US 20210004315A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2021/0004315 A1

DORE et al. (43) Pub. Date: Jan. 7, 2021

(54) SELF-DEBUGGING

(71) Applicant: **NAGRAVISION SA**, Cheseaux-sur-Lausanne (CH)

(72) Inventors: **Laurent DORE**, Thorigne-Fouillard (FR); **Asfandyar ORAKZAI**, Oslo (NO); **Brecht WYSEUR**, Panthalaz (BE); **Yihui XU**, Oslo (NO)

(73) Assignee: **NAGRAVISION SA**, Cheseaux-sur-Lausanne (CH)

(21) Appl. No.: **16/766,768**

(22) PCT Filed: **Nov. 27, 2017**

(86) PCT No.: **PCT/EP2017/080481**

§ 371 (c)(1),
(2) Date: **May 26, 2020**

Publication Classification

(51) **Int. Cl.**

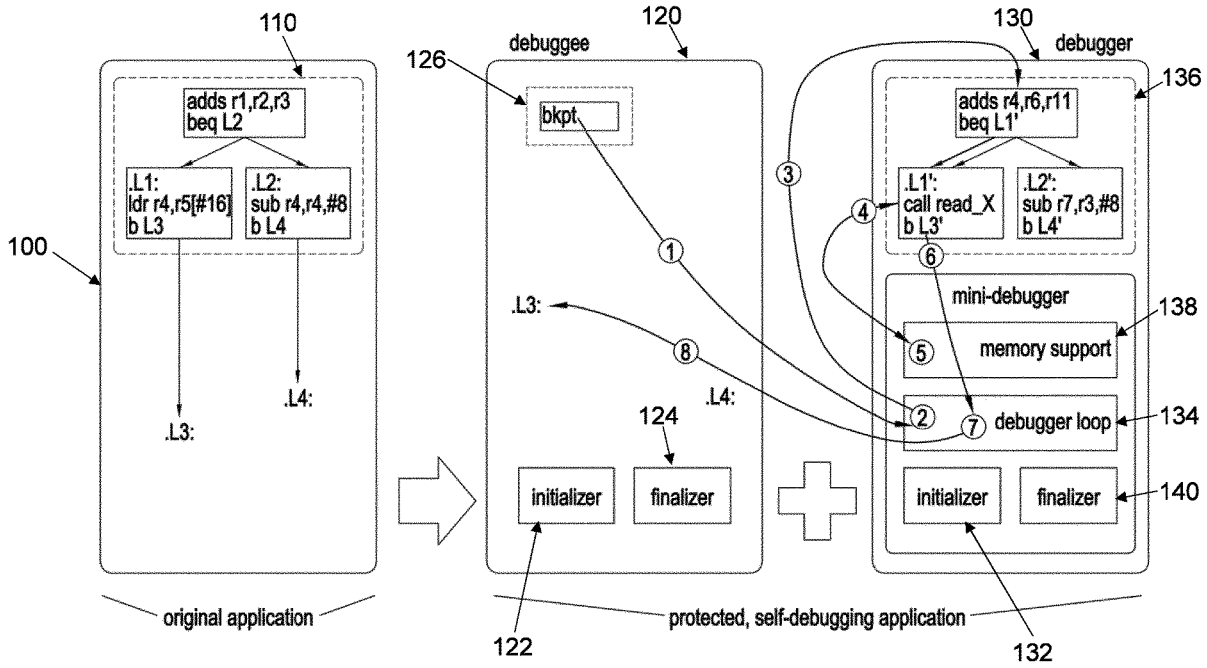
G06F 11/36	(2006.01)
G06F 8/41	(2006.01)
G06F 9/48	(2006.01)

(52) **U.S. Cl.**

CPC ***G06F 11/3624*** (2013.01); ***G06F 8/447*** (2013.01); ***G06F 11/3636*** (2013.01); ***G06F 11/3644*** (2013.01); ***G06F 9/4856*** (2013.01)

(57) **ABSTRACT**

In overview, methods, computer programs products and devices for securing software are provided. In accordance with the disclosure, a method may comprise attaching a debugger process to a software process. During execution of the software process, operations relevant to the functionality of the code process are carried out within the debugger process. As a result, the debugger process cannot be replaced or subverted without impinging on the functionality of the software process. The software process can therefore be protected from inspection by modified or malicious debugging techniques.



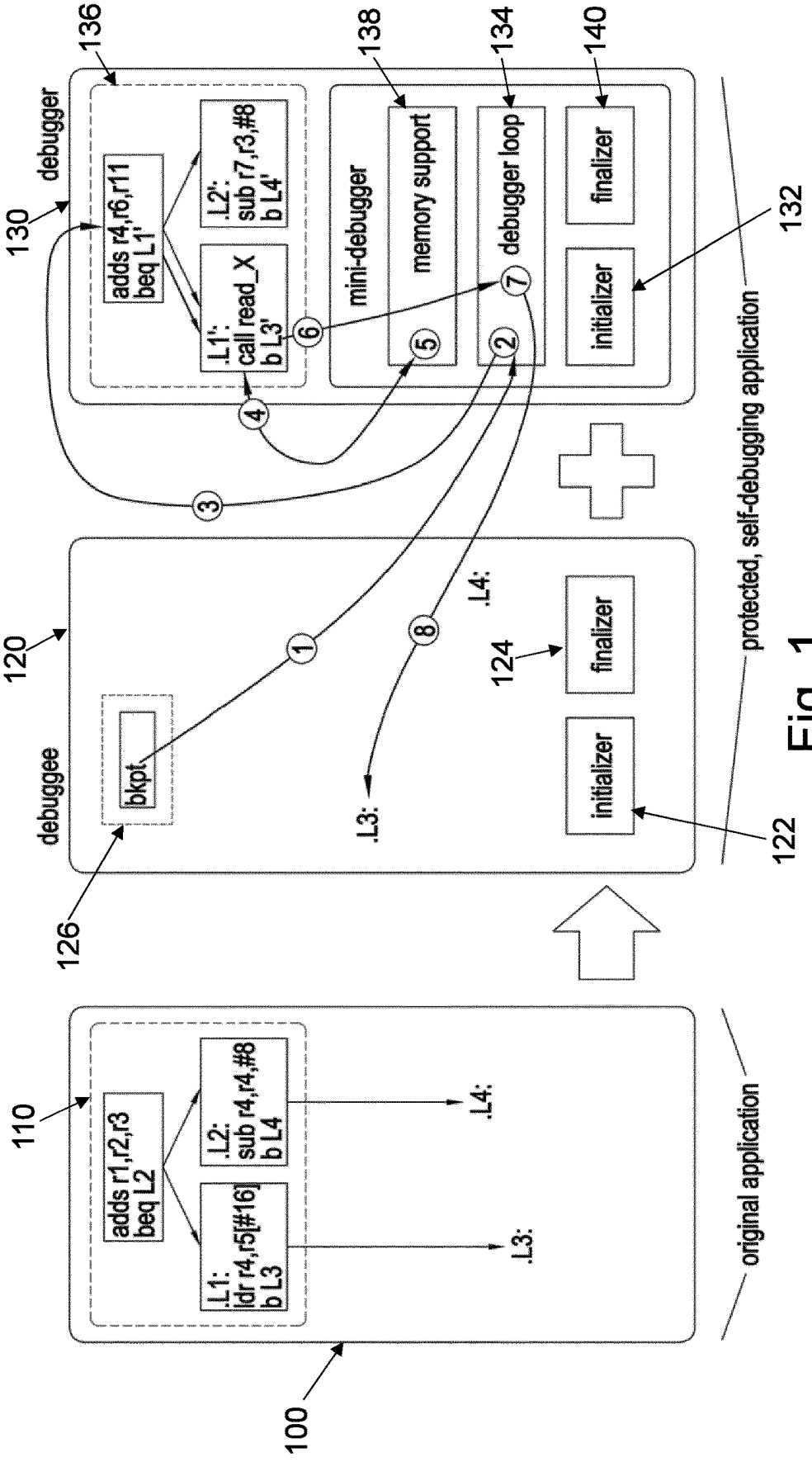

100
110
adds r1,r2,r3
beq L2
.L1:
ldr r4,r5[#16]
b L3
.L2:
sub r4,r4,#8
b L4
.L3:
.L4:
original application
120
debuggee
bkpt
126
.L3:
.L4:
124
finalizer
initializer
122
protected, self-debugging application
132
130
debugger
136
adds r4,r6,r11
beq L1'
.L1':
call read_X
b L3'
.L2':
sub r7,r3,#8
b L4'
138
mini-debugger
memory support
134
debugger loop
140
finalizer
initializer
1
2
3
4
5
6
7
8

Fig. 1

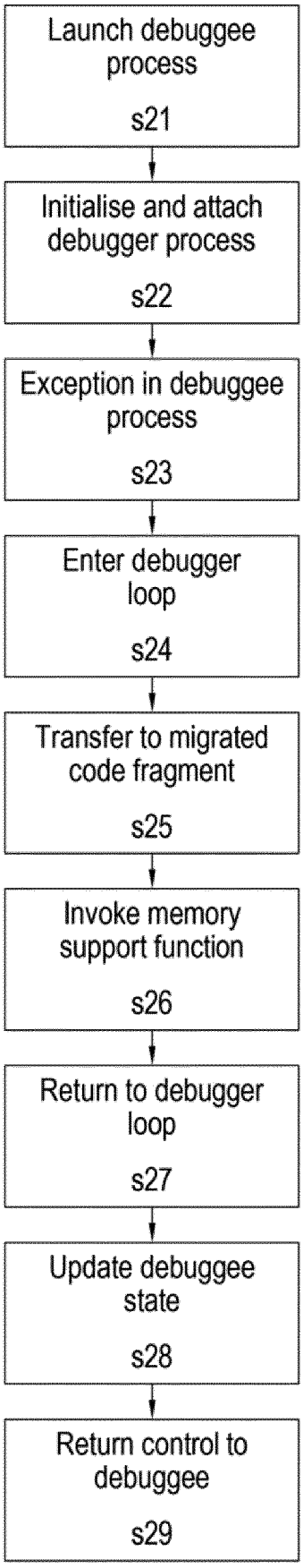

Launch debuggee process
s21
Initialise and attach debugger process
s22
Exception in debuggee process
s23
Enter debugger loop
s24
Transfer to migrated code fragment
s25
Invoke memory support function
s26
Return to debugger loop
s27
Update debuggee state
s28
Return control to debuggee
s29

Fig. 2

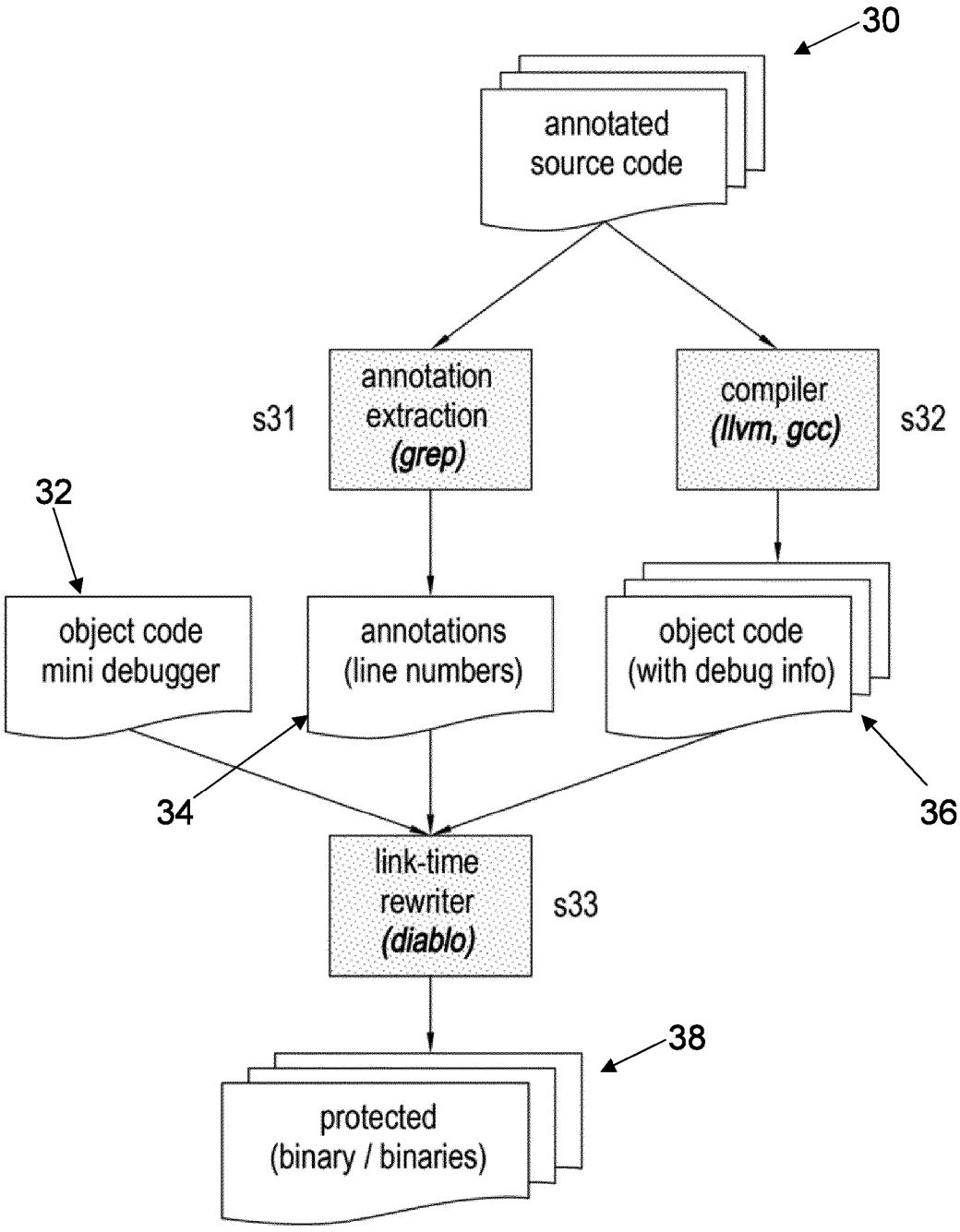

30
annotated source code
annotation extraction (grep)
s31
compiler (llvm, gcc)
s32
32
object code mini debugger
annotations (line numbers)
object code (with debug info)
34
36
link-time rewriter (diablo)
s33
38
protected (binary / binaries)

Fig. 3

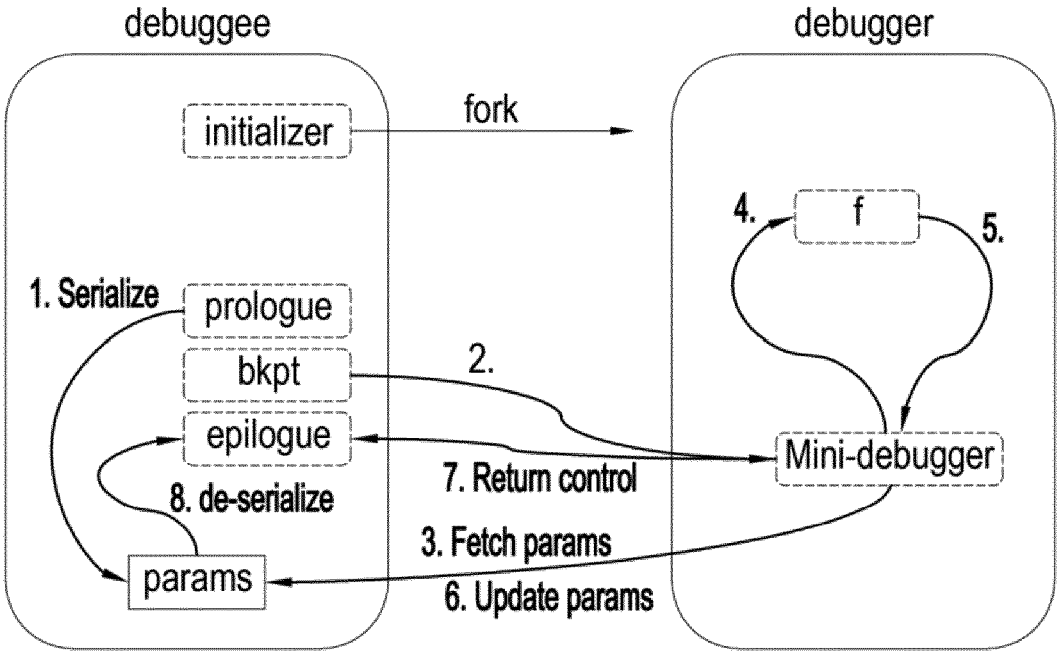

debuggee
debugger
initializer
fork
4.
f
5.
1. Serialize
prologue
bkpt
2.
epilogue
Mini-debugger
7. Return control
8. de-serialize
3. Fetch params
params
6. Update params

Fig. 4

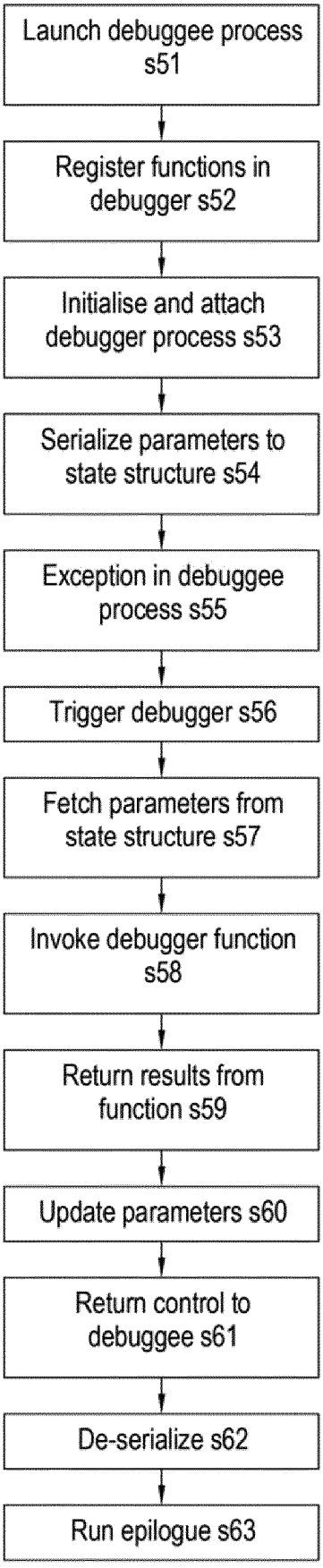

Launch debuggee process s51
Register functions in debugger s52
Initialise and attach debugger process s53
Serialize parameters to state structure s54
Exception in debuggee process s55
Trigger debugger s56
Fetch parameters from state structure s57
Invoke debugger function s58
Return results from function s59
Update parameters s60
Return control to debuggee s61
De-serialize s62
Run epilogue s63

Fig. 5

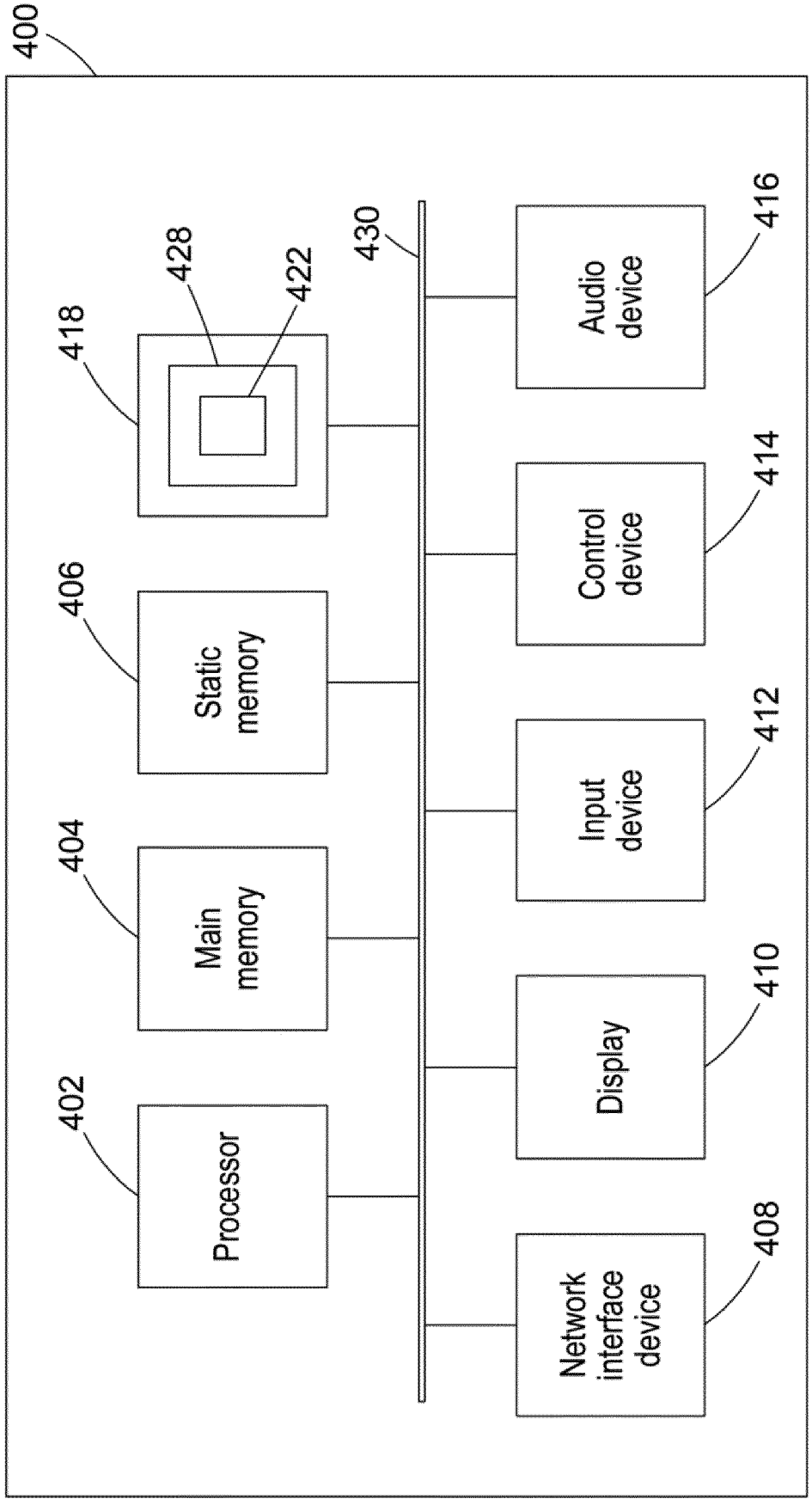

400
Processor
402
Main memory
404
Static memory
406
418
428
422
Network interface device
408
Display
410
Input device
412
Control device
414
Audio device
416
430

Fig. 6

SELF-DEBUGGING

FIELD

[0001] The present disclosure relates to software security, particularly to protection of software such as applications or libraries from attacks using debugging techniques.

BACKGROUND

[0002] Debugging is the process by which errors in code can be identified. One tool for this is the debugger, a type of utility which many operating systems allow to be paired with code to be debugged. When an exception or other error occurs, this is reported to the debugger which is able then to inspect the code and identify the origin of this problem.

[0003] The ability to pair a debugger with code has been utilised by malicious parties in order to compromise the security of that code. In particular, since a debugger is able to identify the operation of code, it can be a source of vulnerability.

[0004] Techniques have been developed to try to protect code against such attack. These techniques include attempts to allow code to identify when an active debugger has been illicitly coupled to the code. Another approach is to design the code to itself initiate a debugger when executed (this debugger can be termed a "self-debugger"). Most operating systems will only allow a single debugger to be paired with a given process, meaning the self-debugger occupies the space a malicious debugger may otherwise wish to use.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] FIG. **1** is a schematic illustration of the main features of a prior art code process, and the coupled code process and debugger process of a first embodiment;

[0006] FIG. **2** is a flow chart showing runtime steps according to the first embodiment;

[0007] FIG. **3** shows primary aspects of the generation of binaries according to the first embodiment;

[0008] FIG. **4** is a schematic illustration of the coupled code process and debugger process of a second embodiment;

[0009] FIG. **5** is a flow chart showing run time steps according to the second embodiment; and

[0010] FIG. **6** shows a hardware infrastructure for implementing a preferred embodiment.

DETAILED DESCRIPTION OF THE DRAWINGS

[0011] In overview, methods for securing the operation of code are provided. In accordance with the disclosure, a method may comprise launching a code process and initialising a debugger process attached to the code process. During execution of the code process, operations critically relevant to the functionality of the code process can be carried out within the debugger process. As a result, the debugger process cannot be replaced or subverted without impinging on the functionality of the code process. The code process can therefore be protected from inspection by modified or malicious debugging techniques.

[0012] In this context, "critically" can be understood to mean that the output produced by those operations carried out in the debugger process serves as input for the remaining part of the code process, and that that input is necessary to allow the code process to generate its correct output given that code process' other input.

[0013] In some aspects of the disclosure a method for securing software is provided. The method may comprise launching a software process and attaching a debugger process to the software process. The code process can then be executed such that the debugger process is invoked at least once. Upon invocation, one or more functions may be performed within the debugger process, these functions having an output dependent on data associated with the software process. Since the output can vary in dependence on data associated with the software process (i.e. it is not predetermined), the overall functionality is only achieved when both software process and debugger process are operating correctly. This does not leave space for interference with the debugger process to analyse the code.

[0014] The software process of this aspect may be considered a "debuggee" process since it takes the place of the process being debugged by the debugger. The debugger process may be initialised when the software process is launched or at a later time. For example, a debugger process may be initialised when certain functionality (e.g. a library) is loaded into the software process). In some examples, the software process forks to initialise the debugger process. In other examples, the debugger process may be initialised first and then fork to generate the software process.

[0015] In some embodiments, the output comprises a data output for use by the software process. Thus the output of the functions within the debugger process can directly influence the later operation of the software process, thereby tightly coupling the two processes in a way which is not easy to break. The output of the function in the debugger process comprises a data input for the software process, said data input being critical for the execution of the software process.

[0016] The software process may generate a data structure comprising parameters required for performance of the one or more functions within the debugger process prior to invocation of the debugger process. Thus the software process may make preparations to allow easy access to data from its memory to the debugger process. This is particularly applicable where source code or bitcode rewriting has been employed to generate the program associated with debugger process. Rewriting at this level can allow implementation of techniques to facilitate the generation of appropriate data structure for functions performed by the debugger process. The data structure may be a state structure.

[0017] In some examples, the software process acts to debug the debugger process. As such, a "circular" debugging arrangement is provided, in which both processes act to debug the other. This may prevent an external debugger processes attaching any process.

[0018] The method may launch an additional process to debug the debugger process. Further additional processes may be provided to continue the cascade, with each process being debugged by another. Again, a circular arrangement may be provided. For example, software process may debug the additional process so that no process is available for an external debugger.

[0019] In some embodiments, the output of a given function may indicate multiple points of return within the software process for continued execution. As such, the point of return for at least one function is variable (rather than fixed). The control flow is thus variable according to the behaviour of the debugger process and cannot be readily inferred or recreated.

[0020] In some embodiments, the debugger process provides memory support capabilities to enable the one or more functions to retrieve data from memory within address space of the software process. As such, the program-relevant functions can have an ability to process data as if they were carried out within the software process.

[0021] The debugger process can be invoked when a break point within the code process is reached. The debugger process can be detached from the software process when the software process finishes. The software process may finish because it is complete, or otherwise (such as when aborted). Alternatively, the debugger process may be detached from the software process when functionality within the software process finishes rather than waiting for the process as a whole to finish.

[0022] In some embodiments, the software process implements an executable, such as an application. In others, the code process implements a library.

[0023] In another aspect of the disclosure, there is provide method of generating protected code. One or more functions in code to be compiled for a first process is identified to be migrated to a second process, wherein the one of the first and second processes is a debugger for the other of the first and second processes. The migration is then carried out and the first process modified to allow transfer of state between the first and second processes. The first and second processes are then to generate binary code. The binary code at runtime may cause a debugger process to attach to a software process, the identified function or functions being executed within the debugger process

[0024] The code to be compiled may be source code or bitcode. In general, it may be code at a higher level than the binary.

[0025] An initializer may be injected into one of the first and second processes to invoke execution of the other of the first and second processes. This initializer may invoke execution of the first or second process which acts as a debugger to the other of the first and second process. In this manner, the debugger is automatically launched.

[0026] One or more Initializers may be injected into the first or second program to register functions present in the other of the first and second process. As such, each process is able to account for and take steps in recognition of functions carried out elsewhere. For example, initializers may facilitate the generation of a data structure for the one or more functions performed in the other process.

[0027] In some examples, each of the first and second processes is a debugger for the other of the first and second processes. As such, a "circular" debugging arrangement is provided, in which both processes act to debug the other. This prevents an illicit debugger attaching itself to the debugger program.

[0028] The method may provide a third process which is a debugger for one of the first and second processes. For example, the second process may debug the fist, and the third may debug the second. Additional processes may be provided to continue the cascade, with each process being debugged by another. Again, a circular arrangement may be provided. For example, the where the second processes debugs the first and the third process debugs the second, the first process may debug the third.

[0029] In another aspect of the disclosure, there is provided a method for generating protected code. Code fragments within object code to be migrated to a debugger can be identified. Binary code can then be generated, where the binary code at runtime causes a debugger process to attach to a software process, the identified code fragments being executed within the debugger process. The software process and the debugger process may be forked from a single process. For example, the software process may initialise the debugger process.

[0030] The step of generating may comprise incorporating predefined code corresponding to generic debugger functionality within the binary code. The generating step may be a linking step incorporating some predefined aspects of the debugger such aspects may be termed a "mini-debugger". As such, the overall debugger includes some generic aspects as well as some aspects specific to the source code by virtue of the inclusion of the identified code fragments.

[0031] The method may comprise extracting from source code one or more annotations identifying code fragments to be migrated to a debugger. The source code is then compiled to generate the object code. Binary code can then be generated from the object code, with the identified code fragments being integrated with a debugger in the binary code. In this way, the generation of binary code may be linking step which includes an element of re-writing to move identified fragments to another location. When the binary code is then used, a debugger is generated comprising aspects of the original source code, which can be pertinent to the functionality of the source code.

[0032] In some embodiments, the binary code comprises a first binary code file corresponding to the source code but excluding the identified code fragments and a second binary code file corresponding to the debugger. Alternatively, a single binary code file may incorporate both source code and debugger.

[0033] Further aspects of the disclosure relate to computer executable program products comprising computer executable instructions to carry out the methods of the aspects described above. Aspects of the disclosure may also relate to devices configured to carry out the methods of the aspects described above.

[0034] Some specific embodiments are now described by way of illustration with reference to the accompanying drawings in which like reference numerals refer to like features.

[0035] Through, binary rewriting techniques, the present disclosure can migrate whole chunks of functionality from the original software to a self-debugger. This offers several advantages. First, the input-output behaviour 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 deconstruction. 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. In combination, 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.

[0036] Overall Self-Debugger Design

[0037] FIG. **1** illustrates the basic concepts of a self-debugging scheme according to the present disclosure. This embodiment targets Linux (and derivatives such as

Android), the principles may also be applied to other environments such as Windows and OS X.

[0038] On the left of FIG. **1**, an original, unprotected application is depicted, including a small control flow graph fragment. The shown assembly code is (pseudo) ARMv7 code. 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. This particular embodiment supports all single-entry, multiple-exit code fragments that contain no inter-procedural control flow such as function calls.

[0039] The migration of such fragments is more than simple copying: memory references such as the LDR instruction should be transformed because in the protected application, the migrated code executing in the debugger address space can preferably access data that still resides in the debuggee address space. All relevant components and transformations will be discussed in more detail in later sections.

[0040] The migrated fragments are preferably critical to the operation of the application. That is to say, the output produced by those operations carried migrated to the debugger process serves as input for the remaining part of the code process, and that that input is necessary to allow the code process to generate its correct output given that code process' other input. This requirement is easy to miss in practice. For example, a typical programmer might consider executing the initialization of variables of the code process in the debugger context. However, in general it does not suffice to execute the initialization of variables from the code process in the debugger process, because in practice, in processes it happens quite often that variable initialization (e.g., of local variables upon entry to a function) is performed as a result of good programming practices and to meet the source programming language definition requirements, without actually being required for the correct functioning of the process and for generating correct outputs. This may be because variables are simply not used in the executed paths in the code process, or because the initial values are overwritten before they can impact the code process' execution or output.

[0041] At run time, the operation of this protected application is as follows. First, the debuggee is launched at step s**21**, 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. Thus the debugger process is launched and attached to the dubuggee process at step s**22**.

[0042] When later during the program's execution the entry point of the migrated code fragment is reached, one possible flow of control in the application follows the arrows in FIG. **1**. In the application/debuggee, the exception inducing instruction is executed and causes an exception at step s**23** (labelled **1** in FIG. **1**). The debugger is notified of this exception and handles it in its debugger loop at step s**24** (labelled **2** in FIG. **1**). 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 to the entry point of that fragment at step s**25** (labelled **3** in FIG. **1**). 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 at step s**26**. 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** at step s**27**, which updates the state of the debuggee with the data computed in the debugger at step s**28**, and **8** control is transferred back to the debuggee at step s**29**. 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, the debugger of the present disclosure behaves in a more complex manner than existing self-debuggers, which implement a one-to-one mapping between forward and backward control flow transfers between debuggee and debugger.

[0043] Eventually, when the application exits, the embedded finalizers will perform the necessary detaching operations.

[0044] 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.

[0045] Although the following description principally refers to protecting applications, implicitly the teaching applies equally applications and libraries. One aspect which is particularly relevant for libraries is the need for proper initialization and finalization of the debugger. 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, they can consist of multiple threads when libraries are loaded and unloaded.

[0046] Tool Support

[0047] FIG. **3** depicts one possible conceptual tool flow.

[0048] Source Code Annotations

[0049] For determining the code fragments to be migrated to the debugger, a number of options exist. One, depicted in the figure—and also what we use in our implementation—is to annotate source code at step s**31** with pragmas, comments or any other form of annotations that mark the beginnings and ends 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 at step s**32**.

[0050] 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.

[0051] 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 be-tween the debuggee and the debugger, it suffices to raise exceptions relatively frequently, but this does not need to be on the hottest code paths. Further considerations for the selection of fragments will be detailed below. Since every raised exception will introduce a meaningful amount of

overhead (context switch, many ptrace calls, . . .) it is important to minimize their number without compromising the level of protection.

[0052] Standard Compilers and Tools

[0053] For the disclosed self-debugging approach to be deployed, any "standard" compiler can be used at step s**33**. The technique does not impose any restrictions on the code generated by the compiler. In experimental evaluations, both GCC and LLVM have been used, in which there was no requirement to adapt or tune the code generation.

[0054] 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 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.

[0055] 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 their specific fragments.

[0056] 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.

[0057] Binaries, Libraries, and Processes

[0058] The link-time rewriter has two options to generate a protected application at step s**35**. 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.

[0059] The alternative option—used in the further examples below—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.

[0060] Implementation

[0061] Initialization & Finalization

[0062] An extra initialization routine can be added to a protected binary. This routine is invoked as soon as the binary has been loaded (because it assigned a high priority), after which all the other routines listed in the .init section of the binary are executed.

[0063] This initialization routine invokes fork( ), thus creating two processes called the parent and the child. Once the initialization routine is finished the parent process will continue execution, typically by invoking the next initialization routine.

[0064] 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.

[0065] The former option is preferred. The parent process (i.e. debuggee) 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 the loading and forking of a protected library.

[0066] This scheme does come with its own problems, however. As already mentioned, shared libraries can be loaded and unloaded (using dlopen( ) and dlclose( )) 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 hasn't finished its initialization yet. 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.

[0067] 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 SIGUSR**1** signal, causing the mini-debugger to detach from all the debuggee's threads and to shut down the debugger process.

[0068] Multithreading Support

[0069] 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. 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.

[0070] 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.

[0071] Control Flow

[0072] 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 informing the debugger of what fragment is to be executed, and the customized pro- and epilogues that are added to every code fragment.

[0073] Raising Exceptions

[0074] The actual 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 of course 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).

[0075] Transferring IDs

[0076] 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.

[0077] 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.

[0078] Depending on the type of instruction used to raise the exception, other methods can be envisioned as well. For example, the divisor operand of a division (by zero) instruction could be used to pass the ID as well.

[0079] Prologues and Epilogues

[0080] 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.

[0081] 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 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 were overwritten in the code fragment.

[0082] Memory Accesses

[0083] 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.

[0084] 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' 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.

[0085] 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.

[0086] 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.

[0087] 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.

[0088] Security-wise, the different options will likely also have an 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.

[0089] Combining Self-Debugging with Other Protections

[0090] To provide strong software protection against MATE attacks, additional protection techniques may be employed. For example, on top of self-debugging, obfuscation to prevent static analysis may be employed, together with anti-tampering techniques to prevent all kinds of attacks.

[0091] For example, the binary rewriter that implements the self-debugging approach may also applies a number of other protections, such as one or more of:

- **[0092]** Control flow obfuscations: the well-known obfuscations of opaque predicates, control flow flattening, and branch functions;
- **[0093]** Code layout randomization: during code layout, code from all functions is mingled and the layout is randomized;
- **[0094]** Code mobility: a technique in which code fragments are removed from the static binary and only down-loaded, as so-called mobile code, into the application at run time;
- **[0095]** 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.
- **[0096]** Control flow integrity: a lightweight technique in which return addresses are checked to prevent that internal functions are invoked from external code.
- **[0097]** Instruction set virtualization: a technique with which native code is translated to bytecode that is inter-preted by an embedded virtual machine instead of executed natively.

[0098] Combining the self-debugging technique with all of those protections poses no problem in practice. In the link-time rewriter, it is not difficult to determine a good order to perform all the transformations for all of the protections, and to prevent that multiple techniques are applied on the same code fragments when those techniques do not actually compose.

[0099] For example, mobile code is relocated to randomized locations. Handling all protections correctly requires some bookkeeping, but nothing complex.

[0100] As for the run-time behaviour, 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.

[0101] Evaluation

[0102] Evaluation Platform

[0103] One implementation of the self-debugger targets ARMv7 platforms. Concretely, this implementation targeted and extensively evaluated the implementation on Linux 3.15 and (unrooted) Android 4.3+4.4. It has further been confirmed that the techniques still work on the latest versions of Linux (4.7) and Android (7.0), and that is indeed the case.

[0104] The testing hardware consisted of several developer boards. For Linux, a Panda Board was used 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.MX6q processor. The latter board was also used for the Android versions.

[0105] In the tool chain, GCC 4.8, LLVM 3.4, and GNU binutils 2.23 were used. Code was compiled with the following flags: -Os-march=armv7-a-marm-mfloat-abi=softfp-mfpu=neon-msoft-float.

[0106] Use Cases

[0107] The self-debugging scheme has been shown to function in multiple use cases. For example, in a digital rights management scenario, the following practical considerations were encountered.

[0108] This use case consisted of two plugins, written in C and C++, 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.

[0109] During experiments and development, several features were observed that make this use case a perfect stress test for this technique. 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 doesn't. 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. Nevertheless, successful implementation was achieved based on the principles described above.

[0110] The techniques of the present disclosure may be applied in many other use case scenarios. For example, in mobile banking on any other scenario in which security is desirable.

Second Embodiment

[0111] In the examples presented above with respect to FIGS. **1** to **3**, the binary file is rewritten to transfer elements to the debugger process. In a second embodiment, the technique can be deployed at source-level or at another higher level than the binary (e.g. at bitcode level) during the build process of the software. This will be described below with reference to FIGS. **4** and **5**. This process transfers program state between debugger and debuggee in a different manner to the example presented above.

[0112] In this approach, the application may be sliced into two or more parts at source, or bitcode, level using a rewriting tool. This slicing may be carried out at a function level, such that the rewriting process transfer certain functions from the initial program to another program and the initial program is modified to be able to transfer state to the other program. The initial program may take the role of debuggee during later execution while the other program can take the role of debugger. Alternatively, the separation of roles of debugger and debuggee may be reversed.

[0113] Furthermore, additional code is injected in the program which is to be first launched (which may be either program within the sliced application) to act as an initializer which allows the application to fork itself or launch another process to enable attachment of the debugger to the debuggee. Moreover, additional initializers may be incorporated into the program which is to be first launched to register those functions which are to be carried out by the other program.

[0114] Run-time operation of the sliced application can be understood with reference to FIGS. **4** and **5**. In particular, FIG. **4** is a schematic illustration of the coupled code process and debugger process of a second embodiment while FIG. **5** is a flow chart showing run time steps according to the second embodiment.

[0115] In the example shown in FIGS. **4** and **5**, the program first launched carries out the debuggee process. As such, at run time, the debuggee is launched at step s**51**, as if it was the original application. The initializers injected during the build process then register the functions targeted by the debugger at step s**52** and fork off a new process for the debugger. Since the initializers can be generated above the binary level in knowledge of the division of functions between the debugger and debuggee processes, beneficially the registration of functions can be suitably targeted.

[0116] Once the new debugger process is forked, the debugger's initializer immediately attaches to the debuggee process. Thus the debugger process is launched and attached to the dubuggee process at step s**53**.

[0117] When later during the program's execution the entry point of a function which has been placed in the debugger is reached, one possible flow of control in the application follows the arrows in FIG. **4**. Firstly, prologue (preferably architecture independent) code may serialize function parameters into a state structure at step s**54** (labelled **1** in FIG. **4**). An exception inducing instruction is then executed, which causes an exception at step s**55** (i.e. a breakpoint, labelled "bkpt" in FIG. **4**). This triggers the debugger at step s**56** (labelled **2** in FIG. **4**), which identifies the debuggee location of the exception/breakpoint.

[0118] The mini-debugger routine/loop in the debugger process is able to infer from the debuggee location of the exception/breakpoint both the target code (i.e. function "f" to be carried out within debugger process) and how to retrieve parameters serialized to the state structure from debuggee memory. It then fetches parameters from the debuggee at step s**57** (labelled **3** in FIG. **4**). In addition, since the state structure can reference elements elsewhere in the dubuggee memory, these extended state elements can be identified at source level such that they can also be retrieved from the debuggee memory to be available to the debugger.

[0119] As the execution through the mini-debugger of the correct function and retrieval of state parameters is conditional on the location of the breakpoint/exception this provides additional security as an incorrect trigger point would not cause proper execution.

[0120] With the parameters successfully retrieved, the mini-debugger may invoke the function "f" at step s**58** (labelled **4** in FIG. **4**). This function "f" was migrated from the debuggee application during source-code rewriting as described above. The function "f" is performed and returns results to mini-debugger at step s**59** (labelled **5** in FIG. **4**). Parameters can then be updated in the state structure, including extended state parameters at step s**60** (labelled **6** in FIG. **4**). The mini-debugger then returns control to the debuggee at step s**61** (labelled **7** in FIG. **4**), and the debuggee writes state and extended state parameters back into debuggee memory through a process of de-serialization at step s**62** (labelled **8** in FIG. **4**). At step s**63**, the debuggee may run epilogue code in order to restore system parameters and/or variables. While such code may be reliant on current architecture, elements can be formulated based on portable complier intrinsic aspects.

[0121] In the example described above with reference to FIGS. **4** and **5**, an initial program is associated with one of the debugger or debuggee and a second program is associated with the other. However, the skilled person will recognise that in some architectures it may not be necessary to assign processes in this way. For example, a single program may launch two processes, one of which debugs the other.

[0122] Moreover, it is recognised that in some environments, such as Linux or Android for example, a "forking" procedure may be adopted. For example, Fork( ) is a system call that will duplicate the process from which it is called. It will copy the process memory, and then both processes will continue in parallel. As such, in this approach a single program is run twice. The program can be implemented as such that it has a different behaviour in case it is the parent or in case it is the child. For example, the child process may act as debugger of the parent process, or vice versa.

[0123] Accordingly, within the ambit of the present disclosure is provided the possibility of different programs being associated with debugger and debuggee processes, a single program generating independent debugger and debuggee processes, and a single program being forked into multiple (identical) processes where one of these assumes the role of debugger and one the role of debuggee. In another alternative, a new (idenitical) process can be created from scratch not using the forking procedure, again with one process carrying out debugging functions for the other. Any alternative division between programs and processes may also be adopted as appropriate.

[0124] Additional Features

[0125] The features below headed as "Circular Debugging", "Nested Debugging", "Re-entrance", "Re-attaching", "Detaching detection" and "Mutual Checking" may be provided in combination with any example or embodiment described above and in any combination with one another.

[0126] Circular Debugging

[0127] Attaching a debugger to the debuggee does not prevent the debugger from being debugged by a third-party. To prevent this, the debuggee can itself debug the debugger, creating a debugging loop, where each process prevents other debuggers from attaching to its debuggee. This is because each process will have a debugger attached and there is therefore no opportunity for an external debugger to attach itself.

[0128] Nested Debugging

[0129] In the example of circular debugging above, there are only two processes operating, with process P**1** debugging process P**2**, and process P**2** debugging process P**1**. However, additional processes may be provided, in order to provide a cascade of debugging relationships between a process Pn and a subsequent process Pn+1. For example, where n is the integer sequence from 1 to N, process Pn may debugging process Pn+1 where n does not equal N. Moreover, process PN may debug process P**1**, thus closing the loop and ensuring that every process has a debugger attached.

[0130] Re-Entrance

[0131] After the performance of a function, the debugger code can transfer control to the debuggee at locations other than the one near the exception inducing instruction, for example by calling functions in the debuggee context. This has the benefit of hiding the control flow from static analysis, as the flow decision is delegated to the debugger process. For example, Function Debuggee**1** triggers function Debugger**2**, which calls Debuggee**2**; no call from Debuggee**1** to Debugee**2** is visible by static analysis of the debuggee alone.

[0132] Reattaching

[0133] To complicate analysis, the debugger process can continuously attempt to re-attach to the debuggee. This would ensure that, should the debugger ever be detached from the debuggee, it would likely re-attach before another debugger attaches.

[0134] Detaching Detection

[0135] To complicate analysis, if the debugger process cannot attach to the debuggee because another debugger is attached, it can try to detach/kill this other debugger.

[0136] Mutual Checking

[0137] Debugger and debuggee, once attached, can check that their respective process ids are consistent. This would detect debugger substitution and/or insertion in the chain. Consistency checks could include whether the debugger/debuggee pid changes, whether the debugger/debuggee pid are parent/child (extended to nesting)

[0138] FIG. **6** illustrates a block diagram of one implementation of a computing device **400** within which a set of instructions, for causing the computing device to perform any one or more of the methodologies discussed herein, may be executed. In alternative implementations, the computing device may be connected (e.g., networked) to other machines in a Local Area Network (LAN), an intranet, an extranet, or the Internet. The computing device may operate in the capacity of a server or a client machine in a client-server network environment, or as a peer machine in a peer-to-peer (or distributed) network environment. The computing device may be a personal computer (PC), a tablet computer, a set-top box (STB), a Personal Digital Assistant (PDA), a cellular telephone, a web appliance, a server, a network router, switch or bridge, or any machine capable of executing a set of instructions (sequential or otherwise) that specify actions to be taken by that machine. Further, while only a single computing device is illustrated, the term "computing device" shall also be taken to include any collection of machines (e.g., computers) that individually or jointly execute a set (or multiple sets) of instructions to perform any one or more of the methodologies discussed herein.

[0139] The example computing device **400** includes a processing device **402**, a main memory **404** (e.g., read-only memory (ROM), flash memory, dynamic random access memory (DRAM) such as synchronous DRAM (SDRAM) or Rambus DRAM (RDRAM), etc.), a static memory **406** (e.g., flash memory, static random access memory (SRAM), etc.), and a secondary memory (e.g., a data storage device **418**), which communicate with each other via a bus **430**.

[0140] Processing device **402** represents one or more general-purpose processors such as a microprocessor, central processing unit, or the like. More particularly, the processing device **402** may be a complex instruction set computing (CISC) microprocessor, reduced instruction set computing (RISC) microprocessor, very long instruction word (VLIW) microprocessor, processor implementing other instruction sets, or processors implementing a combination of instruction sets. Processing device **402** may also be one or more special-purpose processing devices such as an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), a digital signal processor (DSP), network processor, or the like. Processing device **402** is configured to execute the processing logic (instructions **422**) for performing the operations and steps discussed herein.

[0141] The computing device **400** may further include a network interface device **408**. The computing device **400** also may include a video display unit **410** (e.g., a liquid crystal display (LCD) or a cathode ray tube (CRT)), an alphanumeric input device **412** (e.g., a keyboard or touchscreen), a cursor control device **414** (e.g., a mouse or touchscreen), and an audio device **416** (e.g., a speaker).

[0142] The data storage device **418** may include one or more machine-readable storage media (or more specifically one or more non-transitory computer-readable storage media) **428** on which is stored one or more sets of instructions **422** embodying any one or more of the methodologies or functions described herein. The instructions **422** may also reside, completely or at least partially, within the main memory **404** and/or within the processing device **402** during execution thereof by the computer system **400**, the main memory **404** and the processing device **402** also constituting computer-readable storage media.

[0143] The various methods described above may be implemented by a computer program. The computer program may include computer code arranged to instruct a computer to perform the functions of one or more of the various methods described above. The computer program and/or the code for performing such methods may be provided to an apparatus, such as a computer, on one or more computer readable media or, more generally, a computer program product. The computer readable media may be transitory or non-transitory. The one or more computer readable media could be, for example, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, or a propagation medium for data transmission, for example for downloading the code over the Internet. Alternatively, the one or more computer readable media could take the form of one or more physical computer readable media such as semiconductor or solid state memory, magnetic tape, a removable computer diskette, a random access memory (RAM), a read-only memory (ROM), a rigid magnetic disc, and an optical disk, such as a CD-ROM, CD-R/W or DVD.

[0144] In an implementation, the modules, components and other features described herein (for example control unit **410** in relation to FIG. **6**) can be implemented as discrete

components or integrated in the functionality of hardware components such as ASICS, FPGAs, DSPs or similar devices as part of an individualization server.

[0145] A "hardware component" is a tangible (e.g., non-transitory) physical component (e.g., a set of one or more processors) capable of performing certain operations and may be configured or arranged in a certain physical manner. A hardware component may include dedicated circuitry or logic that is permanently configured to perform certain operations. A hardware component may be or include a special-purpose processor, such as a field programmable gate array (FPGA) or an ASIC. A hardware component may also include programmable logic or circuitry that is temporarily configured by software to perform certain operations.

[0146] Accordingly, the phrase "hardware component" should be understood to encompass a tangible entity that may be physically constructed, permanently configured (e.g., hardwired), or temporarily configured (e.g., programmed) to operate in a certain manner or to perform certain operations described herein.

[0147] In addition, the modules and components can be implemented as firmware or functional circuitry within hardware devices. Further, the modules and components can be implemented in any combination of hardware devices and software components, or only in software (e.g., code stored or otherwise embodied in a machine-readable medium or in a transmission medium).

[0148] Unless specifically stated otherwise, as apparent from the following discussion, it is appreciated that throughout the description, discussions utilizing terms such as "receiving", "determining", "comparing", "enabling", "maintaining," "identifying," "replacing," or the like, refer to the actions and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[0149] It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other implementations will be apparent to those of skill in the art upon reading and understanding the above description. Although the present disclosure has been described with reference to specific example implementations, it will be recognized that the disclosure is not limited to the implementations described, but can be practiced with modification and alteration within the spirit and scope of the appended claims. Accordingly, the specification and drawings are to be regarded in an illustrative sense rather than a restrictive sense. The scope of the disclosure should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

1. A method of generating protected code, comprising:

identifying one or more functions in code to be compiled for a first process to be migrated to a second process, wherein the one of the first and second processes is a debugger for the other of the first and second processes;

migrating the identified function or functions to the second process;

modifying the first process to allow transfer of state between the first and second processes; and

compiling the first and second processes to generate binary code.

2. A method according to claim **1**, wherein the code to be compiled is source code or bitcode.

3. A method according to either claim **1** or claim **2**, further comprising injecting an initializer into one of the first and second processes to invoke execution of the other of the first and second processes.

4. A method according to any one of the preceding claims, further comprising injecting one or more initializers into the first or second processes to register functions present in the other of the first and second process.

5. A method according to any one of the preceding claims, wherein each of the first and second processes is a debugger for the other of the first and second processes.

6. A method according to any one of the preceding claims, further comprising providing a third process which is a debugger for one of the first and second processes.

7. A computer executable program product comprising computer executable code for carrying out the method of any one of the preceding claims.

8. A device configured to carry out the method of any one of claims **1** to **6**.

9. A method for securing software, comprising:

launching a software process;

attaching a debugger process to the software process;

executing the software process such that the debugger process is invoked at least once;

performing one or more functions within the debugger process in response to invocation of the debugger process, the one or more functions having an output dependent on data associated with the software process,

wherein the software process generates a data structure comprising parameters required for performance of the one or more functions within the debugger process prior to invocation of the debugger process.

10. A method according to claim **9**, wherein the output comprises a data output for use by the software process.

11. A method according to claim **9** or claim **10**, wherein the data structure is a state structure.

12. A method according to any one of claims **9** to **11**, wherein the software process acts to debug the debugger process.

13. A method according to any one of claims **9** to **12**, further comprising launching an additional process to debug the debugger process.

14. A method according to any one of claims **9** to **13**, wherein the output of a given function may indicate multiple points of return within the software process for continued execution.

15. A method according to any one of claims **9** to **14**, wherein the debugger process provides memory support capabilities to enable the one or more functions to retrieve data from memory within address space of the software process.

16. A method according to any one of claims **9** to **15**, wherein the debugger process is invoked when a break point within the software process is reached.

17. A method according to any one of claims **9** to **16**, further comprising detaching the debugger process from the software process when the software process is complete.

18. A method according to any one of claims **9** to **17**, wherein the software process implements an executable, such as an application.

19. A method according to any one of claims **9** to **17**, wherein the software process implements a library.

20. A computer executable program product comprising computer executable code for carrying out the method of any one of claims **9** to **19**.

21. A device configured to carry out the method of any one of claims **9** to **19**.

* * * * *