philip Korsholm. 2021. Remote attestation: A literature review. *arXiv preprint arXiv:2105.02466* (2021).

[4] David Blackman and Sebastiano Vigna. 2021. Scrambled linear pseudorandom number generators. *ACM Trans. Math. Softw.* 47, 4 (2021), 1–32.

[5] Stephen Cass. 2021. Top programming languages: Our eighth annual probe into what's hot and not. *IEEE Spectrum* 58, 10 (2021), 17–17.

[6] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. 2007. Barrier slicing for remote software trusting. In *7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07)*. IEEE, 27–36.

[7] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. 2009. Trading-off security and performance in barrier slicing for remote software entrusting. *Autom. Softw. Eng.* 16, 2 (2009), 235–261.

[8] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. 2017. How professional hackers understand protected code while performing attack tasks. In *IEEE/ACM 25th International Conference on Program Comprehension (ICPC'17)*. IEEE, 154–164.

[9] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empir. Softw. Eng.* 24, 1 (01 Feb. 2019), 240–286. DOI: https://doi.org/10.1007/s10664-018-9625-6

[10] Mariano Ceccato, Paolo Tonella, Mila Dalla Preda, and Anirban Majumdar. 2009. Remote software protection by orthogonal client replacement. In *ACM Symposium on Applied Computing*. 448–455.

[11] Haehyun Cho, Jongsu Lim, Hyunki Kim, and Jeong Hyun Yi. 2016. Anti-debugging scheme for protecting mobile apps on android platform. *The Journal of Supercomputing* 72, 1 (January 2016), 232–246. DOI: https://doi.org/10.1007/s11227-015-1559-9

[12] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. 2011. Principles of remote attestation. *Int. J. Inf. Secur.* 10 (2011), 63–81.

[13] Christian Collberg, G. R. Myles, and Andrew Huntwork. 2003. Sandmark—A tool for software protection research. *IEEE Secur. Privac.* 1, 4 (2003), 40–49.

[14] Christian S. Collberg and Clark Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation—Tools for software protection. *IEEE Trans. Softw. Eng.* 28, 8 (2002), 735–746.

[15] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. 2016. A survey of man in the middle attacks. *IEEE Commun. Surv. Tutor.* 18, 3 (2016), 2027–2051. DOI: https://doi.org/10.1109/COMST.2016.2548426

[16] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *18th ACM Conference on Computer and Communications Security*. 275–284.

[17] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E. Eghan, and Bram Adams. 2020. On the impact of interlanguage dependencies in multilanguage systems empirical case study on Java native interface applications (JNI). *IEEE Trans. Reliab.* 70, 1 (2020), 428–440.

[18] Erik Kain. 2012. "Diablo III" Fans Should Stay Angry about Always-Online DRM. Retrieved from https://www.forbes.com/sites/erikkain/2012/05/17/diablo-iii-fans-should-stay-angry-about-always-online-drm/?sh=131cd5691853

[19] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Cryptanalytic attacks on pseudorandom number generators. In *International Workshop on Fast Software Encryption*. Springer, 168–188.

[20] Wen Li, Ming Jiang, Xiapu Luo, and Haipeng Cai. 2022. POLYCRUISE: A cross-language dynamic information flow analysis. In *31st USENIX Security Symposium (USENIX Security'22)*, 2513–2530.

[21] Kyeonghwan Lim, Jaemin Jeong, Seong-je Cho, Jongmoo Choi, Minkyu Park, Sangchul Han, and Seongtae Jhang. 2017. An anti-reverse engineering technique using native code and Obfuscator-LLVM for Android applications. In *International Conference on Research in Adaptive and Convergent Systems*. ACM, 217–221. DOI: https://doi.org/10.1145/3129676.3129708

[22] Kyeonghwan Lim, Younsik Jeong, Seong-je Cho, Minkyu Park, and Sangchul Han. 2016. An android application protection scheme against dynamic reverse engineering attacks. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 7, 3 (September 2016), 40–52. DOI: https://doi.org/10.22667/JOWUA.2016.09.31.040

[23] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification* (Java SE 8 edition ed.). Addison-Wesley, Upper Saddle River, NJ. Retrieved from https://docs.oracle.com/javase/specs/jvms/se8/html/

[24] Felix Matenaar and Jeffrey Forristal. 2014. Mobile devices with inhibited application debugging and methods of operation. Retrieved from https://patents.google.com/patent/US8925077/en
[25] Michael McWhertor. 2022. Gran Turismo 7, offline for more than 24 hours, shows its always-online problem. Retrieved from https://www.polygon.com/22984748/gran-turismo-7-maintenance-downtime-credits-car-prices
[26] Jerome Miecznikowski and Laurie Hendren. 2002. Decompiling Java bytecode: Problems, traps and pitfalls. In *International Conference on Compiler Construction*. Springer, 111–127.
[27] Bernhard Mueller, Sven Schleier, Jeroen Willemsen, and Carlos Holguera. 2022. *OWASP MSTG: Mobile Security Testing Guide*. Open Web Application Security Project. Retrieved from https://github.com/OWASP/owasp-mastg/
[28] Ilya Nevolin and Bjorn De Sutter. 2017. Advanced Techniques for Anti Debugging. Retrieved from https://lib.ugent.be/catalog/rug01:002367296
[29] John Papadopoulos. 2019. Call of Duty: Modern Warfare is always-online on the PC, even for its single-player campaign. Retrieved from https://www.dsogaming.com/news/call-of-duty-modern-warfare-is-always-online-on-the-pc-even-for-its-single-player-campaign/
[30] Davide Pizzolotto and Mariano Ceccato. 2018. Obfuscating Java programs by translating selected portions of bytecode to native libraries. In *IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM'18)*. 40–49. DOI: https://doi.org/10.1109/SCAM.2018.00012
[31] Davide Pizzolotto, Roberto Fellin, and Mariano Ceccato. 2019. OBLIVE: Seamless code obfuscation for Java programs and Android apps. In *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. 629–633. DOI: https://doi.org/10.1109/SANER.2019.8667982
[32] Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies (WOOT'09)*.
[33] Bert Abrath, Stijn Volckaert, and Bjorn De Sutter. 2018. Self-debugging. Retrieved from https://patents.google.com/patent/EP3330859A1/en
[34] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens. 2016. Reactive attestation: Automatic detection and reaction to software tampering attacks. In *ACM Workshop on Software Protection*. 73–84.
[35] Alessio Viticchié, Cataldo Basile, and Antonio Lioy. 2017. Remotely assessing integrity of software applications by monitoring invariants: Present limitations and future directions. In *International Conference on Risks and Security of Internet and Systems*. Springer, 66–82.
[36] Alessio Viticchié, Leonardo Regano, Cataldo Basile, Marco Torchiano, Mariano Ceccato, and Paolo Tonella. 2020. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empir. Softw. Eng.* 25, 1 (01 Jan. 2020), 1–48. DOI: https://doi.org/10.1007/s10664-019-09738-1
[37] Jia Wan, Mohammad Zulkernine, and Clifford Liem. 2018. A Dynamic App Anti-Debugging Approach on Android ART Runtime. In *2018 IEEE 16th International Conference on Dependable, Autonomic and Secure Computing, 16th International Conference on Pervasive Intelligence and Computing, 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech'18)*. IEEE, 560–567. DOI: https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00105
[38] Junfeng Xu, Li Zhang, Yunchuan Sun, Dong Lin, and Ye Mao. 2015. Toward a secure Android software protection system. In *IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE, 2068–2074. DOI: https://doi.org/10.1109/CIT/IUCC/DASC/PICOM.2015.307

Received 26 September 2022; revised 25 September 2023; accepted 13 October 2023