Advanced Software Protection:
Integration, Research and Exploitation

D2.03

Binary Code Splitting Report

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	1st November 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D2.03 / 1.0
WP and tasks contributing:	WP 2 / Task 2.3
Due date:	Oct 2014 - M12
Actual submission date:	30 Oct 2014
Responsible Organization:	UGent
Editor:	Bjorn De Sutter
Dissemination Level:	Public
Revision:	1.0

Abstract:
This deliverable presents the status of the client-side binary code splitting work performed during year 1 in Task T2.3. Beyond the reference architecture and the compilation tool flow of this protection, individual components are discussed in more detail and so is future work. On two examples, the results are demonstrated.
Keywords:
SoftVM, X-translator, stubs, bytecode

Editor

Bjorn De Sutter (UGent)

Contributors (ordered according to beneficiary numbers)

Jens Van den Broeck (UGent)

Andreas Weber, Werner Dondl (SFNT)

The ASPIRE Consortium consists of:

Ghent University (UGent)	Coordinator & Beneficiary	Belgium
Politecnico Di Torino (POLITO)	Beneficiary	Italy
Nagravision SA (NAGRA)	Beneficiary	Switzerland
Fondazione Bruno Kessler (FBK)	Beneficiary	Italy
University of East London (UEL)	Beneficiary	UK
SFNT Germany GmbH (SFNT)	Beneficiary	Germany
Gemalto SA (GTO)	Beneficiary	France

Coordinating person: Prof. Bjorn De Sutter
E-mail: coordinator@aspire-fp7.eu
Tel: +32 9 264 3367
Fax: +32 9 264 3594
Project website: www.aspire-fp7.eu

Disclaimer
The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609734.

Executive Summary

For the sake of self-containment, this deliverable first repeats the overall principle, the reference architecture, and the compiler tool flow of client-side code splitting. These contributions are copied literally from previous deliverables D1.02, D1.04 and D5.01.

Next, the individual components' design and development are discussed. For the identification and extraction of native code chunks from the software to be protected, a new front-end tool based on the Diablo link-time rewriting framework is presented. For the X-translation of the extracted native code to bytecode, an LLVM-based tool contributed by SFNT is presented, along with its adaptations to be useable in the ASPIRE context. For the bytecode interpretation, SFNT's SoftVM is presented, along with the adaptations that were made to make it useable in ASPIRE. And another Diablo-based tool is presented that replaces the extracted native code in the software to be protected by invocations of the SoftVM.

Next, we report that the basic binary code obfuscation techniques (i.e., for control flow obfuscations) that were foreseen to be developed in year 2 of the project, are already available. They are opaque predicates, control flow flattening, branch functions, and code layout randomization.

The effect of the client-side code splitting is demonstrated on a small example, illustrating that not only the X-translated code is protected, but that also the control flow of the surrounding code is obfuscated. Furthermore, we show that code layout randomization effectively succeeds in intermingling the original application code and the SoftVM code, thus hiding the functionality of the SoftVM.

Whereas the current implementation of all tools only supports single-entry, single-exit code chunks, an extension is presented to enable the translation and interpretation of single-entry, multiple-exit code chunks, and to do so in a stealthy manner.

Throughout this deliverable, the important design considerations regarding a clear separation of concerns is discussed repeatedly. This separation of concerns improves the maintainability, and hence the integration of ongoing research into the integrated ACTC, and ensures minimal IP dependencies among the contributing partners.

Contents

List of Figures

Section 1 Introduction

Chapter Authors:

Bjorn De Sutter (UGent), Werner Dondle (SFNT), Andreas Weber (SFNT)

This deliverable documents the progress made in Task 2.3 of WP2 regarding the design and implementation of techniques for client-side code splitting. At this point in the project, the techniques concern the actual splitting itself, not the determination of the split points from a security or performance optimization perspective.

Many aspects of the design have already been presented in other deliverables. To make this (public) document self-contained, we repeat those presentations here. Whenever we do so, we explicitly clarify this, such that readers that already read previous deliverables can save time reading this document.

When tool component identifiers are used in this document, they correspond to the identifiers introduced in deliverable D5.01.

Due to the public nature of this deliverable, we purposely put a limit on the level of detail with which the design, development and implementation of certain components are described.

1.1 Principle

The content of this Section 1.1 is copied literally from ASPIRE deliverable D1.02 Section 5.1.

1.1.1 Definition

Virtual Machines (VMs) can be used to obfuscate a program or parts thereof. The VMs emulate a custom instruction set, which we will call bytecode, and therefore provide the necessary run-time functionality to have native code from the program replaced by bytecode. The emulator can be completely customized, i.e., it can implement many different forms of bytecode instruction sets, support for different languages, etc.

Such VMs can make it much harder to reverse-engineer programs because standard disassemblers and standard tracing tools (e.g., debuggers) do not target the custom bytecodes, and because the attackers are not as familiar with the bytecodes as they are with native, standardized and extensively documented instruction sets.

1.1.2 State of the art

In some VM-based obfuscation approaches, an application's executable code is first statically translated (and possibly encrypted) into a custom instruction set, i.e., some form of bytecode, after which that bytecode is linked with a VM. When the application launches, the VM gets control first, after which its dynamic binary translation (DBT) engine begins executing the application by translating the bytecode back to native code that is then executed in a so-called software cache under control of the VM [Anc06,Hu06]. In this approach, the VM supervises the application's execution. These types of VMs are susceptible to VM replacement attacks, in which an attacker replaces the original VM that implements a number of security features by one that lacks those features [Gho12]. The attackers can do so, because the application itself is not bound to the specific VM. As a countermeasure, techniques have recently been proposed to inject such bindings [Gho13]. Furthermore, such VMs are susceptible to tracing attacks [Rab13,Sha09], in which attackers collect and analyse execution traces to separate VM engine code (e.g., the code that maintains the software cache) from the original application code being executed in the

software cache. From the remaining application code trace, they can then reconstruct the original program.

In this section, however, we consider attacks on another type of VMs, i.e., VMs that interpret the bytecode. By avoiding a software cache that stores native code, such interpreting VMs avoid that the original, native code becomes available for inspection. As interpretation comes with a considerable performance penalty, however, such VMs typically only interpret parts of the application, i.e., those parts that are not executed too frequently, and of which the removal of the native code impacts the reverse-engineering most. The rest of the application is still in native code format.

When the application is launched in the latter approach, the VM does not control or supervise the execution of the application. Instead the application's native code directly starts executing. Through inserted call stubs, the native code invokes the VM whenever a fragment of bytecode needs to be interpreted. The stubs also pass the necessary parameters to the VM, such as a pointer to the bytecode to be interpreted and the current state of the program (i.e., register contents and condition flag status).

After interpreting the fragment, the VM returns control to the application, where a return stub translates the results of the VM into the processor registers and flags, after which the normal program execution continues. This form of protection comes with the additional benefit that the native code in the application originates from both the VM and from part of the original application. Both parts being in native code, they are harder to distinguish by an attacker.

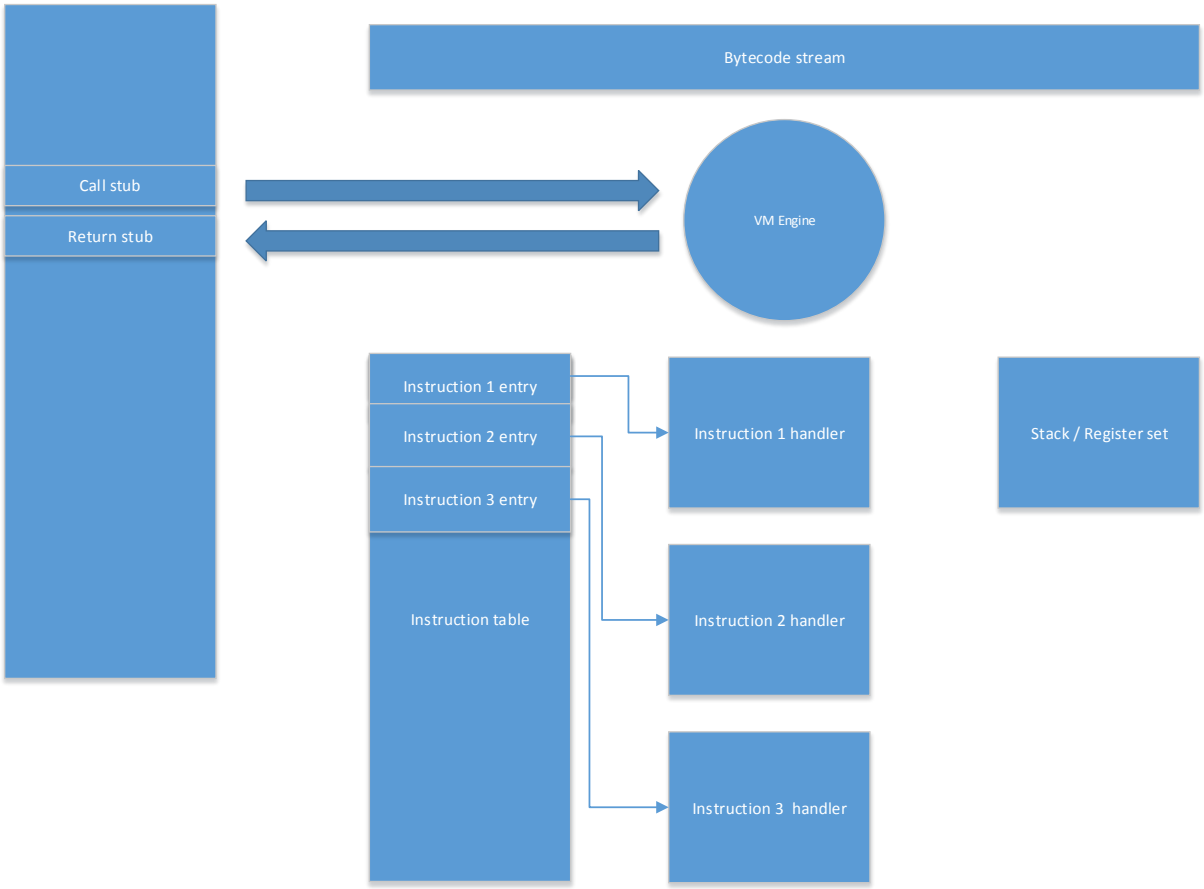

Figure 1 - Static interpreting VM

1.1.2.1 Static VMs

Figure 1 visualizes the internal structure and operation of a static interpreting VM. After such a VM engine is given control through the call stub, it fetches the bytecode instructions from the instruction table, and interprets them one by one. The bytecode instructions most often

represent an index into an instruction handler table, which contains pointers to the instruction handlers corresponding to each instruction. For each instruction, one handler is invoked, which performs the intended operations on the registers, stack, and memory. Handling one instruction at a time, this type of VM resembles classic bytecode interpreters and processor emulators.

The generation of the bytecode is done offline, by a so-called cross-translator that translates the extracted native code sequences into corresponding bytecode sequences. So the cross-translator effectively translates from one machine (the native, typically register-based instruction set architecture (ISA) to another (the bytecode, stack-based or register-based ISA).

1.1.2.2 Dynamic VMs

Dynamic VMs move away from the classic processor emulation. With dynamic VMs, bytecode is no longer passed to a general-purpose VM engine that can execute all individual bytecode instructions, and native code is not cross-translated to sequences of bytecode instructions.

Instead, each native code fragment to be replaced by bytecode is translated directly into a series of custom handlers. These custom handlers can correspond to original bytecode instruction handlers, but they can also combine multiple bytecode instructions, feature multiple (diversified) implementations for the same instruction, be obfuscated using control flow and data flow properties of the original native code, etc.

Obviously, code translated in this way can become quite big. It offers the advantage, however, that the VM and its internal operation cannot be modelled like a standard interpreter, and will hence be much harder to reverse-engineer. In particular, it will be harder to write tools that automatically analyse the interpretation and recover the original program from it [Sha09].

1.2 Reference Architecture

The content of this Section 1.2 is copied literally from ASPIRE deliverable D1.04 Section 3.1.

1.2.1 Client-side components

The following components are added to the application to implement the protection technique.

1.2.1.1 The embedded Virtual Machine

The VM consists of a collection of procedures that together implement the functionality of a custom bytecode interpreter. This code is linked into the application binary by the ASPIRE tool chain. Furthermore, its code will be dispersed throughout the application code by means of Diablo's code layout randomization support. As such, this VM component will not be a single, easily identifiable code region.

During its execution, the application will from time to time invoke the VM and pass it the relevant program state and the address of the bytecode to interpret as a replacement of some original, native code that was removed from the application to hide it from inspection and tampering. The VM will then fetch the bytecode and, starting from the passed program state, interpret the bytecode. This will include the computation of the address at which the execution of native code should continue after the interpretation has finished.

The ASPIRE tool chain will customize the VM, i.e., its instruction set and/or implementation, to some extent, so that an attacker cannot simply reuse results such as a bytecode disassembler from previous analysis without modification. The specific diversification techniques and their scope (e.g., diversify per application, per copy of the application, per

code fragment) are not yet defined but once they are, they will be implemented by SafeNet's cross translator.

1.2.1.2 Bytecode to be interpreted

For each code fragment that is removed from the application, a corresponding bytecode image is provided instead. All bytecode images are provided in object files that can be linked into the application binary by the ASPIRE tool chain linker. Again, Diablo's layout randomization capabilities will be used to disperse the bytecode images throughout the app's own data and code.

1.2.1.3 VM Invocation Stubs

Each bytecode image is accompanied by a distinct native code stub. This stub is responsible for passing the relevant program state to the VM according to the interface accepted by this particular VM, for passing control to the VM, for translating the updated state computed by the VM back to the native app, and for passing control back to the native app at the correct address. More concretely, the stub captures the contents of the physical processor registers and then calls the VM with the captured register values and the address of the corresponding bytecode image. When the VM finished the execution of the bytecode, the stub writes the updated values back into the physical processor registers and passes control back to the application. The necessary continuation address is provided by the just interpreted bytecode image.

Inside the application, the original instructions are replaced with a jump to the corresponding native code stub.

Once the stubs are linked into the application, and the jumps have been inserted, Diablo will optimize and obfuscate the stubs in its surrounding code. The result will again be that the stubs are not easily recognizable code fragments.

1.2.2 Run-time behaviour of client-side code splitting

Figure 2 presents the basic sequence diagram, depicting the run-time behaviour of this protection technique.

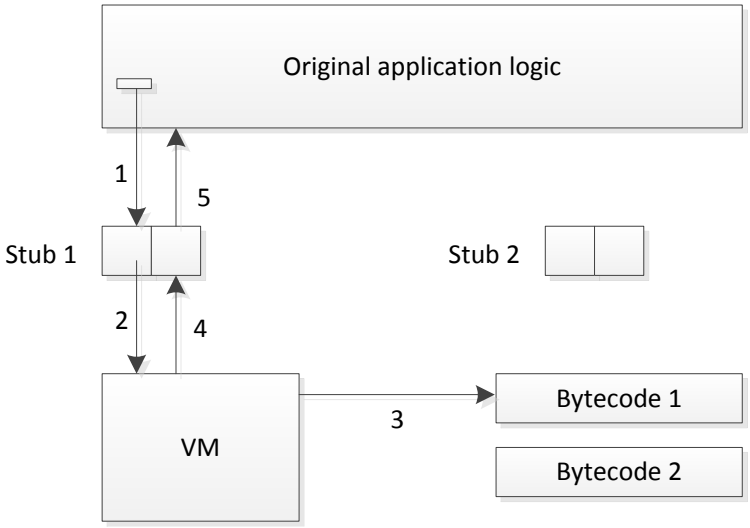

Figure 2 – Client-side code splitting run-time behaviour

A detailed description of each step depicted in Figure 2 is presented below.

Seq#	Operation description
1	The original application transfers control to the stub.

Details: This could be implemented in a first phase as an unconditional jump into the first part of the stub 1 code. The jump could probably be removed by means of branch forwarding, so, in practice, the stub will be inlined in the application code.

2	The stub sets up state for VM and transfers control.

Details: The stub collects the contents of the physical ARM processor registers and calls the VM, passing the address of the corresponding bytecode (VM-image) as argument.

When different stubs have different entry points into the VM, those entry points can be inlined in the stubs as well.

3	The VM fetches the Bytecode and interprets it.

Details: In case the bytecode is stored in encrypted form, the VM will need to decrypt it during this process.

4	After interpretation is finished, control is transferred to second part of the stub.

Details: The bytecode comprises code to calculate the address where the native execution should continue. This address and the updated register values are returned to the stub.

5	The stub cleans up and transfers control back to the application.

Details: The stub updates the physical ARM registers with the values the VM returned and jumps to the continuation address, transferring control back to the application.

1.3 Compiler Support

The content of this Section 1.3 is copied literally from ASPIRE deliverable D5.01 Sections 9.1, 9.3, and 9.4.

The overall binary code protection approach in ASPIRE consists of four major steps, as depicted in Figure 3.

In the first step, BLP01, the binary code is analyzed to decide where and how to apply the binary-level protections that require the generation and integration of additional custom software components.

For example, for the client-side code splitting by means of an embedded SoftVM (see D1.04 Section 3.1), bytecode needs to generate to replace native code sequences in the protected application, and this bytecode needs to be linked into the application. Also the SoftVM itself needs to be linked into the application, and in the more advanced version of the client-side code splitting, the SoftVM internals will be customized for the application in which it will be embedded. Before generating the bytecode and the customized VM, the code to be protected needs to be analyzed, and decisions need to be made about which native code will be replaced by bytecode. The necessary analyses and decisions are part of the first step, the result of which is a set of configuration files to drive the components that will generate the custom software components.

This generation of custom components in the form of object files BC03 constitutes the second step BLP02 of the binary-level ACTC.

In the third step, BLP03, the custom components of BC03 are integrated: they are compiled and linked in into the application, toghether with other, fixed software components BC09 that are also part of the protection, but that were compiled separately and independently of the preceding parts of the ACTC because those fixed components BC09 do not need to be customized for the application at hand.

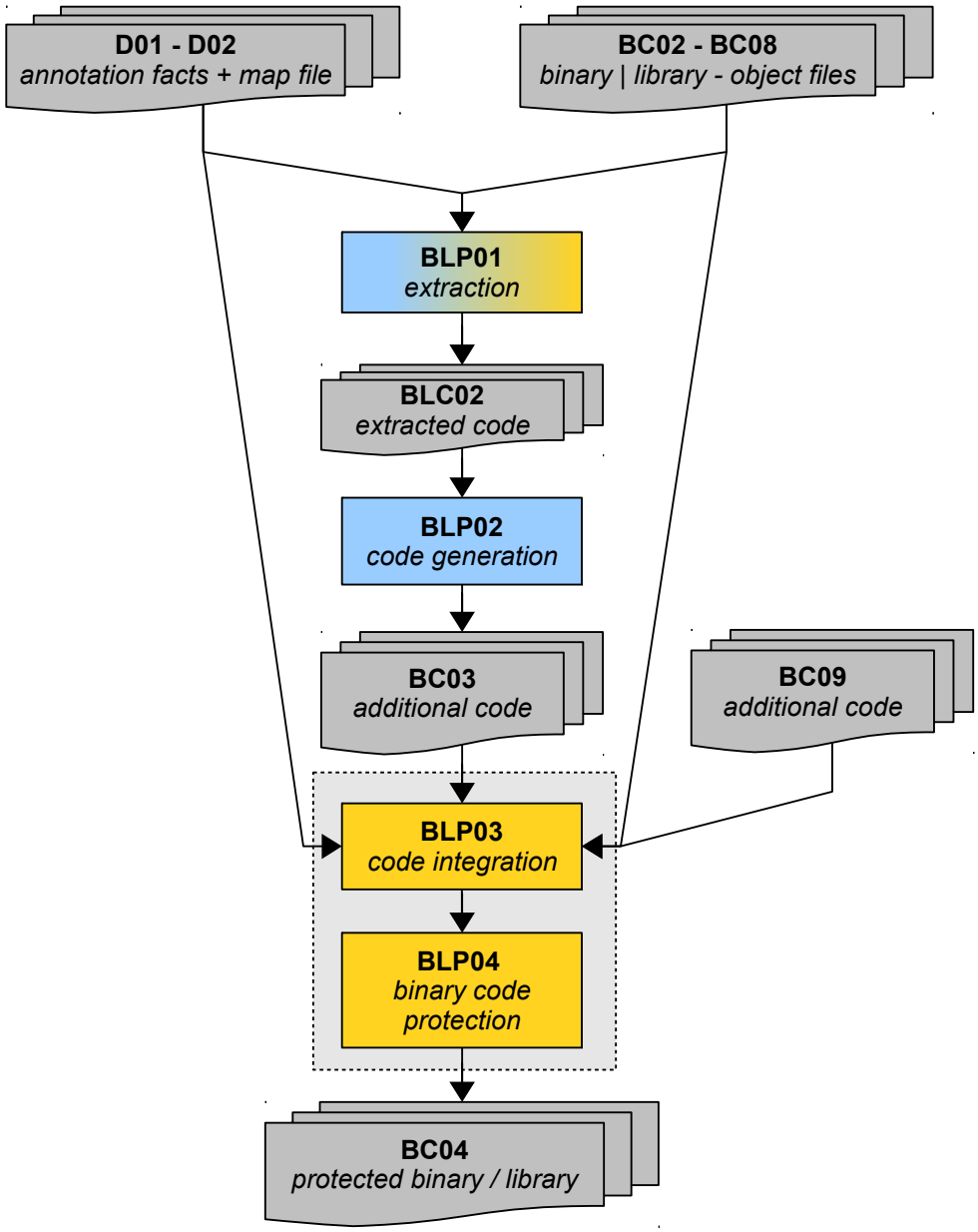

Figure 3 - Four steps of the binary-level part of the ACTC

Furthermore, the original code of the application is rewritten to actually invoke the custom components as needed. For example, for client-side code splitting, the previously selected native code fragments are replaced by stubs that invoke the linked-in SoftVM to interpret the corresponding, linked-in bytecode fragments. This integration will happen protection-per-protection.

In the fourth step, BLP04, all rewritten code and all integrated components are further protected by applying obfuscation and anti-tampering protections. Furthermore, the final code layout is determined, and the code is assembled and relocated (when necessary), such

that the final binary code is finally known. At that point, placeholders that might have been inserted during the rewriting in steps three and four can be filled in.

In practice, most of BLP03 and BLP04 will be performed during one invocation of a binary rewriting tool based on the Diablo link-time rewriting framework. For that reason, we have grouped these steps in Figure 3. In its initial implementation, a fixed SoftVM, which requires no customization, will be embedded in the code to protect.

1.3.1 BLP01: Native Code Extraction

As indicated in Figure 4, in BLP01.01 a Diablo rewriter will collect the code fragments that need to be translated from native code to bytecode. It does so on the basis of the annotation facts D01 assembled by the source-level component SLP04, and based on its usual inputs, which in this case correspond to the application BC02 to be rewritten, the corresponding map file (D02) and the object code (BC08) that was linked into the original application by the standard linker.

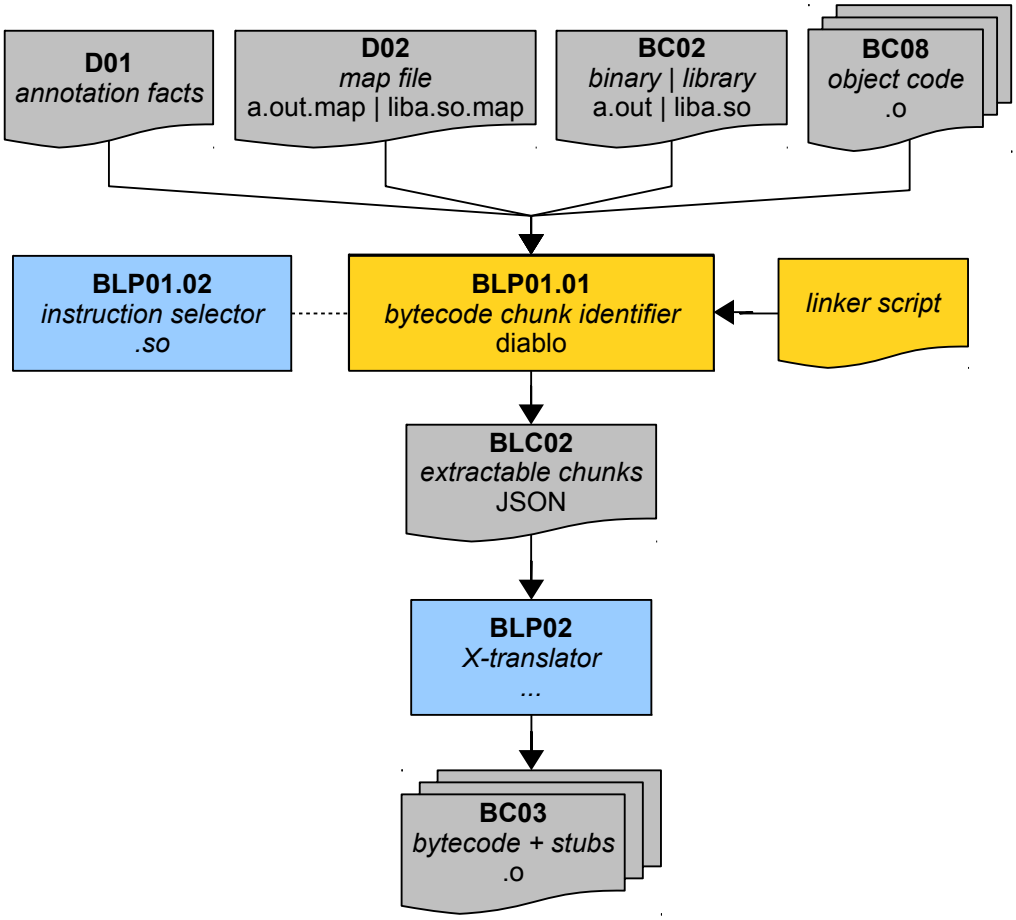

Figure 4 - Tool flow components for chunk extraction and bytecode generation

Diablo produces a description of the native code chunks in the form of JSON files (BLC02). The specification for this interface is presented in Appendix D.

To select the native code fragments to be translated to bytecode, the Diablo tool will consider procedures marked as such in the annotation facts D01. Within these fragments, all possible fragments will be selected, i.e., all fragments of which the instruction selector indicates that the instructions in them are supported by the X-translator and the SoftVM.

1.3.2 BLP02: Bytecode Generation

The second tool BLP02 in support of client-side code splitting is the X-translator. Based on the JSON files of BLC02 it generates bytecode, as well as stubs that will replace the selected native code fragments. The responsability of the stub is to invoke the SoftVM that will be embedded in

the application in BLP03, to let it interpret the generated bytecode that replaces the original native code, as well as to pass the program state to the SoftVM before its invocation, and back after its invocation.

The stubs and the bytecode will be generated as code and data sections in ELF object files, that can simply be linked into the application to protect.

UGent is responsible for the code extraction in the Diablo rewriter, and SFNT is responsible for the X-translator (as well as the SoftVM). This separation of concerns ensures a clear separation of Foreground IP, and a tool flow design in which components can easily be replaced by alternative ones after the project to facilitate exploitation of the project results.

However, unless special care is taken, this separation of concerns could introduce some (unwanted) dependencies between the involved partners' tools. Over time, the subset of the ARMv7 instruction set that is supported by the X-translator and the SoftVM will grow. So over time, the code fragments to be selected by the Diablo rewriter will grow. To avoid the need to keep the three tools spread over two partners synchronized with respect to the supported instruction set, we have decided to lift that responsibility from the Diablo rewriter, and to move it into a small dynamically linked library BLP01.02, the so-called instruction selector in Figure 4, that will be maintained by SFNT, and that will be invoked by UGent's Diablo rewriter to select the instructions that can be translated to bytecode.

1.3.3 BLP03: Code Integration

In the initial implementation of the ACTC, with the fixed SoftVM, the integration of the generated bytecode and code stubs, as well as of the SoftVM itself is straightforward. First, as shown in Figure 5, the SoftVM source code SC09 is compiled with the same tool chain used to compile the code to be protected. The result of this compilation process consist of the SoftVM object code files BC09.

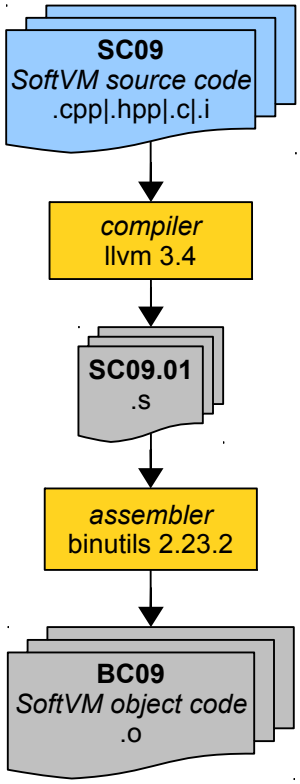

Figure 5 - Compilation of SoftVM

Next, the originally generated object code BC08 of the original application protected at source level (see Figure 10 in D5.01), is relinked with the generated bytecode and the stubs (BC03), and with the SoftVM object code (BC09), as depicted in Figure 6. This produces a new application BC04, with the corresponding map file D04. We use the names c.out and libc.so3 to indicate that these files denote extended version of the original binary a.out or the original library liba.so of BC02.

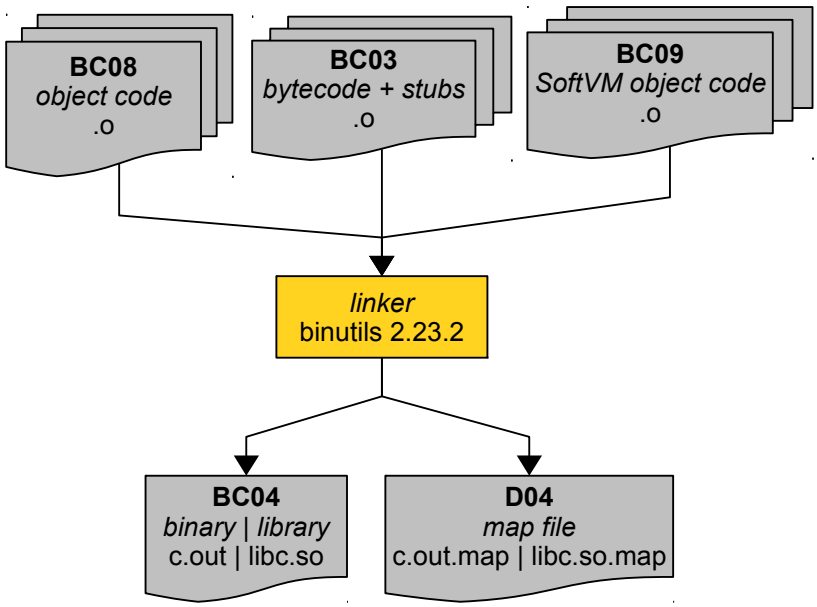

Figure 6 - Linking of the SoftVM

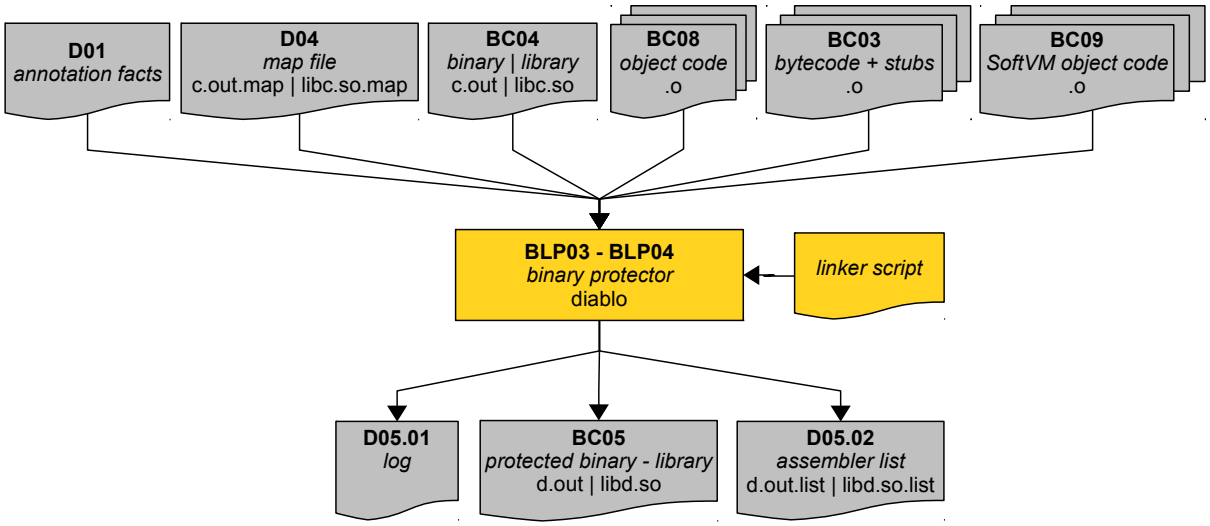

Figure 7 - Integration of the SoftVM and application of binary code obfuscation

Finally, Figure 7 shows the last step, in which a second tool BLP03 based on Diablo will rewrite that application to finalize the protection. This tool will replace the native code fragments that have been translated by the X-translator in step 2 by control flow transfers to their corresponding stubs. UGent is responsible for all of this integration in the Diablo tool.

1.4 Structure of the remainder of this document

In the remainder of this document, we provide additional information and progress reports on the tool components that have been developed in year 1 of the project. We do so in four sections for the code extractor, the X-translator, the SoftVM, and the integration of the SoftVM in the protected software. We also show a code example, i.e., and example in which native code has been replaced by code to invoke the SoftVM. Finally, we added a section that discusses near-future extensions to the SoftVM design that will allow for more extensive and more stealthy protection.

Section 2 Code Extractor

Chapter Authors:

Bjorn De Sutter (UGent), Jens Van den Broeck (UGent)

To implement BLP01, UGent developed a custom front-end on top of its Diablo link-time rewriting framework (see Section 9.2 in D5.01). In this section, we present this front-end, and the support for this front-end that was developed in the generic Diablo infrastructure. We first discuss their current status, and then discuss future work.

2.1 JSON support (Diablo Framework)

The code extractor, i.e., the Diablo frontend for BLP01.01 is configured in part by the source code annotations, that have been extracted in SLP04 of the ACTC and that have been stored in document D01, in a JSON format as specified in Appendix C of deliverable D5.01. Furthermore, in document BLC02, the code extractor has to export descriptions of the code chunks to be translated. That document also uses JSON, as specified in Appendix D of deliverable D5.01. For that reason, the necessary interfaces to Jansson, an open source JSON reader/writer/manipulator library (http://www.digip.org/jansson/), were implemented in Diablo.

2.2 External Instruction Selector (BLP01.02)

To obtain a clear separation of concerns (i.e., to optimize the development and integration processes in ASPIRE) and of developed IP ownership and IP rights (i.e., to optimize the exploitation potential of ASPIRE partners), the design decision was made to couple Diablo with an external instruction selector, as already discussed in Section 1.3.2.

This external instruction selector consists of functionality from the X-translator, of which SFNT extended the API, and exposed that API to Diablo in the form of a shared library. In Diablo, functionality was added to translate an instruction's features as recorded in Diablo's internal data structures into the description supported by the API. That way, Diablo can query the external instruction selector to check whether or not some instruction is supported by the X-translator and the SoftVM.

2.3 Code Extraction Functionality (BLP01.01)

As the Diablo framework cannot yet parse DWARF debugging information correctly (this support will be implemented in T5.1 in year 2 of the project), it cannot yet identify binary code fragments by their corresponding source code line number. It can, however, identify procedures of which the name is linked to a symbol in the symbol information stored in the object files of BC08.

We therefore developed functionality in the code extraction front-end BLP01/01 to identify the whole procedures in BC02 that are annotated as candidates for client-side code splitting in the annotation facts of D01. Within those functions, the front-end then first marks the instructions that can be X-translated by querying the instruction selector BLP01.02. Next, consecutive marked instructions are grouped into single-entry, single-exit code regions (so called chunks). Furthermore, an interprocedural, bi-directional context-sensitive liveness analysis is performed to identify the dead and live registers (and condition flags) on entry and exit of the code fragments. A description of each chunk is then exported in JSON format in BLC02, together with the relevant liveness information.

Section 3 X-translator

Chapter Authors:

Andreas Weber (SFNT)

The core of the X-translator that implements ACTC tool BLP02 is contributed by SFNT as background IP because substantial parts had already been developed prior to the ASPIRE project. The tool is provided to the project partners as a 32-bit Linux x86 executable.

The X-translator is a compiler that translates ARMv7 machine code fragments into functional equivalent bytecode interpretable by a suitable virtual machine. The virtual machine accepts a bytecode image and a machine context as input and calculates an updated machine context. The input machine context is a snapshot of the physical ARM processor state (registers and flags) and; the output machine context contains an updated processor state identical to the results the original machine code would have produced in the physical ARM registers.

As depicted in Figure 4, the BLP02 tool first reads the JSON BLC02 file that specifies the ARMv7 machine code chunks. The tool then generates an ARM assembly file containing for each chunk a native code stub in ARM assembly and an associated bytecode image as data. This assembly file is then assembled into the object code of BC03.

Both the JSON-parser and the assembly writer have been specifically developed for ASPIRE to allow the integration of SFNT compiler technology into the ACTC.

Internally the X-translator uses LLVM-IR as intermediate representation and relies on the LLVM libraries for code optimization. LLVM is an open source project building reusable compiler components and tools (frontends, optimizers, backends, linkers, debuggers, etc.) around a formally specified, strongly typed pseudo assembly language named LLVM-IR (LLVM-Intermediate-Representation). With LLVM-IR as its intermediate representation the X-translator follows a traditional compiler design with a front-end, an optimizer and a back-end.

The front-end consists of two phases. The first phase is the JSON-parser that reads the JSON file and constructs an in-memory representation of the code fragments defined in the JSON file. Each code fragment is represented by a `Chunk` object, which models the fragment's control flow graph as a set of connected `Basicblocks`. Each `Basicblock` consists of a list of `Instructions`, where each `Instruction` represents a disassembled ARM instruction. The JSON-parser is implemented as a descending push parser using the event-driven yajl library (Yet Another JSON Library - https://lloyd.github.io/yajl/) as lexer. For disassembly the frontend uses the Capstone engine (http://www.capstone-engine.org/)

The frontend's second phase translates each `Chunk` into an equivalent LLVM-IR program. LLVM-IR is an intermediate representation mandating static single assignment form (SSA form). From a front-end perspective this means the LLVM processor (a processor capable of executing LLVM-IR) has an infinite number of registers and that each register is defined exactly once.

Translating a `Chunk` into LLVM-IR requires the following steps:

1. Create an LLVM module as a top-level container.
2. Inside the LLVM module create a function that represents the entry point of the code fragment. This function takes as single parameter a pointer to a machine context struct.
3. Create instructions that load the register values from the passed machine context:

   a. For each value of the machine context create an LLVM-IR load instruction that makes the value available as a SSA-register.
   b. Make the arithmetic flags (N, Z, C, V) available as individual SSA-registers using appropriate LLVM-IR bitwise-shift and bitwise-and instructions on the SSA-flags-register.
   c. Track which emulated ARM registers maps to which SSA-register.
4. For each ARM instruction create an appropriate sequence of LLVM-IR instructions that emulate the effects of the ARM instruction (registers and flags) while updating the mapping between emulated ARM registers and their associated SSA-registers.
5. Create instructions that write the computed results back into the machine context struct:
   a. Combine the individual arithmetic SSA-flags into the SSA-flags-register using appropriate LLVM-IR bitwise shift and bitwise or instructions.
   b. Create LLVM-IR store instructions, which write the final SSA-registers back into their corresponding machine context slots.

The resulting LLVM-IR is rather big and inefficient as it loads and stores every register from the machine context although the emulated ARM instructions do not necessarily use all of them. Also not every emulated ARM register might be live at all times, leading to LLVM-IR code that calculates results that are never used. Both problems are addressed using the LLVM optimizer as a library, configured to perform a standard set of optimizations effectively eliminating unnecessary loads and stores as well as unneeded calculations.

The optimized LLVM-IR code is then fed into the back-end. The back-end is organized in two phases: The first phase goes over every `Chunk` and translates the optimized LLVM-IR into a bytecode image for the VM. The second phase generates the final assembly file containing the native stub code and the bytecode image for every code fragment.

A code fragment's native stub creates the machine context by capturing the contents of the physical ARM registers and then calls the VM passing the machine context and the associated bytecode image as parameters. After the VM returns, the stub updates the physical ARM registers with the values from the updated machine context and then resumes execution of the native application.

The stub is created by the X-translator, rather than by the Diablo-based SoftVM integrator of BLP03, to allow a clear separation of concerns. Indeed, this way Diablo does not need to know how to invoke the VM.

For the first phase the back-end implements a subset of LLVM's IR specification, which is sufficient to lower the front-end-generated LLVM-IR to the instruction set of the VM.

During the assembly generation, the back-end defines a text section containing all native code stubs labelled as `vmStart<chunkNumber>`. After that, it defines a data section with the bytecode images, each labelled as `vmImage<chunkNumber>`. The assembly file serves as input for the GNU ARM assembler to produce the ELF object file BC03, which in turn can be integrated into the protected application using Diablo.

At the moment the X-translator supports the following ARM instructions:

- `adc reg, reg, reg, shift`
- `adcs reg, reg, reg, shift`
- `adc reg, reg, imm`
- `adcs reg, reg, imm`
- `add reg, reg, reg, shift`
- `adds reg, reg, reg, shift`
- `add reg, reg, imm`
- `adds reg, reg, imm`
- `and reg, reg, reg, shift`

- `ands reg, reg, reg, shift`
- `and reg, reg, imm`
- `ands reg, reg, imm`
- `asr reg, reg, reg`
- `asrs reg, reg, reg`
- `asr reg, reg, imm`
- `asrs reg, reg, imm`
- `bfc reg, imm, imm`
- `bfi reg, reg, imm, imm`
- `mov reg, reg`
- `movs reg, reg`
- `mov reg, imm`
- `movs reg, imm`
- `mul reg, reg, reg`
- `muls reg, reg, reg`
- `sub reg, reg, ,reg`
- `subs reg, reg, reg`
- `udiv reg, reg, reg`

Section 4 SoftVM

Chapter Authors:

Andreas Weber (SFNT)

The SoftVM is a static virtual machine contributed by SFNT as background. It exposes the following function:

```
void vmExecute( MachineContext* mCtx,
                uint8_t* bytecodeImage,
                uint32_t sizeBytecodeImage
);
```

A chunk's native code stub is expected to call `vmExecute` with the correct parameters for this particular chunk. The `mCtx` is a pointer to the machine context struct containing the values of the physical ARM registers. It has the following structure:

```
typedef struct MachineContext
{
  uint32_t flags;
  uint32_t r0;
  uint32_t r1;
  uint32_t r2;
  uint32_t r3;
  uint32_t r4;
  uint32_t r5;
  uint32_t r6;
  uint32_t r7;
  uint32_t r8;
  uint32_t r9;
  uint32_t r10;
  uint32_t r11;
  uint32_t r12;
  uint32_t r13;
  uint32_t r14;
  uint32_t continuationAddress;
} MachineContext;
```

At run time `vmExecute` allocates a VM instance, loads the provided bytecode into the VM and triggers the interpretation of the bytecode with the supplied machine context as input. When the VM finished interpretation, the native code stub will write the updated machine context back into the physical ARM registers and jump to the continuation address.

The SoftVM core is a small, general-purpose stack-based virtual machine. It supports 32-bit integer but not floating point arithmetic. It uses 1-byte opcodes within a variable length instruction encoding. The machine's memory model supports global variables, local variables and heap memory with manual memory management (no garbage collection). The actual bytecode interpreter is a dispatch loop, which uses the opcode as index into a table of function pointers to retrieve the address of the corresponding opcode handler.

For ASPIRE the existing SoftVM core had to be adapted, so it compiles in the ASPIRE build environment and runs correctly on the ARM processor. As portions of the core had been originally written in C++, the adaption mainly required porting these parts to plain C, so that Diablo can rewrite the compiled code of the SoftVM, and can hence apply the other protections to the SoftVM code as well. Additionally new interface code (essentially

`vmExecute.c`, `softvm.c` and `softvm_test.c`) had to be developed, which enables the usage of the existing core in the ASPIRE binary code splitting scenario.

Currently the SoftVM runs correctly on x86 and ARM, where it produces expected results both with bytecode generated by the X-Translator and with bytecode of manually assembled test cases. At this stage care was taken solely on functional correctness, which means so far the SoftVM has not been benchmarked to estimate the introduced overhead.

Section 5 Integration of SoftVM and bytecode

Chapter Authors:

Bjorn De Sutter (UGent), Jens Van den Broeck (UGent)

To implement BLP03 (combined with BLP04) another frontend of Diablo has been developed. The design, development and status of that frontend tool is described in this section.

5.1 Chunk Recollection

The first thing the front-end does for BLP03, is to recollect the chunks that were selected for X-translation in BLP01. Now this collection of chunks happens on the basis of inputs D04, BC04, BC08, BC03 and BC09, instead of on D02, BC02 and BC08 that were used for BLP01. The code that implements the selection in BLP01 is reused for that purpose in BLP03. As the same annotation facts D01 are used for BLP03 as for BLP01, this recollection is in fact limited to the code originating from BC08, i.e., the original application code. This recollection hence results in exactly the same chunks as were exported by BLP01.

5.2 Chunk replacement

Next, for each chunk the front-end looks up the corresponding stub code in BC03, and simply replaces the chunk with (1) a push of the return address on the stack, i.e., of the address of the first instruction following the chunk, and (2) a branch instruction that transfers control to the stub. This way, the separation of concerns is maintained: the Diablo-frontend does not need to know anything about the SoftVM or the stub code to insert invocations of the SoftVM in the binary code to be protected.

At run time, instead of executing a native chunk

1. The return address is pushed.
2. Control is transferred to the stub that was linked in as part of BC03.
3. The stub is executed. As it was generated by the X-translator (based chunk descriptoins and on liveness information obtained from the code extractor), this code is tuned to invoke the SoftVM correctly, i.e., to pass all the necessary data (live-in values, return address, bytecode address, ...) through the correct API.
4. The VM is invoked and interprets the bytecode.
5. After the VM finished, it returns to the native code by popping the return address from the stack.

5.3 Providing Stealthiness

After replacing the code chunks by branches to the stubs, the Diablo front-end can reuse Diablo's existing infrastructure for optimizing, obfuscating and reordering binary code to complete the integration of the SoftVM and the bytecode. For example, the branch instructions can be removed by branch forwarding, and the whole-program code layout of the protected binary BC05 will be such that code of the SoftVM, the stubs, and original application code (all of which can also be transformed using Diablo's control flow obfuscations) will be mixed and intertwined in one big code section. This will significantly improve the stealthiness of the client-side code splitting approach.

Section 6 Binary Code Obfuscation

Chapter Authors:

Bjorn De Sutter (UGent)

In the ASPIRE DoW, and in particular in Task T2.4 of WP2, the development of binary code obfuscation support was foreseen to start in M13 of the project. Because opportunities surfaced much earlier to exploit the Foreground IP resulting from the basic infrastructure development foreseen in T2.4, we advanced this development in time. As a result, we have already been able to integrate basic control flow obfuscations in the Diablo tool that implements steps BLP03-BLP04 of the ACTC. We will present this work, its extensions and more advanced features, in detail in deliverable D2.06 at M18 of the project, as foreseen in the DoW.

Here, we only summarize the current results.

- The existing Diablo support for opaque predicates for the x86 architecture in Diablo has been ported to the ARMv7 architecture.
- Similarly, the existing Diablo support for code flattening has been ported to the ARMv7 architecture.
- Similarly, the existing Diablo support for branch functions has been ported to the ARMv7 architecture.
- All of the above obfuscations can now be implemented in position independent code, such that they can be applied to dynamically linked libraries.
- The necessary functionality has been developed to control the deployment of the above obfuscations by means of source code annotations.
- The x86 code layout randomization functionality in Diablo has been ported to the ARMv7 architecture.

As we show in Section 7.2, this basic support already enables interesting combinations of different types of protections, in line with the ASPIRE approach of combining different lines of defence in which different protections reinforce each other.

Section 7 Example Results

Chapter Authors:

Bjorn De Sutter, Jens Van den Broeck, Bart Coppens (UGent)

7.1 SoftVM code rewriting

As a testing example, consider the following procedure, named `asmFunction`, which was written manually in assembler to test the whole approach:

```
asmFunction:
    stmfd   sp!, {fp, lr}
    ldr     r0, m1Addr
    bl      puts
    bl      dumpRegs_entry
    mov     r0, #15
    mov     r1, #25
    add     r2, r0, r1
    bl      dumpRegs_entry
    ldr     r0, m2Addr
    bl      puts
    ldmfd   sp!, {fp, pc}
```

The `bl` instructions are calls to the `puts` and `dumpRegs_entry` functions that were inserted to allow us to check the correct execution of the protected program. The `mov` and `add` instructions in between are the instructions that in this example should be replaced by an invocation of the SoftVM and the interpretation of corresponding bytecode.

The Diablo framework can export control flow graphs (CFGs, exported in the .dot format) of the procedures in the software it rewrites, before and after the transformations. Figure 8 shows the CFG of the above procedure. This CFG uses alternative mnemonics (stmfd = push, ldr = adr,, ldmfd = pop) and alternative register names (sp = r13, fp = r11, lr = r14, pc = r15), but the corresponding instructions can easily be identified. For each basic block and instruction in the CFG, the addresses of the instructions in the original binary (BC02) are shown. Red arrows model call edges, blue arrows model the corresponding return edges, and black arrows model intra-procedural control flow. The four procedure calls are clearly visible, as is the fact that this `asmFunction` is called from within the `main` function. The chunk of three instructions to be replaced by bytecode are clearly visible in the left-most block in the CFG. Note that the instructions from the assembly file are all put at consecutive addresses. So when that program is disassembled by an attacker, the simple control flow in this procedure is manifest.

Figure 9 shows the corresponding CFG after that chunk has been replaced by instructions that push the return address 0x9e14 onto the stack, i.e., the address of the instruction following the chunk, and a branch to the chunks stub. These instructions can be found in the block that has not been assigned an address yet (indicated by the temporary address 0x0). That block contains an additional push and pop to temporarily free the register r1 that is needed to hold the return address before it can be pushed onto the stack.

The stub itself is not shown here, as Diablo considers it part of another procedure in the program.

To model the return from the SoftVM after the bytecode has been executed, an edge from the so-called hell node has been added to the basic block at address 0x9e14. With this conservative model, Diablo will correctly perform data flow analyses as needed for applying further protections, such as control flow obfuscations. In this example, this modelling also already hints to the fact that Diablo will be able to put the code executed right before the SoftVM invocation at a location in the program's code section that is completely independent of the location of the code that will be executed immediately after the return from the SoftVM. This feature, in combination with Diablo's capability to effectively randomize the code section layout, indirectly provides another level of obfuscation: not only the X-translated bytecode itself is obfuscated, but also its surrounding control flow.

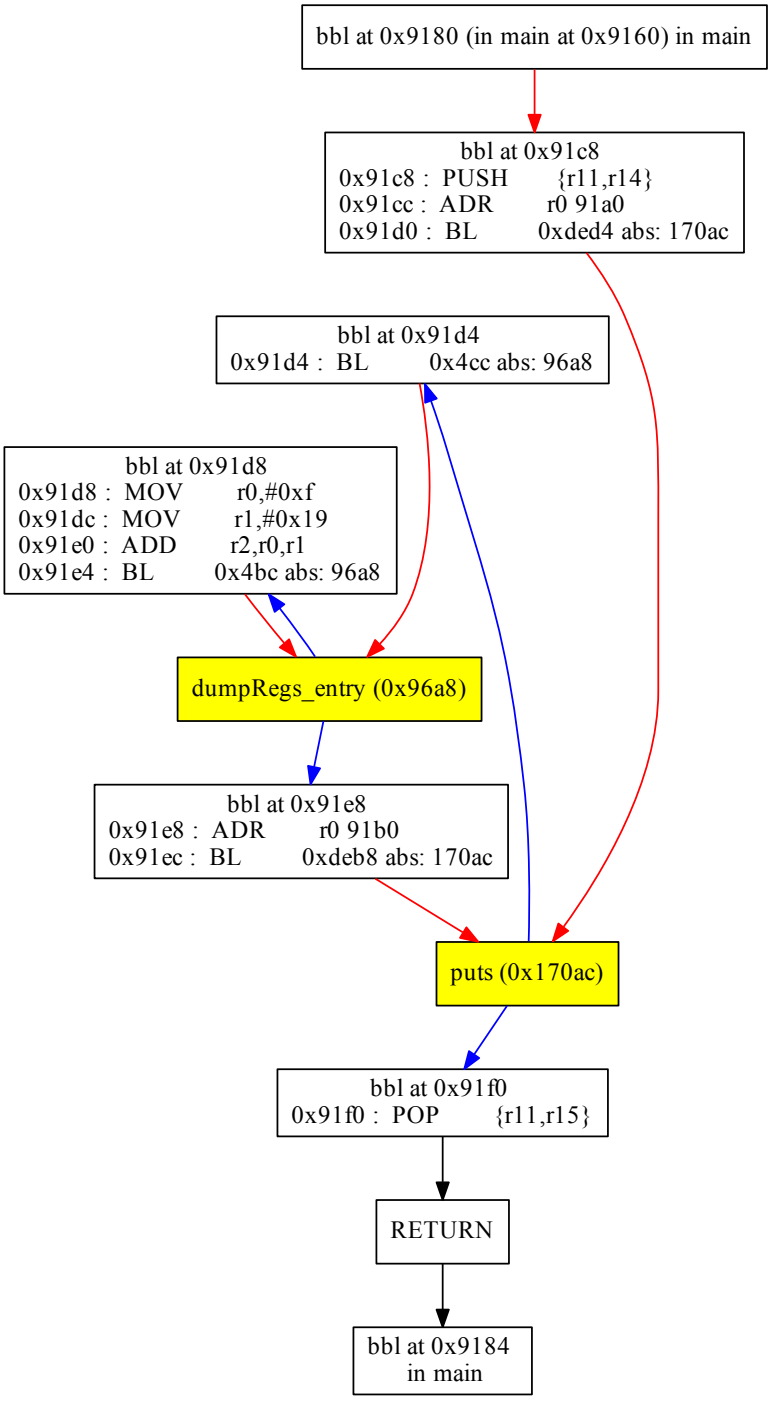

Figure 8 - Example control flow graph before client-side code splitting

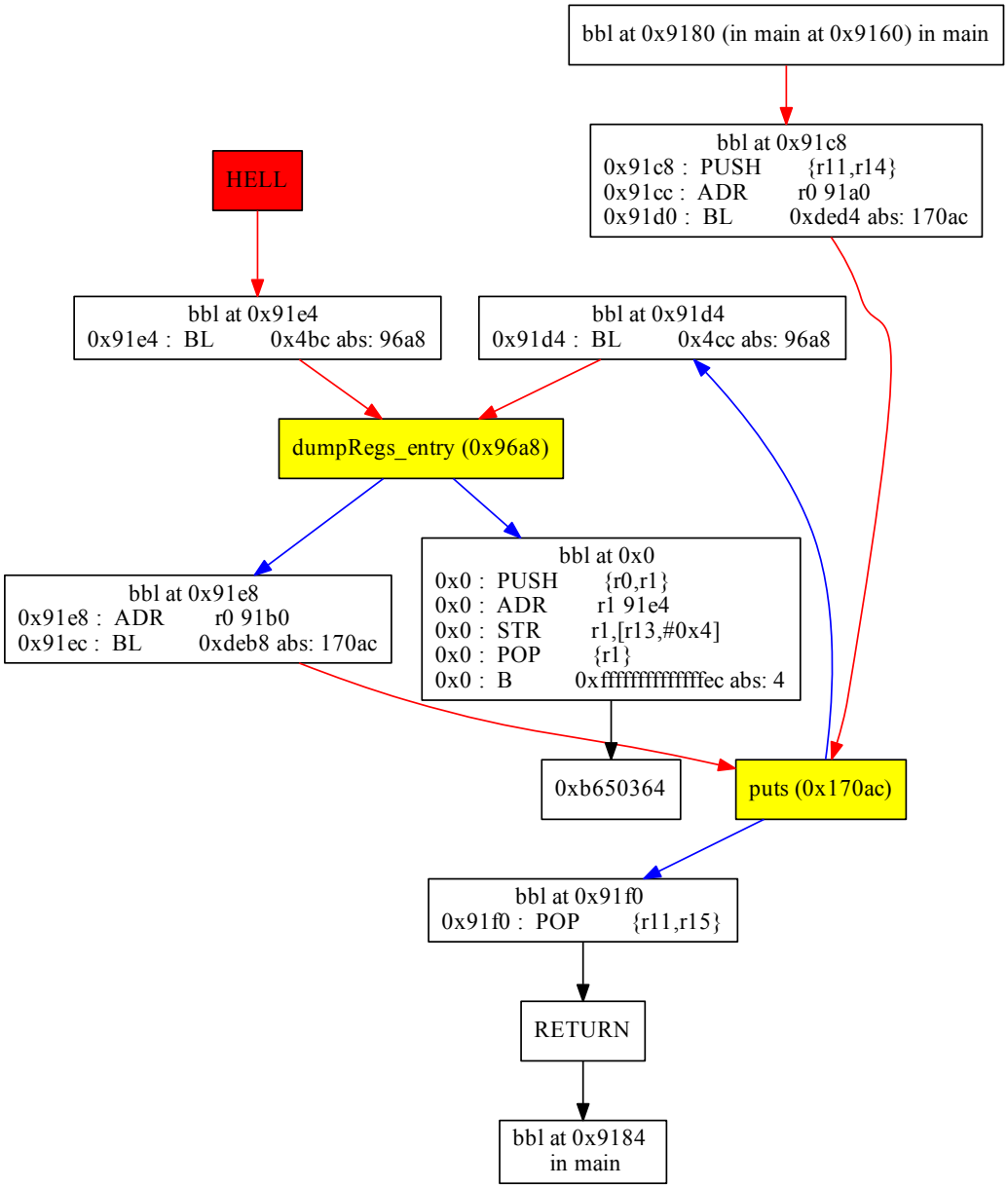

Figure 9 - Corresponding control flow graph after client-side code splitting

7.2 Combination of SoftVM with binary code obfuscations

As a second example, we applied the binary-level ACTC tools to bzip2, one of the SPECint2006 benchmarks. From within two major procedures of the benchmark (main and compressStream), we translated 40 native code fragments to bytecode. Furthermore, we applied our basic control flow obfuscations to the majority of the benchmark procedures. We then applied basic-block-level code factoring as well as code layout randomization.

The result is visualized in Figure 10. In this figure, each pixel represents one instruction of the benchmark's code section. Its colour indicates the origin of the instruction, according to the following legend:

- red: original application code;
- blue: SoftVM code;
- white: 40 stubs inserted to invoke the SoftVM;

- black: code that was factored out;
- green: code that was inserted to implement control flow obfuscations (i.e., opaque predicates, code flattening, and branch functions).

The most interesting aspect of this visualization is that the SoftVM code is truly spread throughout the whole code section. Moreover, the number of blue instruction sequences is far higher than the number of SoftVM procedures. The reason is of course that procedure bodies are not stored consecutively as in unprotected binaries, but spread throughout the binary. It is also visible that the SoftVM stubs, even though they are injected into only two procedures (main and compressStream), are spread over the whole binary.

This clearly demonstrates that even in its current basic implementation, the code layout randomization code is effective in mixing application code and SoftVM code. To the best of our knowledge, the ASPIRE tools are the only existing tools with this capability, which helps with hiding the functionality of the SoftVM, thus making the protection more stealthy.

For demonstrating and visualizing that the code factoring combined code fragments from the application and the SoftVM, our initial prototype still lacks the necessary support. When we first report the work of Task T2.4 (in deliverable D2.06 at M18), we will be able to evaluate and report this aspect in detail.

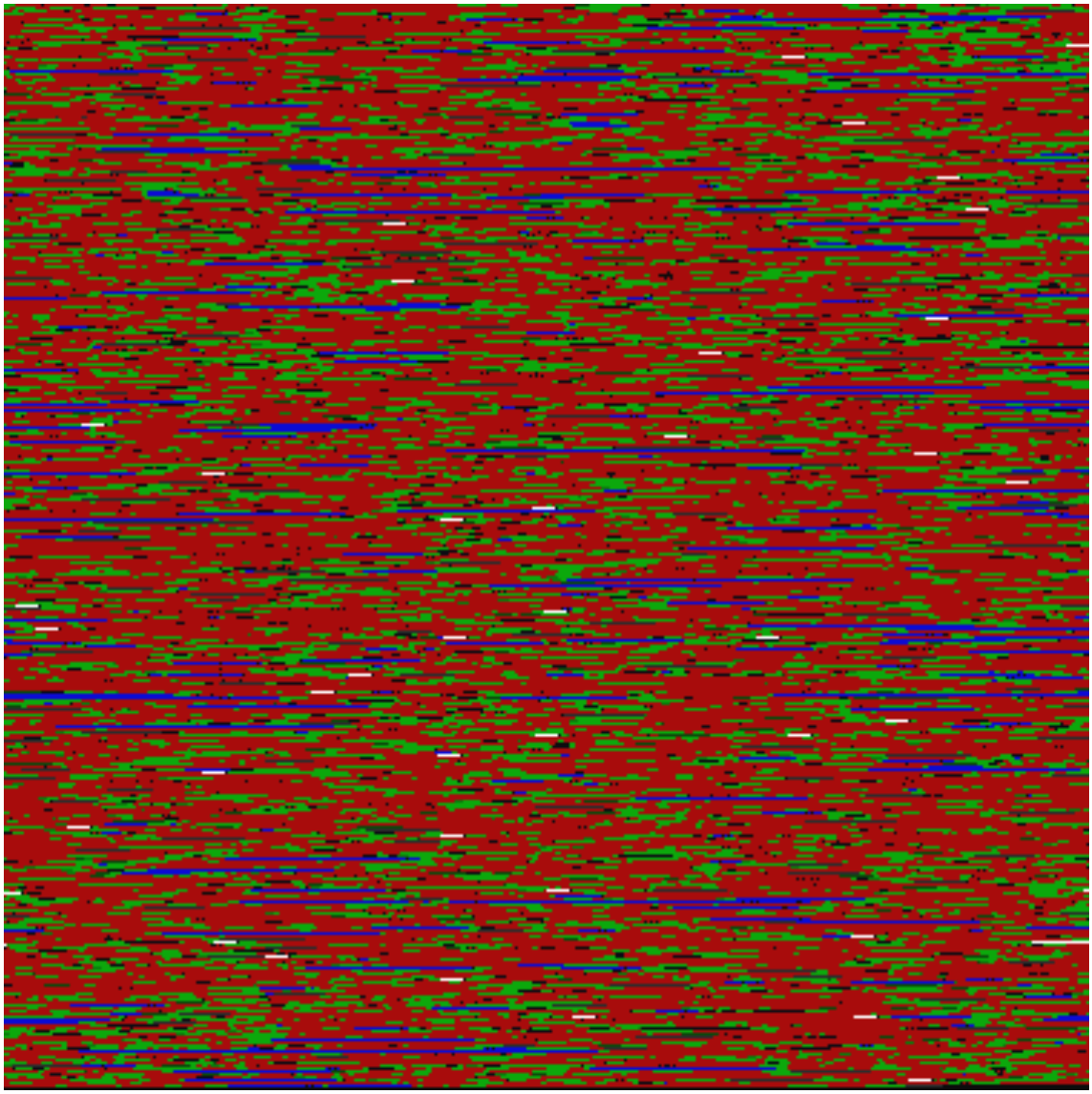

Figure 10 - Visualization of code origin in a protected benchmark

Section 8 Future Work

Chapter Authors:

Bjorn De Sutter (UGent)

In year two, and regarding the SoftVM client-side code splitting approach, we plan to work on

1. Complete the support for single-entry, multi-exit code chunks that might also contain internal control flow transfers.
2. Extending the set of supported instructions, i.e., instructions that can be translated from native code to bytecode.
3. Support for more fine-grained extraction of instructions, i.e., support for line-number based identification of extraction candidates by means of DWARF debugging information.
4. Support for considering profile information while selecting candidates, to balance performance and provided protection. Profiling support is already available in the Diablo framework, but is not used yet.
5. Set up a research collaboration with FBK, which was so far not involved in Task 2.3, to use their techniques based on barrier slicing to determine the code to translate from native to bytecode.

Furthermore, we will design and develop the necessary interfaces to the ADSS as its development continues in T5.2.

For multi-exit chunks, the VM, when it has finished the bytecode interpretation of a chunk, by default has to be able to continue execution of native code at different program points, corresponding to the multiple exit points of the chunk. For that reason, it will no longer suffice to simply push a return (i.e., continuation) address onto the stack before invoking a stub. To replace this, we have designed a technique based on passing symbolic information between the different tools of the ACTC. This technique will be implemented during year 2 of the project.

As presented in Appendix D of D5.01, the chunks described in BLC02, can contain multiple basic blocks. Each of them will get a label, and control flow transfers within a chunk, i.e., from one block to another in the chunk, will be described as edges between the basic blocks as identified with their labels. This will allow the X-translator to implement the necessary control flow within chunks.

To support multiple exits, the Diablo frontends will also assign labels to the multiple continuation points corresponding to all exits. The X-translator will then be adapted to generate bytecode that, upon exit of the chunk, transfers control directly to the address of the label of that exit. So the multiple continuation addresses will be hardcoded in the bytecode, rather than pushed onto the stack before invoke a stub (as described in Section 5.2).

To make the hardcoded continuation addresses stealthy, we foresee that they may be encrypted or encoded in many different ways. To achieve a clean separation of concerns, and sufficient renewability of the encryption or encoding used, the Diablo frontends should not need to know which precise forms of encryption or encoding are used. However, only Diablo knows the final continuation addresses, and it only knows so after all protections have been applied, and the final layout of the protection application is determined.

To solve this phase-ordering, separation-of-concerns problem, we will add a small additional processing step to the ACTC, which we call a fix-up step. Before it generates the final binary, Diablo will invoke the X-translator again, now providing it also the chunks' final continuation

addresses, as well as the final addresses of all bytecode fragments in the protected application.

In sub-step BLP05, shown in Figure 11, the X-translator will use this information to encrypt or encode the continuation addresses, thus obtaining the values that need to be hardcoded into the bytecode. It will generate the bytecode again, and return it to Diablo, which will overwrite the previously generated bytecode fragments (from BC03) with the new ones. This effectively will mean that the bytecode in BC03, generated during the invocation of the X-translator in BLP02 (see Figure 4), only serves as a placeholder for the final bytecode. We will still keep the existing step BLP02, however, as it is needed to compute the size of the (placeholder) bytecode fragments before they are linked into the program, and as that separate phase will later also be useful to generate custom SoftVMs.

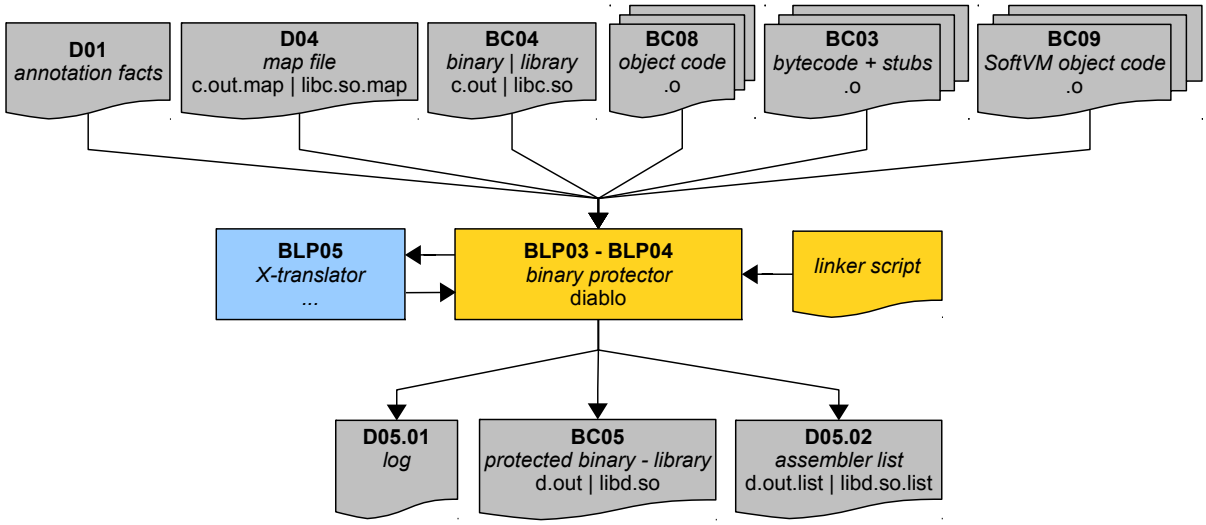

Figure 11 - Extra fix-up step in the ACTC

In addition to the discussed future work on the operation and capabilities of the SoftVM and its compilation support, we will also start researching the use of the SoftVM in support of other protection techniques, i.e., techniques other than client-side code splitting itself.

Furthermore, we will research how to integrate the custom SoftVMs of SFNT into the ACTC, i.e., SoftVMs of which the code (and hence the supported bytecode instruction set) is not fixed a priori, but dependent on the chunks that need to be interpreted in the code to protect at hand.

Section 9 List of Abbreviations

ACTC	ASPIRE Compiler Tool Chain
ADSS	ASPIRE Decision Support System
API	Application Program Interface
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
BCxx	Binary code document nr. xx
BLCxx	Binary-level configuration file nr. xx
BLPxx	Binary-level software processing step nr. xx
CFG	Control Flow Graph
DWARF	Debugging With Attributed Record Formats
Dxx	Datum produced or used by the ASPIRE ACTC identified with nr. xx
Dx.y	ASPIRE deliverable # y in work package x, y is a two digit number
GUI	Graphical User Interface
IP	Intellectual Property
IR	Intermediate Representation
JSON	JavaScript Object Notation
LLVM	Low Level Virtual Machine
RTD	Research and Technology Development
SB	(ASPIRE) Steering Board
SSA	Static single assignment
SVN	Subversion
QAP	Quality Assurance Plan
URL	Uniform Resource Locator
VM	Virtual Machine

Bibliography

[Anc06] Anckaert, Bertrand, Mariusz Jakubowski, and Ramarathnam Venkatesan. "Proteus: virtualization for diversified tamper-resistance.". In *Proceedings of the ACM workshop on Digital rights management (DRM 2006)*, ACM, 2006.

[Gho12] Ghosh, Sudeep, Jason Hiser, and Jack W. Davidson. “Replacement attacks against VM-protected applications.” ACM SIGPLAN Notices. Vol. 47. No. 7. ACM, 2012.

[Gho13] Sudeep Ghosh, Jason Hiser, and Jack W. Davidson. 2013. “Software protection for dynamically-generated code”. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW 2013)*. ACM, New York, NY, USA, Article 1 , 12 pages. DOI=10.1145/2430553.2430554

[Hu06] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill “Secure and practical defense against code-injection attacks using software dynamic translation”. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*. ACM, New, 2006.

[Rab13] Jason Raber, “Virtual Deobfuscator – A DARPA Cyber Fast Track Funded Effort”, Black Hat USA 2013.

[Sha09] Monirul Sharif , Andrea Lanzi , Jonathon Giffin , Wenke Lee, “Rotalumé: A tool for automatically reverse engineering of malware emulators.” In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.