System for Executing Encrypted Java Programs

Michael Kiperberg[1], Amit Resh[2], Asaf Algawi[2] and Nezer J. Zaidenberg[3]
[1]*Faculty of Sciences, Holon Institute of Technology, Holon, Israel*
[2]*Department of Mathematical IT, University of Jyvaskyla, Jyvaskyla, Finland*
[3]*School of Computer Science, The College of Management, Academic Studies, Rishon LeZion, Israel*
{*mikiperberg, amitr44, asaf.algawi*}*@gmail.com, nzaidenberg@me.com*

Keywords: Java, Trusted Computing, Hypervisor, Virtualization, Remote Attestation.

Abstract: An important aspect of protecting software from attack, theft of algorithms, or illegal software use, is eliminating the possibility of performing reverse engineering. One common method to deal with these issues is code obfuscation. However, it is proven to be ineffective. Code encryption is a much more effective means of defying reverse engineering, but it requires managing a secret key available to none but the permissible users. Adequate systems for managing secret keys in a protected trust-zone and supporting execution of encrypted native code have been proposed in the past. Nevertheless, these systems are not suitable as is for protecting managed code. In this paper we propose enhancements to these systems so they support execution of encrypted Java programs that are resistant to reverse engineering. The main difficulty underlying Java protection with encryption is the interpretation that is performed by the JVM. The JVM will require the key to decrypt the encrypted portions of Java code and there is no feasible way of securing the key inside the JVM. To solve this, the authors propose implementing a Java bytecode interpreter inside a trust-zone, governed by a thin hypervisor. This interpreter will run in parallel to the standard JVM, both cooperating to execute encrypted Java programs.

1 INTRODUCTION

Digital content such as games, videos, and the like may be susceptible to unlicensed usage, which has a significant adverse impact on the profitability and commercial viability of such products.

One way of preventing circumvention of the software licensing program, may be using a method of ”obfuscation”. The term obfuscation refers to making software instructions difficult for humans to understand by deliberately cluttering the code with useless, confusing pieces of additional software syntax or instructions. However, even when changing the software code and making it obfuscated, the content is still readable to the skilled hacker (Rolles, 2009; Bohne, 2008).

Additionally, publishers may protect their digital content product by encryption, using a unique key to convert the software code to an unreadable format, such that only the owner of the unique key may decrypt the software code. Such protection may only be effective when the unique key is kept secured and unreachable to an adversary. Hardware based methods for keeping the unique key secured are possible (Schellekens et al., 2008; Pearson, 2002; England et al., 2003; Zaidenberg et al., 2015), but may have significant deficiencies, mainly due to an investment required in dedicated hardware on the user side, making it costly, and, therefore, impractical. Furthermore, such hardware methods have been successfully attacked by hackers (Tarnovsky, 2012).

We would like to stress the key difference between native and managed execution environments. While it is possible to guarantee that a sequence of native instructions cannot be intercepted (read or modified) during its execution by a CPU (Averbuch et al., 2011; Averbuch et al., 2013), such a guarantee cannot be made for a managed execution environment, since an unexpected behavior can be introduced into the software that implements the managed execution environment. There is, therefore, a need for a technique for executing safely encrypted managed programs on the available managed execution environments. In addition to protection of native code, we previously researched the protecting of video streaming (Zaidenberg and David, 2013; Zaidenberg and David, 2014).

In this paper, we present a system that allows encrypting and executing programs written for the Java Virtual Machine (JVM) (Lindholm et al., 2013). The system execution engine is based on a thin hypervi-

Kiperberg, M., Resh, A., Algawi, A. and Zaidenberg, N.
System for Executing Encrypted Java Programs.
DOI: 10.5220/0006078902450252
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 245-252
ISBN: 978-989-758-209-7

sor and works in cooperation with the JVM through the standardized JVM Tool Interface (JVM TI) and the Java Native Interface (JNI). The hypervisor acquires the decryption key during the initialization of the JVM and is responsible for decrypting and executing the encrypted parts of the Java program.

2 JAVA BYTECODE

Java bytecode is the instruction set of the JVM (Lindholm et al., 2013). Programs written in Java are compiled to the Java bytecode and stored, together with additional information, in class files.

JVM is a stack machine: the arguments of an instruction are pushed onto a stack, the instruction is executed, which pops the arguments off the stack and the result is pushed back onto the stack.

Each method has its own stack and its own area of local variables (which also includes method's parameters). Any constants used in a method, e.g. numerical constants, type names, or method names, are stored in a constant pool belonging to the class in which the method is defined. The method references these constants via their indices.

Many languages, Java among them, have introduced a notion of exceptions. An exception is an abnormal condition detected by the program, which cannot be handled locally, i.e. in the method which detected this condition. Most exceptions occur synchronously as a result of an action by the thread in which they occur. An asynchronous exception, by contrast, can potentially occur at any point in the execution of a program. Asynchronous exceptions are not covered in our work. Synchronous exceptions can occur as a result of execution of the *athrow* instruction or any other instruction that specifies an exception as a possible result. For example, the *idiv* instruction, which divides two integers, throws an ArithmeticException if the value of the divisor is 0.

Each method may be associated with zero or more exception handlers. An exception handler specifies the range of instructions for which the exception handler is active, and describes the type of exception that the exception handler is able to handle. When an exception is thrown, the JVM searches for a matching exception handler in the current method. If a matching exception handler is found, the system branches to the exception handling code specified by the matched handler.

3 JAVA FILE STRUCTURE

Java is an object-oriented programming language. All code in a Java program is written in classes. The source code is compiled into intermediate bytecode that is stored in *class files* (Lindholm et al., 2013). Each *class file* contains the compiled bytecode of a class along with descriptions of its fields, interfaces and methods.

Similarly to other instrumentation tools (Chander et al., 2001; Lee and Zorn, 1997; Harkema et al., 2002), our encryption tool (Algawi et al., 2014) analyzes and modifies each *class file* that was defined to be protected.

4 JNI/JVM TI

JVM TI is an application programming interface (API) provided by the JVM that allows the inspection and the controlling of the state of the JVM and the program it executes. This API is usually used by performance profilers and debuggers (Binder and Hulaas, 2006; Luedde, 2012). JVM TI is a two-way interface. A client of JVM TI, an *agent*, can be notified of interesting occurrences through events. An agent can query the JVM through many functions, either in response to events or independent of them. An agent is realized in a dynamic library, a dynamic link library on Windows or a shared object on Linux.

In addition to events interception, JVM TI allows an agent to inspect and manipulate the state of the JVM and the state of the program it executes. For example, it allows to retrieve the methods of a particular class, obtain the method's name and bytecode. Another family of JVM TI functions allows inspection of dynamic aspects of the execution. For example, the GetLocalVariable functions retrieves the value of method's local variable (in Java parameters are also variables). Finally, another family of JVM TI functions allows modifying the state of the program. This family includes functions such as SetLocalVariable, which assigns a value to method's local variable, SetBreakPoint, which sets a breakpoint at a specified location of a specified method.

JNI is an API provided by the JVM that enables Java programs to call and be called by native programs. JNI provides functions that can inspect and manipulate Java objects. These functions can be subdivided to the following families: class operations, exceptions, accessing fields, calling methods, etc.

The class operations family includes functions such as FindClass, which loads a class by its name, and IsAssignableFrom, which determines whether an

object of one class can be safely cast to another class. The exceptions family includes functions such as Throw and ThrowNew, that request to handle the specified exception, and ExceptionOccurred, which determines whether an exception is being handled. The accessing fields family includes functions such as GetObjectField, which retrieves the value of the specified field in the specified object, and SetObjectField, which assigns a value to the specified field in the specified object. The calling methods family includes functions such as CallVoidMethod, which calls the specified method of the specified object, and CallNonvirtualVoidMethod, which calls the specified method of the specified class (not necessarily object's class).

5 THIN HYPERVISOR

A hypervisor, also referred to as a Virtual Machine Monitor (VMM), is software, which may be hardware-assisted, to manage multiple virtual machines on a single system (Popek and Goldberg, 1974). The hypervisor virtualizes the hardware environment in a way that allows several virtual machines, running under its supervision, to operate in parallel over the same physical hardware platform, without obstructing or impeding each other.

The authors propose the use of a type-1 hypervisor environment for securing a single guest stack. Rather than wholly virtualizing the hardware platform, a special breed of hypervisor, called a thin-hypervisor, is used (Chubachi et al., 2010). The thin-hypervisor is configured to intercept only a small portion of the system's privileged events. All other privileged instructions are executed without interception, directly, by the OS.

A thin-hypervisor is less susceptible to being hacked as a result of vulnerabilities, since its code and complexity are greatly reduced when compared to a full-blown hypervisor.

In the proposed system, to execute encrypted Java bytecode, the thin-hypervisor capabilities are exploited to decrypt the encrypted Java bytecode (using the secret key) into protected memory regions and following up with interpretation and execution of the decrypted instructions while in host mode. The hypervisor obtains the secret key through a process of remote attestation (Seshadri et al., 2004; Kiperberg and Zaidenberg, 2013; Kiperberg et al., 2015) Following Attestation our hypervisor protects the keys. (Resh and Zaidenberg, 2013).

6 SYSTEM DESIGN

The system we present comprises four main components: (1) encryption tool, (2) JVM TI agent, (3) thin hypervisor, (4) attestation server. Figure 1 depicts the relationship between these components.

The encryption tool processes each *class file* by first de-serializing it into memory based structures. The code bytes of each method are located and zeroed out to create a sequence of *nop* instructions except for the very first code byte and the last three bytes. In the first code-byte (offset 0) it always inserts an *aconst_null* opcode (a single-byte instruction that pushes a NULL on the operand stack). In the last 3 bytes it inserts a *jump* instruction that loops back to the 1st *nop* instruction (offset 1). The reason for this pattern is to allow a means to synchronize the JVM processing of the decrypted bytecode, as will be detailed later. The *aconst_null* opcode at the beginning of the method's code is required to appease the Java verifier. Without the *aconst_null*, the verifier contemplates that in the event of exception handling, the loop back to the method's start may have a stack depth of 1, while during other loops, stack depth is 0. This discrepancy is not allowed. With the *aconst_null* opcode, the stack depth is always 1 regardless, and thus allowed by the verifier. Methods that are smaller than 5 bytes and cannot accommodate this pattern are simply not encrypted.

The encryption tool extends the existing constant pool to make room for encrypted versions of protected methods' bytecode. The original bytecodes of each method are encrypted and inserted in a new record appended at the end of the constant pool table.

After adding all the new encrypted entries, the encryption tool adds a trailer record at the very end, detailing the number of preceding encrypted entries. When the Java class is loaded for execution, the runtime decryption and execution engine can find this information by looking up the trailer-record at the end of the constant pool.

Once encryption is completed, the encryption tool serializes the class structures back into a modified *class file*, which replaces the original one in the *jar file*.

During the initialization of the JVM TI agent, it deploys the hypervisor and installs interception functions for the following events (1) class loading, (2) breakpoint, (3) exception catch. The class loading event occurs whenever the JVM loads a class and before any of the class code is executed. Upon this event the agent inspects the class and determines whether it is encrypted. If so, the agent installs a breakpoint at the first instruction of each method. These break-

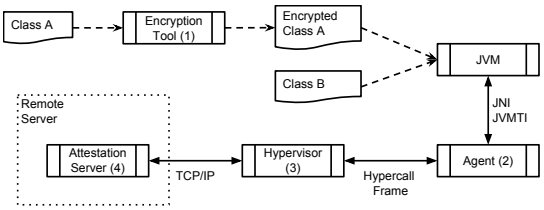

Figure 1: Relationship between the different components of the described system. The encryption tool (1) transforms regular Java classes into encrypted ones. Regular and encrypted Java classes are then loaded by the JVM. The JVM loads a JVM TI agent (2) through a JVM TI interface. The agent links the hypervisor to the JVM and assists in the interpretation process. The agent communicates with the JVM through JVM TI and JNI. The communication between the agent and the hypervisor is based on hypercalls and execution frames. The hypervisor receives the decryption key from a remote server, which attests the validity of the hypervisor and the hardware on which it executes.

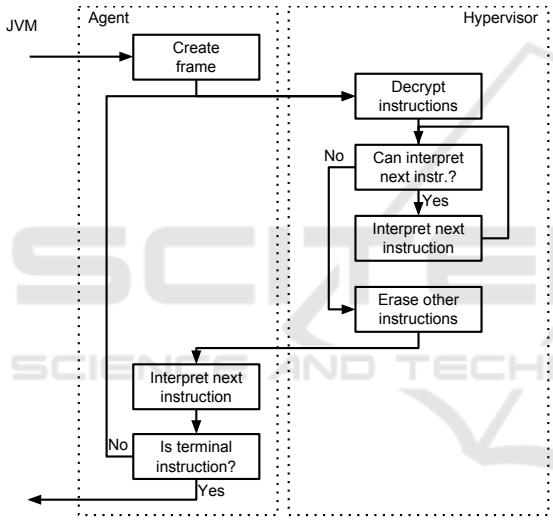

Figure 2: A simplified control flow during encrypted method execution. The JVM reaches the breakpoint installed by the agent, and transfers the control to the JVM TI agent. The agent creates a frame and transfers the control to the hypervisor. The hypervisor decrypts the instructions, and interprets them until an uninterpretable instruction is reached. Then the hypervisor erases all the other instructions and returns control to the JVM TI agent, which interprets the instruction and either transfers the control to the hypervisor or returns control back to the JVM.

points induce a breakpoint event on each entry to the encrypted methods. The agent intercepts the breakpoint event, resolves the method that hosts the hit breakpoint, and begins the interpretation process.

The interpreter constructs a frame, a data structure which constitutes the execution environment of the current method invocation (including the encrypted bytecodes of the current method), and transfers control to the hypervisor. The hypervisor decrypts the

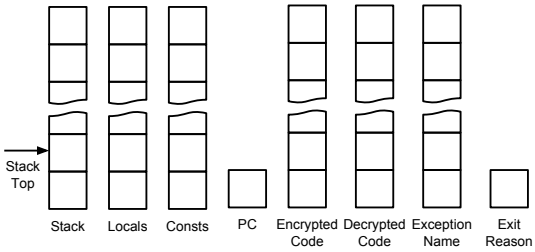

Figure 3: Frame's structure. The frame contains the computation stack and a pointer to its top element. The frame includes copies of all the local variables and the constant pool. The program counter contains the location of the next instruction to be executed. The "encrypted code" buffer contains the encrypted bytecode of the current method, which is then decrypted and interpreted by the hypervisor. The "decrypted code" buffer contains the last instruction that could not be interpreted by the hypervisor. If the hypervisor encountered an abnormal condition, it reports its nature through the exception name buffer and sets the "exit reason" field accordingly.

bytecodes and starts interpreting them one-by-one until it reaches an opcode which requires cooperation with the JVM. At this point, the hypervisor returns control to the agent and provides it with the instruction, which it could not interpret, in decrypted form. The agent proceeds by interpreting the instruction using JVM TI and JNI and then transfers control back to the hypervisor. Figure 2 presents the control flow diagram of the system operation.

7 CO-INTERPRETATION

The interpretation is performed by two interpreters: one is embedded in the JVM TI agent and the other is embedded in the hypervisor (further reference to the thin-hypervisor will be simply: "hypervisor"). Each opcode is interpreted by only one of the two interpreters. When one interpreter cannot continue interpretation, it transfers the control to the other interpreter. The interpreters share a data structure, which we call a frame, and in which they store the intermediate results of the interpretation as well as some additional information. The Frame's structure is depicted in Figure 3 and explained below.

We want to enable the interpreter, which is embedded in the hypervisor, to interpret as many instructions as possible. Many instructions operate only on the stack and the program counter (PC). These instructions include the following groups of instructions: arithmetic/logic, type conversion, stack management, control transfer. These instructions require the following information to be included in the stack: (a) PC, (b) stack. The load and store group of instruc-

tions allow the pushing of the value of local variables and constants onto the stack. In order to enable the hypervisor to interpret instructions in these groups, we include the (c) constant pool and the (d) local variables in the frame.

Instructions that belong to the following groups are interpreted by the JVM TI agent: object creation and manipulation, method invocation and return, and others. These instructions require cooperation with the JVM. For example, the *getfield* instruction, which pushes onto the stack the value of the specified field in the specified object, must inspect the internal representation of the object as defined by the JVM. Another example is the *return* instruction, which terminates execution of the current method. This instruction must modify the internal representation of the stack trace, which is managed by the JVM. Therefore, all these instructions are interpreted by the JVM TI agent via JNI and JVM TI functions.

In addition to the data structures which are used during interpretation, the frame includes three data structures which are used for communication between the two interpreters: (a) encrypted code, (b) decrypted code, (c) exit reason. Before transferring the control to the hypervisor, the encrypted code buffer is filled with the encrypted bytecodes of the current method by the JVM TI agent. The hypervisor decrypts the buffer and begins interpretation until it reaches an instruction, which cannot be interpreted (inside the hypervisor). This instruction is written to the decrypted code buffer and the exit reason is set to signify that the interpretation was suspended due to an uninterpretable instruction. The JVM TI agent interprets this single instruction and the process continues.

Exceptions are an essential part of Java; they are embedded into the low level bytecode instructions. Our interpreter supports exceptions (actually the support for asynchronous exceptions is partial) both in instructions that are interpreted by the JVM TI agent and those that are interpreted by the hypervisor. Clearly, our interpreter must cooperate with the JVM since it is possible that an encrypted method throws an exception which is handled by a non-encrypted method and vice-versa.

The implementation of exceptions in our interpreter can be divided into two parts: exception generation and exception handling. We begin our discussion with exception generation. The interpreter should generate an exception when it executes an instruction which generates an exception either explicitly (by executing the *athrow* instruction) or implicitly (e.g. by executing the *idiv* instruction with invalid arguments). The JVM TI agent delivers an exception to the JVM by calling the Throw or ThrowNew functions of JNI.

Unfortunately, the hypervisor cannot call JNI functions directly. Therefore, whenever the hypervisor detects an abnormal condition, it transfers the control to the JVM TI agent. The nature of the exception is delivered through the exception field of the current frame. The JVM TI agent then delivers the exception on the hypervisor's behalf.

In order to locate the correct exception handler, the JVM traverses the call stack of the currently executing methods. For each method, the JVM inspects the location in which the execution of the method was suspended and transferred to another method. These locations are part of the internal state of the JVM. The JVM updates these locations during program execution.

Our interpreters, however, cannot modify these locations directly, leaving the locations at 0 in all the encrypted methods. Whenever our interpreters need to modify the location of the currently executing method, they install a breakpoint at a desired location and return control to the JVM. The JVM executes the instrumented bytecode of the method (generally NOPS and a jump to the beginning at the end), as described in section 6, until it reaches the installed breakpoint, and transfers the control back to the JVM TI agent, which continues the interpretation process. Since our interpreter can affect only the location of the currently executing method, it must update the location before calling other methods.

8 MEASUREMENTS

According to (Collberg et al., 2007), invokevirtual is the second most popular instruction (appears with 8.9% frequency) and getfield is the fourth most popular instruction (5.4%). Unfortunately, these instruction cannot be interpreted inside the hypervisor, and, therefore, they are delivered in a decrypted form to the JVM TI agent, which is not considered secure. The hypervisor delivers about 38% of the instructions in a decrypted form back to the JVM TI. Therefore, in practice, only about 60% of the instructions in an encrypted class are actually hidden from an adversary. To compare performance of protected-Java vs. non-protected-Java, two empirical measurements were conducted.

The purpose of the first measurement was to compare code interpretation of decrypted Java bytecode in the hypervisor to regular, non-protected, JVM code interpretation. Algorithm 1 presents the pseudo code of a method that was run in protected and unprotected mode.

Algorithm 1: Algorithm measuring the correlation between instruction sequence length and its execution time.

```
for i = 1,10000 do
    t = System.nanoTime()
    baseTime += System.nanoTime() - t
end for
for all k ∈ {10, 20, ..., 190, 200, 400, ..., 10000} do
    for i = 1,10000 do
        t = System.nanoTime()
        for i = 1,k do
            nothing
        end for
        sumTime += System.nanoTime() - t
    end for
end for
Output: baseTime/10000, sumTime/10000
```

The first repeat block measures the overhead associated with system time measurement during 10000 iterations. The second repeat block performs the actual timed measurement. The number of interpreted instructions measured are a function of parameter k. The Java bytecodes measured are an empty *for* loop and include the instructions to manipulate the control variable and to cycle the loop. The value of k governs the number of instructions processed during each iteration. Measurements are performed for k assuming values 10 to 200 at increments of 10 and then 200 to 1000 — at increments of 200. To cancel out random measurement errors as a result of asynchronous events, 10000 iterations are performed for each value of k and the average value is output. The difference between *sumTime* and *baseTime*, the overhead measurement, is the net time duration of the instruction interpretation process. When measuring protected Java interpretation the *baseTime* of the non-protected java is subtracted from the measured result, in order to include in the time measurement a hypervisor exit and a hypervisor entry from/to the agent which is performed in order to call the System.nanoTime() method.

We observed that *baseTime* is 55ns in a non-protected program, and 28726 — when executed in a protected program. Since the System.nanoTime() method call is always executed by the JVM (it is not protected), the overhead time difference (28671 nanoseconds) is attributed to the transitions between the hypervisor and the agent.

The measurement result for the span of k values is plotted in Figure 4 on a logarithmic scale. Note that the transition overhead is significant (as compared to code interpretation) up until $k = 1000$. Since the size of the empty *for* loop in bytecode is about 10 bytes, it can be determined that the transition overhead is

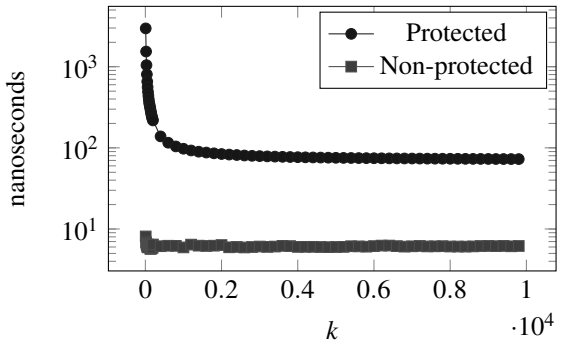

Figure 4: Execution time of Algorithm 1 in nanoseconds. The figure presents two graphs that correspond to two execution modes: (1) protected mode, in which the algorithm is realized by an encrypted method, (2) non-protected mode, in which the algorithm is realized by a regular, non-encrypted method.

significant for bytecode sizes of up to 10000 bytes. For larger bytecodes the performance comparison is stable at about a 1:10 factor.

While interpretation performance in the hypervisor can be optimized to achieve a result better than 1:10, this measurement shows that for all practical purposes the transition time will overshadow this. Therefore, optimization efforts should be concentrated there.

The purpose of our second study was to measure the overhead of calling a protected method as a function of the number of its parameters. When a protected method is called, the agent needs to construct the frame context and transfer control to the hypervisor (via a hypercall). The hypervisor needs to locate the encrypted bytecode, decrypt it and perform the local interpretation. When complete, it needs to return control to the agent, which will adjust the JVM frame context.

To measure this entire process, functions of a

Algorithm 2: Algorithm measuring the correlation between function's number of arguments and its invocation time.

```
baseTime = System.nanoTime()
for i = 1,40000 do
    nothing
end for
baseTime = System.nanoTime() – baseTime
for all k=0,15 do
    T_k = System.nanoTime()
    for i = 1,40000 do
        Call f_k(0, 1, ..., k)
    end for
    T_k = (System.nanoTime() – T_k) / 40000
end for
Output: baseTime, T_0, T_1, ..., T_15
```

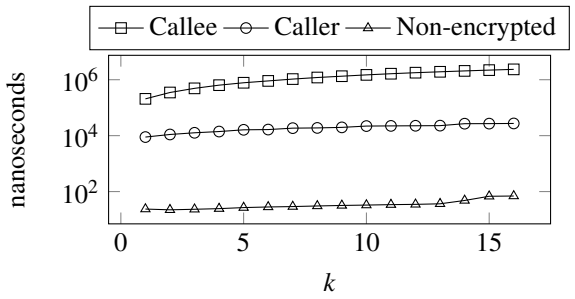

Figure 5: Execution time of Algorithm 2 in nanoseconds. The figure presents three graphs that correspond to three execution modes: (1) the callee, i.e. the function f_k, is encrypted (2) the caller is encrypted (3) neither the caller nor the callee are encrypted.

varying number of parameters were called and the call procedure was timed. The function contents were identical: $f_k(\text{int } p_0, ..., \text{int } p_k)\ \{p_0* = p_0;\}$ As in the previous study, each measurement was carried out multiple times (400000) to reduce error, and a *baseTime* was calculated to reflect the overhead associated with managing the loop process, as shown in Algorithm 2.

The *baseTime* was subtracted from the function call measurements results. Three types of measurements were conducted, each for all values of k: (a) non-protected program, (b) protected caller which calls an non-protected callee, (c) non-protected caller which calls a protected callee. Figure 5 plots these measurements on a logarithmic scale.

The largest overhead was acquired when the callees were encrypted, since this operation is the most involved: requiring preparation of the environment, transferring to the hypervisor, decrypting, interpreting in hypervisor and restoring the environment. It is roughly 1.5 to 2 orders of magnitude greater as compared to the case where only the caller was protected, since this has moderate overhead, only requiring the hypervisor to prepare the environment and transfer control to the agent.

The number of parameters generally increase the timing linearly, as can be expected. However, when comparing the two protected cases, it can be seen that the gap between the results increases with the number of parameters. This indicates that the overhead of preparing and restoring the environment is more significant when the callee function is protected.

9 CONCLUSIONS

As has been shown, Java programs can be, at least partially, protected from an adversary. We believe that this degree of protection is sufficient in cases where traditionally obfuscation was used. In other cases, which require a higher degree of protection, we suggest either avoiding using uninterpretable (by the hypervisor) instructions or use a tool which can reduce the frequency of uninterpretable instruction (by inlining methods, for instance).

While the performance penalty can be significant (2-2.5 orders of magnitude) on frequent transitions between the hypervisor and the JVM TI agent, the performance improves when longer sequences of instructions can be interpreted in the hypervisor at once. Obviously, sporadic execution of the protected parts of a Java program has little effect on the overall performance of the program.

REFERENCES

Algawi, A., Neittaanmaki, P., Zaidenberg, N. J., and Parisinos, T. (2014). In kernel implementation of rsa routines. In *ICCSM*, pages 149–153.

Averbuch, A., Kiperberg, M., and Zaidenberg, N. J. (2011). An efficient vm-based software protection. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 121–128.

Averbuch, A., Kiperberg, M., and Zaidenberg, N. J. (2013). Truly-Protect: An Efficient VM-Based Software Protection. *Systems Journal, IEEE*, 7(3):455–466.

Binder, W. and Hulaas, J. (2006). Exact and portable profiling for the jvm using bytecode instruction counting. *Electronic Notes in Theoretical Computer Science*, 164(3):45–64.

Bohne, L. (2008). Pandora's Bochs: Automated Unpacking of Malware. In *Pandora's Bochs: Automated Unpacking of Malware*.

Chander, A., Mitchell, J. C., and Shin, I. (2001). Mobile code security by Java bytecode instrumentation. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 2, pages 27–40. IEEE.

Chubachi, Y., Shinagawa, T., and Kato, K. (2010). Hypervisor-based Prevention of Persistent Rootkits. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 214–220, New York, NY, USA. ACM.

Collberg, C., Myles, G., and Stepp, M. (2007). An Empirical Study of Java Bytecode Programs. *Softw. Pract. Exper.*, 37(6):581–641.

England, P., Lampson, B., Manferdelli, J., Peinado, M., and Willman, B. (2003). A Trusted Open Platform. *Computer*, 36(7):55–62.

Harkema, M., Quartel, D., Gijsen, B., and van der Mei, R. D. (2002). Performance monitoring of Java applications. In *Proceedings of the 3rd international workshop on Software and performance*, pages 114–127. ACM.

Kiperberg, M., Resh, A., and Zaidenberg, N. J. (2015). Remote Attestation of Software and Execution-Environment in Modern Machines. In *CSCloud*.

Kiperberg, M. and Zaidenberg, N. J. (2013). Efficient Remote Authentication. In *The Journal of Information Warfare*, volume 12.

Lee, H. B. and Zorn, B. G. (1997). BIT: A Tool for Instrumenting Java Bytecodes. In *USENIX Symposium on Internet technologies and Systems*, pages 73–82.

Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2013). *The Java Virtual Machine Specification*. Oracle Corporation.

Luedde, M. (2012). Low impact debugging protocol. US Patent 8,312,438.

Pearson, S. (2002). *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Popek, G. J. and Goldberg, R. P. (1974). Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421.

Resh, A. and Zaidenberg, N. (2013). Can keys be hidden inside the cpu on modern windows host. In *European Conference on Information Warfare*, pages 231–235.

Rolles, R. (2009). Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 1–1, Berkeley, CA, USA. USENIX Association.

Schellekens, D., Wyseur, B., and Preneel, B. (2008). Remote Attestation on Legacy Operating Systems with Trusted Platform Modules. *Sci. Comput. Program.*, 74(1-2):13–22.

Seshadri, A., Perrig, A., van Doorn, L., and Khosla, P. (2004). SWATT: softWare-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282.

Tarnovsky, C. (2012). Attacking TPM part two. In *Defcon*.

Zaidenberg, N. and David, A. (2013). Truly protect video delivery. In *European Conference on Information Warfare*, pages 405–407.

Zaidenberg, N. and David, A. (2014). Maintaining streaming video drm. In *ICCSM*, pages 149–153.

Zaidenberg, N. J., Neittanmaki, P., Kiperberg, M., and Resh, A. (2015). Trusted computing and drm. In Martti Lehto, P. N., editor, *Cyber Security: Analytics, Technology and Automation*, chapter 13, pages 205–214. Springer, Springe.