Lessons Learned and Challenges of Deploying Control Flow Integrity in Complex Software: the Case of OpenJDK's Java Virtual Machine

Sabine Houy
Umeå University
Umeå, Sweden
sabine.houy@cs.umu.se

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

***Abstract*—This research explores integrating LLVM's Control Flow Integrity (CFI) into the OpenJDK Java Virtual Machine (JVM) to mitigate memory corruption vulnerabilities. We present a manual approach to CFI integration that offers a solution applicable to various real-world projects. Using the DaCapo benchmark suite, we conduct a thorough performance evaluation of the CFI-integrated JVM version. Our work reveals that introducing CFI results in an average performance overhead of approximately 11.5% and a 34% increase in binary size. Remarkably, we identify specific CFI subcategories that, when implemented individually, induce performance improvements for the JVM. This finding highlights CFI's potential to enhance security and performance in Java and general applications. Our research advances the understanding of CFI integration in complex software such as the JVM, shedding light on the challenges and opportunities in securing software systems against memory corruption attacks.**

***Index Terms*—cfi, jvm, control flow integrity, memory corruption, C/C++ vulnerabilities, security methodology**

I. Introduction

Today, Java is one of the top programming languages and is used to write popular applications from development environments to reverse engineering tools and games. These include Ghidra, a reverse engineering framework from the NSA [1], the Eclipse Integrated Development Environment [2], the Minecraft game [3], and most mobile applications for Android, such as Netflix[4]. Java is considered a memory-safe language because Java applications do not directly manage or have direct access to memory. However, Java applications are executed with the help of the Java Virtual Machine (JVM), which is mainly written in C and C++. Therefore, the JVM is also susceptible to vulnerabilities affecting C/C++. As C and C++ are memory-unsafe languages, memory corruption is one of the most common vulnerability types. A successfully exploited memory corruption vulnerability can allow an attacker to gain control over a program's control flow and, thereby, control over the entire system [1]. These attacks are also known as control-flow hijacking attacks, as the attacker can redirect a program's expected execution to an unintended function or code segment.

Despite decades of research, memory corruption vulnerabilities remain among the most critical and challenging. Memory corruption vulnerability is an umbrella term encompassing various software vulnerabilities, including stack and heap buffer overflows and use-after-free. MITRE, a not-for-profit corporation, publishes several rankings every year. One of these lists is part of the CWE (Common Weaknesses Enumeration) Rankings [2] and summarizes the most severe Known Exploited Vulnerabilities (KEV) [3]. The top three KEVs in 2023 are part of the memory corruption vulnerability family, highlighting the ongoing need for research in this area, and number one in the "*2023 CWE Top 25 Most Dangerous Software Weaknesses*" ranking [4], highlighting the urgent need for further research.

Over the years, extensive research has yielded a plethora of mitigation techniques. These are usually applied in combination in systems to achieve the highest possible level of security. Mitigation techniques such as ASLR (Address Space Layout Randomization) or the non-executable stack are deployed at the operating system level and *do not modify the code of applications*. Nowadays, they are present by default in all major operating systems, such as Windows, Ubuntu, Android, and iOS. In recent years, an effort has been made to complement these mechanisms through program-based techniques, which *modify applications to inject security mechanisms executed at runtime* within the application. One of the most promising and effective of them is Control Flow Integrity (CFI).

CFI is injected into the application by the compiler, and its code is executed at runtime. During compilation, the compiler generates a program's control-flow graph (CFG) and identifies all indirect branching and cast instructions that could potentially be vulnerable. These instructions include indirect calls or jumps of virtual or non-virtual functions or bad casts. It then identifies all call or jump targets with the same signature. A signature consists of a function or destination's return and parameter types. All destinations with the same signature are grouped together. The code verifying that the target has the

[1]https://github.com/NationalSecurityAgency/ghidra
[2]https://github.com/eclipse-platform/eclipse.platform
[3]https://www.minecraft.net/en-us/article/programmers-play-minecrafts-inner-workings
[4]https://netflix.github.io/

same signature as the intended destination based on the CFG is injected during compilation and executed during runtime before an indirect branching is executed.

The fundamental theoretical idea of CFI was first mentioned in a 2005 paper by Abadi et al. [5], which was revised and republished in 2009 [6]. Designing and implementing a deployable version of control flow integrity took several approaches and multiple years [7]–[10]. The critical factor that caused it to take this long to create a customizable version was the introduced performance overhead. Indeed, Wang et al.'s [8] implementation of CFI published in 2015 has a runtime overhead of almost 30% [11], while today's implementation's performance overhead is between less than 1% and at most 15% [12]. In 2015, Tice et al. [13] introduced CFI to two standard compilers for C and C++ from the GNU Project (GCC) and LLVM (Clang). While GCC implements CFI as virtual-table-verification with very limited customizable options [14], LLVM provides *seven different sub-variants of Control Flow Integrity* and the option to use an ignore list to exclude only specific files or functions that should not be compiled with CFI [12]. Moreover, Xu et al. [15] evaluated the compatibility of different CFI implementations, including Clang's and GCC's, and concluded that Clang's CFI has the best compatibility. Compatibility, in this context, refers to CFI's ability to work effectively with various types of software without disrupting execution or degrading performance. This may be why most software developers have decided to use Clang as their compiler. For instance, Android switched from GCC to Clang over time (Android 7/8) [16] and ended support for GCC in 2020 [17]. Around the same time, they started incorporating CFI into their operating system [18].

However, the practical integration of CFI is highly challenging. Android had to make some changes to Clang's implementation before they could use it in their operating system [19]. Due to the nature of CFI, a program's functionality can be broken – resulting in the program not working anymore – if it is not 100% CFI compatible. This results in a trade-off between security and usability. The Chromium Project, therefore, only deploys two variants of the seven available in Chrome for Linux systems [20]. Mozilla has been looking for a CFI integration for over ten years and is currently building Firefox completely without CFI [21]. So far, there is no clear documentation on the challenges of deploying CFI in real applications. This work aims to shed light on the missing information.

The above examples exhibit the significant challenges in implementing control-flow integrity in real-world projects. Since many real-world and widely used programs and projects are primarily written in memory-unsafe languages (C/C++), CFI should be more widely deployed to protect against memory corruption vulnerabilities. One of these widely used projects is the Java Virtual Machine, which, as mentioned above, is mainly written in C and C++. Previous work has shown that, as for other large C/C++ projects, exploitable memory corruption vulnerabilities do exist in the JVM [22].

To better understand and characterize the difficulties of CFI integration in a large software project, we provide a manual methodology and apply it to deploy CFI to the JVM of OpenJDK[5], called HotSpot[6]. OpenJDK is an open-source implementation of the Java Platform and, thus, offers access to its source code. The availability of the source code is essential to introducing CFI as it is a compiler-based mitigation technique. After successfully introducing control flow integrity to the JVM (CFI-JVM), we evaluate the results of excluded files that would otherwise break the functionality. Moreover, we assess the performance of CFI-JVM using the DaCapo benchmark suit [23]. Hence, we provide the following contributions:

1) A manual methodology to deploy CFI in a large software project such as the JVM.
2) The application of the methodology to deploy CFI in OpenJDK's JVM (version 21) and compile it with LLVM's implementation of Control Flow Integrity (CFI-JVM). The binary size overhead is 34.25%.
3) A performance evaluation of the CFI-integrated JVM based on the DaCapo benchmark suit. The results show that the JVM works on the DaCapo benchmarks with CFI. However, an average performance overhead of 11.47% is introduced. Surprisingly, specific individual CFI variants can increase the performance.

The remainder of the paper is structured as follows: In Section II, we provide the necessary background on Control Flow Integrity (II-A) and the Java Virtual Machine (II-B). Afterward, we present our manual approach to introducing CFI into the JVM (Section III), followed by our results in Section IV. We then discuss our results in Section V, including our main takeaways. In Section VI, we outline the related literature. Lastly, we summarize our approach and main findings in Section VII.

II. Background

In Section II-A, we provide an overview of Control Flow Integrity (CFI) and its essential concepts. Section II-B shifts the focus to the JVM, discussing necessary information in the context of memory corruption vulnerabilities.

A. Control Flow Integrity (CFI)

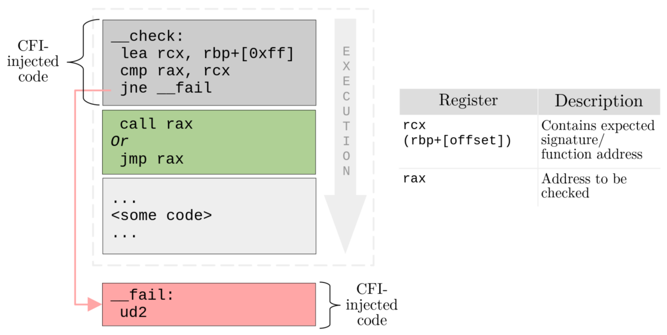
Control Flow Integrity (CFI) is a mitigation technique tailored to protect systems against control-flow hijacking attacks that are based on a memory corruption vulnerability. Although not all memory corruption vulnerabilities are exploitable, the impact on a system in the event of an exploitable vulnerability is critical. Many vulnerabilities go entirely unnoticed by users, as they do not visibly affect the program's control flow and, thus, its functionality from the user's perspective. If the vulnerabilities are nevertheless visible to the user, this is usually because they "only" cause the program to crash. However, an attacker might be able to write or use an exploit for the vulnerability to gain complete control of the system, possibly resulting in remote code execution (RCE). Due to the

[5]https://openjdk.org/
[6]https://openjdk.org/groups/hotspot/

severe repercussions, protecting systems against control-flow hijacking attacks is essential. The name originates from an attacker taking over the intended control flow of the execution and deriving it according to their goals. As mitigation techniques cannot entirely prevent such exploits, it is generally recommended to combine and stack multiple techniques to make a system as secure as possible. In the context of memory corruption vulnerabilities and control-flow hijacking attacks, techniques such as ASLR, non-executable stacks, and stack canaries are commonly deployed simultaneously. In addition, Control Flow Integrity is known to be one of the most effective techniques [24]. Therefore, over the years, efforts have been made to add CFI to applications such as the Chrome browser or operating systems like Android and Windows.

Multiple implementations have been proposed since the formal introduction of CFI in 2005 [5]. However, barely any of them could establish themselves due to performance difficulties, e.g., up to 30% [8], [11], and lack of real-world implementations. The primary implementations are based on the publication by Tice et al. [13], providing the fundamentals to integrate CFI into GCC and LLVM compilers. For the LLVM compiler, Clang, the introduced overheads are less than 1% up to 15% in the Chromium binary [12]. Microsoft's implementation [25] uses a similar approach to LLVM Clang's. Since we rely on LLVM for the experiments, we explain CFI's functionality based on LLVM's implementation. CFI can be divided into *forward- and backward-edge enforcement*. Backward-edge protects function returns using a so-called ShadowCallStack. However, the backward-edge enforcement is not widely supported as its implementation only exists for ARM-based systems since LLVM had to remove it for x86_64 systems due to security issues caused by race conditions and critical performance overheads [26].

Forward-edge CFI protects indirect and virtual calls. For the simplicity of the paper, we will refer to forward-edge CFI as CFI. In general, CFI is introduced during a program's compilation process. First, the compiler (Clang) statically builds the whole program's Control Flow Graph (CFG). This CFG represents all expected possible execution paths. In the second step, Clang identifies potentially vulnerable code parts, i.e., all indirect branchings such as indirect jumps or virtual calls. It then generates equivalence sets of functions or jump targets with the same signatures. The signatures are based on the return and parameter types. By that, a set of legitimate call and jump targets is created. Additional code is introduced before each potentially vulnerable branching instruction, verifying that the destination of the indirect or virtual call is in the determined set of legitimate targets. Figure 1 illustrates a simplified version of this approach. `rax` contains the address of the to-be-called function. The verification signature is loaded into `rcx`, then the contents of `rax` and `rcx` are compared. If they are the same, the execution proceeds normally. Otherwise, the program crashes using the `ud2` instruction. Beyond covering function calls and jumps, the LLVM CFI implementation extends its protection to avert bad type casts. By validating that only proper casts occur—such as disallowing casts between unrelated classes—CFI mitigates type confusion attacks.



```
__check:
 lea rcx, rbp+[0xff]
 cmp rax, rcx
 jne __fail
```

```
 call rax
Or
 jmp rax
```

```
...
<some code>
...
```

```
__fail:
 ud2
```

Register	Description
`rcx` (`rbp+[offset]`)	Contains expected signature/ function address
`rax`	Address to be checked

Fig. 1. CFI control flow example

The following describes the different CFI schemes provided by LLVM's implementation. All seven schemes are enabled when simply using the "`fsanitize=cfi`" flag [12].

1) `cfi-cast-strict`: It checks that an object of a base class is not cast to a derived class if the derived class has the same layout and virtual function semantics as the to-be-cast base class. This is the case if the base class is the only non-virtual base class to the derived class and the derived class does not define any additional virtual member functions or fields or override existing ones. The only declaration allowed is a virtual instructor.

2) `cfi-derived-cast`: It is not directly part of the initial idea of Control Flow Integrity. Still, it can also lead to memory corruption vulnerabilities and is thus included in the CFI implementation of Clang. Its purpose is to check that a pointer's cast is only made to an object of the correct dynamic type. This means that "*the dynamic type of the object must be a derived class of the pointee type of the cast*" [12]. It handles casts from a base class to a derived class. The C++ standard defines a bad cast of this type as undefined behavior.

3) `cfi-unrelated-cast`: Just as II-A2, it is also part of bad cast checking and not initially proposed CFI. It verifies that no bad cast from a pointer of type `void*` or another unrelated type, e.g., not derived, is performed. Usually, a bad cast of this type is handled automatically, and the pointer is cast back to its original type. However, this is impossible if the pointer is uninitialized or defined as `static_cast`. Sometimes, bad casts cannot be prevented and must conform to an external API, e.g., using a member function of a standard library.

4) `cfi-nvcall`: This scheme ensures that object-based non-virtual calls have the correct dynamic type. The type is considered correct if the dynamic type of the callee object is a derived class of the static type of the callee object.

5) `cfi-vcall`: This CFI variant verifies that the virtual pointer (`vptr`) used for virtual calls has the correct dynamic type. The definition of a correct type is the same as for II-A4.

6) `cfi-icall`: This option ascertains that function calls only occur using a function of the appropriate dynamic type, meaning the function's dynamic type should correspond to the static type utilized during the call.

7) `cfi-mfcall`: The object used for indirect calls utilizing a member function pointer must have the correct dynamic type. The dynamic type is correct if the type of the member function referenced by the member function pointer aligns with the function pointer component. Additionally, the member function's class type must correspond to the member function's base type. For simplicity, we omit the option `-fcomplete-member-pointers`, which enables non-conforming language extensions requiring the member pointer base types to be complete.

If one of the above schemes cannot be applied correctly, meaning is not successfully validated during runtime, a CFI violation is detected, and the program terminates. This can be prevented by removing specific files or functions from the CFI introduction and compiling them without CFI protection by adding them to the so-called *`ignorelist`*.

B. Java Virtual Machine (JVM)

The Java platform is widely used on servers, desktops, mobile phones, etc. The platform can generally be divided into two main components: (i) the Java Virtual Machine (JVM) written in C/C++, and (ii) the Java Class Library (JCL) written in Java. The second one is not of interest for this paper. The JVM is the heart of the Java platform, supplying the fundamental execution functionalities, which include the bytecode parser, interpreter, just-in-time compiler, and garbage collector. Its code is primarily located in the shared library `libjvm.so` (25.4MB in size) and is loaded by the java executable. In total, the code consists of 3955 classes, resulting in approximately 511K lines of code. The JVM takes Java bytecode as input. This bytecode can be untrusted, such as when running a potentially malicious Java plugin within Ghidra. Ghidra[7] is a software reverse engineering tool that relies on the JVM to execute its plugins [27]. This reliance means that untrusted bytecode could pose a security risk within the Ghidra environment. In this case, the applet functions as the exploit payload, so the attacker's crafted input exploits the vulnerability and possibly leads to remote code execution. Figure 2 shows the generalized pipeline for executing a Java program. First, the Java compiler (javac) receives the `.java` as input. These are compiled into so-called `.class` files, which contain Java bytecode. The Java bytecode files are then input to the Java Virtual Machine and executed there.

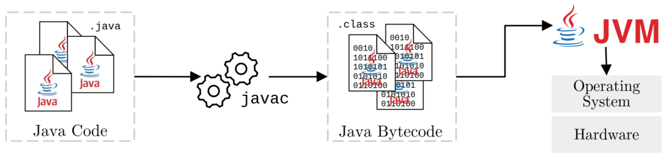


Fig. 2. Simplified layout of execution in Java platform

The JVM is a native application, meaning it is written in C and C++ and is therefore susceptible to vulnerabilities specific to these languages. These include memory corruption vulnerabilities such as buffer and integer overflows. CVE-2015-4843 [28] describes a known and fixed integer overflow vulnerability in the JVM that could lead to RCE [22].

In Java, signed integers are represented in 32-bit. An overflow can occur if an integer exceeds the maximum size. This can happen when an arithmetic operation is performed without ensuring that the result is not too large. Suppose we have an integer `max_int` that contains the maximum value represented by an integer in Java, namely $2^{31} - 1$. Adding 1 to this value will cause an overflow because the result exceeds the maximum value. In Java, if an overflow occurs during integer arithmetic, the value is reversed and becomes a negative number. The observed value of the `max_int` variable would be -2147483648, the smallest value that a 32-bit integer can represent. This unexpected behavior can lead to errors or security vulnerabilities. The overflowed value can be used to allocate memory or index arrays outside their bounds. An attacker can exploit this to overwrite critical data structures or execute arbitrary code. CFI prevents the execution of arbitrary code by restricting attackers from accessing, loading, and executing code segments that are outside the authorized boundaries of a user's access.

III. Methodology

In this and the subsequent sections, we aim to answer the following research questions:

RQ1: How can we deploy Control Flow Integrity in projects like the JVM?

RQ2: What challenges arise when deploying CFI in such projects?

RQ3: What is the performance impact of implementing CFI in these environments?

In Section III-A, we outline the necessary information to prepare the OpenJDK Java Virtual Machine for the implementation of CFI. Following this, in Section III-B, we present a comprehensive manual approach to introducing CFI into the JVM, addressing *RQ1*. Subsequently, in Section V, we explore the encountered challenges stemming from this process to answer *RQ2*. Our performance evaluation setup is detailed in Section III-D, aimed at addressing *RQ3*.

A. JVM Setup

We worked with the latest LTS version of OpenJDK (openjdk 21-internal 2023-09-19, Server VM), the same version installed on a system using its package manager. A root JDK is needed to build the JDK; we used version 21.0.2_linux_x64. Our JDK is compiled on an Ubuntu 22.04.3 LTS x86_64 machine. OpenJDK offers the option only to make its JVM called HotSpot; however, this does not include the Java executable, meaning we cannot execute and thus test the JVM. Therefore, we chose to build the base JDK module using `make jdk`, which includes, among others, the Java executable. By default, OpenJDK uses GCC as compiler, but since we focus on the CFI implementation of LLVM, we have to pass Clang as the compiler to `make`. To get a more accurate comparison, we

[7]https://github.com/NationalSecurityAgency/ghidra

compile the JDK once without CFI but with Clang, which we then use as ground truth.

To introduce CFI step-by-step into the JVM, we need debugging symbols to determine which files to add to the ignore list. This process is explained in detail in the following Section III-B. However, adding debugging symbols meant that we could no longer compile the JDK at all, as the Makefiles define that warnings are handled as errors during the build process. We, therefore, had to adapt the Makefiles and set the field `WARNINGS_AS_ERRORS` to `False`. After that, compiling the JDK with and without CFI was possible. In order to compile a program with LLVM's CFI, the `-flto` and `-fvisibility=hidden` flags must be added [12]. Moreover, we created an initial ignorelist containing a list of all source files not belonging to the JVM. By that, we ensure that CFI will only be added to the JVM-related code.

B. Manual CFI Introduction Approach

Figure 3 shows our pipeline to introduce Control Flow Integrity into the JVM. We use this pipeline separately for each of the seven CFI variants (described in the Background Section II-A) and generate an ignorelist for each. Once each version works, we merge the ignorelists and compile it using the `cfi` option, which means all seven versions are combined. To explain our pipeline better, we will use one specific CFI version; however, it works identically for all others.

We want to identify all files that must be excluded when compiling the JVM protected by `cfi-icall`. This means that we must add the flags displayed in Table I to the configuration.

TABLE I
CONFIGURATION FLAGS USED FOR BUILDING PROCESS OF `CFI-ICALL`

FLAG	DESCRIPTION
`--with-toolchain-type=clang`	Use Clang as compiler instead of GCC
`--with-boot-jdk=<path_to>/jdk-21`	Path to root JDK needed to build JDK locally
`--enable-debug`	Use debugging symbols
`--with-native-debug-symbol=internal`	Store debugging symbols directly in binary file itself
`--with-conf-name=cfi-icall`	Specific configuration name defined for each CFI variant
`--with-extra-cflags= "-flto -fvisibility=hidden -fsanitize=cfi-icall -fsanitize-ignorelist= <path_to>/ignorelist.txt"`	Addition C flags Required CFI flags and CFI variant Path to the ignorelist
`--with-extra-cflags= "-flto -fvisibility=hidden -fsanitize=cfi-icall -fsanitize-ignorelist= <path_to>/ignorelist.txt"`	Addition C flags Required CFI flags and CFI variant Path to the ignorelist
`--with-extra-cflags= "-flto -fvisibility=hidden -fsanitize=cfi-icall -fsanitize-ignorelist= <path_to>/ignorelist.txt"`	Addition C flags Required CFI flags and CFI variant Path to the ignorelist

Then, the source code, depicted in Figure 3-①, of OpenJDK is compiled to build the JDK containing the java executable that loads `libjvm.so`, basically, the JVM.

Once the building process is finished, we run the resulting java executable as $Test_{version}$ (using the argument `--version`), initializing the JVM (②) to test that the basic functionalities, such as the VM initialization and basic class loading, work as expected. The building process automatically incorporates an optimization step at the end, following the successful compilation of the java executable and `libjvm.so`. This optimization step could fail as it involves calling the newly built. However, since both the java executable and the `libjvm.so` have been generated normally before that step, it does not impact our CFI introduction process.

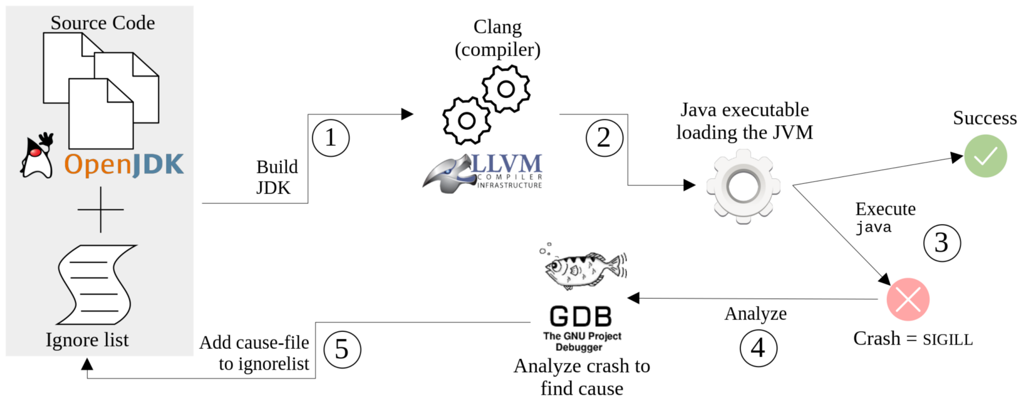
We then verify if the execution exits normally or crashes due to receiving a SIGILL (*Illegal Instruction signal*) in step ③. In the first case, we are done and have a working JVM with CFI or CFI version, which executes correctly on the '–version' use case. In the other one, we must dig deeper to determine which exact file caused the SIGILL. This file needs to be added to the `ignorelist`.

We run the java executable with $Test_{version}$ in gdb [29], see Figure 3-④. Since Java uses speculative loads, we must tell gdb not to handle SIGSEGV and pass them on to Java. After that, we can start the actual debugging process, which means running it once and looking at the generated backtrace when it crashes. We can extract the information from the backtrace where we set our first breakpoint, namely at the frame (#1), just before the SIGILL was caught (#0). Next, we re-run everything to hit the breakpoint. We single step into (*gdb-command* `step` or `s`) the function we hit with the breakpoint and, from there, start to step over (*gdb-command* `next` or `n`) until we receive the SIGILL. Then we repeat the process but replace the last step over with a step into, and we keep stepping over again until we encounter the SIGILL. We must repeat this process until a step into (`s`) causes the SIGILL. From there, we extract the file to be added to the `ignorelist`. The file is the source file containing the code just before the last step into (`s`) causing the crash. This results in a *gdb-trace* that can be described by

$$\mathtt{s} \cdot k_0\mathtt{n} \cdot \mathtt{s} \cdot ... \cdot \mathtt{s} \cdot k_i\mathtt{n} \cdot \mathtt{s},$$

with $k, i \in \mathbb{N}$ and k donating the amount of step overs (`n`) before the next step into (`s`). Three special cases can occur during the process. (i) Since the JVM uses multi-threading, a SIGILL might occur simultaneously in two or more threads. In general, that is not a problem, but determining which file is causing the SIGILL, as gdb switches between threads, can complicate it. This can be solved by telling gdb only to execute the current thread being debugged (`(gdb) set scheduler-locking on`). (ii) If the set breakpoint is not hit, we set a new breakpoint at the next frame in the backtrace, meaning that if we had our breakpoint at #1, the next one would be at #2, and so on, until one is hit. (iii) The file added to the `ignorelist` does not fix the problem. We approach that problem similarly to (ii); we use the next breakpoint to see if we can identify another cause.

When inspecting a program's stack frame after a crash, for instance, caused by an illegal instruction (SIGILL), not all



Source Code
OpenJDK
Ignore list
Build JDK
1
Clang (compiler)
LLVM COMPILER INFRASTRUCTURE
2
Java executable loading the JVM
Success
Execute java
3
Crash = SIGILL
Analyze
4
GDB The GNU Project Debugger
Analyze crash to find cause
Add cause-file to ignorelist
5

Fig. 3. Manual CFI Introduction pipeline

previous function calls may be displayed for several reasons. Stack corruption caused by the crash can disrupt critical information like return addresses and frame pointers, making it difficult for gdb to trace back through the stack accurately. Additionally, modern compiler optimizations, such as inlining functions, tail call optimizations, and frame pointer omission, can obscure the stack trace by eliminating or altering the representation of function calls in the stack. Furthermore, crashes occurring within the prologue or epilogue code of functions responsible for setting up and tearing down stack frames can leave the stack in an inconsistent state, further complicating gdb's ability to interpret the stack frames correctly. These issues collectively hinder obtaining a complete and accurate stack trace during debugging. When we inspect the stack frame, the illegal instruction signal has occurred, meaning the stack frame is already corrupted. Therefore, the information displayed in frame #1 is simply the first intact frame discovered by gdb.

The last step in our pipeline (Figure 3-⑤) of introducing CFI to the JVM is to add the identified file to the `ignorelist` and re-start the whole step until we receive a success in step ④. Files that were added to the `ignorelist` but did not fix anything will be removed from `ignorelist`.

In the end, we have seven separate `ignorelist` for each of the CFI variants described in the Background Section II-A. After finishing the process detailed above, we merge the seven `ignorelists` and, by that, receive our final one, which we use to compile the JVM with the general `-fsanitize=cfi` flag. This flag means that all seven versions are combined.

The described process is generic and can be used to deploy CFI on other software than the JVM. However, it is likely that some adjustments will be made, such as those described for the Makefiles or special cases in ④.

C. Discussion of Approach

The manual process described above is a subtractive approach, meaning that first, all files are compiled with CFI, and then each file causing a crash, thus being incompatible with CFI, will be excluded and compiled without CFI. An alternative approach we considered is the additive approach. Here, all files would be compiled without CFI, and then CFI would be introduced on a file-by-file basis. While the additive approach is more straightforward and less complex, it is more time-consuming. Take the example of the JVM, its source base consists of 3955 files. Adding CFI file-by-file requires compilation after each newly added file to determine if it is compatible with CFI, meaning it does not cause a crash. This results in a time consumption of

$$\mid source\_files \mid \cdot\ compile\_time.$$

Our subtractive method requires 3.5 compilations on average per crash. The time used for debugging varies significantly but can be averaged on 120 minutes per session. We needed about 2.5 debugging sessions per crash. Therefore, we have

$$\mid excluded \mid \cdot\ (3.5 \cdot compile\_time\ +\ 2.5 \cdot debug\_time).$$

The overall time consumption for the additive results in $3955 \cdot 15min$, summing up to 988.75 hours or 41.2 days. On the contrary, we have the subtractive method, and as we can see in Section IV, 41 files needed to be excluded. Thus, we have $41 \cdot\ (3.5 \cdot 15min\ +\ 2.5 \cdot 120min)$, resulting in only 240.875 hours. The subtractive approach is, therefore, more efficient than the additive approach. However, the additive approach might be more desirable for smaller projects with only a few source files.

D. Performance Evaluation Setup

We use the DaCapo benchmark suit [23] to assess the performance of the different JVMs resulting from the introduction of the seven CFI variants. We chose DaCapo because it includes a diverse set of real-world applications, providing realistic and meaningful performance data. It is a well-established and standardized tool in the Java community, offering comprehensive metrics such as execution time, memory usage, and garbage collection behavior. We downloaded the most recent release of DaCapo, *dacapo-23.11-chopin*[8]. The suit contains 22 benchmarks, of which four (`cassandra`, `h2o`, `tradebeans`, and `tradesoap`) could not be run as they

[8]https://www.dacapobench.org/

only work for JDK versions $\leq$ 17. We run the performance evaluation on a Dell Precision with Intel® Core™ i7, 16 processors, and 32GB of RAM. We ran each experiment five times and took the average time.

IV. Results

In Section IV-A, we delve into our findings from the CFI introduction process, supported by concrete examples. Following this, Section IV-B details the performance evaluation of our CFI-integrated JVM.

A. CFI Introduction Process

We have compiled the JVM with LLVM's implementation of Control Flow Integrity. In our context, it means successfully executing $Test_{version}$ and the DaCapo benchmark suit to make the JVM work. In total, we added 41 files to the `ignorelist`, which resulted in 1.04% of JVM files needing to be compiled without CFI. Figure 4 shows the excluded file distribution based on the seven CFI variants explained in the Background Section II-A.

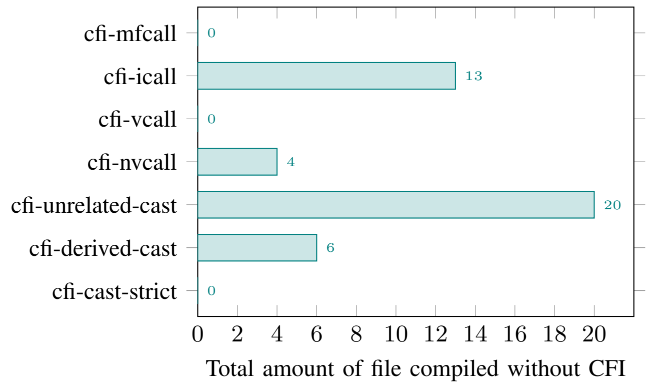


Fig. 4. Files added to the `ignorelists` based on CFI variants

However, when broken down to the line numbers, 13.41% of the total JVM code needs to be compiled without CFI in order to work correctly. 34 files needing to be excluded were identified by the $Test_{version}$ testing; the remaining seven were determined by running the DaCapo benchmarks. We used the same approach as described in the Methodology Section III-B with the benchmarks.

One of the most challenging and time-consuming task when introducing CFI was to determine the exact file causing the JVM crash due to CFI. Splitting the introduction process into the different CFI options also gives us more insights into which ones are more complicated to introduce. Additionally, it is more straightforward to identify the exact reason why CFI is not compatible with that specific code, which causes the JVM to crash. In the following, we discuss two example files that must be excluded.

1) cfi-unrelated-cast: We examine a CFI violation example caused by a cast between two unrelated classes. Figure 5 highlights the affected lines in the file `/src/hotspot/share/code/codeCache.cpp`, where a cast on line 4 triggers a crash. The classes involved are

```
...                                                   1
CodeHeap* heap = get_code_heap(code_blob_type);       2
...                                                   3
cb = (CodeBlob*) heap->allocate(size);                4
...                                                   5
```

Fig. 5. Code snipped taken from /src/hotspot/share/code/codeCache.cpp

`CodeCache`, `CodeHeap`, and `CodeBlob`, which are not related. We examine each class separately to better understand the example. The `CodeCache` class oversees a memory area dedicated to storing compiled native code by the JIT (*just-in-time*) compiler during Java application execution. It optimizes performance by implementing caching strategies to reuse compiled code and reduce execution time. `CodeBlob` represents individual compiled code objects, such as methods or code segments, and manages their lifecycle, including loading, unloading, and garbage collection.

Lastly, the `CodeHeap` class handles memory allocation and organization for storing compiled code segments within the JVM's memory space, organizing regions based on compilation level, code type, and size. In summary, the `CodeCache` class stores `CodeBlob` instances representing compiled code objects within memory regions managed by `CodeHeap`. This enables efficient storage, management, and execution of compiled native code within the JVM's runtime. In line 2, `CodeCache` retrieves a `CodeHeap` instance based on a `CodeBlob` type. However, in problematic line 4, a cast is made when attempting to associate this instance with a `CodeBlob` object for further processing. Since `CodeBlob` and `CodeHeap` are unrelated, this results in a CFI violation. Hence, excluding the `CodeCache` file from the CFI introduction is necessary. A possible fix might be to introduce an additional *heap* field in the `CodeBlob` and store the `CodeHeap` object there. Thus, the cast is not needed anymore.

2) cfi-icall: The complexity of this example exceeds that of the unrelated-cast case. In the first instance, the problematic file was pinpointed directly within the notification of the illegal instruction. Contrastingly, in this case, identifying the problematic file necessitated an extensive debugging session, as detailed in the Pipeline section. Consequently, for the first example, a single run through our pipeline sufficed, whereas the second demanded approximately three runs with several breakpoints adjusted. Line 20 in Figure 6 triggers a CFI violation due to an indirect call via a function pointer and a discrepancy between the caller's and callee's static and dynamic types. Examining the JVM's interpretation of Java bytecode into machine code is essential to understand the code comprehensively. Interpreters generally map individual Java bytecode instructions to corresponding machine code instructions. However, the JVM employs a technique known as a template interpreter to accelerate the interpretation process. Unlike traditional mapping, the template interpreter utilizes precompiled machine code templates. These templates allow entire Java bytecode sequences to be mapped to these tables instead of an instruction-by-instruction approach. These machine code templates are precompiled code snippets optimized

```
// interpreter/templateTable.hpp
class Template {
    typedef void (*generator)(int arg);
    ...
    // template code generator
    generator _gen;
    ...
    friend class TemplateTable;
    ...
};

// interpreter/templateTable.cpp
#include "interpreter/templateTable.hpp"
...
void Template::generate( InterpreterMacroAssembler*
↪  masm) {
  // parameter passing
  TemplateTable::_desc = this;
  TemplateTable::_masm = masm;
  // code generation
  _gen(_arg);
  masm->flush();
}
```

Fig. 6. Code snipped taken from /src/hotspot/share/interpreter/templateTable.hpp and /src/hotspot/share/interpreter/templateTable.cpp

to enhance the JVM's performance. The templateTable class dynamically generates and manages these templates. The class generates these code tables at runtime based on bytecode patterns encountered during program execution triggered by the code snippet displayed in Figure 6. The mismatch happens because the static type void does not match the dynamic generator type.

These examples are concrete. Nevertheless, there are more general reasons why the CFI introduction might fail, not depending on the specific CFI variant used. For instance, projects deploy legacy code or non-standard coding practices that do not conform to CFI's strict control flow requirements. Additionally, projects involving low-level operations or those with highly dynamic and complex control flow, such as dynamic code generation or extensive use of reflection, may face verification challenges with CFI mechanisms. These factors make it difficult for CFI to enforce control flow integrity accurately in such contexts.

In summary, we successfully integrated CFI into the Java Virtual Machine of OpenJDK, achieving compatibility with 86.39% of its code, with only 41 out of 3955 source files being excluded. Our approach involved introducing CFI variants individually, offering deeper insights into the causes of CFI violations within the source code. As illustrated in our first example (Section IV-A1), there is potential to further minimize the percentage of excluded code through source code refactoring and adjustments. Our manual introduction of CFI into the JVM demonstrates the application of LLVM's Control Flow Integrity implementation to a real-world project, effectively addressing *RQ1*.

B. Performance Evaluation

We evaluated the JVM's performance with all seven CFI versions separately, once combined and once without any CFI. 18 out of 22 DaCapo benchmarks ran successfully. The four that did not work are designed for older versions of the JDK than the one we chose and will not work even on an unmodified OpenJDK version 21. The performance evaluation results are displayed in Table II.

As can be observed, all CFI variants applied individually, but *cfi-derived-cast*, perform better than the JVM version without CFI (*no cfi* in Table II). Possible reasons for this observation might be the additional information the compiler has to collect to introduce CFI. This information enables the compiler to use it for additional optimization purposes such as devirtualization [30], polymorphic inline caching [31], and redundant code elimination. Moreover, CFI might influence the placement of code segments so that it is beneficial for the performance, e.g., by improving the cache locality or reducing instruction cache misses [32]. CFI can guide the compiler in generating more efficient code paths for dynamic dispatches [33], [34], reducing overhead associated with function pointer resolution and call-site lookups [35]. In the best-observed case, the performance is reduced by half (*cfi-vcall* compared to *no cfi* on the benchmark avrora). In the future, we plan to conduct in-depth research about the reasons, which might not only help to improve the JVM's security but also its performance. Nevertheless, adding all CFI versions has worsened the performance by almost 11.5%. Considering the typical overhead introduced by LLVM's CFI ($\leqslant$1%-15%), this result comes as little surprise. Furthermore, the size overhead is 34.25% for the CFI-injected JVM (all CFI variants).

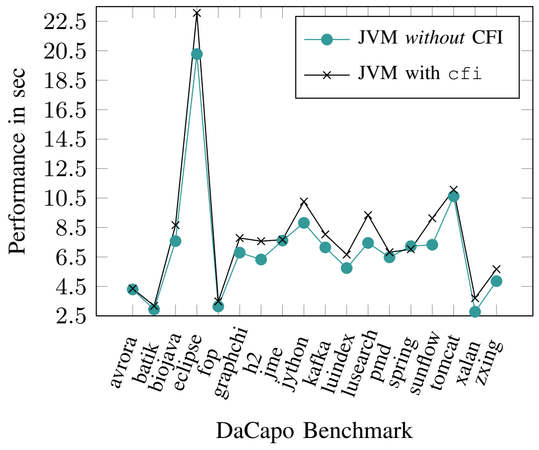


Fig. 7. Comparison of JVM performance with and without CFI

Figure 7 illustrates the performance difference between no CFI and all CFI variants combined. Generally, the JVM performs better without CFI. The smallest overhead observed is −48.1% for `jme`, and the largest is 37.3% for `xalan`. The average performance overhead is 11.47%. There are even two benchmarks (`spring`, `jme`) for which the JVM version with CFI outperforms the one without by −3.2% and

TABLE II
DaCapo Performance Evaluation in msec and difference to compiled without CFI

DaCapo	CFI variant performance in msec																
Benchmark	cfi-mfcall		cfi-icall		cfi-vcall		cfi-nvcall		cfi-unrelated-cast		cfi-derived-cast		cfi-cast-strict		cfi		no cfi
avrora	4198	−0.2%	2229	−47.0%	2184	−48.1%	2202	−47.7%	2220	−47.2%	4206	−48.0%	2188	**+0.0%**	4354	**+3.5%**	4207
batik	2964	−0.9%	2130	−28.8%	2142	−28.4%	2228	−25.5%	2164	−27.6%	2974	−0.5%	2081	−30.4%	3177	**+6.3%**	2990
biojava	7019	−9.3%	6100	−21.1%	6430	−16.9%	6449	−16.7%	7431	−4.0%	7627	−1.5%	6375	−17.6%	8659	**+11.9%**	7740
eclipse	14667	−29.1%	14537	−29.7%	14567	−29.6%	15776	−23.7%	20536	−0.7%	20561	−0.6%	14636	−29.2%	23073	**+11.6%**	20682
fop	2304	−30.4%	2217	−33.1%	2234	−32.5%	2397	−27.6%	3157	−4.7%	3397	**+2.6%**	2180	−34.2%	3481	**+5.1%**	3312
graphchi	5489	−20.9%	5425	−21.8%	5646	−18.6%	5655	−18.5%	6791	−2.1%	7684	**+10.8%**	5368	−22.6%	7786	**+12.3%**	6936
h2	4587	−29.2%	4987	−23.1%	4944	−23.7%	5000	−22.9%	7183	**+10.8%**	6977	**+7.7%**	4719	−27.2%	7571	**+16.8%**	6481
jme	7383	−4.0%	7357	−4.4%	7380	−4.1%	7446	−3.2%	7605	−1.1%	7997	**+4.0%**	7353	−4.4%	7673	−0.3%	7693
jython	6855	−21.1%	7234	−16.7%	7117	−18.1%	7247	−16.6%	8757	**+0.8%**	10415	**+19.9%**	6963	−19.8%	10261	**+18.1%**	8687
kafka	6627	−11.2%	6469	−13.4%	6585	−11.8%	6735	−9.8%	7136	−4.4%	7333	−1.8%	6376	−14.6%	8031	**+7.6%**	7467
luindex	4564	−20.6%	4589	−20.1%	4583	−20.3%	4704	−18.1%	5851	**+1.8%**	6733	**+17.2%**	4632	−19.4%	6662	**+15.9%**	5747
lusearch	4939	−33.8%	4736	−36.5%	4815	−35.5%	4972	−33.4%	7263	−2.7%	8812	**+18.1%**	4747	−36.4%	9341	**+25.1%**	7464
pmd	4481	−31.1%	4423	−32.0%	4660	−28.3%	4841	−25.5%	6293	−3.2%	6546	**+0.7%**	4647	−28.5%	6821	**+4.9%**	6502
spring	5192	−28.5%	5233	−27.9%	5200	−28.4%	5542	−23.7%	7024	−3.2%	7916	**+9.1%**	5195	−28.4%	7024	−3.2%	7259
sunflow	6732	−13.4%	6197	−20.2%	7051	−9.3%	7481	−3.7%	7881	**+1.4%**	9834	**+26.6%**	5955	−23.4%	9129	**+17.5%**	7770
tomcat	9139	−14.5%	9077	−15.1%	9295	−13.0%	9278	−13.2%	10599	−0.8%	11203	**+4.8%**	9189	−14.0%	11063	**+3.5%**	10689
xalan	1939	−28.1%	2112	−21.7%	1938	−28.1%	2115	−21.6%	2688	−0.3%	2985	**+10.7%**	1974	−26.8%	3703	**+37.3%**	2697
zxing	3429	−31.8%	3475	−30.9%	3494	−30.6%	3783	−24.8%	4752	−5.5%	5156	**+2.5%**	3556	−29.3%	5672	**+12.7%**	5031
Average Overhead		−19.90%		−24.64%		−23.62%		−20.90%		−5.16%		**+7.22%**		−25.24%		**+11.47%**	

−0.3%, respectively. The variations occur since the DaCapo benchmark suit tests different performance aspects, which CFI affects differently. For instance, the `xalan` benchmark evaluates XML processing tasks' performance, particularly XSLT (Extensible Stylesheet Language Transformations) transformations. XSLT is a language for transforming XML documents into other XML documents, HTML, or text formats. Xalan[9] is an open-source XSLT processor developed by the Apache XML Project. Overall, CFI might affect the performance of the `xalan` benchmark due to the nature of XML processing tasks, which often involve indirect function calls, dynamic dispatch operations, code size considerations, and interactions with compiler optimizations. Another example is the Spring Framework[10]. It is a popular framework for building Java-based enterprise applications, providing comprehensive support for dependency injection, aspect-oriented programming, transaction management, and other enterprise features. The `spring` benchmark typically involves simulating a web-based application scenario in which multiple concurrent clients interact with a server-side application built using the Spring Framework. The benchmark measures the application's throughput, response time, scalability under various load conditions, resource usage, and other relevant metrics. Overall, CFI might improve the performance of the `spring` benchmark by optimizing code generation and minimizing runtime overhead for the reasons mentioned above regarding the CFI variants, outperforming the JVM compilation without CFI. These factors contribute to a more efficient and scalable execution environment for Java enterprise applications built using the Spring Framework.

In addition to the performance evaluation, we evaluated the resulting size of the JVM, namely the `libjvm.so` file. The results are shown in Figure 8. The largest file is clearly generated using all CFI variants, and the smallest file is generated without any CFI. We could not observe a connection between the size, performance, or amount of ignored files.

[9]https://xalan.apache.org/

[10]https://spring.io/projects/spring-framework

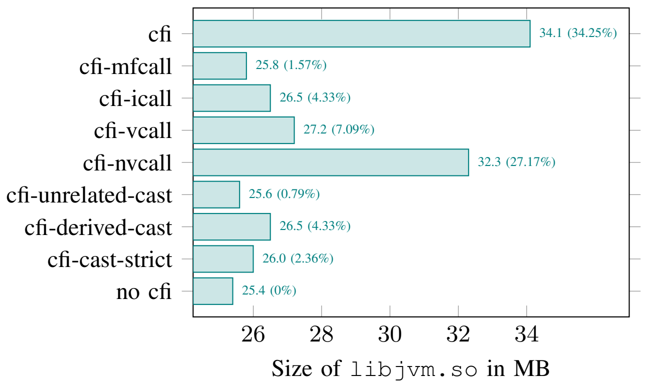


Fig. 8. Comparison of JVM sizes based on CFI variants and overhead (%)

We conducted performance evaluations of the CFI-integrated JVM discussed in Section IV-A using the DaCapo benchmark suite. Our findings reveal a performance overhead of 11.49%, falling within the expected range (⩽1%-15%) outlined in LLVM's documentation [12] and answering *RQ3*. Of particular interest is the unexpected discovery that applying the seven CFI variants individually results in performance enhancements compared to the JVM version without CFI. This intriguing outcome suggests avenues for further research into Control Flow Integrity, exploring its potential not only to enhance security but also to improve system performance.

V. Discussion - Lesson Learnt

Introducing Control Flow Integrity (CFI) into the Java Virtual Machine (JVM) of OpenJDK presents challenges and opportunities. In this Section, we discuss how the negative aspects of CFI integration can be mitigated by leveraging positive elements and vice versa. Integrating CFI into the JVM poses significant challenges, notably in terms of time and resource consumption. The manual nature of our CFI introduction process and the time-consuming compilation of the JDK results in a cumbersome workflow. Adjustments to Makefiles and complex debugging further contribute to the time expenses. In addition, there is the negative aspect of a significant increase in the JVM's performance by an average of almost 11.5% when using the DaCapo benchmark suit for evaluation. This performance penalty renders CFI impractical for deployment in performance-critical environments such as servers. Additionally, the 34% increase in size presents challenges, particularly for resource-constrained platforms like mobile devices.

Despite these challenges, our research uncovers several positive aspects that can mitigate the negative impact of CFI integration. Notably, only a tiny fraction of files (41 out of 3955) require exclusion from CFI protection, resulting in a relatively small amount of unprotected code (13.41%). Analyzing the root causes and, e.g., refactoring the code might further reduce this percentage. Furthermore, the size overhead is negligible, primarily for servers and desktops, although it may be more significant for mobile devices. While time-consuming, the introduction process can be partially automated, reducing manual interaction and time requirements. This automation can help alleviate the burden associated with the CFI integration process. Significantly, performance improves for single CFI variants, offering promising possibilities for enhancing both security and performance in the JDK.

While the challenges of CFI integration are substantial, they can be addressed by leveraging the positive aspects disclosed in our research. For example, automating the introduction process can reduce manual effort and time consumption, mitigating the negative impact of manual adjustments and debugging. Additionally, the performance improvements observed with single CFI variants motivate further research to optimize CFI integration and performance. Conversely, the positive aspects of CFI integration, such as the minimal amount of unprotected code and the potential for performance improvements, can help justify the time and resource investment required for integration. By optimizing our CFI integration processes and taking advantage of the observed performance benefits, the JVM can benefit from both increased security and decreased performance. This renders CFI a viable option for improving the security posture of the Java Virtual Machine.

Introducing CFI into the JVM presents significant challenges, including time-consuming integration processes, notable performance penalties, and increased code size. In addition to the general time-consuming integration, instructions and necessary adjustments, such as modifying Makefiles before starting the debugging process and excluding CFI-incompatible files, are necessary. These specific findings directly address *RQ2*. Despite these hurdles, our research reveals positive aspects, such as minimal file exclusion from CFI protection and potential performance improvements with individual CFI variants. By addressing these challenges and optimizing the integration process, CFI becomes a viable option for enhancing the security of programs like the JVM.

VI. Related Work

We split the Related Work Section into two main parts, other existing CFI approaches, Section VI-A), and Section VI-B focusing on previous works aiming to enhance the Java Virtual Machine's security.

A. Control Flow Integrity Approaches

Over the years, numerous CFI approaches [7], [8], [36]–[38] have been formalized but barely implemented. The implementation for the LLVM compiler Clang is among the most prominent ones, as it is for examples used by Google [18]. Most other available CFI tools are tailored for a particular purpose, while LLVM's CFI implementation is a more general-purpose solution. In the following, we discuss four of these alternative approaches.

EC-CFI [39] is one of the most recent CFI implementations, published by Nasahl et al. in 2023. EC-CFI proposes a novel approach to enhancing CFI using code encryption to mitigate fault attacks. The approach is to encrypt and dynamically decrypt critical code segments at runtime, thereby thwarting attackers' attempts to manipulate control flow by inducing faults in the execution.

CFIXX [40] is a CFI solution designed to provide fine-grained control flow protection. It generally incurs low performance overhead due to its efficient design and hardware support. By leveraging lightweight metadata and optimizing runtime checks, CFIXX achieves effective control flow protection with minimal impact on application performance. The performance overhead of CFIXX is typically less than 5%, making it suitable for use in various software environments without significant degradation in execution speed.

PICFI [10] verifies that each indirect branch target matches a valid control flow graph (CFG) node at runtime. PICFI is designed to work with position-independent executables, binaries that can be loaded at any memory address. This makes it suitable for modern software environments where address space layout randomization (ASLR) is commonly employed to thwart memory-based attacks.

binCFI [8] is a CFI implementation focusing on binary-level. It analyzes binary executables and inserts runtime checks to ensure that the control flow adheres to a predefined control flow graph. binCFI offers fine-grained control over the enforcement policies and can be tailored to specific application requirements. Yan Lin [11] evaluated the performance of binCFI and found that it comes with an overhead of up to almost 30%.

Zhang et al. [41] introduce CCFIR, a practical method that combines control flow integrity and code section randomization in binary executables to prevent control flow hijacking attacks. The performance overhead, on average, is only 3.6% [42]. The approach is designed for real-world effectiveness, aiming to provide robust protection for binary executables.

Nevertheless, none of the aforementioned CFI approaches provides a real-world deployable implementation.

B. JVM Security Efforts

Since the JVM's birth, efforts have been made to strengthen its security. These initiatives encompass a spectrum of methods, from mandatory access control to fuzzers. In the subsequent paragraphs, we outline various tools and approaches employed in this pursuit.

CONFUZZION [43] is a fuzzing tool designed specifically for testing the security and robustness of JVM implementations. It works by generating and executing a large number of Java programs to trigger unexpected or erroneous behavior in the JVM. CONFUZZION applies random mutations to the bytecode of Java programs, such as inserting incorrect instructions or modifying existing ones, to explore different code paths and uncover potential vulnerabilities or bugs in the JVM. By fuzzing the JVM, CONFUZZION helps identify and mitigate security issues, memory corruptions, or other vulnerabilities that attackers could exploit.

Brennan et al. [44] introduce a method to find JIT-induced side-channel vulnerabilities in JVMs using fuzzing. It outlines a systematic process to generate and run Java programs aiming to trigger JIT optimizations that might leak sensitive data through side channels. By fuzzing the JVM, the paper seeks to reveal vulnerabilities exploitable by attackers to infer secrets or alter Java program execution.

Classming [45] is a security tool designed to detect class boundary violations in Java programs. It analyzes bytecode to identify instances where a class in one package attempts to access a class in another package in violation of the defined access control rules. By detecting these violations, Classming helps identify potential security vulnerabilities related to access control within Java applications.

Venelle et al. [46] propose adding Mandatory Access Control (MAC) capabilities to the JVM. This enhancement aims to enforce fine-grained access control policies on Java applications, enhancing security by restricting the actions that Java code can perform based on security labels or attributes.

Pridgen et al. [47] address the issue of reducing persistent latent secrets in the HotSpot JVM. They focus on mitigating the risk posed by sensitive data remaining in memory after it is no longer needed, which attackers could access. The paper proposes techniques to minimize the exposure of such secrets by implementing memory management improvements within the HotSpot JVM.

Ion et al. [48] discuss extending the JVM to enforce fine-grained security policies on mobile devices. It proposes enhancements to the JVM architecture to enable more granular control over security policies, especially in mobile environments where security is crucial. These enhancements may include mechanisms for runtime monitoring, access control, and policy enforcement within the JVM.

Riom et al. [49] examine Android's Java Class Library (JCL), focusing on its evolution and security implications. The insights are relevant while it does not directly address the JVM's security. It analyzes changes in the Android JCL over time, identifying security-related modifications and their impact on Android app security. By understanding the evolution of the Android JCL and its security implications, the paper informs strategies for enhancing security within the JVM ecosystem, especially concerning mobile application security and Java bytecode execution on Android devices. All these efforts highlight the importance of further improving the JVM's security.

C. Discussion of Control Flow Integrity's Limitations

LLVM's Control Flow Integrity can suffer from limitations in precision, which may allow specific attacks to bypass its safeguards. For example, CFI often depends on indirect control flow transfers, such as indirect function calls, restricted to a predetermined set of allowed targets. However, if the set of allowed targets is too large, attackers may still be able to exploit the system by targeting the remaining permissible options [50]. This issue, commonly known as the over-approximation challenge, stems from the algorithm used to generate the control flow graph (CFG) and define the set of permissible jump and call targets. CFI operates on the assumption that the targets of indirect control flows are clearly defined and that a legitimate set of control flows can be established. However, in complex systems, especially those involving dynamic code loading or reflection, this assumption may not hold, leading to potential security gaps [51].

In addition, although CFI is an efficient mitigation technique, it does not encompass all potential attack vectors. Its primary focus is on indirect control flow transfers, but it might not fully shield against other types of attacks, such as direct memory corruption or data-oriented exploits [40]. This highlights the importance of integrating CFI within a broader security strategy, including additional measures to address these vulnerabilities. Moreover, sophisticated attackers might leverage advanced techniques, such as Return-Oriented Programming (ROP) or Jump-Oriented Programming (JOP), to craft attacks that circumvent CFI's protections [52], [53]. This is especially concerning when the CFI implementation lacks fine granularity, allowing these methods to slip through the cracks. Despite efforts to minimize the performance impact of LLVM's CFI, an overhead still exists [12]. This performance cost can be particularly problematic in applications where efficiency is critical, rendering CFI less desirable for some use cases.

CFI might not be compatible with all code bases or development practices, especially in legacy systems where assumptions about control flow integrity were not considered during initial design [15], [54]. This highlights the importance of early security planning and the need to consider control flow integrity from the outset of a project. In addition, it underscores the importance of our work and its future extension.

VII. Conclusion

In this work, we enhanced the security of the OpenJDK JVM by integrating LLVM's Control Flow Integrity (CFI) implementation to mitigate the impact of memory corruption vulnerabilities. Our manual approach to CFI integration offers a generic solution applicable to a wide range of real-world projects beyond the JVM ecosystem. Through our evaluation using the DaCapo benchmark suite, we assessed the performance implications of our CFI-integrated JVM version.

Our findings reveal that while introducing CFI incurs an average performance overhead of nearly 11.47% and a 34.25% increase in binary size, it introduces crucial security enhancements. Notably, our research uncovers that specific CFI subcategories, when used individually, improve the JVM's performance. This discovery underscores the potential for CFI to strengthen security and enhance application performance overall.

Overall, our study contributes to advancing the understanding of CFI integration in software, offering insights into both the challenges and opportunities associated with securing software systems against memory corruption attacks. Future research may explore further optimizations to minimize performance overhead while maximizing security benefits, paving the way for more resilient and efficient applications.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[2] The MITRE Corporation, "CWE - Common Weakness Enumeration," https://cwe.mitre.org/, accessed 2024-05-01.

[3] ——, "CWE - 2023 CWE Top 10 KEV Weaknesses," https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html, Dec. 2023, accessed 2024-05-01.

[4] ——, "CWE - 2023 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, Nov. 2023, accessed 2024-05-01.

[5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[6] ——, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[7] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[8] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *Proceedings of the 31st annual computer security applications conference*, 2015, pp. 331–340.

[9] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 927–940.

[10] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 914–926.

[11] Y. Lin, *Novel techniques in recovering, embedding, and enforcing policies for control-flow integrity*. Springer Nature, 2021.

[12] The Clang Team, "Control Flow Integrity - Clang 19.0.0git documentation," https://clang.llvm.org/docs/ControlFlowIntegrity.html, accessed 2024-05-01.

[13] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM}," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 941–955.

[14] Jonathan Wakely, "vtv - GCC Wiki," https://gcc.gnu.org/wiki/vtv, Nov. 2014, accessed 2024-05-01.

[15] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "{CONFIRM}: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1805–1821.

[16] Google Groups, "Android Clang/LLVM Prebuilts," https://android.googlesource.com/platform/prebuilts/clang/host/linux-x86/+/main/README.md, accessed 2024-05-01.

[17] ——, "Android GCC 4.9 Deprecation Schedule," https://android.googlesource.com/platform/prebuilts/clang/host/linux-x86/+/refs/heads/main/GCC_4_9_DEPRECATION.md, accessed 2024-05-01.

[18] The Chromium Projects, "Control Flow Integrity | Android Open Source Project," https://source.android.com/docs/security/test/cfi, accessed 2024-05-01.

[19] Jake Edge, "Control-low integrity for the kernel [LWN.net]," https://lwn.net/Articles/810077/, Jan. 2020, accessed 2024-05-01.

[20] The Chromium Projects, "Control Flow Integrity," https://www.chromium.org/developers/testing/control-flow-integrity/, Nov. 2023, accessed 2024-05-01.

[21] Bugzilla, "510629 - (cfi)[meta] Ship Control Flow Integrity," https://bugzilla.mozilla.org/show_bug.cgi?id=510629, Aug. 2009, accessed 2024-05-01.

[22] I. Eauvidoum and disk noise, "Twenty years of Escaping the Java Sandbox," *Phrack Magazine*, vol. 0x10, no. 0x46, p. 0x07, 2021.

[23] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[24] Marco Benatto, "Fighting exploits with Control-Flow Integrity (CFI) in Clang," https://www.redhat.com/en/blog/fighting-exploits-control-flow-integrity-cfi-clang, May 2020, accessed 2024-05-08.

[25] Windows App Development, "Control Flow Guard - Win32 apps | Microsoft Learn," https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, accessed 2024-05-06.

[26] ShadowCallStack – Clang 19.0.0git documentation, "ShadowCallStack," https://clang.llvm.org/docs/ShadowCallStack.html, accessed 2024-05-06.

[27] National Security Agency (NSA), "ghidra/DevGuide.md at master · NationalSecurityAgency/ghidra," https://github.com/NationalSecurityAgency/ghidra/blob/master/DevGuide.md, accessed 2024-07-31.

[28] Red Hat Bugzilla, "1273053 - (CVE-2015-4843) CVE-2015-4843 OpenJDK: java.nio Buffers integer overflow issues (Libraries, 8130891)," https://bugzilla.redhat.com/show_bug.cgi?id=1273053, accessed 2024-05-08.

[29] The GDB developers, "GDB: The GNU Project Debugger," https://www.sourceware.org/gdb/, Mar. 2024, accessed 2024-05-11.

[30] Piotr Padlewski, "Devirtualization in LLVM and Clang - The LLVM Project Blog," https://blog.llvm.org/2017/03/devirtualization-in-llvm-and-clang.html, Mar. 2017, accessed 2024-05-12.

[31] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *ECOOP'91 European Conference on Object-Oriented Programming: Geneva, Switzerland, July 15–19, 1991 Proceedings 5*. Springer, 1991, pp. 21–38.

[32] Sergei Larin, Harsha Jagasia, Tobias Edler von Koch, "Impact of the current LLVM inlining strategy on complex embedded application memory utilization and performance - Impact-of-the-current-LLVM-inlining-strategy.pdf," https://llvm.org/devmtg/2017-02-04/Impact-of-the-current-LLVM-inlining-strategy.pdf, Feb. 2017, accessed 2024-05-12.

[33] D. Bounov, R. G. Kici, and S. Lerner, "Protecting c++ dynamic dispatch through vtable interleaving." in *NDSS*, 2016.

[34] The Clang Team, "Control Flow Integrity Design Documentation - Clang 19.0.0git documentation," https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html, accessed 2024-05-12.

[35] Chris Lattner, Dinakar Dhurjati, Gabor Greif, Joel Stanley and Reid Spencer, "LLVM's Programmer's Manual," https://releases.llvm.org/2.5/docs/ProgrammersManual.html#TypeResolve, accessed 2024-05-12.

[36] W. He, S. Das, W. Zhang, and Y. Liu, “Bbb-cfi: lightweight cfi approach against code-reuse attacks using basic block information,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 1, pp. 1–22, 2020.

[37] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, “Ccfi: Cryptographically enforced control flow integrity,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.

[38] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, “Opaque control-flow integrity.” in *NDSS*, vol. 26, 2015, pp. 27–30.

[39] P. Nasahl, S. Sultana, H. Liljestrand, K. Grewal, M. LeMay, D. M. Durham, D. Schrammel, and S. Mangard, “Ec-cfi: Control-flow integrity via code encryption counteracting fault attacks,” in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2023, pp. 24–35.

[40] N. Burow, D. McKee, S. A. Carr, and M. Payer, “Cfixx: Object type integrity for c++ virtual dispatch,” in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[41] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in *2013 IEEE symposium on security and privacy*. IEEE, 2013, pp. 559–573.

[42] S. Sayeed and H. Marco-Gisbert, “On the effectiveness of control-flow integrity against modern attack techniques,” in *ICT Systems Security and Privacy Protection: 34th IFIP TC 11 International Conference, SEC 2019, Lisbon, Portugal, June 25-27, 2019, Proceedings 34*. Springer, 2019, pp. 331–344.

[43] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. Le Traon, “Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities,” in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 586–597.

[44] T. Brennan, S. Saha, and T. Bultan, “Jvm fuzzing for jit-induced side-channel detection,” in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 1011–1023.

[45] Y. Chen, T. Su, and Z. Su, “Deep differential testing of jvm implementations,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1257–1268.

[46] B. Venelle, J. Briffaut, L. Clévy, and C. Toinard, “Security enhanced java: Mandatory access control for the java virtual machine,” in *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. IEEE, 2013, pp. 1–7.

[47] A. Pridgen, S. Garfinkel, and D. Wallach, “Present but unreachable: Reducing persistentlatent secrets in hotspot jvm,” 2017.

[48] I. Ion, B. Dragovic, and B. Crispo, “Extending the java virtual machine to enforce fine-grained security policies in mobile devices,” in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. Ieee, 2007, pp. 233–242.

[49] T. Riom and A. Bartel, “An in-depth analysis of android’s java class library: its evolution and security impact,” in *2023 IEEE Secure Development Conference (SecDev)*, 2023, pp. 133–144.

[50] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[51] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, “Control-flow integrity: Precision, security, and performance,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[52] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[53] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, “Finding cracks in shields: On the security of control flow integrity mechanisms,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1821–1835.

[54] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, “Out of control: Overcoming control-flow integrity,” in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 575–589.