Tightly-Coupled Self-Debugging Software Protection

Bert Abrath
Computer Systems Lab
Department of Electronics and Information Systems
Ghent University
bert.abrath@ugent.be

Bart Coppens
Computer Systems Lab
Department of Electronics and Information Systems
Ghent University
bart.coppens@ugent.be

Stijn Volckaert
Secure Systems Lab
University of Califoria at Irvine
stijnv@uci.edu

Joris Wijnant
Ghent University

Bjorn De Sutter
Computer Systems Lab
Department of Electronics and Information Systems
Ghent University
bjorn.desutter@ugent.be

ABSTRACT

Existing anti-debugging protections are relatively weak. In existing self-debugger approaches, a custom debugger is attached to the main application, of which the control flow is obfuscated by redirecting it through the debugger. The coupling between the debugger and the main application is then quite loose, and not that hard to break by an attacker. In the tightly-coupled self-debugging technique proposed in this paper, full code fragments are migrated from the application to the debugger, making it harder for the attacker to reverse-engineer the program and to deconstruct it into the original unprotected program to attach a debugger or to collect traces. We evaluate a prototype implementation on three complex, real-world Android use cases and present the results of tests conducted by professional penetration testers.

Keywords

Reverse Engineering; Anti-Debugging; Self-Debugging; Binary Rewriting

1. INTRODUCTION

Recently, there has been a trend to restrict more and more debugging capabilities in Android by blocking access to parts of the ptrace kernel interface. This fits in the broader trend to give apps as few permissions as possible, and to block access to generic OS features that typical, benign apps do not require, in an attempt to block malware from exploiting those features. In this case, blocking the kernel's debugging interface prevents malware from intervening in a running process' memory space and I/O.

SSPREW '16, December 05 - 06, 2016, Los Angeles, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4841-6/16/09...$15.00
DOI: http://dx.doi.org/10.1145/3015135.3015142

While this approach is generally sound, one problem is that its provisioning of security guarantees depends completely on the run-time environment. In this case, Google's targeted environment is that of end users that run an unrooted Android OS with the Developer Options disabled, and that have the legitimate desire to be protected against malware.

In at least one relevant scenario, however, the run-time environment can divert from the targeted one. This scenario is that of man-at-the-end (MATE) attacks. Many mobile applications embed assets from service providers, software providers, and content providers. Examples of such assets are cryptographic keys, algorithms that offer competitive advantages, and software components that control access to content and functionality. Vendors need to protect those assets against malware running on devices of benign users, but also against reverse engineering and tampering attacks conducted by malicious users. In many circumstances, such attacks are not performed on end-user consumer devices, but on developer boards in labs where the attackers control the whole environment, incl. the OS. It is then rather trivial for attackers to run the software on an OS with a re-enabled ptrace support, which then enables trace collection, live debugging, and other dynamic MATE attacks.

To provide protection against the malicious use of ptrace or other debugging interfaces in the MATE scenario, we propose to build the necessary protection into the software to be protected itself, rather than in the environment. We are not the first to do so, as many so-called anti-debugging techniques have been presented in the past. A myriad of simple techniques —most of which are hacks really— consist of dynamic checks that a process can execute to query the runtime environment for signs of active debugging [10, 23, 21, 5]. These techniques do not provide strong protection, however; many counter-techniques (i.e., debugger hacks) have been proposed to thwart the checks [18, 4, 11, 21].

Secondly, self-debugging has been proposed [20, 13]. This technique builds on the fact that all major OSs (Windows, Linux, Android, OS X, ...) support only one debugger process per debuggee process. What's more, hardware support for debugging (such as debug registers and hardware breakpoints) is designed for one debugger only (even though they might also be useful beyond that limitation). Investing in the development of effective and efficient OS support for multiple concurrent debuggers per debuggee is for the time being considered infeasible. So the strength of a self-debugger comes from occupying the single available debugger seat with a custom "debugger" that offers no useful debugging capabilities and that attackers cannot trivially replace with their own true debugger.

To achieve the latter with self-debugging, control flow transfers in the application are replaced by exception inducing instructions (typically breakpoint instructions) and a special-purpose debugger is injected into the original application. When the application launches, the self-debugger is launched as well, and attaches itself to the application. Whenever an exception is then thrown, the debugger intervenes and implements the original control flow transfer by transferring control to the original continuation point in the application. When the debugger is detached to make room for an attacker's debugger, the application itself lacks the necessary control flow and can hence no longer be traced or debugged live.

A major shortcoming of existing self-debugging schemes, however, is the simplicity of the self-debuggers. In essence, they implement an obfuscated, but relatively simple interprocess control flow transfer mechanism, of which the implementation (typically based on simple address translation table lookups) is completely predetermined by the developers of the protection tools. This feature, combined with the fact that only simple binary code patching is performed to convert individual control flow transfers to breakpoints, implies that it is relatively straightforward for knowledgeable hackers to inject a debugger-in-the-middle that iteratively resets each breakpoint to its original instruction, thus iteratively reconstructing and deobfuscating the unprotected program to re-enable standard tracing and live debugging techniques. Because the injection of breakpoints has left all other instructions in the program in their exact original location, the reconstruction only requires the attacker to flip some code bytes back to their original values, which can be determined from the simple input-output relation of the self-debugger, or even be obtained statically from its address translation tables.

In this paper, we push the state of the art in self-debugging. Relying on advanced binary rewriting techniques, we propose to migrate whole chunks of functionality from the original software to the self-debugger. This offers several advantages. First, the input-output behavior of the self-debugger is no longer pre-determined: Every time the self-debugger intervenes, it executes different functionality that is not predetermined, but that can instead vary as much as functionality in protected programs can vary. This makes the protection much more resilient against automated analysis, deobfuscation, and decompilation. Secondly, even if the attacker can figure out the control flow and the data flow equivalent of the original program, it becomes much harder for an attacker to undo the protection and to reconstruct that original program. Combined, we believe these two strengths make it much harder for an attacker to detach the self-debugger while maintaining a functioning program to be traced or live-debugged.

The contributions of this paper are the following:

- We present the design of a self-debugger that executes part of the original program functionality to make it harder for an attacker to detach the self-debugger and to deobfuscate the overall control and data flow.
- We present an open-source prototype implementation and tool support for protecting stand-alone programs as well as shared libraries.
- We discuss how to engineer the tool support to make it compatible with other software protections.
- We evaluate the tools and prototype on complex, real-life security-sensitive use cases, ranging from native libraries embedded in Android APKs and invoked with the JNI interface, to native plugins of the Android DRM server and the Android media server.
- We discuss the impact on attacker capabilities based on observations we made when professional penetration testers were hired to attack the protected use cases.

The remainder of this paper is structured as follows. In Section 2, we discuss the overall design and applicability of our self-debugger approach. Section 3 then discusses the necessary tool support. In Section 4, a number of implementation aspects are discussed in more detail, after which a prototype implementation is evaluated in Section 5. Additional attack vectors are discussed in Section 6, after which Section 7 draws conclusions and briefly discusses future work.

2. OVERALL SELF-DEBUGGER DESIGN

Figure 1 illustrates the basic concepts of our self-debugging scheme. Our design, prototype implementation, and presentation in this paper target Linux (and derivatives such as Android), but to the best of our knowledge, all aspects of the design are relevant and have direct counterparts on Windows, BSD variants, and OS X.

On the left of Figure 1, an original, unprotected application is depicted, incl. a small control flow graph fragment. The shown assembly code is (pseudo) ARMv7 code [22]. This unprotected application is converted into a protected application consisting of two parts: a debuggee that corresponds mostly to the original application as shown in the middle of the figure, and a debugger as shown on the right. Apart from some new components injected into the debuggee and the debugger, the main difference with the original application is that the control flow graph fragment has been migrated from the application into the debugger. Our design and our current implementation support all single-entry, multiple-exit code fragments that contain no interprocedural control flow such as function calls.

The migration of such fragments is more than simple copying: Memory references such as the `LDR` instruction need to be transformed because in the protected application the migrated code executing in the debugger address space needs to access data that still resides in the debuggee address space. All relevant components and transformations will be discussed in more detail in later sections.

At run time, the operation of this protected application is as follows. First, the debuggee is launched, as if it was the original application. A newly injected initializer then forks off a new process for the debugger, in which the debugger's initializer immediately attaches to the debuggee process.

When later during the program's execution the entry point of the migrated code fragment is reached, *one possible flow*

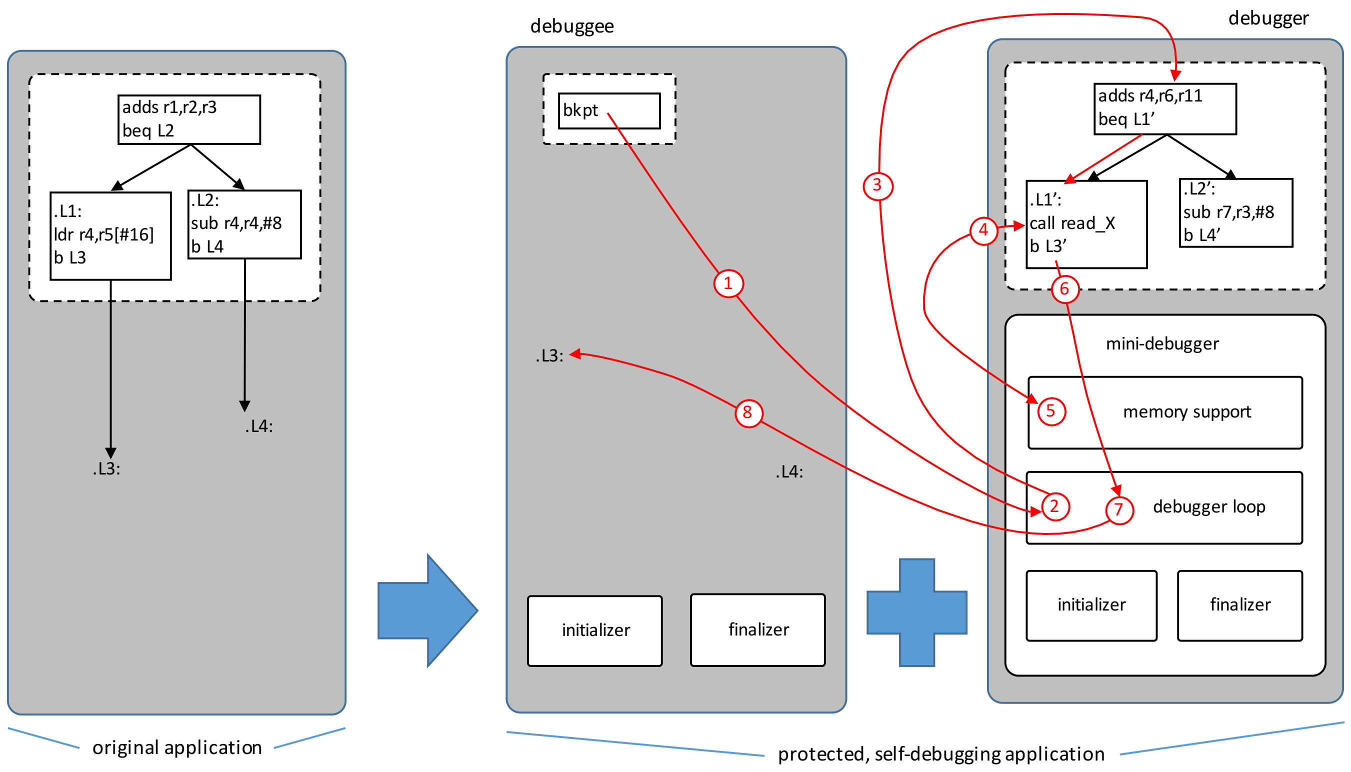

Figure 1: On the left, the original, unprotected application with a control flow graph fragment. In the middle, the protected application from which the fragment has been omitted, but with minimal functionality to launch and kill the debugger component. On the right, the debugger with the migrated fragment and mini-debugger functionality. Red, numbered edges and points show the control flow in the running, self-debugging application.

of control in the application follows the red arrows in Figure 1. In the application/debuggee, the exception inducing instruction is executed and causes an exception (①). The debugger is notified of this exception and handles it in its debugger loop (②). Amongst others, the code in this loop is responsible for fetching the process state from the debuggee, looking up the corresponding, migrated code fragment, and transferring control (③) to the entry point of that fragment. As stated, in that fragment memory accesses cannot be performed as is. So they are replaced by invocations (④) of memory support functions (⑤) that access memory in the debuggee's address space. When an exit point (⑥) is eventually reached in the migrated code fragment, control is transferred to the corresponding point in the debugger loop (⑦), which updates the state of the debuggee with the data computed in the debugger, and (⑧) control is transferred back to the debuggee. For code fragments with multiple exits, such as the example in the figure, the control can be transferred back to multiple continuation points in the debuggee. In this regard, our debugger behaves more complex than existing self-debuggers, which implement a one-to-one mapping between forward and backward control flow transfers between debuggee and debugger.

Eventually, when the application exits, the embedded finalizers will perform the necessary detaching operations.

It is important to note that this scheme cannot only be deployed to protect executables (i.e., binaries with a `main` function and entry point), but also to protect shared libraries. Just like executables, libraries can contain initializers and finalizers that are executed when they are loaded or unloaded by the OS loader. At that time, all of the necessary forking, attaching and detaching can be performed as well.

In the remainder of this paper, we will mostly write about protecting applications, but implicitly, we denote applications and libraries. The only aspect specific to libraries is the need for not only proper initialization, but for proper finalization of the debugger as well. This is necessary because it is not uncommon for libraries to be loaded and unloaded multiple times within a single execution of a program. For example, repetitive loading and unloading happens frequently for plug-ins of media players and browsers. Furthermore, whereas main programs consist of only one thread when they are launched themselves, at the point in time where they load and unload libraries, they can already consist of multiple threads. This complicates the attaching/detaching of debuggers to libraries.

3. TOOL SUPPORT

Figure 2 depicts one possible conceptual tool flow. The main components are discussed in the following sections.

3.1 Source Code Annotations

A number of options exist for determining the code fragments to be migrated to the debugger. One, depicted in the

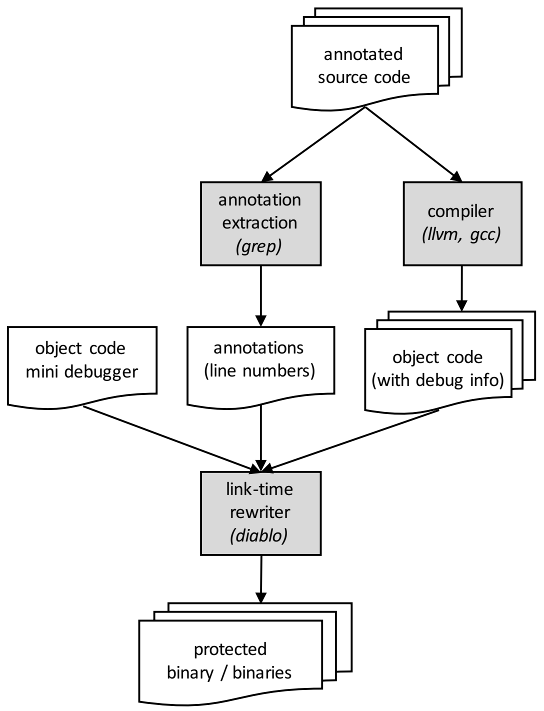
annotated source code
annotation extraction (grep)
compiler (llvm, gcc)
object code mini debugger
annotations (line numbers)
object code (with debug info)
link-time rewriter (diablo)
protected binary / binaries

Figure 2: Tool flow of self-debugging support

figure —and also used in our implementation— is to annotate source code with pragmas, comments, or any other form of annotations that mark the beginning and end of the code regions to be migrated to the debugger process. A simple `grep` suffices to extract annotations and their line numbers, and to store that information in an annotations file.

Alternative options would be to list the procedures or source code files to be protected, or to collect traces or profiles to select interesting fragments semi-automatically.

In that regard, it is important to note that the fragments to be migrated to the debugger should not necessarily be very hot fragments. To achieve a strong attachment between the debuggee and the debugger, it suffices to raise exceptions relatively frequently, but this does not need to be on the hottest code paths. We will discuss good strategies to select fragments in more detail later in the paper. Obviously, every raised exception will introduce a significant amount of overhead (context switch, many ptrace calls, ...), hence it is important to minimize their number without compromising the level of protection.

3.2 Standard Compilers and Tools

To deploy our self-debugging approach, any "standard" compiler can be used: Our technique does not impose any restrictions on the code generated by the compiler. In our experimental evaluation, we used both GCC and LLVM, in which we did not need to adapt or tune the code generation.

One requirement, however, is that the compiler and the binary utilities (the assembler and linker) provide the link-time rewriter with sufficiently accurate symbol and relocation information. This is required to enable reliable, conservative link-time code analyses and transformations to implement the whole self-debugging scheme, including the migration and transformation of the selected code fragments. Providing sufficiently accurate information is certainly within reach for commonly used tools. ARM's proprietary compilers, e.g., have done so for a long time by default, and for the GNU binutils, GCC, and LLVM, very simple patches[1] suffice to prevent those tools from performing overly aggressive symbol relaxation and relocation simplification, and to force them to insert mapping symbols to mark data in code. These requirements have been documented before, and it has been shown that they suffice to perform reliable, conservative link-time rewriting of code as complex and unconventional as both CISC (x86) and RISC (ARMv7) versions of the Linux kernel and C libraries, which are full of manually written assembly code [25].

A large, generic part of the debugger —the *mini-debugger*— can be precompiled with the standard compiler and then simply linked into the application to be protected. Other parts, such as the debug loop's *prologues* and *epilogues* for each of the migrated fragments, are generated by the link-time rewriter, as they are customized for each fragment.

To allow the link-time rewriter to identify the fragments that were annotated in the source code, it suffices to pass it the line number information extracted from the source code files, and to let the compilers generate object files with debug information. That debug information then maps all addresses in the binary code to source line numbers, which the rewriter can link to the line numbers from the annotations. To the best of our knowledge, all compilers and binary utilities support the generation of debug information.

3.3 Binaries, Libraries, and Processes

The link-time rewriter has two options to generate a protected application. A first option is to generate two binaries, one for the application/debuggee, and one for the debugger. From a security perspective, this might be preferable, because the application semantics and its implementation are then distributed over multiple binaries, which likely makes it even harder for an attacker to undo the protection, i.e., to patch the debuggee into the original application. This option does introduce additional run-time overhead, however, as the launching of the debugger then also requires loading the second binary.

The alternative option —that we use in our implementation— is to embed all debuggee code and all debugger code into one binary. In that case, simple forking will suffice to launch the debugger. Whether or not, and to what extent, this eases attacks on the protection provided by self-debugging is an open research question.

4. IMPLEMENTATION

4.1 Initialization & Finalization

We add an extra initialization routine to a protected binary. This routine is invoked as soon as the binary has been loaded (because it is assigned a high priority), after which all the other routines listed in the `.init` section of the binary are executed.

This initialization routine invokes `fork()`, thus creating two processes called the parent and child [16]. Once this routine is finished, the parent process will continue execution, typically by invoking the next initialization routine.

[1]Our patches are available at https://github.com/diablo-rewriter/toolchain-patches.

Two options exist for assigning the debugger and debuggee roles: After the fork, either the child process attaches to the parent process, or vice versa. In the former case, the child becomes the debugger and the parent becomes the debuggee, in the latter case the roles are obviously reversed.

The former option is highly preferred. The parent process remains the main application process, and it keeps the same process ID (PID). This facilitates the continuing execution or use of all external applications and inter-process communication channels that rely on the original PID, e.g., because they were set up before loading a protected library.

This scheme does come with its own problems, however. With `dlopen()` and `dlclose()` [15], shared libraries can be loaded and unloaded at any moment during the execution of a program. There is hence the potential problem that a protected shared library can be unloaded and *loaded again* while the originally loaded and forked off debugger has not yet finished its initialization. This can result in the simultaneous existence of two debugger processes, both attempting (and one failing) to attach to the debuggee. In order to avoid this situation, we block the execution of the thread that called `dlopen()`. So until that time, that thread cannot invoke `dlclose()` using the handle it got with `dlopen()`, and it cannot pass the handle to another thread either. An infinite loop in the debuggee's initialization routine prevents the loading thread from exiting the initialization routine before the debugger allows it to proceed.

The initialization routine also installs a finalizer in the debuggee. This finalizer does not do much. At program exit (or when the shared library is unloaded) it simply informs the mini-debugger of this fact by raising a `SIGUSR1` signal, causing the mini-debugger to detach from all the debuggee's threads and to shut down the debugger process.

4.2 Multithreading Support

Attaching the debugger is not trivial, in particular in the case of protected shared libraries. When a library is loaded, the application might consist of several threads. Only one of them will execute the debuggee initialization routine during its call to `dlopen()`. This is good, as only one fork will be executed, but it also comes with the downside that only one thread will enter the infinite loop mentioned in the previous section. The other threads in the debuggee process will continue running, and might create new threads at any point during the execution of the debuggee initialization routine or of the debugger initialization routine.

To ensure proper protection, the debugger should attach to every thread in the debuggee process as part of its initialization. To ensure that the debugger does not miss any threads created in the debuggee in the meantime, we use the `/proc/[pid]/task` directory, which contains an entry for every thread in a process [17]. The debugger process attaches to all the threads by iterating over the entries in this directory, and by keeping iterating until no new entries are found. Upon attachment to the thread, which happens by means of a `PTRACE_ATTACH` request, the thread is also stopped (and the debugger is notified of this event by the OS), meaning that it can no longer spawn new threads from then on. So for any program that spawns a finite number of threads, the iterative procedure to attach to all threads is guaranteed to terminate. Once all threads have been attached to, the infinite loop in the debuggee is ended and its stopped threads are allowed to continue.

When additional threads are created later during the program execution, the debugger is automatically attached to them by the OS, and it gets a signal such that all the necessary bookkeeping can be performed.

4.3 Control Flow

Transforming the control flow in and out of the migrated code fragments consists of several parts. We discuss the raising of exceptions to notify the debugger, the transferring of the ID that informs the debugger about the fragment it needs to execute, and the customized prologues and epilogues that are added to every migrated code fragment.

4.3.1 Raising Exceptions

The notification of the debugger can happen through any instruction that causes an exception to be raised. In our implementation, we use a software breakpoint (i.e., a `BKPT` instruction on ARMv7) for simplicity. Other, less conspicuous exceptions can be used, such as those caused by illegal or undefined instructions. When such instructions are reachable via direct control flow (direct branch or fall-through path), they can of course easily be detected statically. But when indirect control flow transfers are used to jump to data in the code sections, and the data bits correspond to an illegal or undefined instruction, static detection can be made much harder. Likewise, legal instructions that throw exceptions only when their operands are “invalid” can be used to conceal the goal of the instructions. Such instructions include division by zero, invalid memory accesses (i.e., a segmentation fault), or the dereferencing of an invalid pointer (resulting in a bus error).

4.3.2 Transferring IDs

We call the thread in the debuggee that raises an exception the *requesting thread*, as it is essentially asking the debugger to execute some code fragment.

The debugger, after being notified about the request by the OS, needs to figure out which fragment to execute. To enable this, the debuggee can pass an *ID* of the fragment in a number of ways. One option is to simply use the address of the exception inducing instruction as an ID. Another option is to pass the ID by placing it in a fixed register right before raising the exception, or in a fixed memory location. In our implementation, we used the latter option. As multiple threads in the debuggee can request a different fragment concurrently, the memory location cannot be a global location. Instead, it needs to be thread-local. As each thread has its own stack, we opted to pass the fragment's ID via the top of the stack of the requesting thread.

Depending on the type of instruction used to raise the exception, other methods can be envisioned as well. For example, the dividend operand of a division (by zero) instruction could be used to pass the ID as well.

Finally, many data obfuscation techniques [5] can be used to hide the values of the passed IDs, thus complicating the reverse engineering of the control flow in the original application.

4.3.3 Prologues and Epilogues

The debugger loop in the mini-debugger is responsible for fetching the program state of the debuggee before a fragment is executed, and for transferring it back after its execution. Standard `ptrace` functionality is used to do this: With one

API call, the status of all registers in the debuggee can be retrieved in a struct in the debugger. Likewise, one API call suffices to update all registers in the debuggee with the values in a struct.

For every migrated code fragment, the debug loop also contains a custom prologue and epilogue to be executed before and after the code fragment resp. The prologue loads the necessary values from the struct into the debugger's registers, the epilogue writes the necessary values back into the struct. The prologue is customized in the sense that it only loads the registers that are actually used in the fragment (the so-called live-in registers). The epilogue only stores the values that are live-out (i.e., that will be consumed in the debuggee) and that can have been updated by the executed code fragment.

4.4 Memory Accesses

For every load or store operation in a migrated code fragment, an access to the debuggee's memory is needed. There exist multiple options to implement such accesses.

The first is to simply use `ptrace` functionality: The debugger can perform `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` requests to read and write in the debuggee's address space. In this case, per word[2] to be read or written, a ptrace system call is needed, which results in a significant overhead. Some recent Linux versions support wider accesses, but those are not yet available everywhere, such as on Android.

The second option is to open the `/proc/[pid]/mem` file of the debuggee in the debugger, and then simply read or write in this file. This is easier to implement, and wider data can be read or written with a single system call, so often this method is faster. Writing to another process's `/proc/[pid]/mem` is not supported on every version of the Linux/Android kernels, however, so in our prototype write requests are still implemented with the first option.

A third option builds on the second one: if the binary-rewriter can determine which memory pages will be accesses in a migrated code fragment, the debug loop can actually copy those pages into the debugger address space using option 2. The fragment in the debugger then simply executes regular load and store operations to access the copied pages, and after the fragment has executed, the updated pages are copied back to the debuggee. This option can be faster if, e.g., the code fragment contains a loop to access a buffer on the stack. Experiments we conducted to compare the third option with the previous two options revealed that this technique might be worthwhile for as few as 8 memory accesses. We did not implement reliable support for it in our prototype, however: A conservative link-time analysis for determining which pages will be accessed by a code fragment remains future work at this point.

A fourth potential option is to adapt the debuggee, e.g., by providing a custom heap memory management library (malloc, free, ...) such that all allocated memory (or at least the heap) is allocated as shared memory between the debuggee and the debugger processes. Then the code fragments in the debugger can access the data directly. Of course, the fragments still need to be rewritten to include a translation of addresses between the two address spaces, but likely the overhead of this option can be much lower than the overhead of the other options. Implementing this option and evaluating it remains future work at this point.

[2]The ptrace word size depends on the processor architecture.

Security-wise, the different options will likely also have a different impact, in the sense that they will impact the difficulty for an attacker to reverse-engineer the original semantics of the program and to deconstruct the self-debugging version into an equivalent of the original program. Without penetration tests, we are not in a position yet to make strong statements in any one direction, however.

4.5 Combining self-debugging with other protections

To provide strong software protection against MATE attacks, one protection technique does typically not suffice. For example, on top of self-debugging, a good protection also requires obfuscation to prevent static analysis, and anti-tampering techniques to prevent all kinds of attacks.

The binary rewriter that implements our self-debugging approach also applies a number of other protections, incl.

- Control flow obfuscations: the well-known obfuscations of opaque predicates, control flow flattening, and branch functions [14, 26, 6].
- Code layout randomization: code from all functions is mingled and the order and layout are randomized.
- Code mobility: a technique in which code fragments are removed from the static binary and only downloaded, as so-called mobile code, into the application at run time [3].
- Code guards: online and offline implementations of techniques in which hashes are computed over the code in the process address space to check that the code has not been altered.
- Control flow integrity: a lightweight technique in which return addresses are checked to prevent that internal functions are invoked from external code.
- Instruction set virtualization: a technique with which native code is translated to bytecode that is interpreted by an embedded virtual machine instead of executed natively.

Combining the self-debugging technique with all those protections poses no problem in practice. In the link-time rewriter, it is not difficult to determine a good order to perform the transformations for all of the protections, and to prevent that multiple techniques are deployed on the same code fragments when those techniques do not compose.

For example, in our prototype implementation[3] we do not yet support mobile code in the debugger. Also, the debugger needs to know the exact continuation points to transfer control to in the debuggee. But mobile code is relocated to randomized locations. So at least for the time being, our prototype implementation of self-debugging does not compose, on the same fragment, with code mobility. Handling all protections correctly requires some bookkeeping, but nothing complex.

[3]The source code of the Diablo link-time rewriter supporting all other protections is available at https://github.com/diablo-rewriter. The source code annotation extractor is available at https://github.com/aspire-fp7. The self-debugger source code and the Diablo support for the injection of the mini-debugger and the auxiliary routines, as well as for migrating the code fragments is available at https://github.com/diablo-rewriter/diablo-selfdebugger.

As for the run-time behavior, the techniques compose as well. Multiple techniques require initializers and finalizers, but in the debugger process we do not want to execute the initializers of the other protections, as that debugger process should only be a debugger, and not another client for code mobility or any other technique. To prevent the other initializers from executing, the self-debugger initializers are given the highest priority. They are executed first when a binary or library is loaded, and the debugger initialization routine implements in fact both the real initializer, as well as the debug loop. The routine therefore never ends (that is, as long as the finalizer is not invoked), and hence control is never transferred to the other initializers that might be present in the binary.

Finally, we should point out one limitation of our current design and tool support. As presented, it can only be deployed once in a running process. In other words, with the design and implementation details presented in the remainder of this paper, either the main application or one of the shared libraries can be protected, but not more. This limitation stems from the fact that in order to protect multiple libraries (incl. possibly the main program), one debugger needs to contain, or have at least have access to, the migrated code fragments and all auxiliary code and data of all protected libraries. The extensions required to our scheme to support this are future work at this point.

5. EVALUATION

5.1 Evaluation Platform

Our prototype implementation of the self-debugger targets ARMv7 platforms. Concretely, we targeted and extensively evaluated the implementation on Linux 3.15 and (unrooted) Android 4.3+4.4. We also checked whether the techniques still work on the latest versions of Linux (4.7) and Android (7.0), and that is indeed the case.

Our testing hardware consist of several developer boards. For Linux, we used a Panda Board featuring a single-core Texas Instruments OMAP4 processor, an Arndale Board featuring a double-core Samsung Exynos processor, and a Boundary Devices Nitrogen6X/SABRE Lite Board featuring a single-core Freescale i.MX6 processor. The latter board was also used for the Android versions.

In our tool chain, we used GCC 4.8.1, LLVM 3.4, and GNU binutils 2.23. We compiled code with the following flags: `-Os -march=armv7-a -marm -mfloat-abi=softfp -mfpu=neon -msoft-float`.

5.2 Use cases

To evaluate the real-world potential of our self-debugging scheme, we deployed it on three use cases that were developed independently in three market leader companies The three use cases were hence developed using different development approaches, different software architectures, and even different build systems. Each use case consists of one or more shared libraries to provide us with software of sufficient complexity to be representative for real software products and with embedded, security-sensitive assets representative of the assets (and corresponding security requirements) in the companies' real products. We chose the fragments to migrate from the application into the debugger together with the application architects and developers, and with security architects from the companies.

5.2.1 Digital Rights Management

The first use case consists of two plugins, written in C and C++ at Nagravision S.A., for the Android media framework and the Android DRM framework. These libraries are necessary to obtain access to encrypted movies and to decrypt them. A video app programmed in Java is used as a GUI to access the videos. This app communicates with the mediaserver and DRM frameworks of Android, informing the frameworks of the vendor of which it needs plug-ins. On demand, these frameworks then load the plug-ins. Concretely, these servers are the `mediaserver` and `drmserver` processes running on Android.

During our experiments and development, we observed several features that make this use case a perfect stress test for our protection. First, the mediaserver is multi-threaded, and creates and kills new threads all the time. Secondly, the plug-in libraries are loaded and unloaded frequently. Sometimes the unloading is initiated even before the initialization of the library is finished. Thirdly, as soon as the process crashes, a new instance is launched. Sometimes this allows the Java video player to continue functioning undisrupted, sometimes it does not. This makes debugging the implementation of our technique even more complex than it already is for simple applications. Fourthly, the mediaserver and drmserver are involved in frequent inter-process communications.

5.2.2 Software License Manager

The second use case is a software license manager that stores credentials and controls access to licensed content and functionality, e.g., through time-limited licenses, key-enabled licenses. This license manager is programmed in C at SafeNet Germany GmbH (which has since been acquired by Gemalto S.A.). The library includes the JNI interface, and is embedded in an Android app. This native library thus functions as a license manager for a Java application. In this case, the Java application is relatively simple: It is a riddle game of which the solutions are protected by the license manager.

To test our self-debugger technique, this use case is also interesting. In particular, the library is loaded into Android's Dalvik execution environment, which features multiple threads (such as for the JIT compiler, garbage collector, ...), and over which we have absolutely no control [2].

Fortunately, a command-line version of the riddle game, programmed in C is also available. It uses the same library (except the JNI wrapper). On top of providing an easier target to debug on our Android developer boards, this command-line version can also be compiled for Linux. This way, we could also test our implementation on Linux.

5.2.3 One-time password generator

The third use case is a one-time password generator developed in C and C++ at Gemalto S.A. In this case, the native library is responsible for storing and accessing counters and seed values necessary to generate one-time passwords, and also for provisioning the original counter and seed values. This library is again embedded in a Java Android application, which in this case simply provides GUI functionality on top of the functionality in the library.

Table 1 lists a number of features of the three use cases as an indication of their representativeness of real-world software. The number of source code lines includes all the men-

use case	developer	nr. of src lines	included third-party libraries	embedded assets
DRM	Nagravision S.A.	306.247	OpenSSL	keys
software license manager	SafeNet Germany GmbH	55.487	libtomcrypt, libtommath	keys
OTP generator	Gemalto S.A.	360.446	OpenSSL, libcurl	seed, counter

Table 1: Feature matrix of the use cases

Transformation	Overhead
Control Flow	1.7 ms
Memory Read	3.4 μs
Memory Write	2.3 μs

Table 2: Overhead of our transformations.

tioned third-party libraries that are compiled and statically linked into the shared libraries to be protected. This static linking is a security requirement, because dynamic linking leaks too much symbolic information to attackers. Whereas the linked-in libraries do not contain any assets, they operate on assets such as keys, and the flow of control into them needs to be protected against reverse engineering as well.

5.3 Correctness

We tested the technique for correctness by extensively testing them, first on toy examples and then on the three use cases. For the use cases, we tested self-debugging combined with many different combinations of the protections listed in Section 4.5. After a considerable amount of engineering effort, we reached the status of reliable correct execution, even in complex situations such as native code libraries loaded into Google's Dalvik environment and plugins loaded into the Android DRM and media servers. The latter server proved to be particularly testing, as described in Section 5.2.1.

5.4 Execution Overhead

On the use cases, the self-debugging did not introduce significant overhead. This is due to the nature of the assets and code fragments protected with the technique, which are not located on hot code paths.

To get an idea of the actual overhead, we also performed measurements on micro-benchmarks. Our aim was to measure the overhead introduced by the transformations we use on control flow and memory accesses. Each micro-benchmark migrated a little code fragment to the debugger. For the memory accesses it was a load from memory or a store to memory that was executed in a loop (the entire loop being migrated). For the control flow the migrated code fragment consisted of a single instruction (an `ADD` instruction), also contained in a loop. As these transformations only replace a single instruction, the original execution time of the micro-benchmarks is simply that of the respective instructions (which is on the order of nanoseconds).

We tested these micro-benchmarks on our Linux board (see Section 5.1), and made sure the loops had sufficiently high trip counts to make the execution time measurable and to ensure the execution time of the micro-benchmark was completely dominated by the transformation we wanted to benchmark. Table 2 lists the results.

5.5 Security Analysis

We describe four categories of possible vectors of attack against our technique: circumventing or avoiding the migrated control flow fragments, reverting the binary transformations, attacking the mini-debugger directly, and full system emulation.

5.5.1 Circumvention & Avoidance

One possible method of attack is to prevent the mini-debugger ever being invoked. This requires the attackers finding a path between an entry point of the protected binary and the area they are interested in, that does *not* contain any migrated control flow fragment. Attackers can then disable the mini-debugger, attach their own debugger, and debug the found path without any consequences. Finding such a path will be easier in shared libraries. Whereas an executable possesses a single entry point, shared libraries usually have multiple.

Even if no unprotected path to an area of interest exists, attackers can debug this area if they manage to disable the mini-debugger at the right moment. That is, after the last migrated fragment but before the area is entered. We will not go into the question of exactly how one would deduce from the application's execution that the right moment for intervention has arrived. A plethora of side channels might be used for this.

5.5.2 Reverting the Transformations

If the attacker simply reverts all the transformations that were applied to migrate control flow fragments, the mini-debugger can be disabled without problem. We differentiate between memory access transformations and control flow transformations. Both classes of transformations need to be reverted for this attack to succeed, but reverting a control flow transformation requires determining the address to which control should be transferred. In the current implementation this can easily be done through static analysis. For example, when a migrated fragment is invoked one can simply find the destination address by looking up the fragment ID in a datastrucure (see Section 4.3.2).

5.5.3 Attacking the Mini-Debugger

The mini-debugger itself obviously also forms an attack surface through which the application can be compromised. As the child process containing the mini-debugger is not protected, an attacker can attach a debugger to it and attempt to observe and manipulate the application through it.

Through observation of the mini-debugger the attacker can learn more about the control and data flow of migrated fragments, which can be used in the other attacks discussed in sections 5.5.1 and 5.5.2.

Instead of attaching their own debugger to the application, attackers can also subvert the mini-debugger for their own purposes. They can attach their own debugger to the mini-debugger process and subvert its `ptrace` privileges over the target application for their own purposes. Using this indirection, `ptrace` requests can be inserted into the application to one's heart desire.

Another possibility would be for attackers to develop their own debugger that incorporates the mini-debugger's functionality and that augments it with real debugging functionality. The mini-debugger could then be safely disabled and replaced with this new debugger.

5.5.4 *Full System Emulation*

Obviously, full system emulation could also be used to trace and debug our self-debugging applications. To the best of our knowledge, however, and as confirmed to some extent by the penetration tests described below, no such emulators are available for our targeted platform. And of course analysing and understanding the interaction between two processes, with all kernel interactions in between, will be harder than debugging a single program in isolation.

5.6 Penetration Tests

For each of the three Android use cases, professional penetration testers were hired for several weeks to attack the assets in the code protected with many techniques, as discussed in Section 4.5.

Whereas all of them tried to use tracing and live-debugging techniques, within the time frame of the pen tests none of them succeeded in collecting full traces or attaching debuggers for live-debugging of the most interesting code fragments being executed in-situ. The latter of course followed from the manual selection of migrated code fragments by the use case developers, which ensured that migrated code fragments occur on all execution paths to the relevant code fragments.

The evaluated tools that break on self-debugging libraries include gdb, valgrind used both as a standalone tracer tool and as a gdbserver for gdb and IDA Pro, and QEMU [1, 7, 19, 8]. The fact that gdb breaks is not surprising, as it depends on the ptrace interface to which access is blocked by the self-debugging technique. Valgrind currently does not support the `BKPT` instruction correctly, and it cannot emulate ptrace calls, so it cannot emulate a self-debugging program correctly. The QEMU version corresponding to our Android targets also does not support the `BKPT` instruction correctly.

Other tracing tools, such as dtrace [12] and systemtap [9] were not evaluated because they do not support our Android platform and because they do not support interactive debugging anyway. Their tracing and debug actions need to be scripted beforehand.

Some pen testers did succeed, however, in tracing and live-debugging code out of context. They then loaded a library into a specially crafted main program that directly invokes some of the library functions in isolation. In essence, they were able to create an execution path leading to some of the interesting internal functions without first executing a migrated fragment.

6. PRACTICAL CONSIDERATIONS

6.1 Fragment Selection

As explained in Section 2 the decision of what fragments are to be migrated to the debugger rests with the programmer. This is an important decision, as selecting the wrong fragments will result in a weakened protection, as was already discussed in the previous section. Therefore the location of the selected fragments in the control flow should be right before and inside all of the code regions an attacker might be interested in.

The fact that this selection is not straightforward was clear when the experts chose the fragments to be migrated. For example, at some point they mistakenly chose to migrate variable initialization code of which the initial values later proved to be dead. While their choice still resulted in control flow being transferred to the debugger, and control flow hence being obfuscated, an attacker could relatively easily undo the protection by rewriting the exception inducing instruction, thus circumventing many of the challenges that general code rewriting exhibits.

To hinder the transformations being reverted, a selected fragment should contain sufficient code, and code that is sufficiently complex. Some examples are control flow internal to the fragment, memory accesses, and complex computations.

6.2 Impact On Multithreading

While we did not observe this in our use cases and test programs, a potential issue with our technique might be that the debugger process is single-threaded, while the debuggee process is multi-threaded. Handling all requests to execute migrated fragments in the debugger might therefore become a bottleneck. Clearly the solution to this problem is engineering a more complex, multi-threaded mini-debugger.

6.3 OS Limitations

In the current implementation the debugger forks from the protected application and attaches to it using `ptrace`. However, the `ptrace` interface is quite powerful, and over the past years a number of protections placing restrictions on its use have been introduced and adopted by some Linux distributions. When enabled, these protections can hinder our technique or even make it impossible to use.

One of the protections introduced is `ptrace_scope`, which places restrictions on attaching to another process [27]. In Ubuntu, e.g., the enabled restriction level allows a process to attach only to its children [24]. In our case this can still be overcome however, as we have the ability to execute code in the protected application: During initialization the application can explicitly allow the debugger process to attach (using `prctl(PR_SET_PTRACER, debugger, ...)`).

Still, it is possible for Linux distributions to choose higher restriction levels of `ptrace_scope`. In that case, our self-debugger will not work.

7. CONCLUSIONS

In this paper, we proposed to migrate code fragments from an application to a debugger that serves as an anti-debugger. This way, we can make attacks on self-debuggers significantly harder: The semantics of the code in the debugger is not predetermined, and multiple control flow paths are possible per invocation of the debug loop.

Our open-source prototype implementation works on complex, real-world use cases, as demonstrated by protecting complex shared Android libraries. We also discussed multiple implementation issues and options.

As for future work, a number of issues remain to be tackled. The first issue is circular debugging, in which the debuggee of the current protection would not merely serve as the main application being debugged, but also as a debugger of the other process by which it is being debugged. We believe that there are no technical issues that make circular debugging impossible, but a significant engineering effort is still required. Circular debugging would certainly make certain attack paths harder, e.g., by requiring the inclusion of a full emulation step on the attack paths to simply observe the control flow implemented by the debugger.

The second open issue is the development of support for protecting multiple libraries that are loaded within the same application.

A third issue that we wish to investigate in the near future is the impact of different implementation aspects (such as ways to transfer and obfuscate fragment IDs) on the effort required by attackers.

Acknowledgements

Stijn Volckaert contributed to the results presented in this paper as a PhD student in the Computer Systems Lab at Ghent University. He thanks the Agency for Innovation by Science and Technology in Flanders (IWT) for supporting his research at the time.

Most of the presented results were obtained in the context of the ASPIRE FP7 research project. The ASPIRE project ran until October 2016 and has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.

8. REFERENCES

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] D. Bornstein. Dalvik VM internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008.

[3] A. Cabutto, P. Falcarin, B. Abrath, B. Coppens, and B. D. Sutter. Software protection with code mobility. In *Proceedings of the Second ACM Workshop on Moving Target Defense, MTD 2015, Denver, Colorado, USA, October 12, 2015*, pages 95–103, 2015.

[4] Carbon Monoxide. Scyllahide. https://bitbucket.org/NtQuery/scyllahide.

[5] J. N. Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 2009.

[6] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[7] G. Developers. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[8] C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler.* No Starch Press, 2011.

[9] F. Eigler, V. Prasad, W. Cohen, H. Nguyen, and M. Hunt. Architecture of systemtap: a Linux trace/probe tool. http://sourceware.org/systemtap/archpaper.pdf, 2005.

[10] P. Ferrie. The "ultimate" anti-debugging reference. http://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf, April 2011.

[11] Ferrit. OllyExt 1.8. https://tuts4you.com/download.php?view.3392.

[12] B. Gregg. DTrace Tools. http://www.brendangregg.com/dtrace.html.

[13] jean. hack.lu CTF - Challenge 12 WriteUp. Technical report, Sogeti ESEC Lab, 2010.

[14] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[15] Linux Programmer's Manual. dlopen(3) - Linux man page.

[16] Linux Programmer's Manual. fork(2) - Linux manual page.

[17] Linux Programmer's Manual. proc(5) - Linux manual page.

[18] mrexodia. TitanHide. https://bitbucket.org/mrexodia/titanhide.

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 2007.

[20] Pellsson. Starcraft 2 anti-debugging. http://www.bhfiles.com/files/StarCraft%20II/Wings%20of%20Liberty%20%28Beta%29/0x1337.org%20-%20SCII%20Anti-Debug.htm, March 2010.

[21] M. Schallner. Beginners guide to basic linux anti anti debugging techniques. *CodeBreakers Magazine*, 2006.

[22] D. Seal. *ARM architecture reference manual.* Pearson Education, 2001.

[23] T. Shields. Anti-debugging – a developers view. Technical report, Veracode, 2009.

[24] Ubuntu Wiki. SecurityTeam/Roadmap/KernelHardening - Ubuntu Wiki. https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening{\#}ptrace{_}Protection.

[25] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. IEEE, 2005.

[26] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 193–202. IEEE, 2001.

[27] Yama. ptrace_scope. https://www.kernel.org/doc/Documentation/security/Yama.txt.