Next generation debuggers for reverse engineering

Julien Vanegue[1], Thomas Garnier[2], Julio Auto[3], Sebastien Roy[4], Rafal Lesniak[5], and Rafael Villordo[6]

[1] Master Parisien de Recherche en Informatique
vanegue@ens.fr

[2] Ecole Superieure d'Informatique
thomas.garnier@supinfo.com

[3] Federal University of Pernambuco
jam@cin.ufpe.br

[4] Devhell Labs.
ceb@coldev.org

[5] Leibniz University of Hannover
rafal.lesniak@stud.uni-hannover.de

[6] Immunity Incorporation
rvillordo@immunityinc.com

Abstract. Classical debuggers make use of an interface provided by the operating system in order to access the memory of programs while they execute. As this model is dominating in the industry and the community, we show that our novel embedded architecture is more adapted when debuggee systems are hostile and protected at the operating system level. This alternative modelization is also more performant as the debugger executes from inside the debuggee program and can read the memory of the host process directly. We give detailed information about how to keep memory unintrusiveness using a new technique called allocation proxying. We reveal how we developed the organization of our multiarchitecture framework and its multiple modules so that they allow for graph-based binary code analysis, ad-hoc typing, compositional fingerprinting, program instrumentation, real-time tracing, multithread debugging and general hooking of systems. We reveal the reflective essence of our framework by embedding its internal structures in our own reverse engineering language, thus recalling concepts of aspect oriented programming.

1 Introduction

As general purpose debuggers do a very good job in debugging our development projects, it is difficult to understand how to improve such environments for programming purposes. Some debuggers [2] are even so portable that we clearly don't catch at first sight what is to be improved in the concept itself of being a debugger, as the software seems so adaptable.

1.1 Limitations of classical debuggers

The first point we should notice about debuggers is the shape of their models, which is adapted for development but not for reverse engineering. Especially, both forensic analysis and advanced machine code analysis are required for the analysis of malwares, protected software, and the control of systems in hostile configurations.

Encrypted binaries, rootkits and malwares are often coming without section table information, without symbols, and are compiled statically so that the program cannot be debugged through the interactions it has with other component libraries. Well often they are encrypted, so it is necessary to retrieve their memory image while being executed instead of trying to perform a static analysis on the binary file.

Constrained systems with hostile configurations can be realized using ACL systems and non-execution protections such as PaX and grsecurity [3], and special linking in position independant executables in order to make the address space layout randomization easier to the protected system. Other variants of this protection is provided in OpenBSD and involve the same limitations while trying to use a classic debugging tool. Sometimes the system even comes with no debugging interface (like in embedded systems without usable JTAG), or the debugging interface may be simply disabled (as when a production system has the ptrace syscall disabled).

Additionally, the use of the classic debugging API usually involves heavy communications between the debugger and the debuggee process. While this is not a problem when doing development debugging, more elaborated analysis techniques such as tracing or fuzzy testing requires more fluidity, as existing tools [4] [5] really lack a more real-time answer.

Finally, classic debuggers make the assumption that source code is available most of the time and they don't take advantage of internal binary format information. The most basic reverse engineering is made harsh on raw assembly code without code analysis techniques, fingerprinting primitives, and the lack of a language adapted to the discovery of information in raw disk or memory dumps.

1.2 Advantages of our framework

Our debugger is made for reverse engineers on the ELF format. While the global architecture does not implies a scope limited to one binary format, we have focused on the standard used on almost all Operating Systems (both free and commercial), except Microsoft Windows and the OsX from Apple. Our choice of the ELF format has been historically to fill the gap between the analysis software already available on those commercial OS's, and the poor playground

of the UNIX world, after the remark that an hegemony of GNU tools for those OS's would not adapt to reverse engineering.

The ELF shell project [1] started 6 years ago and implemented an interactive and scriptable machine for manipulation of on-disk ELF binaries. After some first insights in this binary format, we implemented novel techniques for program analysis only using on-disk modifications [7]. At this time, malwares for UNIX were quite primitive and protections on top of ELF files were almost absent, despite the release of burneye [8] binary encryptor, following the way of the UPX packer [9]. Our platform was already compatible with PaX systems for multiple architectures and we succesfully instrumented a wide amount of binary programs for auditing and hooking purposes.

After experiments in the real world, we concluded that control over analyzed binaries was insuficient, and adding runtime capable analysis, among other additional improvements such as different types of control flow redirections, and partial relinking of missing resources on both dynamic and static binaries [10] had become necessary. We managed to reuse our whole API using an inspired abstraction of data accesses for selecting source and destination buffers, depending on the choice of on-disk or in-memory requests. We also added at that time the first support of major techniques for other architectures such as ALPHA or MIPS, and improved the scripting langage to make it nearer a real reverse engineering oriented interpreter.

Today, we bring one more layer of techniques for more advanced handling of the mentioned problematic systems. Our framework is more intuitive to use in the everyday life of reverse engineers and forensic analysts on UNIX platforms. Our internal representations have been formalized to a type system adapted for the inference of information, using a lazy way to abstract and concretize data object types as we need to manipulate them, that make the interpreted language more flexible to the addition of features of interactions. We standardize our reconstructed information into a debugging format which can also benefit from the information of other debugging formats like DWARF or STABS, if those are available in analyzed binaries by any chance. The debugger has become compatible with multithreaded programs and also keep memory unintrusiveness by proxying all allocations and disallocations happening in the debuggee. Thus, we provide a real world debugging environment for hostile systems which does not suffer from performance penalties due to the use of debug interfaces provided by the Operating System.

2 Contributions

The ERESI framework [6] brings a new environment for reverse engineering on UNIX operating systems. In this section we will introduce it with a high-level

perspective. In the next parts, we will enter more and more in details about each component, starting with in-depth explanations of the used programming techniques that made it possible.

2.1 A modular framework for reverse engineering

The organization of the project is as follow :

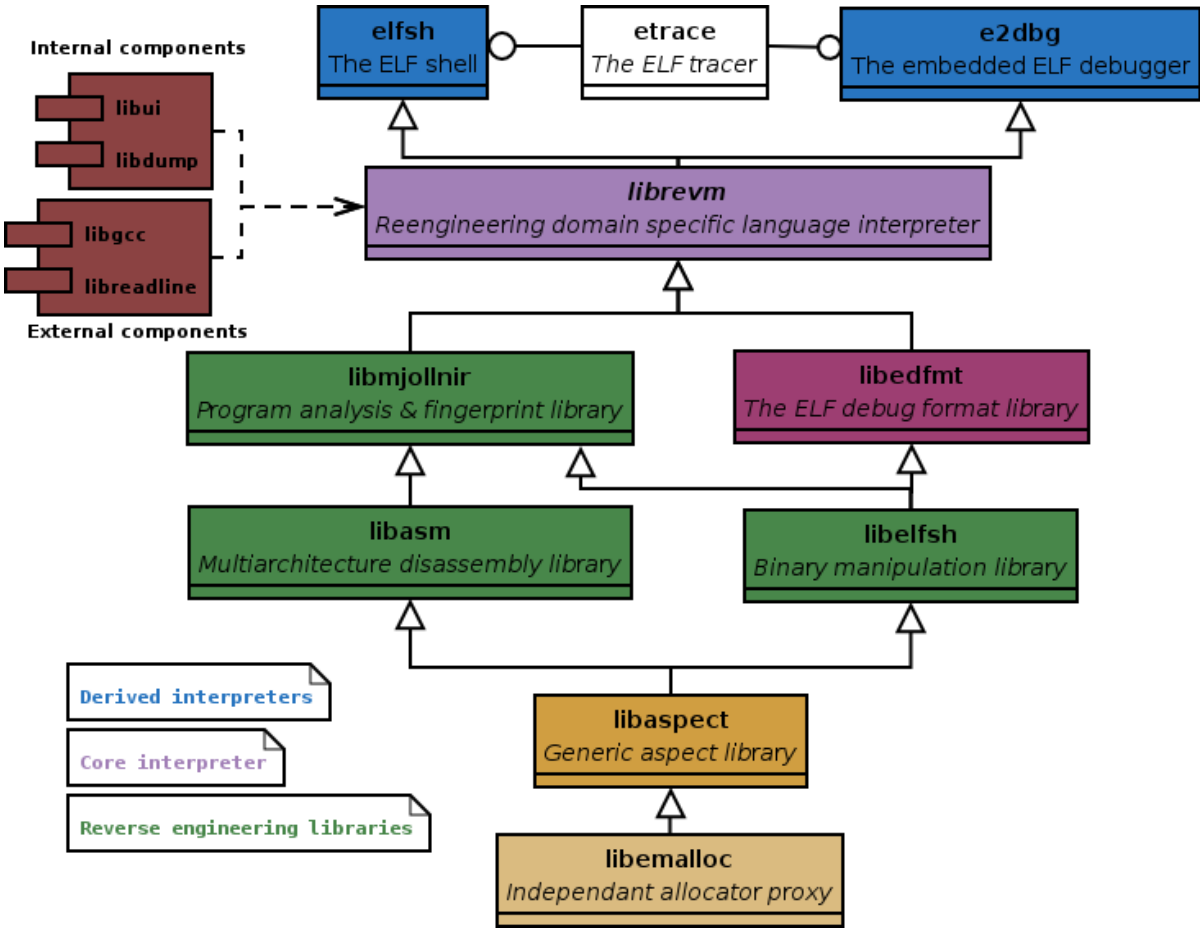

Fig. 1. The framework model

Let's understand this better, starting from the highest level components to the lowest level components :

- e2dbg stands for the Embedded ELF debugger. It can hook processes without the need of OS-level debugging API.

- etrace stands for the Embedded ELF tracer. It can trace processes at the normal execution frequency.

- elfsh stands for the ELF shell. This is the ondisk analysis tool of the framework.

- librevm is the template ERESI interpreter. It provides a small virtual machine for the ERESI language, the first reverse engineering language with types and reflection.

- libmjollnir is the fingerprinting and analysis library. It handles the construction of graphs using generic containers data structures.

- libedfmt is the library that deal with the ERESI debug format. It can convert stabs and dwarf formats to the eresi format, and make the framework aware of program types as indicated by debug information.

- libelfsh is the binary manipulation library for the ELF format. It can run both ondisk and embedded in the debugger using a unified interface. Libelfsh is currently the biggest and oldest component of the framework.

- libasm is the typed disassembly library. It currently supports entirely Intel and Sparc architectures. It provides a vector that allow for overloading its features on demand (see next point).

- libaspect handles the vectors and hash tables which allow for reflection of the whole framework. It also provide the type system for the ERESI language. The idea to implement the type system in such a low-level component make it possible for handling types in an unified way in both the analyzed program and all components of the framework itself.

- libemalloc is the allocation proxying library. It allows the embedded debugger and the legit program to keep a separate memory pool for dynamic memory allocation, thus providing heap memory unintrusiveness, making the debugger useful for the analysis of heap-contained structure, for instance in the development of heap exploits.

As you can notice, the vector data structure is central in our framework. Part 6 of this article is dedicated to the explanation of it. Before entering such

level of details, we will sum up the various contributions that make the ERESI framework a unique environment for efficient analysis of hardened programs.

2.2 Effective analysis

The first contribution of our framework is the ability to debug and trace programs efficiently. The usual debugging framework suffers from a lack of integration with the debuggee program. Because the normal architecture of a debugger is to be a separate component, there is the need of context switching each time the debugger wants to access variables of the debuggee program. The worse case arises when the debugger is made scriptable. In that case, the number of context switching is proportional to the number of variable access made by the script. As this might not seem like a limitation, this make automated analysis much slower. In our case, the interpreter of the ERESI language in which the scripts are made is mapped directly in the debuggee program address space, so there is no need of any context switching during debugging. Thus, our framework is much more adapted for the automated runtime analysis of program. As an example, advanced fuzzy testing techniques that make use of feedback information [11] to optimize the choice of analyzed program paths is made a lot more efficient without the use of OS-level API such as ptrace.

2.3 Tracing and Debugging on hostile systems

Another consequence of not using OS-level API is the ability to trace (and debug under certain conditions) programs even if debug API is disabled. For instance, our tracer and our base debugger are not blocked because of grsecurity, when systems comes with the ptrace system call disabled. Because we inject the debugger in the debuggee process, we are also not restricted for reading and writing the memory content of such programs once the library is injected.

Because we use multiple techniques for data and code injection [7], we never have to write in unauthorized areas when we do static injections. Nevertheless, in some cases it happens that we may need to write in the code sections of the program while debugging, for instance when we want to install breakpoints by the special processor opcode, or by function hijacking. For this, we use a special technique that is not blocked by the PaX kernel patch, even when all options of it are enabled. The technique simply consists in remapping the memory area from userland instead of trying to change their rights. This is smarter than just disabling the mprotect PaX option for the desired binary (which is possible as well in case the binary is read-accessible and PaX is compiled in soft mode. As this is the general case, we do not restrict our analysis to this situation).

Obviously, this technique can be countered by explicitely refusing such remapping, for instance when creating a special grsecurity ACL configuration, but the experience shows that real-world systems often lack of such configuration, or do not enforce it for other reasons. In case this grsecurity option is enforced and the

binaries are not readable, other PaX related features, such as the handling of text relocations, could be used to allow write access to code zones in order to perform function hijacking or breakpoint installation, but this technique is not included in the current version of the debugger. However, we support all the relocation API that would make possible to specially craft relocation entries for performing such operation. If none of these techniques make it possible to hjack functions, we can still modify the content of function pointers (that includes the content of special ELF sections such as the Global Offset Table, or the heap management *morecore* function pointer on ptmalloc [12] implementation of Linux) so that we keep being capable to redirect library functions or indirectly called functions.

2.4 A unified debug format

A debugging format is an important source of information as it permits to retrieve a description of each element that composes a binary. There are many different formats to fulfil the same goal, this diversity is an obstacle for an useful implementation and most reverse engineers do not bother implementing the support for them. Libedfmt parses every formats to create a uniform representation. This uniform representation is created on 3 steps. First we analyze a specific format and create an interface that make possible to read it. Then we transform this format using the uniform API. During this transformation, we keep only important information that we can find in all debugging format. A part of this information can be extracted with analysis tools but a debugging format provides you all types and names for functions and variables. The last step is about cleaning allocated memory so we have enough space to fetch the rest of the information.

For the moment libedfmt supports stabs and dwarf2. This made us realize that a different parsing engine had to be done for each debugging format. Stabs manages types by identifiants without any reference to the position in the debug section. You have to keep in memory all elements to be able to parse the information. Dwarf2 contains more information and you cannot store all of it without wasting a big part of your memory. At least, it contains a clear reference system and you can find a dependence without parsing everything. Even in libedfmt, you cannot read stabs and dwarf2 the same way, and each transformation has to be implemented differently.

When a file is compiled, a format is chosen to store the debugging information. The final binary can use more than one format. That is why it is needed to support more than one debugging format backend. Our library parses everything available and creates an unique representation. Once this representation is made, we add every retrieved types directly into the ERESI type engine. We update a hash table for each type with the list of all variables typed as such. The combination of ERESI and libedfmt creates a powerful debugging and reversing environnement that allow for saving and retrieving the types information.

In the futur libedfmt will be able to parse more debugging formats and should be used in e2dbg to display information that we cannot rebuild without source code, such as current file lines of the debuggee program.

2.5 Ad-hoc types recovery

When performing reverse engineering of closed source software, a very useful information is the recovery of types for the analyzed program data structures are variables. While this can be acheived with the use of debug information (as explained in the previous section), we bring a very useful innovation that make the user capable to give complex types for memory zones of the program, without any debug or source code information. This is made possible using our builtin type system, which is sufficiently expressive to handle complex types, including (mutually) recursive structures, pointers or multidimensional arrays. We also bring the capability to define partial types, in which a part of the type is only defined by its size. This is very useful when doing incremental reverse engineering and recovering types as the manual binary code analysis is going.

This is a major contribution to reverse engineering, as no current framework is capable to perform this kind of analysis. The GNU debugger only allows for existing types (indicated by the debug format) to be manipulated and does not provide any support for partial types. The IDA framework allows to define data structure templates in order to perform incremental type recovery, but this feature is purely manual and is not naturally integrated in the IDA scripting language (which does not have the expressivity of our type system). Unlike IDA, we provide a special language binding that allow for creating types, which make the feature useful inside ERESI programs that perform analysis automatically. It is not clear how complex and extended is the ability of IDA to manipulate the structure templates automatically (it might be possible using the API of the project in C language, thus outside their scripting language) and how far this framework is capable to create complex types such as partial or recursive structures, including unbounded dimensions arrays of those. Our approach is much clearer and unified with our language, which make the feature useful even for reverse engineers without knowledge of the ERESI framework internals. This feature could be defined as the underlying language type system for the automated type-based decompilation as presented by Mycroft [13].

Finally, this feature leads fo a very promising runtime reflection of the analyzed program data structures. For instance, hooking allocation functions such as malloc and free make it possible to *inform* a type about a precise variable (given its address). In that particular case, we can simply define the heap chunk type in the ERESI language and inform the chunk type of each address that is returned by the malloc function, so that the list of chunks is accessible through the chunk type hash table of variables directly from the language, opening the

door to an advanced analysis of the heap evolution directly in the ERESI language. Obviously, ad-hoc types recovery is not limited to heap variables, but this example was chosen for pedagogical purposes as it is really easy to hook those functions and automatically know the exact list of legit heap chunks available in the analyzed program in a sound way, at any moment of the execution.

3 Programming techniques

3.1 Aspects weaving

Features of the framework are modularized in a way that their interface is accessible to the user. This allow to refine analysis in runtime, when features are implemented differently depending on user-definable criterions, or when feature needs runtime updates. Those concepts originally comes from the notion of aspect oriented programming [15] which was developped some years ago to fill the lack of flexibility of object oriented designs.

This is acheived by the libaspect component of the framework, which implements vectors and hash tables primitives. Those data structures are the same than those used for representing the language objects, so that all vectors and all hash tables inside the project can be modified by the user. This correspondance allows for a framework that is entirely capable of reflection and reification, as its internal structures are accessible from the programming language, that make the user capable of plugging modifications of the framework itself just as plugging modifications on the debuggee program.

We applied this modelization to various points in the project. Each vector is used to make the implementation of a particular feature to be modular, updatable, portable, and give it the capability to be traced, monitored, or improved, or adapted to a particular requirement.

As most of those features are quite obvious and simple, this organization allows for porting the framework on other architectures and OS just by registering alternative handlers for them. Obviously, retreiving the program counter, getting the next stack frame pointer, or enabling stepping is made in a different manner depending on the operating system and architecture. This implementation is a portable way to abstract those differences and this make our framework very attractive because minimal efforts are needed to make it work on an originally unsupported platform. More details about the vector data structure is given in the part describing the ERESI reverse engineering language.

A new feature that comes with the current version of the libasm is its vector-based architecture. This allow to overload the handling of disassembled instruction, which has a potentially wide amount of applications, such as dataflow analysis or opcode tracing. Let's detail a little bit more our advanced disassembling interface.

Vector name	Role	Dimensions	Discriminant criterions	Host module
REL	ET_REL injection	3	Architecture, Object type, OS	libelfsh
CFLOW	Control flow redirection	3	Architecture, Object type, OS	libelfsh
PLT	Original PLT redirection	3	Architecture, Object type, OS	libelfsh
ALTPLT	Alternative PLT redirection	3	Architecture, Object type, OS	libelfsh
ENCODEPLT	Encoding of regular PLT entries	3	Architecture, Object type, OS	libelfsh
ENCODEPLT	Encoding of first PLT entry	3	Architecture, Object type, OS	libelfsh
BREAK	Breakpoint installation	3	Architecture, Object type, OS	libelfsh
EXTPLT	External symbols relinking	3	Architecture, Object type, OS	libelfsh
ARGC	Function arguments counting	3	Architecture, Object type, OS	libelfsh
GETREGS	Retreive registers context	3	Architecture, Host type, OS	e2dbg
SETREGS	Modify registers context	3	Architecture, Host type, OS	e2dbg
GETPC	Retreive program counter	3	Architecture, Host type, OS	e2dbg
GETRET	Retreive return address	3	Architecture, Host type, OS	e2dbg
GETFP	Retreive frame pointer	3	Architecture, Host type, OS	e2dbg
NEXTFP	Follow next frame pointer	3	Architecture, Host type, OS	e2dbg
SETSTEP	Enable singlestep mode	3	Architecture, Host type, OS	e2dbg
RESETSTEP	Disable singlestep mode	3	Architecture, Host type, OS	e2dbg
ASM	Opcode hooking	2	Architecture, Instruction opcode	libasm

Table 1. Registered vectors in libaspect

3.2 Typed disassembling

Libasm is a binary disassembling library designed for multiple architectures. It's one of the lowest-level components and, as such, must do a variety of tasks to support all the components on top of it. Libasm currently supports SPARC V9 and Intel IA-32. Currently in progress, there is also efforts to port this library to support MIPS code.

Besides providing the basic disassembling functionalities, libasm implements extra features to better support the services provided by the upper layers that rely on it. One of these features is its type system. By labeling all the disassembled instructions with common types, libasm makes program analysis tasks easier.

It is important to notice that these types are not mutually exclusive, this way you can have an arithmetic instruction that is known to modify some processor flags. Furthermore, this type system is shared between all portings of libasm, so programs or other libraries lying on top of it can, for instance, detect forward control flow changes just by checking if the current instruction being analyzed is of one of the 3 types, without even caring about on what kind of machine code this analysis is being done.

Another very interesting aspect that libasm features is the fact that every opcode handling function written is stored inside a vector of the kind provided by libaspect. When disassembling an instruction, libasm retrieves the correct

Type	Description
IMPBRANCH	Imperative branch (jump)
CONDBRANCH	Conditional branch
CALLPROC	Call to a procedure
RETPROC	Return from a procedure
ARITH	Arithmetic or logic operations
LOAD	Memory data load
STORE	Memory data store
ARCH	Architecture-dependent instruction
FLAG	Flag-modifier instruction
INT	Interrupt or call-gate instruction
ASSIGN	Assignment instruction
TEST	Comparison or test instruction
NONE	Instruction that does not fit any of the above

Table 2. Libasm instructions types

handler by querying this vector on its 4 dimensions: 1 regarding the machine type and 3 about opcode information (including architecture-specific requirements, such as SPARC's secondary opcode). Storing the addresses to these functions in this vector brings to the user the advantages of being able to dump and modify vectors from inside ERESI's scripting language. So, in practice, the function that does the job of disassembling a given instruction can be replaced in runtime by other code of the user's choice, allowing for easy opcode tracing, among other applications.

3.3 Generic containers

Libmjollnir is currently the main ERESI component for code fingerprinting and analysis. Among its analysis capabilities there is the construction of control flow graphs, both at basic block and at function level. Besides having separate structures for representing both entities, libmjollnir also stores them in generic structures called containers. Containers have link information (input and output links), a pointer to the actual data encapsulated, and information about the type of the data object, so the data can be accordingly interpreted.

The use of containers abstracts type information, thus giving the possibility to write analysis routines that work at this higher level of abstraction, walking through the graph of containers. Currently we only store blocks and functions inside containers, what suffices the needs of control flow analysis for these entities. In the future, we may store data nodes inside containers too, in order to perform data flow analysis. Finally, there is also the idea of having containers of containers, providing a "zoomed out" view of other graphs, eg. the linking between modules as a more abstract view of the linking between functions.

3.4 Allocation proxying

As the core of our framework of analysis runs in the same process than the analyzed process, it is very important to separate the memory used by the reverse engineering framework from the memory used by the legit debuggee process. Obviously, this matters only when we are debugging programs, and not when performing static analysis, as static analysis does not execute the program it does not need to care about memory unintrusiveness. Unintrusive debugging is based on the fact that the heap of the debugger and the heap of the debuggee must stay separated. This remark also holds for the debugger and debuggee's stacks, as we do not want to mess the debuggee stack when we debug our programs.

The stack unintrusiveness is realized using a modelization choice, as we make the debugger to execute in a separate thread, so it also inherits from a separate stack whoose base pointer is swapped when each thread is taking control (as provided by POSIX standard thread libraries of the various UNIX operating systems). The heap separation is more subtle, as the memory allocation is done entirely in runtime, unlike the stack allocations which are partially realized at compilation time (so we just have to take care about stack-related registers in runtime).

Our technique was named after the Syscall Proxying [14] idea which is mostly useful for the writing of vulnerability exploits in order to simulate remote execution when some particular required system calls are not available on the target machine, making it possible to execute entire binaries (such as a UNIX shell) but only executing some particular syscalls (like file systems accesses) on the target machine. However, heap separation cannot be implemented simply by allocation function proxying, since the returned values of those functions are memory pointers that are accessed for reading and writing in the subsequent code of the debugger. Using an external allocator would also require to proxy the accesses of all memory zones allocated in such way.

Our choice was to isolate a dedicated portable heap management system for the debugger, that allocate and destroy all its memory chunks in a single mapped memory region. We then garantee that the heap allocator is totally unintrusive, modulo the fact that a certain memory range will not be allocatable for the legit heap. The choice of the base address for this zone should adapted from OS to OS and from architecture to architecture, or even from debugged program to debugged program. A judicious default choice is the one of a very low virtual address (in the very first pages of the address spaces) that is rarely used, unless special modification of the OS or behavior of the debuggee program implies so.

The challenge that represented allocation proxying is more than the choice of the alternative heap base address. Indeed, as the analysis module of the framework is mapped inside the debuggee process itself, it can shows the need of a

dynamic allocation before the minimal debugger environment is set up. This case can happens in 2 situations : when the debugger thread is not yet created (as we rely on the POSIX thread library of each operating system), or when the debugger thread is created (so it already installed its own allocation handlers since the debugger is mapped in first in the program using the LD_LIBRARY_PATH enviroment variable.

We noted that it is not a definitive choice to use the LD_LIBRARY_PATH variable, as it implies that the allocator proxy is mapped in first in the program (so its symbol have the strongest priority since the object is located in first in the linkmap linked list of mapped objects). If we want to make the debugger to work using an ad-hoc library injection, we would have to redirect allocation primitive functions in the process before starting to debug the program, but this feature is still to be implemented, as function address lookups and hijacking still have to be realized without using any allocations functions (even at initial conditions).

Once the proxy allocator takes control in the proxyfied {m,c,re}alloc and free functions, we can face two situations. As all dependences of the program are already loaded when the first debuggee calls to malloc happens, the debugger code and data are already mapped in both of these situations. The first initial situation is when the debugger thread has not been created yet. In that case, the debugger memory mapping i already done (so we can use allocation-free functions inside the debugger), but the initialization of the debugger structures (that requires allocations) did not happen yet. Our solution is simply to keep a variable initialized to 0 to hold the thread id of the debugger. In case this variable is 0 or equals to the thread idea of the debugger, we call the separate heap implementation. In the other case, we call the legit allocations functions for the debuggee program to allocate in its legit heap.

Is the technique acheived ? Not yet. Indeed, we still had to make sure that we were able to resolve allocations functions without using dynamic memory allocations. This is realizable using a lookup of the linkmap linked list content, whoose first element is pointed by the second entry of the Global Offset Table section. As each element of the linked list contains a cache of the exported symbol table (the ELF .dynsym and .dynstr sections), we can resolve symbols and guess base address of objects without any external allocations. The cost of this technique is a small function that has to lookup the Global Offset Table address statically in the ondisk file using data statically allocated, in order to know where the linkmap base address stands (as specified by the ELF reference).

It is to be determined if that technique is appliable to other binary format than ELF (for instance in the PE binary format), but we believe that proxy allocation side effects would still keep a minimal place independently of the binary format specifications, so that the technique is potentially portable to other frameworks.

4 Applications

Thanks to the modular approach, we can derive multiple tools for debugging, static and runtime analysis of programs. In that section, we present the architecture of our embedded debugger, the algorithm of our embedded tracer, and an example of modular graph-based analysis taking advantages of our generic containers data structures.

4.1 The embedded debugger : e2dbg

The global architecture of the debugger is as follow :

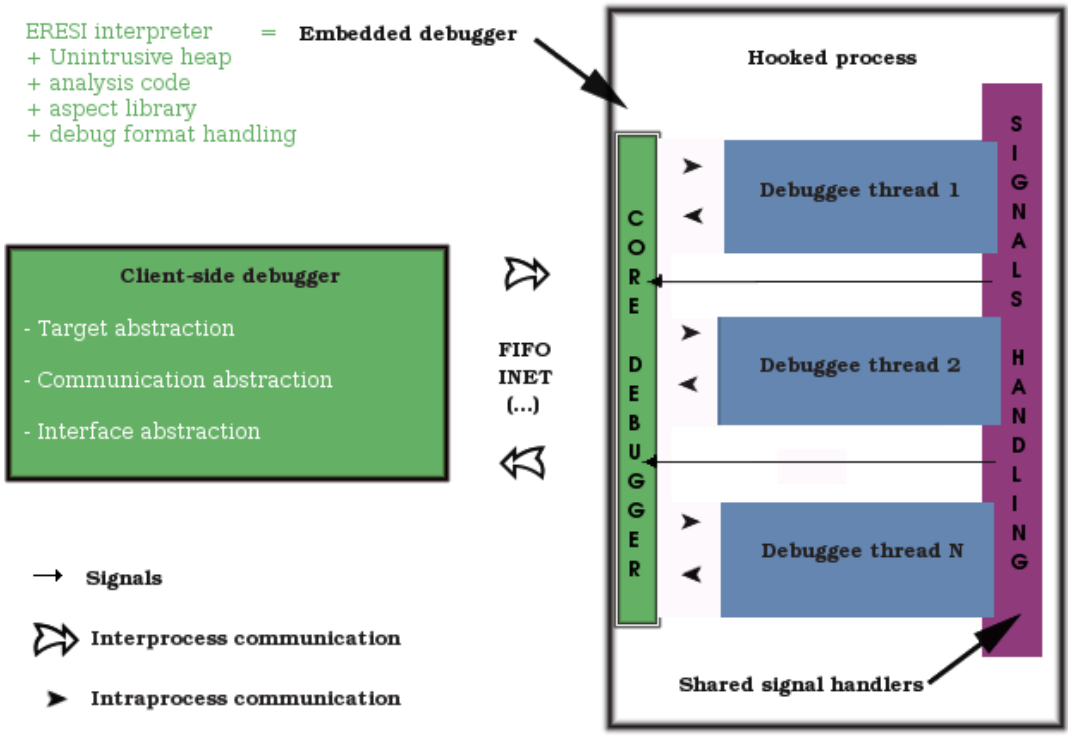

Fig. 2. The embedded debugger

The debugger is made of two parts. First, the client-side that is responsible for communicating the user requests to the embedded debugger. In the classical

scheme, the embedded part takes control before the main function is executed, so that the user can perform analysis as soon as requested. The embedded debugger also handles all important signals (including SIGTRAP, SIGSEGV ..) so that it catches debugging events such as breakpoints, and debuggee program crashes.

The first innovation that comes with this tool is about its architecture. The fact that the debugger is made bipartite allow for a very flexible handling of targets. For now, only the userland debugging is available, but in the future we can imagine any kind of targets (embedded systems, kernel-level code..) being supported by this framework without changing anything in the modelisation. Of course, the way to handle breakpoints, stepping, or other debugging events varies from target to target, but the vector system of the ERESI framework allows for a very fine grained implementation of features, so that most of the code is reusable without any change.

The embedded debugger was not created with this modelisation since the beginning [10] but is an evolution after a lot of experiments with the debugging of multithreads programs in an embedded context. We have been testing various ways to keep a high performance while hooking programs, staying portable and stealth for OS-level protections, and reusing the available code as much as possible since our early development [7] about on-disk analysis of programs.

Our successive experiments looked like this :

1. We first tried to map the complete ELF code as a new dependence library of the program beeing debugged. This worked well, as we just had to create an intermediate function that was responsible for accessing the data of the manipulated object, but keeping the same manipulation code for on-disk and runtime structures modifications. This was the core of our powerful embedded debugger idea. However, this solution was not taking care of untrusiveness in the host process memory as we used the same heap than the debuggee, which made the debugger useless for heap-related debugging, such as heap overflow exploits. Additionally, this first implementation was monolithic and not capable to handle multithreads programs.

2. As a second generation embedded debugger, we first implemented the allocation proxying technique. As already explained, this technique allow for two different heap allocators to reside in the host process memory, which made the debugger completely heap-unintrusive, and improved greatly its usefulness for real-life needs. We also changed the debugger system for running in a separate thread. This had the advantage to naturally handle multithreads program and the debugger execution context using a unified API, by hooking signals and then selecting what to do depending on which thread was breaking the execution. This also had the advantage of beeing stack-

unintrusive, as each thread (including the debugger) had a separate stack. However, this was still not completely bullet-proof and this for two reasons. On the one hand, we introduced intrusiveness in the thread manager table, as we had to switch to a multithread process even if the original process was monothread. On the other hand, we needed a signal-based communication between threads that appeared to be way too complex to manage, especially when we implemented thread context modification feature. Indeed, when a thread is breaking, only its context is available, and the only way to retreive the other threads's context is to send each of them a signal (handled using the sigaction system call) in order to stop them and retreive their register values at the same time. Additionally, a signal had to be delivered to the debugger thread to wake it up, so that it could start the analysis.

3. In order to simplify the threads management, we decided to do a compromise on the architecture of the debugger. We now keep the whole threads API so that we continue to be multithreads capable. However, the debugger is not running in a separate thread anymore but on top of each thread. This reduces the number of signals to be delivered as the debugger does not have to be woke up anymore. This also reduce the number of necessary mutexes from four to one (the remaining one beeing around the breakpoint/step handler, so that we dont try to handle simultaneous breakpoints of threads, but we queue them.). This made the thread-table intrusiveness to disapear. However we restarted to be stack-intrusive. Even if reusing the current thread stack is less critical than reusing the heap (as the stack pointer comes back to the legit value after the debugger is finished), this is still a minor issue to resolve. A potential solution is to use the sigaltstack system call in combination with the sigaction system call, which allow to specify an alternative stack context when a signal is received. However, the current debugger version (0.77) at the time of writing this article is still not implementing the alternative stack technique, but this should be added any time soon.

After describing the techniques we use for the debugger, we will focuss on another component of the ERESI framework : the embedded tracer.

4.2 The embedded tracer : etrace

Etrace is an embedded tracer which was built for tracing internal and external calls. Most tracers do not trace internal calls because they rely on a statically stored function prototypes list. Despite the fact that it provides a correct prototype on those functions, you cannot deal with unknown functions. Etrace is a tracer built to deal with every functions. It means you do not have to create a function prototypes database.

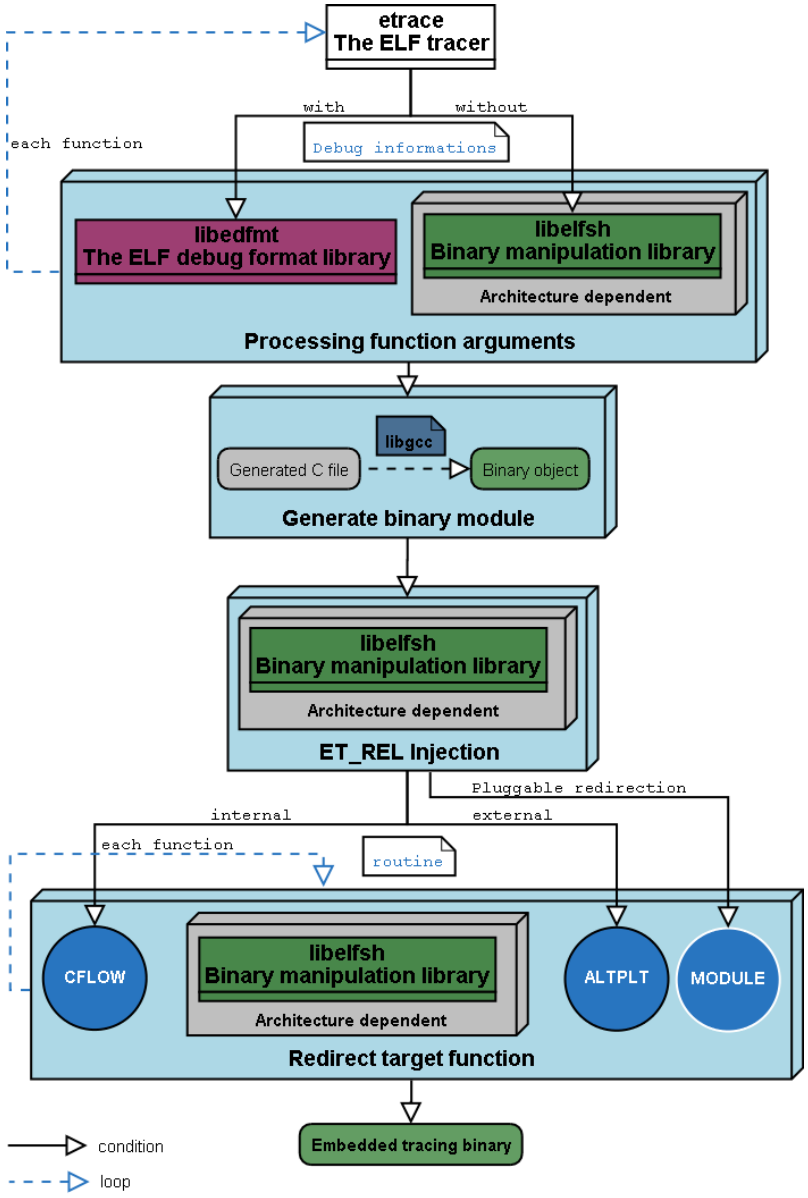

Fig. 3. Tracing algorithm

Our tracing technology is dynamic and supports multiple architectures. We take advantage of the debugging format information of libedfmt to retrieve ex-

act function prototypes. In case the debugging information is not available, an architecture dependent analysis permits to retrieve a deduced prototype. We present our tracer as embedded because we redirect traced functions directly in the target binary. Then we create a new file that includes and reports tracing information. Etrace does not use any kernel debugging interface and only relies on analysis of parameters. This is done using an architecture dependent function that is hooked using a vector.

```
1  Processing function arguments:
2      foreach Traced Functions:
3          if ( Internal function )
4              if ( Debugging information is present )
5                  • Retrieve prototype information
6              else
7                  • Architecture dependent argument counting
8              endif
9          else
10             if ( Dependence library is available )
11                 if ( Debugging information is present )
12                     • Retrieve prototype information
13                 else
14                     • Architecture dependent argument counting
15                 endif
16             else
17                 • Architecture dependent argument counting
18             endif
19         endif
20         • Add a proxy function into a generated C file
21     endforeach
22
23 Architecture dependent argument counting:
24     • Set argument count to 0
25     • Search a call to this function
26         if ( We found a call )
27             • Start a backward argument counting
28         if ( Argument count is 0 )
29             • Start a forward argument counting
30     • Return argument information
31
32 Generate binary module:
33     • Compile the generated C file as a module
34     • Inject this module using ET_REL injection technique
35
36 Redirect target function:
37     foreach Traced Functions:
38         • Redirect the function using CFLOW / ALTPLT technique
```

The previous algorithm shows how our tracer takes advantage of the ELFsh framework features. The CFLOW and ALTPLT techniques were already described in [10]. They allow for redirecting calls on our generated functions. Etrace reduces architecture dependences by generating and compiling a .c file which prints information on every redirected functions.

During function redirection, we check each argument and try to see if it is a pointer, a string or a value. When we try to read each argument as a pointer, we handle the SIGSEGV signal. If it is not a pointer, we can declare it as a value. Otherwise another check is performed which indicates if this pointer is a string. For a simple pointer we display 4 first bytes. The previous SIGSEGV signal handler is restored after our pointer test. Those information help us to create a correct prototype of the function. This runtime check does not rely on debugging format thus allowing us to deduced prototype information on any case.

An example of the output for 3 functions:

```
+ main()
BEFORE !
   + firstfunc(int num: 0x3, *char value: *0x8048648 "this is the text")
arguments: num = 3 / value = this is the text (0x8048648)
      + testcrypt()
         + crypt(*0x804860d "password", *0x8048608 "salt")
         - crypt = b7f26120
      - testcrypt = b7f26120
   - firstfunc = 1
OK !
AFTER
- main = 0
```

Additionally, our tracer provides a way to group functions. That allows for the user to decide which pool of functions he wants to trace. For now, this support is only static, as the tracer is not entirely interfaced with the debugger at the moment of writing this article. However, the extension of this feature will make possible to decide in runtime which function to continue tracing or which function to remove from the inspection list as the analysis is going.

5 ERESI : Towards a reverse engineering language

In this part of the article, we will go one step more abstract, by describing the internals of the reverse engineering language of our framework. First, we detail how the variables of ERESI programs change their type when it is necessary (that could be labelled as "weak typing") and how types are enforced when we cannot do in another way (to avoid unrequested behavior to happen). At the same time, we introduce our major abstract internal data structures : the reflective vector.

A reflective vector, informally, is an container object whoose elements are all of the same type. This type can be, among others, a function pointer, or the type of a set of elements, represented in practice by a hash table. Depending on the type of the vector elements, the vector has a different use. We call it a reflective vector because all existing vectors are accessible from the ERESI language, so that we can change as we want the content of the vector as the analysis is going. As all big features of the framework are registered in vectors, that make it possible to update or swap in runtime the framework behavior, in a single line of ERESI code.

5.1 The ERESI type system

The language of the framework was made primary with reflection and modularity in mind. The concept of a reverse engineering type system follows the idea that manual or automatic type information can be adapted to the required format of data. Thus, each data object in the language does not have a unique type as it evolves as you use the variable. For instance, writing an array of integers in memory would first need to convert this array as a raw data buffer, then write its contents at the desired address. Inversly, extracting data from raw sections must allow for extraction in any desired format.

The language base types can be ordered as a semi-lattice, that is adapted for allowing or refusing type conversion from a given type to another.

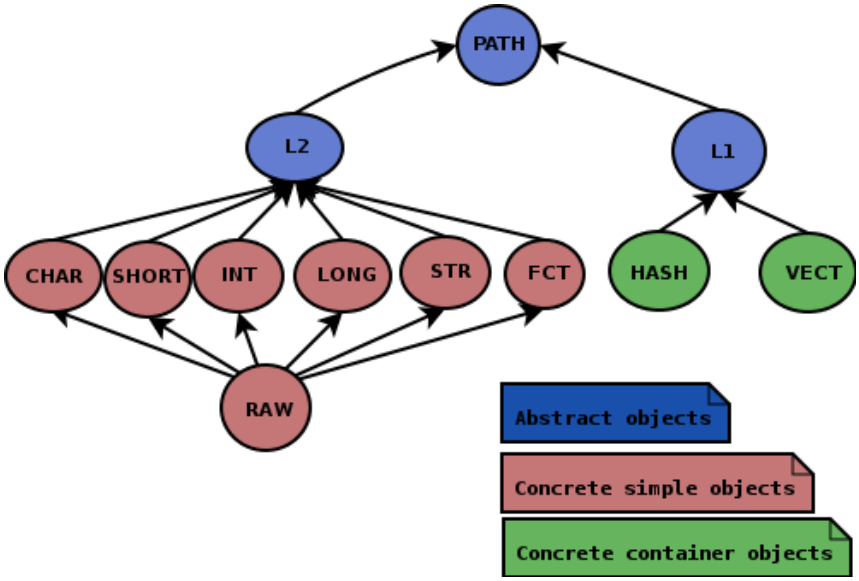

Fig. 4. The extended reverse engineering type system

As the diagram shows, we separate simple types (including uniform function types) and container types such as hash tables and vectors. Those last types are combining aspect oriented advantages with the flexibility requirement of reverse engineering, as the framework objects are contained in vectors and hash tables, so it is possible to change pieces of the framework itself in runtime and overload it on demand. For instance, adding new fingerprinting discriminants or additional binary analysis can be performed by modifying the vectors of libmjollnir or libasm. We have not detailed the fingerprinting capabilities of libmjollnir in that article as we plan to dedicate a complete paper about it in the future.

Formally, we can separate simple, containers, and generic types :

- Simple types are denoted by ψ : { char, short, int, long, str, func } $\subset \Psi$
- Containers types are denoted by γ : { vect, hash } $\subset \Gamma$
- Abstract types are denoted by σ : { L1, L2, path } $\subset \Sigma$

For convenience of notation, we also define $\Lambda = \Psi \cup \Gamma$, as some of our functions acts on both kinds of objects. We define 4 simple operations on types that allow to manipulate data easier in the framework, both manually or automatically :

$$lookup : \Gamma \times \Psi^n \longmapsto \Sigma$$
$$reify : \Lambda \longmapsto \Sigma$$
$$reflect : \Sigma \longmapsto \Lambda$$
$$convert : \Psi \longmapsto \Psi$$

The operation of lookup corresponds to the action of retrieving a simple object from a container, by passing the vector name, and all the n coordinates inside the vectors of dimension n. Reification is the action of making an object of the debugger reverse engineering framework itself available in the programming language. Reflection is the action of concretizing an object from the language into the framework. Finally, the semantics of convert is quite intuitive as it just converts between concrete types.

5.2 Using vectors

In order to understand better this structure, we give additional hints for the use of such system :

1. Reflection and reification of container objects make programming very practical as internal framework structures and routine can be manipulated and updated from the language.

2. The lookup and reflect operations combines as an algebraic (set) separation when $reflect(lookup(Vect \times \Psi^n)) \longmapsto Hash$. Indeed, vector indexations acts as discriminating parameters between sets of objects contained in the vector of hash tables.

3. The lookup and reflect operations combines as a control flow aspect when $reflect(lookup(Vect \times \Psi^n)) \longmapsto Fct$. Features of the framework can be hooked in runtime using such objects in the language.

4. The convert operation only applies to simple types and does not require to enter the reflection and reification cycle. Obvious conversions are done by a $convert : \psi \longmapsto Raw$ or $convert : Raw \longmapsto \psi$. We distinguish particulary some interresting conversions. $convert : Func \longmapsto Raw$ as it allows for converting a function object to a raw data object being the code of the function.

As an example of use of those operations, we have interest for sets separation when it comes to do compositional fingerprinting. For instance, the vector index for each dimension can corresponds to the graph distance from the current object (block, function or instruction) to a certain code pattern in the analyzed program. Additional dimensions for a fingerprinting vector can then discriminate between objects that were sharing a common fingerprint criterion on previous vector dimensions (and then were contained in the same set before introducing those additional criterion). We called this compositional fingerprinting as each dimension of the vector can represent a different way to separate elements for fingerprinting purpose.

As another example, control aspects guarantee a fine-grained modularity of our analysis framework once they are organized using vectors, as explained in previous parts of the article.

6 Related work

In this section, we briefly discuss the work related to modular debugging. Several of our features are available in other debugging framework, such as remote debugging, language based debugging, or graph based debugging. However, none of the existing frameworks manage to provide a unified interface for all of those features. Additionally, the innovation of our work resides not only in new features available nowhere else (such as reflection on binary programs or on-the-fly typing) but also on the way they are implemented in an embedded framework, without any debug API at the OS level.

6.1 Remote debugging with gdb

The GNU debugger has a useful feature often called gdb stubs. Stubs are piece of backend code that allow for reusing the gdb code base for the debugging of

any kind of systems. It comes with a protocol for communicating between the debugging client and server, so called the gdb remote protocol. A Gdb stubs has to implement a minimal set of functions in order to be operational. Those functions essentially include the interface for reading and writing registers, reading and writing memory, stepping and continuing the process. It also include functions for more elaborated features such as memory search, setting of memory variables, console output, or last signal information.

As this feature makes gdb attractive for porting on any kind of systems, the stub has to be programmed in C language and linked with the server side debugger. The vector system of ERESI provide a similar feature than Gdb stubs, but additionally the vector can be overloaded using code in ERESI language itself, so the porting of our debugger to additional targets is made easier than gdb stubs. However, there is a wide amount of available gdb stubs, whereas our framework is fresh and we still lack of experience on porting it to exotic systems to conclude anything yet.

6.2 Reflective debuggers

The work on reflective debuggers [17] has been very academical until now, and we are not aware of any previous reflective debugger that is capable to debug real world binary programs, with or without the need of the source code. Nevertheless, the only related work presenting a minimalistic reflective debugger was made on an interpreted language. Their approach has in common with our work that reflection does not rely on source language construct, unlike approach used in the Java language extension such as AspectJ, but on the runtime environment that handle the execution of interpeted programs. A consequence of providing a proof of concept on an interpreted language is their ability to select the granularity of reflection in a very fine way, so they can decide wether to provide reflection on control structures or data variable on demand.

The reflection in the ERESI language can only handle the control structure of binary program until now. We can think about handling data-level reflection as well but it is not obvious how to acheive this on hardened systems without relocation information in the binary files (as reflection on data variables would be done using inline patching of the code that access those variables, which would result in changing the location of assembly instructions, which is not always possible without information on how to relocate the code, or with a perfect data flow analysis engine for binary code. For instance, instructions performing relative memory accesses cannot be executed in another location unless they are reencoded to do so).

Hardened systems bring a bigger challenge to data-level reflection of binary code as it might not always possible (depending on the hardened configuration) to write on code instruction in runtime, as explained at the beginning of the paper about the debugging of hardened systems. In the hypothesis that the

reflective analysis benefits from the output of a perfect data flow analyzer, then we might be able to realize data-level reflection on hardened systems only using static analysis and without the need of relocation information. However, the existance of such perfect data flow analyzer for binary code is far from beeing confirmed, as various hard problems such as the aliasing of pointers are answered only in a conservative way.

6.3 Embedded debuggers

Among the few efforts to achieve an embedded debugging (also sometimes called in-process debugging) behavior similar to what we have presented, Microsoft's .NET Framework debugging API seems to be the most popular. This API, constituted by a subset of the System.Diagnostics namespace, provides some useful methods and properties to help a program debug itself. However, as far as we are concerned, the power of this API is very limited, restricting itself to some profiling functionalities and lacking the ability to change program state, therefore not being able to set breakpoints or stepping instructions. [16].

6.4 Python debuggers

Some new frameworks, such as IDA-python [22] or the PaiMei debugging suite [21] on the windows operating system, are based on a recent generation of programming language, which make the development faster. However, the use of a general purpose programming language still let the framework language more verbose than necessary. For example, a code iterating on the basic blocks of an analysis program can be done in around 20 lines of python code, whereas you need less than 5 lines in ERESI to realize the same operation.

There are multiple reasons for this. First, our language is dedicated to reverse engineering and its primitives directly include hash tables of existing blocks and functions, and handle regular expression in an attractive syntax. Additionally, those tools are based on the IDA code base written in C language, so it comes as an additional layer on top of an existing tool. On the opposite, our language is integrated directly in the framework and we can adapt its syntax for our own needs. The reflexivity of the framework data structures make possible to access such computed information without any additional interfacing.

Finally, we do not need any general purpose virtual machine for such a complete language as python or ruby, but we rely on the minimalistic vm of ERESI, which is able to scale for the analysis of large programs even when embedded in-process. We are not trying to argue that our language is better than python, but that it is more dedicated to the task of reverse engineering, and the language itself (and not only the framework features) can be adapted to our need as we get inspiration from other programming languages. For instance, features such as program transformation or type-based decompilation will be integrated in the future using 2 simple additional commands (match and transform) whereas

similar features in python would turn into developping a library and provide additional functions to the reverse engineer, which would lack the beauty and the intuitive aspect of the language.

6.5 Graph-based debuggers

Reverse engineering tools such as BinNavi [19] or IDA [18] provides a very attractive graph-based interface for the analysis of programs. In the first case, it comes with a certain latency as the interface code is written in Java. In the second case, the visual aspect is well suited for manual analysis. In both cases, the systematic approach to information visualization (certainly for commercial purposes) made the development of real code analysis features to be slowed down. For instance, none of those frameworks come with an innovative intermediate form other than control flow graphs, when in-depth analysis requires such capability, or at least requires the framework to bring facilities for builting your own intermediate form suited to your problematics (which is what we started to do by typing instructions in libasm). We also bring a graph visualization feature, but we provide it only on request to the user, using the graphviz [20] tool. Obviously our graph visualization is not as good as the mentioned tools but our internal graph structures are more complete than simple cross-references as provided by IDA. Our generic container system also brings the possibility to reuse our whole API for the graphing of any kind of element relations, which open the door to the display of data flow graphs or other kind of dependences relation.

7 Conclusion

We provide an alternative framework for debugging programs in a hostile environment with the perspective of reverse engineering. This novel model of analysis allows for high performance, unintrusive, multiarchitecture analysis of binary programs without needing the source code. Our implementation includes a disassembly engine, a binary manipulation library, a fingerprinting library, and the support for a new powerful debug format that keep compatibility with existing ones. We rely on a minimal aspect oriented interface for modularity and we propose a practical scripting language and its ad-hoc type system. Thanks to this fine-grained architecture, we manage to provide the base interpreter functionalities in a small and extensible core for rapid development of specialized interpreters. We gave three examples of such instances : the ELF shell for ondisk analysis, the Embedded ELF debugger for runtime analysis, and the ELF tracer that combine both facets.

References

1. The ELF shell crew, The ELF shell
$http://elfsh.asgardlabs.org/$

2. The GNU project, The GNU debugger
http://www.gnu.org/software/gdb/

3. The PaX team, Grsecurity project
http://pax.grsecurity.net

4. Dave Aitel, Fuzzy testing tools
http://www.immunitysec.com

5. Michal Zalewski, Fenris
http://lcamtuf.coredump.cx/fenris/

6. ERESI : The ERESI Reverse engineering software interface
http://eresi.asgardlabs.org

7. The cerberus ELF Interface, mayhem, Phrack Magazine issue 61
http://phrack.org/archives/61/p61 − 0x08_The_Cerberus_ELF_interface.txt

8. Scut, Burneye. Phrack Magazine issue 58
http://phrack.org/archives/58/p58 − 0x05

9. UPX team, The Ultimate Packer for Executables
http://upx.sourceforge.net

10. Embedded ELF Debugging, The ELF shell crew, Phrack Magazine issue 63
http://phrack.org/archives/63/p63 − 0x09_Embedded_Elf_Debugging.txt

11. AutoDafe : An Act of Software Torture
http://events.ccc.de/congress/2005/fahrplan/events/606.en.html

12. Wolfram Gloger's malloc homepage
http://www.malloc.de

13. Alan Mycroft, Type-based decompilation
European Symposium and Programming (ESOP99)

14. Maximiliano Caceres, Syscall Proxying : simulating remote execution
BlackHat 2002
http://www.coresecurity.com/files/files/13/BlackHat2002.pdf

15. Aspect Oriented Programming
First International Symposium on Generative and Component-Based Software Engineering

16. Mike Pellegrino, Improve Your Understanding of .NET Internals by Building a Debugger for Managed Code
http://msdn.microsoft.com/msdnmag/issues/02/11/CLRDebugging/

17. M Ancona, W Cazzola - Implementing the essence of reflection: a reflective run-time environment

Proceedings of the 2004 ACM symposium on Applied computing

18. Ilfak Guilfanov and the datarescue team - The Interactive Disassembler
http : //www.datarescue.com/idabase/

19. Halvar Flake and the sabre-security team - BinNavi
http : //www.sabre − security.com/products/BinNavi/

20. Graphviz - Graph Visualization Software
http : //www.graphviz.org/

21. Pedram Amini - PaiMei
http : //pedram.redhive.com/PaiMei/docs/

22. Gergely Erdelyi - IDAPython
http : //d − dome.net/idapython