Software Protection with Code Mobility

Alessandro Cabutto, Paolo Falcarin
University of East London
School of Architecture, Computing and Engineering
E16 2RD London (UK)
{a.cabutto, falcarin} @uel.ac.uk

Bert Abrath, Bart Coppens, Bjorn De Sutter
Ghent University
Sint Pietersnieuwstraat 41, 9000 Ghent (Belgium)
{bert.abrath, bart.coppens, bjorn.desutter}
@elis.ugent.be

ABSTRACT
The analysis of binary code is a common step of Man-At-The-End attacks to identify code sections crucial to implement attacks, such as identifying private key hidden in the code, identifying sensitive algorithms or tamper with the code to disable protections (e.g. license checks or DRM) embedded in binary code, or use the software in an unauthorized manner. Code Mobility can be used to thwart code analysis and debugging by removing parts of the code from the deployed software program and installing it at run-time by downloading binary code blocks from a trusted server. The proposed architecture of the code mobility protection downloads mobile code blocks, which are allocated dynamically at addresses determined at run-time; control transfers into and out of mobile code blocks are rewritten using the Diablo binary-rewriter tool.

Categories and Subject Descriptors
D.2.0 [**Software Engineering**]: General – *Protection mechanisms.*

General Terms
Security, Reverse engineering.

Keywords
Code Mobility, Binary Code, Binary Rewriting.

1. INTRODUCTION
One of the main goals of software protection is to prevent code from being observed and analysed, and then eventually illicitly modified and tampered with. To protect against code analysis, developers usually try to make reverse-engineering harder, by applying different obfuscating transformations [1]; attackers can use binary code inspection tools like IDA Pro [2] and binary instrumentation tools [3] to extract run-time information such as execution traces and memory dumps.

To protect against illicit modifications, anti-tampering approaches are utilized to detect when code has been tampered with and to react by stopping or delaying program execution. Tamper-resistant software typically uses built-in integrity checks to detect code tampering by guarding the code being executed [4] or by checking that the flow of control through the program confirms to the expected flow [5].

Binary obfuscation techniques have been proposed to increase reverse engineering complexity: Linn et al. [6] proposed a tool for inflating binary code with redundant and/or garbage instructions to defeat disassemblers or to produce a very complex assembly code: they evaluate obfuscation strength with their confusion factor, as the percentage of instructions not correctly disassembled because of binary obfuscation.

Aucsmith [22] proposed encryption to resist to code observation: his technique break a binary program into individually encrypted segments, so that the hash value of a block is the secret key for decrypting the next block; if the program was altered the hash value is changed and then the next block cannot be decrypted properly and the program cannot continue to run; in this case finding the first key allows recovering the full chain of keys.

Kanzaki et al. [7] used self-modifying binary code to thwart static analysis and disassembling, while Birrer et al. [8] provide metamorphic binary code by means of program fragmentation, and Giffin et al. [9] used self-modifying code for code guards hardening.

Online protections techniques aim at extending state-of-the-art static protection techniques by leveraging on software updates and trusted network services.

Different online protections use dynamic code replacement to periodically replace the copy of the program running on the untrusted machine with the goal of limiting the amount of time that the attacker has to reverse engineer the application.

The replacement may be implemented for the functional part of the program, and/or for the protection techniques used to protect it [10]. Collberg et al [11][12] and Falcarin et al [13] proposed the continuous replacement of Java and binary code respectively, in which the remote trusted entity frequently sends a set of new code fragments to the untrusted machine.

The technique of Collberg et al [12] has some limitations has it relies on CIL (Common Intermediate Language): this bounds the scenarios in which the technique is usable (e.g., not with dynamically linked libraries), their composability with other protections, and the granularity of the code blocks.

Previous works in Java implemented dynamic replacement of protection code implementing code mobility features on top of dynamic aspect-oriented platforms [14] [14]or by ad-hoc JVM extensions [15].

In this paper, we present the Code Mobility framework, an online protection technique that aims to overcome the drawbacks of local protection techniques by introducing mobile code. A mobile code block is a piece of binary code removed from an application before deployment. A trusted server placed on the network, is in charge of providing mobile code blocks to the untrusted client.

To protect against code analysis, the Code Mobility framework delivers binary code to the client at run-time; the client application self-modifies its own code layout to install the

downloaded code blocks, in order to thwart static analysis and increase the difficulty of dynamic analysis.

The Code Mobility framework has been developed within the ASPIRE project [21], and it is compliant to the software protection reference architecture designed in the project and documented in deliverable D1.04 [16], which is available on the project website.

The main novel contributions of this work are:

1. design and prototype implementation with demonstration on standard Android code;
2. integration with compilers commonly used for native code development, including in the Android NDK;
3. integration in a whole tool-flow (of the ASPIRE project) to ensure as much as possible composability with other protections;
4. very fine-grained code blocks (albeit with a performance overhead);
5. convenient way to specify and control deployment via source code annotations;
6. evaluation on real networks, ranging from local networks, to 3G mobile networks.

The paper is structured as follows: in section 2 we introduce the Code Mobility architecture and all its components, then in section 3 we describe how to create offset-independent mobile code. In section 4 we introduce the automated tools support to instrument and split binary code in code blocks before run-time; then in section 5 we describe the performance analysis of our framework on different network settings, while section 6 draws the conclusions and discusses future work.

2. CODE MOBILITY ARCHITECTURE

In the code mobility architecture we designed and the prototype tool support we developed, a client application (which may also be a dynamically linked library) is stored on the user device as an incomplete executable that does not contain all the application's code. Two components, Downloader and Binder, are introduced for this technique: They are able, respectively, to fetch binary code blocks from a trusted server at run-time, and to patch these into the running process' memory, in a dynamically allocated memory area. These components are not part of the original application and they have to be injected into the protected version. This approach aims for mitigating reverse engineering: instead of preventing analysis of code by making the code complex, we make sure that the code is not available for static analysis on the client-side as long as possible, and deliver the necessary code only when it is actually needed by the control flow. The Code Mobility framework's architecture is depicted in Figure 1: it can be seen as a dynamic binary obfuscation approach based on the deployment of an incomplete application whose code arrives from a trusted network entity (the Code Mobility Server) as a flow of mobile code blocks; such blocks are fetched by the Downloader component and arranged in memory by the Binder component at run-time, with an unpredictable memory layout. The Code Mobility framework is compliant to the ASPIRE project reference architecture [16], defining the ASPIRE portal, which acts as a common entry point for all online protections developed in the ASPIRE project, and the ASPIRE communication control logic (ACCL) library in the client host, which provides native socket support to Android apps.

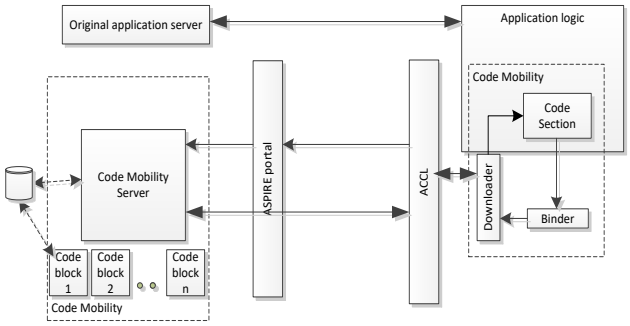

Figure 1 - Code Mobility High-Level Architecture

Mobile code blocks coming from the Code Mobility Server will not be placed in a statically known location in the binary code section, but will instead be placed in dynamically allocated memory. So the location of the code blocks will not be fixed. This implies that the mobile code needs to be position-independent code (PIC) that can be dynamically relocated, and independently from the non-mobile code part of the client's binary code. Thus, indirections need to be inserted in the transformed code to deal with these variable code locations, both in the static, non-mobile parts of the client application and in the mobile code. Fortunately, only local code transformations are required for this: instructions will be replaced with small code snippets that can deal with the a priori unknown addresses at which the code has been loaded.

Our current design only supports mobile code blocks with a single entry point. These can be entire procedures or parts of their control flow graphs (CFGs). This significantly simplifies the implementation of the Binder and its book-keeping data structures.

We should also point out that we only make code mobile. Data allocated statically in the binary's data sections is left static, including statically allocated data that is accessed by the mobile code.

2.1 Binder

The client-side Binder component is in charge of invoking the Downloader when required. The Binder is invoked by the application when the control flow reaches a mobile code block. If that block has not been downloaded from the server yet, the Binder asks the Downloader to retrieve the requested missing code block. Through the ACCL communication library, implementing a socket API in native code, the Downloader queries the Code Mobility server in order to obtain such a block and finally the Code Mobility Server sends the proper block back to the Downloader.

After the fetch process the Binder places the block in memory and makes sure that the just downloaded block will not be downloaded again, reducing the overhead effort introduced by the protection technique. Eventually the Binder redirects the control to the entry point of the downloaded code, where the application can continue normally.

In the original client application, control flow transfers (such as procedure calls) to mobile procedures need to be transformed such that:

1. Upon the first execution of a call to a mobile procedure, the Binder and Downloader components are properly invoked in order to obtain the code from the server;

2. Upon subsequent calls to the same mobile procedure, the control is immediately transferred to the already downloaded mobile code.

By avoiding going through the Binder again, the performance overhead of mobile code can be limited. Figure 2(a) shows the original control flow without mobile code: procedure f() is selected to become mobile. In the transformed program, shown in Figure 2(b), our tools inserted a look-up table with procedure pointers. Look-up table accesses are depicted with dashed arrows whereas control flow transfers are depicted with regular arrows. The pointers in the look-up table either point to stubs that invoke the Binder to start the mobile code downloading process, or they point directly to the already downloaded code. All calls to mobile functions are transformed into a code snippet consisting of a table lookup and an execution control redirection to the address loaded from the table.

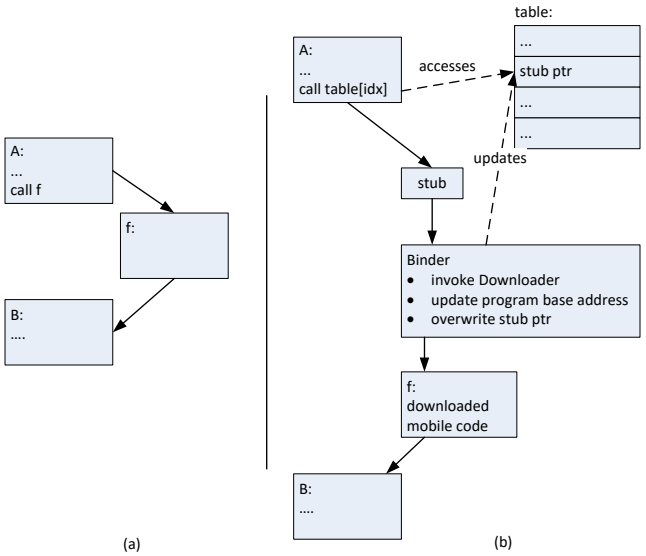

Figure 2 – Function Call: Before (a) and After (b) Code Mobility Transformations

Initially, when the called mobile function f() has not yet been downloaded and bound, the address in the look-up table is that of a stub that invokes the Binder.

This stub calls into the Binder, providing as argument the index at which this stub is installed in the look-up table. This index is then used as an identifier of the mobile function to be downloaded. The Downloader component is then invoked to retrieve the mobile (PIC) version of the function's code body from the Code Mobility Server, and stores this code body in a dynamically allocated buffer.

Finally, the Binder updates the entry in the pointer look-up table by overwriting the address of the stub with the address of the downloaded code, after which it redirects the control to this code, and normal code continues.

Subsequent calls to the already downloaded procedure f() then proceed as indicated in Figure 3. Since the Binder has already updated the pointer in the look-up table at the used index to let it point to the downloaded code, the inserted code snippet (in block A in Figure 3) now loads this procedure pointer and thus transfers control immediately to the previously downloaded mobile code. So for subsequent calls, the overhead is limited to the table look-up, and the necessary spilling and restoring of registers.

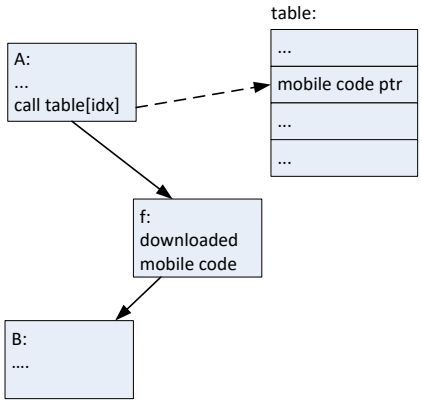

Figure 3 – Calling Function f() passing through already downloaded mobile code

The Binder contains three tables: the GMRT (Global Mobile Redirection Table), a mutex table, and a table that stores whether a certain mobile block is present or not (if it's not, the entry is zero). At program start-up for a certain mobile block its GMRT entry contains the address of the associated stub, the mutex entry is initialized, and the entry in the last table is zero. When control is transferred to the stub through the GMRT, it will itself invoke the Binder with the index for the mobile block as an argument. The Binder locks the corresponding mutex and checks whether the block is present. This is very unlikely to happen, unless another thread just downloaded it.

If the block is not present the Binder instructs the Downloader to download the block. It then writes the base address of the protected binary onto the first four bytes of the mobile block, maps all the pages the block resides on as executable, backs up the current GMRT entry (which is the address of the stub) to the last table, replaces the GMRT entry with an address in the mobile block, and unlocks the mutex. As a small aside: the locking and subsequent unlocking of a mutex is not actually done in single-threaded applications, avoiding unnecessary cost.

2.2 Downloader

The Downloader is invoked by the Binder to request a specific mobile code block (identified by an index) when needed by the client application. After a mobile code block is correctly received a suitable heap-allocated memory area is prepared, filled with mobile code, and passed back to the Binder. The returned memory area must be allocated with respect to a few constraints:

- It must be memory page aligned so that the Binder can apply the proper access rights (execution) later
- Every mobile block must be allocated in one or more dedicated memory pages so that there are no access right conflicts: after a page is declared as execution-only it should not be accessed in write mode to avoid segmentation faults

The first constraint is respected by using the posix_memalign system call which allocates page aligned memory. The latter is respected by simply allocating the minimum number of memory pages able contain to the full mobile code block. These constraints result in an additional overhead (in terms of time and

memory consumptions) because, after receiving the buffer containing the mobile block, the Downloader must copy it into a new memory-aligned one. This overhead could be avoided introducing a new parameterization that instructs the ACCL API to allocate page aligned buffers natively. Furthermore reserving full memory pages for single mobile blocks lead to an additional overhead in memory allocation. This overhead can be computed as:

$$\sum_{i=1}^{N} ps - mbs_i$$

where, N is the total number of mobile code blocks transferred over time, ps is the single memory size, mbs_i is the i^{th} mobile block size. In a scenario where one hundred blocks are extracted from the original application the additional overhead is upper limited by the page size times one hundred. As an example if the page size is 4kB the "wasted memory" would be less than 400 kB. Tuning the amount of original binary code made mobile can mitigate this.

2.3 Server-Side Components

This component reachable by the client via a network link and is trustable by hypothesis. The Code Mobility Server is the back-end invoked by the Downloader component on the client-side. It is in charge of delivering requested mobile code blocks by accessing a repository using a given index.

3. OFFSET-INDEPENDENT MOBILE CODE

When a mobile code block is mapped into the address space of the binary or library, this is done on a randomized address on the heap because of ASLR. The statically allocated, non-mobile code and data of the binary or library is randomized as well. This implies that the offset between the mobile code block and the non-mobile code and data is unknown at compile time. This differs from standard position-independent code, where the offsets between elements in a statically allocated segment are still fixed. Position-dependent code or position-independent code (PIC) in the original binary therefore needs to be rewritten into so-called offset-independent code.

On architectures like the x86, this rewriting is straightforward, as one of the registers is used (by convention) as a so-called global pointer (GP) to the global offset table (GOT) that contains pointers to all code and data fragments of which the absolute address might be needed at some point.

On architectures like ARMv7, however, position-independent code makes heavy use of the visible program counter (PC) register and of PC-relative addressing. So there is no fixed register holding a GP, and PIC code is full of PC-relative offsets.

Figure 4(a) shows an ARMv7 assembly PIC fragment. To load the value at label .Ldata into memory with the instruction at .Lins2, a PC-relative address stored in a so-called literal pool in the .text section is first loaded into a register at .Lins1, and then used in the PC-relative memory access at .Lins.[1] All edges in the code fragments of Figure 4 correspond to offsets that are known at compile time. For that reason, they can be computed by the linker or protection tool, and stored as entries in the literal pools, or they can be encoded as immediate operands of instructions.

Suppose that the three instructions in red become mobile. Figure 4(b) shows the transformed static PIC. In this example, we assume that enough registers are available (like r6 in this fragment) to store temporary values. If not, additional spill code would be needed.

Instead of the original code, the first two inserted instructions in red produce the address of the GMRT. The next instruction loads the address of the mobile block from its (fixed) index in the GMRT, and then control is transferred to that address. When the mobile code block is not yet present, control will be transferred to a stub that invokes the Binder with the requested block index instead. The binder then invokes the downloader and overwrites the address of the stub in the GMRT with that of the downloaded block.

Please notice that in the remaining static code of the shown example, there is absolutely no need to place the instruction at .Lins4 right after the inserted instructions, since the control transfer from the mobile code to that instruction will happen indirectly. Besides hiding the mobile code, this also opens up opportunities to obfuscate the control flow in the code that remains static. When code mobility is combined with code layout randomization in which independent code fragments (i.e., fragments that do not need to be allocated consecutively because there are no fall-through execution paths between the fragments) are reordered and spread throughout the whole text section, the fact that .Lins0 and .Lins4 belonged to the same basic block will no longer be apparent in the static code.

Figure 4(c) shows the offset-independent mobile code block that replaces the three instructions extracted from the static code. The single entry point of this code block (i.e., the address that will be stored in the GMRT by the Binder) is actually the third word in this block (marked by the .Lins1 label). The second word is an instruction that restores some registers and the first word is a kind of GP. In our current implementation, it points to the start of the statically allocated code and data of the binary or library in memory, i.e., to the .Ltext label that marks the start of the .text section. As this address is randomized by ASLR, it is unknown at compile time. Therefore it is the Binder's job to fill in this address in the blocks first word at run time, i.e., when the mobile code block is placed in the process' memory space.

Rather than relying on the PC and a PC-relative address loaded from a literal pool to access statically allocated data as the instruction at .Lins2 did in the original code fragment, the rewritten code in Figure 4(c) uses the .text GP stored in the first word of the block, and an .Ltext-relative address loaded from the literal pool. Likewise, to facilitate the jump from the end of the mobile code back to .Lins4 in the static code, that address of .Lins4 is computed using an .Ltext-relative address.

[1] The +8 in the PC-relative address is due to the ARM specification that a used PC equals the PC of the instruction that uses it plus eight.

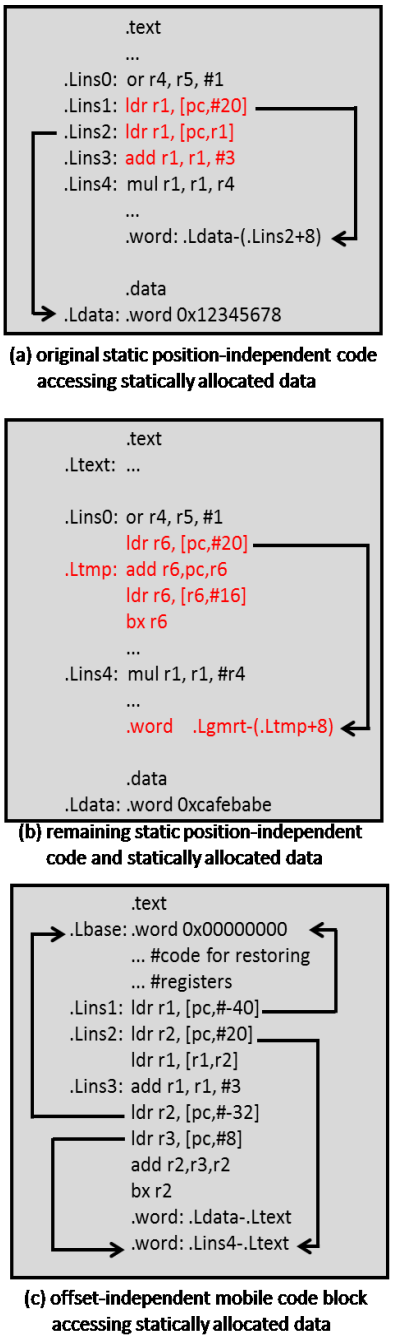

Figure 4: Example of offset-independent code

By combining the different redirection mechanisms discussed above, it is possible to rewrite all direct references, be it in direct memory accesses or in direct control flow transfers from mobile to static code or data, from static code to mobile code and even from mobile to mobile code.

To handle indirect references from static data to mobile code, we require another mechanism, however. This case occurs when pointers to mobile code are stored inside static data sections or when they are computed on the fly to be used in indirect control flow transfers to mobile code. Fundamentally, the problem with such references is that while the origin of the reference can accurately be identified (in source code or in binary code, as we will discuss in the next section), the points of use of those references cannot easily be identified accurately: Once some procedure pointer has been computed and stored in memory, it is very hard if not impossible in most programs (due to aliasing) to decide exactly where that pointer will be used in an indirect transfer. Run-time solutions to rewrite all potential indirect transfers where code pointers are used have been proposed in the SecondWrite binary code rewriting system and in other designs [20], but all of them introduce a significant amount of code and data bloat, which we consider unacceptable in many usage scenarios.

So rather than rewriting the code fragments that indirectly use references to mobile code, we propose to limit code mobility to regions that can only be reached through direct control flow transfers. In practice, this is straightforward: When we detect that a region we want to make mobile is accessed indirectly, an indirection pre-header is generated for this region. This pre-header consists simply of a direct branch to the original entry point of the region, which will later on be converted into an indirect branch. It then suffices to replace all indirect references to the region's original entry point (i.e., statically allocated code pointers or code pointer computations) by references to the indirection pre-header instead. This pre-header then remains static, thus avoiding the whole problem, while the whole region itself can still become mobile.

With the discussed transformation, the code mobility protection can be applied widely. It is clear that rewriting mobile code references to static code or data into offset-independent code can introduce significant overhead, in particular when additional registers will have to be freed. We will evaluate this overhead in the evaluation section.

4. AUTOMATED TOOL SUPPORT

It is not trivial to make the described form of code mobility generally applicable and usable for developers that may not have the time to invest in complex tools and that may have to operate in industrial environments that put a lot of restrictions on the used compilers and development tools.

In the ASPIRE project, we therefore designed a plugin-based tool flow that allows a developer to annotate the source code that he wants to make mobile, and that can be used in combination with open source compilers like LLVM and GCC, as well as with proprietary compilers such as ARM RVDS.

In this tool flow, we make use of three sets of tools, which corresponds to three phases as depicted in Figure 5.

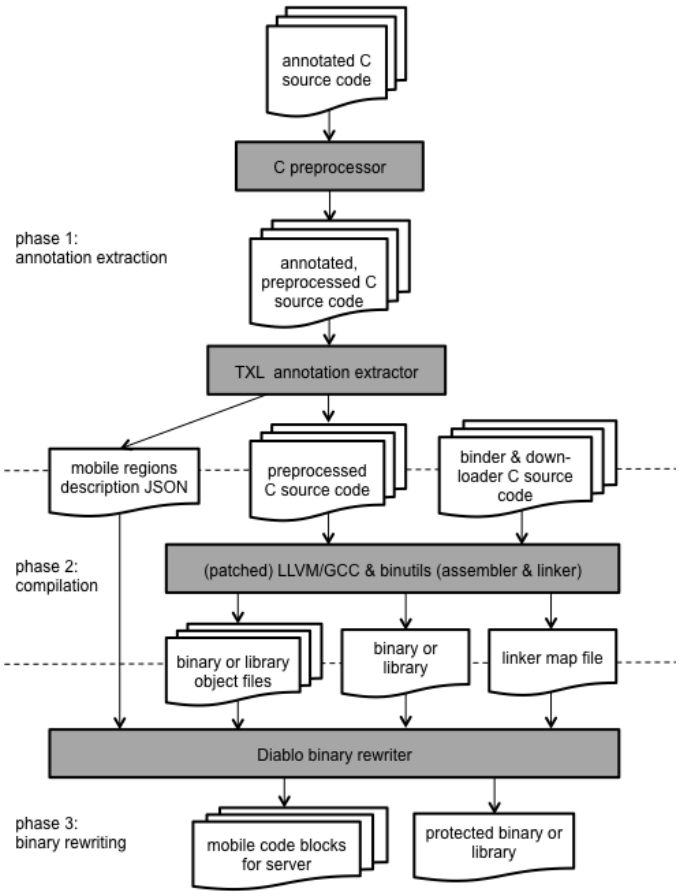

Figure 5: Code mobility tool flow

4.1 Specifying the regions to become mobile

First, we use source code analysis tools based on TXL [17] to extract annotations from the C source code.[2] The annotations are inserted by the programmer in the form of _Pragma directives as defined in the C standard since C99. Figure 6 depicts an example. The ASPIRE begin and ASPIRE end pragmas denote a code region to be protected, in this case with the code mobility protection. Many other protections are also supported by the full ASPIRE tool chain, but are out of scope for this paper. The regions mark by the pragmas have to follow the scoping rules of { ... } blocks in C, but this is not problem, since C programmers are obviously very familiar with this scoping.

The analysis tool extracts the annotations from pre-processed source code, and produces a JSON file that identifies the regions by means of their path and file names, their line numbers, the functions in which the regions were found, as well as the protections that were specified for each region. The tool also removes the ASPIRE pragmas, such that compilers will not complain about unknown pragmas.

In addition, the user can edit the JSON file, for example to mark additional functions that need to be made mobile. Wildcards can be used to denote multiple functions and multiple files. This eases experimenting with regions, for example to find a good balance between overhead and protection. Moreover, it also allows the user to specify that functions need to be made mobile that are not part of the original application, but that are injected into the application to implement other protections, such as code guards, by other plugin components in a protection tool flow. A range of such components is documented in some of the public ASPIRE deliverables available on the ASPIRE website [21].

```
int f(x) {
  int y,z,i;
  y = 2 * x;
  z = 0;
  _Pragma("Aspire begin protection(mobility)");
  for (i=0; i<y; i++)
    z +=  x << i;
  _Pragma("ASPIRE end");
  z /= 2;
}
```

Figure 6: Annotation code example

4.2 Compilation with standard compilers

In the second phase, the pre-processed code without the pragma is compiled, assembled and linked into a binary or library. The compiler, assembler and linker are instructed to generate debug information in the produced object files and in the final binary/library, as well as a linker map file. All compilers and linkers we know can do so. The linker map and the debug information, as well as sufficient relocation and symbol information need to be available in support of the third step, which consists of a link-time rewriting process.

Sufficient relocation and symbol information needs to be present to allow the link-time rewriter to rewrite the generated code conservatively, i.e., without breaking the original program behaviour. For example, so-called mapping symbols are needed that identify data present in the code sections. As another example, relocations should not be relaxed because important information is lost during the relaxation process. A standard linker does not suffer from that loss, but an advanced link-time rewriter does. Some compilers and binary utilities already produce sufficient information, such as ARM's proprietary compilers. Others, like GCC, LLVM and the GNU binutils do not do produce it out of the box. However, about 10 small patches, touching only few lines of code in total, suffice to make them produce it.

4.3 Binary code rewriting

The third phase then consists of the actual extraction of mobile code blocks and the rewriting of all code to insert the necessary indirections. For this, we rely on the Diablo link-time rewriter from Ghent University (http://diablo.elis.ugent.be) [23]. This rewriter has already been used for many different applications, incl. fault injection mitigation; obfuscation; kernel customization; memory safety; software diversity; and program compaction, optimization and instrumentation. In the ASPIRE project and tool chain, it applies many protections besides code mobility, incl. control flow obfuscation, code guards, ISA randomization, and anti-debugging techniques.

The internal program representation in Diablo is a so-called whole-program control flow graph (WPCFG). This WPCFG includes the CFGs of all functions in the program, as well as call and return edges, and additional so-called hell nodes and hell edges that can conservatively model unknown code (such as library code) and unknown (or at least not precisely known) control flow (such as calls through function pointers).

[2] For the time being, we only support C code because the TXL grammar we use is limited to C. C++ grammars exist as well, however, so this is no fundamental limitation.

Diablo first builds the WPCFG of the original application or library by disassembling it with the help of the linker map file and the original object files (and the relocation and symbol information contained in them). After this it annotates the nodes in the WPCFG with line number information that it extracts from the debugging information.

In the WPCFG, it then identifies the regions specified in the JSON configuration file. If a region has multiple entry points, it is split in multiple single-entry regions. Moreover, if a region is reachable through indirect control flow transfers such as calls through function pointers, the already mentioned form of pre-headers is inserted in the code. At that point, all regions are single-entry regions that are only entered through direct control flow transfers. Diablo then rewrites all those direct transfers into indirect ones that go through the Binder's redirection tables.

Next, the code inside each region is rewritten to replace all transfers and references to other mobile code regions or to static code and data by indirect, offset-independent references. Typically, the offset-independent references require more instructions, and often they need to store temporary (relative and absolute) addresses in registers. Diablo relies on its bi-directional, inter-procedural, context-sensitive liveness analysis to maximally find available registers in the code. If none are available at some point, the necessary number of registers is freed by inserting registers spills to the stack.

The rewritten regions are then extracted from the WPCFG, and migrated to separate WPCFGs, one per region. Entries and exits to and from these separate WPCFGs are modelled conservatively with hell edges, as if each region corresponds to a library that can be called by unknown application code. Once the original WPCFG has been split in multiple ones this way, each of them can still be transformed independently: The hell edges insure that dependencies between the blocks are respected automatically.

For each extracted regions, multiple WPCFGs can actually be translated, which are then diversified with the stochastic diversification techniques previously documented in literature [18][19], incl. opaque predicates, branch functions, flattening, and code layout randomization. Obviously, those protections can also be applied to the application code that remains static, including the binder and the downloader.

4.4 Current Status and Limitations

While most Diablo transformations, including the aforementioned diversification transformations can handle both the fixed-width 32-bit ARM code and mixed-width Thumb2 instruction sets of the ARMv7 architecture, as well as combinations of the two sets, the current tool support for producing offset-independent code only handles the 32-bit ARM subset. This is not a fundamental limitation however, only a matter of engineering effort.

Diablo in general can handle position-dependent as well as position-independent code, and so can the mobile code support we implemented on top of Diablo. There is one exception, however. The current tool cannot yet convert position-dependent switch tables (a.k.a. branch tables) into position-independent or offset-independent ones. WPCFG fragments containing such tables are therefore excluded. This is also a matter of engineering effort, not a fundamental issue.

The whole tool flow, including the code mobility support, has already been extensively tested with LLVM 3.3 and 3.4, as well as with GCC 4.8.1 and 4.6.4, and binutils 2.23.2 for ARMv7 software executed on Linaro Linux, as well as with the Android NDK API level 18 (incl. the already mentioned compilers and binutils) for software running on Android JellyBean (4.3). Both standalone binaries (from the SPEC2006 benchmark suite, as well as system utilities) as well as libraries have been tested, incl. security-sensitive plugins for the Android DRM Framework. In terms of structure and other requirements, such as the use of GNU_STACK and GNU_RELRO segments, the generated binaries and libraries conform to the strict security requirements of SELinux.

For the moment mobile blocks can't share pages yet. This is because when a new mobile block has to be loaded into memory, the page(s) it would be placed on would have to be mapped first to non-executable and then back to executable; in Android systems this would require a rooted device.

In case the code from another mobile block present on one of these pages is being executed in another thread at the same moment, this thread would generate a segmentation fault. A future solution for this problem would be to install a signal handler for segmentation faults in the binary that suspends this thread and resumes it when the page is executable again. For this same reason there is also no support yet for removing mobile blocks from memory, but this feature can be eventually added with minimal effort.

A basic version of the tool flow, including the mobile code support, will be open sourced during the course of the ASPIRE project (Nov 2013 - Oct 2016).

4.5 Testing

To make sure rewriting binaries with Diablo and splitting off mobile blocks didn't introduce any bugs it was verified whether rewritten applications still work correctly. For this purpose a stub downloader without an actual network connection was used, which simply maps the requested mobile block from the disk. The testing was done both for ARM Linux and Android, using Position Independent Executables. The applications used are those from the SPEC CPU 2006 benchmark. The testing was done by simply making mobile every named function present in the binary (if that was possible). As an example more than 3000 functions were made mobile for the 403.gcc benchmark.

5. PERFORMANCE ANALYSIS

Our performance analysis was carried out for our Code Mobility framework on three case studies written in the C and C++ languages, taken from SPEC CPU 2006 benchmark, namely libquantum, namd and milc. Library. Tests were performed on a SABRE Lite i.MX6 board with a Quad-Core ARM Cortex A9 processor at 1 GHz clock speed, with 1 GByte of 64-bit wide DDR3 at 532 MHz.

To evaluate the steady-state overhead of the mobile code transformations, i.e., the performance overhead on an application in which all executed mobile code blocks have already been downloaded, we used a customized version of Diablo. It transform the applications by applying the GMRT indirection and by making all mobile code offset-independent as described in Section 3, but it leaves the mobile code blocks in the binary's static code sections, but avoiding mobile blocks dumping.

To evaluate the latency that the downloading of the blocks might incur, we tested four different network scenarios: Localhost, LAN, WiFi, and 3G. In the localhost scenario, all components were configured such that the server, the client, and the code mobility server reside on the same test virtual machine: all communications took place locally, in order to exclude influence of transmission delays from collected data and have to reference measures for the other configurations.

In the LAN configuration, we tested the code on a 100 Mbps wired network; in the WiFi configuration we tested the code on a 54 Mbps wireless network, while in the 3G scenario we tested it on a HSDPA mobile network.

We measured the *latency*, i.e. the time required to establish a new TCP connection, whenever a new code block has to be downloaded; then we calculated the *blocks download time* to measure the time needed to download a mobile block on different network configurations. For the block download we made an arbitrary function mobile and measured the time needed to transfer it from the server to the client. The chosen function has a code footprint of 412 bytes.

Each experiment was repeated 500 times to collect data and we calculated average value and standard deviation of latency and time to download a mobile code block (see Table 1); for latency measures we run the code only 100 times. The last column of Table 1 represents the total execution time of a mobile version of the libquantum application. In this case we made a hot function mobile that represents by itself circa 50% of the executed operations.

Table 1. Summary of Performance Overhead (in *ms*)

Config		Latency	Block download	Libquantum 50% mobile
Localhost	Average Std Dev Overhead	0.12 0.03	9.36 6.63	369.37 66.28 +1.97%
LAN	Average Std Dev Overhead	0.32 0.02	6.98 1.46	370.45 65.74 +2,27%
Wifi	Average Std Dev Overhead	3.43 2.81	29.64 24.49	401.56 68.36 +10,86%
3G	Average Std Dev Overhead	134.27 119.58	228.87 154.44	659.54 173.42 +82,08%

Since most of the overhead comes from downloading blocks, which happens only once per mobile code block in our current implementation, and because our Android boards are relatively slow, we used the test SPEC inputs in our experiments. As expected, the worst overhead (82%) is found in case of mobile network connection while in a LAN scenario the overhead is as low as 2%.

Table 2 shows the performance once all mobile code blocks have been downloaded, i.e., when the redirection via the Binder's GMRT table is applied to all the fragments of an application.

For each benchmark application scenario the average total execution time and its standard deviation are provided, overhead is computed as the increment of execution time with respect to the original application, where no functions have been instrumented to become mobile. Each row indicates a different experiment with a significant percentage (20%, 50%, and 100%) of indirection/mobility, evaluated as the number of instructions executed in mobile functions over total number of executed instructions.

Table 2. Summary of Computational Overhead (in *ms*)

Execution time	Average	Std Dev	Overhead
libquantum			
original	362.23	63.11	
20%	363.18	67.93	+0.26%
50%	355.73	67.14	-1.80%
100%	394.80	62.06	+8.99%
milc			
original	85,697.45	29.98	
20%	85,417.24	46,73	-0,33%
50%	85,985.24	46.73	+0,34%
100%	88,557.82	133.17	+3,34%
namd			
original	92,729.70	107.89	
20%	93,403.56	124.05	+0.73%
50%	94,383.00	115.48	+1.78%
100%	95,503.73	119.98	+2.99%

In both the 20% and 50% coverage example we can see that the overhead is very low and sometimes even less than zero. This is due to the optimizations made to the code by Diablo. Only when 100% of the application's functions are made "mobile" forcing the indirection we can see a significant overhead occur.

6. CONCLUSIONS AND FUTURE WORK

The main contribution of our work is the definition of a new software protection relying on code mobility and the full automation of mobile code blocks generation. Our solution shows that splitting program in code blocks transmitted via network by a trusted server is a suitable and low-cost software protection that can be useful in defending software programs from reverse-engineering. Our protection creates problems for common reverse engineering tools and makes the code comprehension task more difficult for the attacker.

The proposed solution provides stronger protection than the one described in previous works. First of all, the addresses at which the mobile code is downloaded will differ from one run of the program to another. This makes all kinds of dynamic attacks more difficult. Secondly, almost all the necessary support is already available to free the allocated memory of mobile code blocks, and to restore the addresses in the look-up table to their original values, i.e., the stub addresses. Once this is implemented, it will allow us to make sure that not all mobile code is present at once, and to let multiple different mobile code blocks occupy the same memory addresses during a single run of a program. The fact that addresses in the program's address space then no longer map onto instructions in a one-to-one mapping, also complicates many dynamic and hybrid attacks, e.g., because many tools such as IDA Pro are engineered around the central notion that every code byte and address corresponds to at most one instruction.

Further research will be devoted to integrate code mobility with remote attestation in order to integrate tamper-detection techniques to improve the level of protection. Another line of research we want to explore is the combination of code mobility

and software diversity. Software diversity creates many different copies from an initial version of a program: each copy of the protected program is different in its binary shape, but is functionally equivalent to other copies [24]. Thus, attacks designed to work with one version might not work with other customized versions. Along with parameterizing the binary layout (diversity in space) we will explore how to extend it with diversity in time, by making Code Mobility even more configurable, by randomizing the binary structure [25] and parameterizing the number and size of code blocks and their duration in the client code before expiring and being replaced by a new version.

7. ACKNOWLEDGMENTS

The ASPIRE project runs 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] Collberg, C., and Nagra, J. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamper-proofing for Software Protection*. Addison-Wesley.

[2] IDA Pro Disassembler - multi-processor, disassembler and debugger. Online at http://www.hex-rays.com/idapro/

[3] Madou, M., Anckaert, B., De Sutter, B., and De Bosschere, K. 2005. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM Workshop on Digital Rights Management*, 75–82.

[4] Chang, H., and Atallah, M.J. 2001. Protecting Software Code by Guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management,* Springer LNCS 2320, 160–175.

[5] Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., and Jakubowski, M.H. 2002. Oblivious hashing: a stealthy software integrity verification primitive. In the *5th International Workshop on Information Hiding*, 400–414.

[6] Linn, C., and Debray, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *ACM proceedings of Computer and Communications Security Conference*. CCS-03. ACM, 290–299.

[7] Kanzaki, Y., Monden, A., Nakamura, M., and Matsumoto, K. 2003. Exploiting self-modification mechanism for program protection. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*. COMPSAC 2003, 170–179.

[8] Birrer, B. D., Raines, R. A., Baldwin, R. O. , Mullins, B. E., and Bennington, R.W. 2007. Program fragmentation as a metamorphic software protection. *In Proceedings of Third IEEE International Symposium on Information Assurance and Security.*IAS 2007. 369-374.

[9] Giffin, J. T., Christodorescu, M., and Kruger, L. 2005. Strengthening software self-checksumming via self-modifying code. In *21st IEEE Annual Computer Security Applications Conference*. ACSAC-05. 18-27.

[10] Jakobsson, M., and Reiter, M.K. 2002. Discouraging software piracy using software aging. In *Security and Privacy in Digital Rights Management: 1st ACM Workshop on Digital Rights Management*. Springer. 1-12.

[11] Collberg, C., Nagra, J., and Snavely, W. 2008. *bianlian: Remote Tamper-Resistance with Continuous Replacement.* Technical Report TR08-03, Department of Computer Science, University of Arizona.

[12] Collberg, C., Martin, C., Myers, J., and Nagra, J. 2012. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th ACM Annual Computer Security Applications Conference*. 319-328.

[13] Falcarin, P., Di Carlo, S., Cabutto, A., Garazzino, N., and Barberis, D. 2011. Exploiting Code Mobility for Dynamic Binary Obfuscation. In *Proceedings IEEE World Congress on Internet Security.* WorldCIS. 114-120.

[14] Falcarin, P., Scandariato, R., and Baldi, M. 2006. Remote trust with aspect oriented programming. In *Proceedings of the 20th IEEE International Conference on Advanced Information Networking and Applications*. AINA. 451–458.

[15] Scandariato, R., Ofek, Y., Falcarin, P., and Baldi, M. 2008. Application-Oriented Trust in Distributed Computing. In *Proceedings of the IEEE 3rd International Conference on Availability, Reliability and Security*. ARES. 434–439.

[16] ASPIRE project deliverable D1.04: Reference Architecture. Online at https://aspire-fp7.eu/project-deliverables

[17] Cordy, J. R. The TXL source transformation language. 2006. *Science of Computer Programming* 61, 3. Elsevier. 190-210.

[18] Coppens, B., De Sutter, B., and Maebe, J. 2013. Feedback-Driven Binary Code Diversification. *ACM Transactions on Architecture and Code Optimization* 9, 4, (Jan. 2013).

[19] Coppens, B., De Sutter, B., and De Bosschere, K. 2013. Protecting your software updates. *IEEE Security & Privacy*. 11, 2. 47-54.

[20] O'Sullivan, P., Anand, K., Kotha, A., Smithson, M., Barua, R., and Keromytis, A.D. 2011. Retrofitting Security in COTS Software with Binary Rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*. Springer Berlin Heidelberg. 154-172

[21] ASPIRE project website: https://www.aspire-fp7.eu

[22] Aucsmith, D. 1996. Tamper resistant software: An implementation. In *Proceedings of the First International Workshop on Information Hiding*. Springer. 317–333.

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

[24] Larsen, P., Homescu, A.. Brunthaler, S., and Franz, M. 2014. SoK: Automated Software Diversity. *In Proceedings of the 35th IEEE Symposium on Security and Privacy*. 276-291.

[25] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. 2012. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of the ACM conference on Computer and communications security*. CCS-12. 157-168.