Lightweight heuristics to retrieve parameter associations from binaries

Franck de Goër
LIG[*]
UGA[†]
France
degoerdf@imag.fr

Roland Groz
LIG
UGA
France
roland.groz@imag.fr

Laurent Mounier
Verimag
UGA
France
laurent.mounier@imag.fr

ABSTRACT

We present an approach to recover information on function signatures and data-flow relations from stripped binary code. Contrary to most approaches based either on static analysis or fine-grained dynamic instrumentation, we propose lightweight instrumentation and heuristics. Our goal is to get a fast and scalable pre-processing that could serve as a front-end to focus more detailed analysis of particular functions. We infer arity and parameter types, as well as a coupling relation (which we define). We are interested in particular in couples of functions with a data-flow relation, such as memory allocators. We trade-off accuracy for scalability and performance, but our experiments show that the results of the proposed heuristics can be quite accurate, even on a single random execution.

Keywords

Reverse-engineering, data-flow, dynamic type inference

1. INTRODUCTION

Understanding the content and behavior of a binary program is required in numerous situations. For instance, it may help for validation purposes (does a given binary code respect some specification), for reuse purposes (how to call and execute some undocumented pieces of code available only at binary level), and, more often, for security purposes (does a given binary contain some vulnerability, or does it behave as a well-known malware etc.). However, this is a difficult task because most of the information readily available at the source level is no longer explicit in the binary code.

As a consequence, several static and/or dynamic analysis techniques have been proposed in the last decade to help reverse engineers. These techniques are usually dedicated to retrieve specific information, such as function boundaries [2], function APIs [15, 3], source-level data types and data structures [1, 18, 9] and even function semantics [4]. As usual, each of these reverse engineering techniques have been designed as a trade-off between conflicting requirements, in particular:

- **quality:** how much information is retrieved with respect to the source code?
- **accuracy:** how precise is this information?
- **scalability:** is the proposed technique able to operate on large binaries?
- **generality:** does it rely on strong assumptions on the binary under investigation?

In this paper, we propose techniques whose purpose is to retrieve *data-flow relations* between pairs of functions inside a binary code. Such information gives useful indications regarding the semantics of these functions. In particular, it helps reverse engineers in building (partial) data-flow graphs of the binary code under investigation. This is often a tedious and manual task, although being a building block of many code auditing technique, for instance to check if tainted data may flow towards vulnerable functions. Our objective in this work is to address this problem by providing a lightweight pre-processing step, scalable enough to handle very large binary codes, and able to retrieve such *function coupling relations* from a reduced set of executions. As a by-product, our technique also computes abstract function prototypes, based on a limited (yet sufficient for our needs) set of types.

A typical application we have in mind is to detect hand-made memory allocation and liberation functions, which is a mandatory step when looking for specific vulnerabilities such as *use-after-free* [13]. This problem has been addressed in [4], where the authors propose some dynamic criteria to recognize memory allocation/liberation functions. An extra criterion (not proposed in [4]), is that there may exist a data-flow between a memory allocating and freeing function (and not in the reverse way). The approach we propose already allows to easily check such a criterion. Going further, our future plan is to use function coupling relations to infer behavioral abstract models of (parts of) a binary code using active machine learning techniques.

One of the original part of our work is that the technique we propose relies on a low-cost dynamic analysis, based on aggressive heuristics, and requiring only (ultra-)lightweight

[*]Laboratoire d'Informatique de Grenoble
[†]Université Grenoble Alpes

PPREW-5, December 08 2015, Los Angeles, CA, USA
ACM ISBN 978-1-4503-3642-0/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2843859.2843861

instrumentation code. As an example, one of these heuristics allows to track at runtime data propagation across read and write operations without requiring extra memory. Instead, we rely on numeric comparisons: if the same (address) value is returned by a function f and later used by a function g, and if such a situation occurs a sufficient number of times, then we could conclude that a data flow exists from f to g. This leads to a very efficient implementation, compared both with existing static and dynamic analysis techniques, making our solution highly scalable. At the same time, the precision we obtain is still very good: we infer always more than 90% correct function signatures on real binary codes (including a PDF reader, a web browser and a text editor), even from a single random execution. Furthermore, this technique is rather general, it can be applied on any stripped binary code produced by a C compiler (we only assume that function boundaries can be identified, and that there is no object-oriented feature in the code).

Our contributions can be summarized as follows:

- We propose a lightweight and scalable dynamic analysis technique, allowing to retrieve **inter-procedural data-flow relations** from a binary code, based on aggressive heuristics based on values and repetitions.
- This analysis also allows to infer **function signatures**, using a coarse-grained notion of type[1].
- Finally, a prototype tool, called SCAT[2], has been developed and evaluated on several real-life programs; the experimental results we provide show the trade-off we obtain in terms of **scalability** of the approach and **precision** of the results; in particular, even with a very poor code coverage, the function signatures we infer are almost always correct. This prototype, together with our experimental data, are available on-line [5].

In the rest of the paper we present (section 2) and define (section 3) the problem we address, then we describe the dynamic analysis we propose (section 4), discussing some of the choices we made, and we provide experimental results obtained (section 5). Related work is discussed in section 6 and some conclusion and directions for future work are given in section 7.

2. PROBLEM STATEMENT

2.1 Assumptions

2.1.1 White-box binaries

The binaries we deal with are compiled from source code, with no particular compilation option. We assume to have no access to the source code, but we can instrument the binaries and observe execution with dedicated tools. In this work, we use `Pin` [11], a dynamic code instrumentation framework developed by INTEL. Instrumentation code is defined as a *pintool*, which consists of a set of call-back functions triggered by execution events (for instance, each time a register is read).

[1]Considering a more expressive type system is not in the scope of this work

[2]for Strong Coupling, Arity and Type

2.1.2 Identification

To be able to retrieve information about functions, the first thing we need is to detect such functions in a binary. Although function retrieving can be a complicated task (see [2], [16]), we assume in our approach that we already have access to this knowledge. In our implementation, this problem is handled by `Pin`, as we use its API to instrument functions (`RTN_AddInstrumentationFunction`). The functions not recognized by `Pin` are therefore simply ignored.

2.1.3 Known calling-convention

To ensure that a function compiled with one compiler is able to call another function compiled with another compiler, these two functions must agree on the way the parameters are passed. That leads to well-defined calling conventions. There exist several conventions for each architecture, depending for instance on the Operating System. We assume that we know the calling convention being used by the binary under analysis. We also assume that this convention is unique. It means that binaries using multiple calling-conventions are out of the scope of this work.

2.2 Problem

Given a compiled binary, possibly stripped and with no further information (such as debugging information or source code), we aim to retrieve, under efficiency constraints, the following:

1. **Arity** - Find how many parameters each function takes. *This is required to call a function out of the normal execution of the binary, for instance to run active machine learning algorithms.*
2. **Type** - Determine the basic type of each parameter of functions. By basic type, we mean address, integer or float. *This step is both used to be able to call a function with relevant values and to infer function coupling.*
3. **Coupling** Try to infer data-flow relations between two functions of the kind: which output of which function is passed as a parameter to which other function. An example is the couple of functions `(malloc, free)`: the expected behavior of a binary that uses those two functions is that the output of `malloc` will be given as a parameter to `free`. More details on this are given in section 3.2.3.

2.3 Goals

The analysis we propose in this paper should be seen as a *front-end* analysis that aims to reveal some particular points of interest in a (large) binary. It is an upstream work before a deeper analysis which will focus on those points of interest. For instance, the deeper step could be active machine-learning inference focused on the behavior of the allocating functions embedded in the binary. Consequently, we give priority to efficiency rather than accuracy (which will be taken care of by the deeper analysis).
We aim to propose a lightweight approach (in terms of execution speed), efficient enough to be scalable to large programs (such as web browsers or multimedia file readers). Yet, we show in section 5 that even with a lightweight approach, we obtain accurate results in practice.
Another consequence is that each step of our analysis is performed by design on a single and simple execution trace.

2.4 Limitations

2.4.1 *Non-object paradigm*

Analysing object-oriented programs would require further investigation (in particular an analysis of `this` call conventions). We chose to limit our work to non-object programs, but this could be addressed in a future work.

2.4.2 *Calling convention*

We base our approach on the `x86-64` calling convention, and more specifically the `System V AMD64 Application Binary Interface` [12]. This is the convention used on 64-bit `x86` machines that runs on `Solaris`, `Linux`, `FreeBSD` and `Mac OS X`. According to this `ABI`[3], the parameters are primarily passed *through registers* (not through the stack, as it is with `x86-32` for instance). The first six integer or address parameters are passed through `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` ; and the first eight floating point arguments are passed through `%xmm0`-`%xmm7`. Any additional parameter is passed through the stack. Our approach relies specifically on this calling convention, as it assumes that parameters are passed through these particular registers. However, we could adapt our techniques to other calling conventions, but probably neither on custom compilers using their own conventions, nor on programs written directly in assembly language.

2.4.3 *Stripped binaries*

The approach we propose in section 4 does apply to binaries that are not stripped. However, for evaluation purposes, we have split our implementation into three pintools (each one infering an element among arity, type and coupling). We need an invariant to denotate a function from a pintool to another ; and for commodity reasons, we currently use function names. Therefore, our implementation does not support yet stripped binaries. Note that this is **not** a strong limitation of our work and it will be fixed in the next version of our tool.

2.4.4 *Variadic functions*

Our approach does not include the inference of arity for variadic functions (i.e. functions with a variable number of parameters, such as `printf`). This would need specific heuristics that are not discussed here.

2.4.5 `OUT` *parameters*

Some calling conventions allow functions to take `OUT` parameters. It can be either a register or a location on the stack, that will be written by the called function and fetched by the caller later in the execution. This case is not handled by our current implementation. However, the algorithms we propose are easily adaptable to correctly deal with such functions, using the same heuristics.

3. DEFINITIONS

In this section, we present definitions and formal descriptions of the problems we address.

3.1 Notations

We denote by B a binary file under analysis. B calls a set of functions that we denote by B_f. Among those functions, some of them are implemented in B (the binary code corresponding to these functions is then a part of B), and some are dynamically linked this is the case for shared libraries). Consequently, from B_f, we define two disjoint subsets: B_f^e the set of functions embedded in B and B_f^s the set of functions linked to B but implemented elsewhere.
We assume that each function $f \in B_f$ has a fixed number of parameters that we denote by $\#f$ (possibly zero), and a return value (possibly empty for a procedure). We denote by RET_f^e the set of values returned by $f \in B$ during a given execution e. In the same way, we denote by $PARAM(i)_f^e$ the set of values used as an i^{th} parameter of f during a given execution e, with $f \in B$ and $i \in [\![1, \#f]\!]$, and $f[i]$ the i^{th} parameter of f.
We denote by $memloc(a)$ the memory location of the variable a, for any a, including function parameters. This location can be a memory address or a register. For instance, for a function $f \in B_f$ and $i \in [\![1, \#f]\!]$, $memloc(f[i])$ is the register or memory location used to transmit the i^{th} parameter of f.

3.2 Problems

3.2.1 *Arity*

Determining the arity of each function defined in the binary is also a main goal. For each $f \in B$, we want to infer the number of parameters that f takes, denoted by $\#f$. Note that we consider that a function has a fixed number of parameters.

3.2.2 *Type*

We try to recover (partially or totally) the function $type_f$ defined for each function $f \in B$ as follows:

$$type_f : i \to \begin{cases} \text{Type of } f[i] & \text{if } i \in [\![1, \#f]\!] \\ \text{Type of return value of } f & \text{if } i = 0 \end{cases}$$

where a type is an element of $T = \{\texttt{addr}, \texttt{int}, \texttt{float}\}$. Note that the function *type* always returns the same and unique type for a given variable. Consequently, we do not deal with parameters that would have multiple types. Inference of multiple types for a given parameter is out of the scope of this paper.

3.2.3 *Coupling*

We propose here to find functions that are dynamically coupled, *ie* there is a "connection" between the output of one function and one parameter of the other. Let f and g be two functions of B. f and g are said **ρ-coupled** if and only if there exists $i \in [\![1, \#g]\!]$ such that:

- $type_f(0) = type_g(i)$, *ie* the return value of f and the i^{th} parameter of g have the same type.
- For any execution e, if we denote by $nc(g, e)$ the number of times g is called during execution e, and $C_e^{[f \to g]}$ the number of times a *def-use* chain exists between RET_f^e and $PARAM(i)_f^e$ during execution e, then we should have $C_e^{[f \to g]} \geq \rho \times nc(g, e)$.

In other words we say that $f \in B$ and $g \in B$ are ρ-coupled if, for a given formal parameter p of g, a proportion ρ of values taken by p are computed from return values of f.

We say that f and g are **strongly ρ-coupled** if they are ρ-coupled and for a given $i \in [\![1, \#g]\!]$, $type_f(0) = type_g(i) =$

[3]Application Binary Interface

`addr`. This notion is stronger that a simple coupling because a reference allows bilateral data sharing between the caller and the callee, as the callee can read and write the value pointed by the address given in parameter.
Examples of coupling are given in section 4.4.

4. APPROACH

In this section, we present our methods relatively to prototype inference, and coupling inference. As mentioned in section 2.4.2, our approach relies on a particular calling convention, mainly for arity inference. The next section deals with code instrumentation and the following sections are dedicated to the problems defined in 2.3. For each problem, we first present the intuitions and approximations we use, followed by the proposed solutions and a discussion.

4.1 Dynamic instrumentation

The methods we propose are based on dynamic code instrumentation. We assume to be able to instrument any instruction of the binary under analysis, and intercept any call or return of any function. This assumption is realistic with `Pin`.

4.2 Arity

4.2.1 Intuitions and approximations

1 – *To detect function parameters, it is enough to watch read-access to registers.*

The underlying idea is that a parameter location should be read at some point. For f in B_f and $i \in [\![1, \#f]\!]$, if there does not exist an execution e such that $memloc(f[i])$ is read during the execution of f (*ie* after a call to f and before f returns), then $f[i]$ is useless and should not be considered as a parameter of f[4].

2 – *A location always written before being read can be ignored.*

Indeed, if for any possible execution e, and for a given function f and $i \in [\![1, \#f]\!]$, $memloc(f[i])$ is always written before being read, then $f[i]$ is not relevant as a parameter and should not be considered as one. Note that this approximation does not work with `OUT` parameters (as they are written before read by the callee). However, such parameters could easily be detected in the same way we would detect return values (see section 4.2.3, by checking the memory locations accessed by the caller after the call.

3 – **1** *and* **2** *can be checked on a finite (small) number of executions of the function under analysis.*

Previous points mention properties of parameters on all possible executions of a given function. However, checking this would raise code coverage issues, and slow down the execution. Instead, we claim that we can conclude on a finite number of executions of each function (*ie* the number of calls to it), namely `NB_CALLS_TO_CONCLUDE`, with high precision. Section 5.3.1 presents experimental results that confirm this idea.

4 – *A third parameter implies a first and a second one.*

This may seem to be a tautology, but given the calling convention we rely on, it makes sense. Indeed, this calling convention defines an ordered list of registers used to pass parameters. If we detect a parameter passed through register `%rdx`, we can deduce that there are at least three parameters (`%rdi` and `%rsi` must have been used before `%rdx`), even if we did not detect the two others.

[4]We recall that OUT parameters are not considered

4.2.2 Method overview

To infer arity, we essentially instrument four types of instructions at binary level: any register read (Algorithm 1), any register write (Algorithm 2), and function calls and returns (resp. Algorithms 3 and 4). For each instruction, we define handlers taking different parameters from the context. For instance, a register read will induce a call to `on_reg_read` with the register name as first parameter (e.g. `%rdi`). We give a simplified version of each handlers in pseudo-code in the following algorithms. Some details on implementation are discussed in section 4.2.3.

```
Input: reg: register being read
Data: curr_fn: current function being executed,
      last_write: array containing for each register the
      last function that wrote it, ret_since_w: for each
      register reg, ret_since_w[reg] is True if and only
      if a RET instruction occurred since the last time
      reg was written
if !ret_since_w[reg] and last_write[reg] != Null then
    /* This loop is discussed in section 4.2.3
       */
    foreach f in call_stack.from(last_write[reg]) do
        /* Parameter detected for f              */
    end
end
```

Algorithm 1: `on_reg_read` for arity inference

```
Input: reg: register being written
Data: curr_fn: current function being executed,
      last_write: array containing for each register the
      last function that wrote it, ret_since_w: for each
      register reg, ret_since_w[reg] is True if and only
      if a RET instruction occurred since the last time
      reg was written
last_write[reg] ← curr_fn;
ret_since_w[reg] ← False ;
```

Algorithm 2: `on_reg_write` for arity inference

```
Input: fid: id of the function being called
Data: call_stack: stack of functions called and not
      returned yet, curr_fn: function currently being
      executed
call_stack.push(fid);
curr_fn ← fid;
```

Algorithm 3: `on_call` for arity inference

4.2.3 Discussion

Soundness – From a theoretical point of view, our approach is not sound since approximations we make can lead to false positives (*ie* detect parameters that do not exist). In

```
Data: call_stack: stack of functions called and not
      returned yet, curr_fn: function currently being
      executed, ret_since_w: for each register reg,
      ret_since_w[reg] is True if and only if a RET
      instruction occurred since the last time reg was
      written
fid ← call_stack.pop();
curr_fn ← fid;
foreach reg do
    ret_since_w[reg] ← True;
end
```

Algorithm 4: `on_ret` for arity inference

particular, approximation **4** can induce false positives, because a global variable could be passed to a function through one of these registers (this would be highly dependent on the compiler used to generate binary code). Although a global variable could be considered as a parameter at binary level (if we omit side effects), it may lead to important arity errors (if a global variable is passed through register `%r9`, we would immediately deduce that the function takes at least six parameters). In practice, this case never occurred during our tests.

Completeness – Approximation **3** impacts significantly the completeness of our approach. Indeed, running a finite number of times (called `NB_CALLS_TO_CONCLUDE`) each function to conclude on its arity leads to false negatives (*ie* missing a few parameters). In particular, we can miss unused tail parameters. However, as shown in section 5.3.1, increasing `NB_CALLS_TO_CONCLUDE` does not improve arity results anymore after a certain value. To be sound, we would need to use other methods. This will be the subject of a future work.

Ascendant propagation of parameters – It can happen that a function takes a parameter that is not used within its body, but is passed to another function. We give such an example in Listing 1.

```c
void bar(int n);
void foo(int n) {
    bar(n);
    ...
}
```

Listing 1: Example of direct propagation of parameters

In this case, `foo` never uses its parameter `n`. Indeed, at assembly level, the register `%rdi` containing the value of `n` is never read during the execution of `bar`. To detect `n` as a parameter of both `foo` and `bar`, we need to propagate parameters through embedded calls, from the function that wrote a given register (here the function calling `foo` with a given value of `n`) to the function that actually reads the register (here **bar**). Note that the propagation stops at the first `RET` instruction encountered. Indeed, parameter registers are scratch registers, so the compiler cannot assume that their value is preserved after a call. Therefore, after a `RET` instruction, any new propagation should start by the resetting of the relevant register. In Algorithm 1, this is what the loop does: it iterates on each function that has been called (with no `RET` in between calls) since the last write of the accessed register. For each of these functions, we deduce the existence of a parameter.

Return values – This section focuses on the detection of the number of parameters of a given function. However, the sames ideas can be applied to detect if a function returns a value or not. To do so, we would watch accesses to `%eax` register. Let say that f calls g ; we would detect that g returns a value if:

- `%eax` is written by g at some point,
- `%eax` is read before written by f after the call to g.

In addition, we would apply the same techniques described in "ascendant propagation of parameters" to define a "descendant propagation of return values".

Implementation issues – Some technical problems that we omit in previous algorithms (for clarity purposes) occurred during the implementation of our tool. The most important one is the miss of some calls and returns. This makes the maintenance of a consistent calling stack very difficult. We solved this problem by resetting the whole calling stack when a `RET` instruction is reached. Although this may seem inappropriate, it does not affect our results. Indeed, what we need is to propagate arguments through calls. However, all parameter registers are scratch registers, so when a `RET` instruction is encountered, parameter propagation cannot go further, except if the registers are reset with the relevant values after the `RET`. If they are, we deal with such a situation by calling `on_reg_write`.

4.3 Type of parameters and return value

4.3.1 Intuitions and approximations

1 – *There are only three basic types of data, and size does not matter.*

We only consider the three following types of data: **integer**, **address** and **float**. The distinction we make is semantic. An integer is semantically very different from an address, as the kind of operation (at assembly level) we can perform with one or the other are different: we can dereference an address but not an integer, and we can multiply two integers but not two addresses. The same argument stands between integers and floats, and between addresses and floats.

Even though integers may have different sizes in bytes and addresses may point to different types of data, for our purposes we do not need to be that much accurate. Indeed, the abstract behavior of a binary code can be partially understood without distinguishing between a two-byte integer and a four-byte integer. Being able to make these distinctions have been already treated in several papers, but it is not in the scope of this work.

2 – *Floats are passed through specific registers.*

Within this set of three types, we can make another distinction: floats have another difference with integers and addresses, as they are usually handled in specific registers (namely `%xmm0` to `%xmm7` for parameters, `%xmm0` for return value). Thus, by watching registers as we do in section 4.2 for arity is enough to conclude for float parameters and return value.

3 – *Address parameters can be identified when they are used.*

What, by definition, differentiates an address from an integer is that an address can be used as a memory location to read or write, *ie* as an operand to an instruction such as `LOAD` or `STORE`. We assume that an address parameter will,

at some point, be used in such an instruction during the program execution. However, it does not necessarily happen inside the function: an address can be stored at some location during the function call and be used as an address only later in the execution.
4 – **3** *can be checked on a finite (small) number of executions.*
This assumption is the same we made for arity inference.
5 – **3** *can be checked by simple value comparison.*
This is the strongest approximation we make. It allows to keep a light instrumentation, while showing good accuracy. The idea is to keep argument values of each call in some data structure, and detect if they are used as memory operands later in the execution. The detection is made by comparing memory operand of each relevant instruction with the stored values. For instance, at a given call of the function f, we will save the value of parameter value $f[i]$ for a $i \in [\![1, \#f]\!]$, and then watch if any of the future memory instructions use this value as a memory operand. If it is the case, we assume that the correspondence of values is enough to conclude that there exists a data-flow between $f[i]$ and this memory operand, and therefore $f[i]$ is an address. It relies on the following intuitions:

- It is unlikely that an integer has the exact same value as a valid and used address. Therefore, observing values on several calls should statistically lead to correct results.
- An address passed in parameter is rarely modified before being used as a memory operand. Therefore, we should not miss a lot of addresses due to pointer arithmetic (which is usually done inside functions).

In practice, this heuristic leads to an efficient and accurate type inference (see section 5.3.2).
6 – *A value between two addresses is an address.*
If we already detected two addresses, namely $addr_{low}$ and $addr_{high}$, any value within the range $[\![addr_{low}, addr_{high}]\!]$ can be inferred as being an address. This is time saving, as it avoids value comparison with a large set of values each time a memory instruction is performed. In practice, we exclusively rely on this method to decide whether a value is an address or not. To be as accurate as possible, we perform this test (is a given value an address ?) at the end of the execution of the entire program, at which point the bounds inferred are the more accurate. As discussed in section 4.3.3, the drawback is a loss of soundness, but it gives good results in practice as we show in section 5.

4.3.2 Method overview

We give here an overview of instrumentation algorithms we use as handlers for respectively memory access - either read or write - (Algorithm 5) and function call and return (Algorithms 6 and 7). We also give in Algorithm 8 the handler called at the end of the program execution. This is where we conclude on parameter types.

4.3.3 Discussion

In this section, we discuss correctness and soundness of our approach relatively to address inference, assuming that arity has been inferred correctly. A false positive is an integer parameter inferred as an address ; and a false negative is an address parameter inferred as an integer.

```
Input: $addr$: address being accessed
Data: $data_{low}$: lower bound of the address space
      observed so far, $data_{high}$: higher bound of the
      address space observed so far
if $addr < data_{low}$ then
    $data_{low} \leftarrow addr$;
end
else if $data_{high} < addr$ then
    $data_{high} \leftarrow addr$;
end
```

Algorithm 5: `on_mem_access` for type inference

```
Input: $fid$: function being called
Data: $arg\_value$: two-dimensions array, where
      $arg\_value[i][j]$ is a list of values of the $j^{th}$
      argument of function $i$
$i \leftarrow 1$;
foreach $arg$ of $fid$ do
    $arg\_value[fid][i].push(arg)$;
    $i \leftarrow i + 1$;
end
```

Algorithm 6: `on_call` for type inference

Soundness – Approximations **5** and **6** lead to a loss of soundness, for the following reasons:

- **5** – If an integer parameter takes, by chance, a value being a valid address that is used later in the execution, this parameter will be seen as an address. This is the price to pay of not using data-flow analysis to propagate data.
- **6** – In the same way, if an integer parameter takes a value within the addressing space, it will be seen as an address.

In practice, experiments (see section 5) show that those two heuristics does lead to good results, while their implementation remain light.
Completeness – Approximations **3** and **4** induce a loss of completeness. Indeed, due to **3** we are not able to detect unused parameters, and due to **4** we may miss parameters that are not use during the executions we observed, but that are used on particular execution paths we did not explore. Detecting unused parameters can be useful for some purposes (e.g. for code refinement), but in our case missing them is not a problem. However, missing parameters that are used on some execution paths that we did not explore is an issue. In fact, this is the preponderant source of errors of our approach (see section 5.4.2). Another source of error could be a pointer (say `ptr`) to a structure that is shifted before being passed as a parameter (e.g. `ptr + 4`).

4.4 Coupling

We present in this section our proposal to find functions that are ρ-coupled, with definitions given in section 3.2.3. Less formally, we are looking for outputs (that are of type `addr`) of functions that are given in parameter to another function. For instance, the output of `malloc` is often given in parameter to `free`. We parameter our method by the coupling coefficient ρ, which indicates the frequency of coupling we are looking for. For example, `malloc` and `free` in a given implementation that does not use `calloc` nor `realloc`

```
Input: $fid$: function returning, $ret$: return value
Data: $arg\_value$: two-dimensions array, where
      $arg\_value[i][j]$ is a list of values of the $j^{th}$
      argument of function $i$
/* We use the index 0 for return values */
$arg\_value[fid][0].push(ret)$;
```

Algorithm 7: `on_ret` for type inference

```
Data: $arg\_value$: two-dimensions array, where
      $arg\_value[i][j]$ is a list of values of the $j^{th}$
      argument of function $i$
foreach function $fid$ do
    if $nb\_calls[fid] >= NB\_CALLS\_TO\_CONCLUDE$
    then
        for $i$ in $[\![0, \#fid]\!]$ do
            $nb\_addr \leftarrow 0$;
            foreach $v$ in $arg\_values[fid][i]$ do
                if $is\_addr(v)$ then
                    $nb\_addr \leftarrow nb\_addr + 1$;
                end
            end
            if $\frac{nb\_addr}{nb\_calls[fid]} > TYPE\_THRESHOLD$
            then
                /* $i^{th}$ parameter of $fid$ is an
                   address */
            end
        end
    end
end
```

Algorithm 8: `on_exit` for type inference

are theoretically ρ-coupled with $\rho = 1.0$, as the parameter of `free` is always coming from the output of `malloc`. In the next section, we explain why we only try to infer strong coupling, and then we present our method.

4.4.1 Strong ρ-coupling only

In this paper, we specifically target strong coupling (*ie* coupling involving addresses), and do not consider nor try to infer two functions coupled by integer values. For instance, we try to find out the link between the return of `fopen` and the parameter of `fclose` (see Listing 2), but we do not consider the link that may exist between the return of `fclose` and a function that would check the return code to ensure that the file was correctly closed (see Listing 3).

```
FILE *fopen(const char *restrict filename,
const char *restrict mode) {
    FILE *fs;
    /* code relative to file stream opening */
    ...
    return fs;
}
int fclose(FILE *stream) {
    if (stream == NULL)
        return FCLOSE_ERR;
    /* code relative to file stream closing */
    ...
    return FCLOSE_OK;
}
int main(void) {
    FILE *fs = fopen("/etc/shadow", "r");
    ...
    fclose(fs):
    return 0;
}
```

Listing 2: Example of two functions strongly coupled

```
int fclose(FILE *strem) {
    if (stream == NULL)
        return FCLOSE_ERR;
    /* code relative to file stream closing */
    ...
    return FCLOSE_OK;
}
void check_retcode(int retcode) {
    if (retcode == FCLOSE_ERR)
        exit(EXIT_FAILURE);
}
int main(void) {
    FILE *fs = fopen("/etc/shadow", "r");
    ...
    int r = fclose(fs):
    check_retcode(r);
    return 0;
}
```

Listing 3: Example of two functions coupled with integer values

The reason behind this choice is that links between functions involving addresses give more interesting information about the binary under analysis. For instance, knowing that could improve use-after-free vulnerability detection [7].

4.4.2 Problems to deal with

The definition given in section 3.2.3 leads to a natural method to find couplings, consisting in following the data-flow from output of functions, and notice when it is used in parameter of other functions. However, there are at least three practical obstacles:

- The ρ-coupling is defined considering *any* possible execution, whereas we can only observe a finite set of executions in a finite time.
- Following a data-flow (using taint propagation for instance) during a whole execution is possible yet heavy either in time consumption or in memory usage, and could lead to a whole "tainted" memory at the end of an execution (that would probably be the case if we taint the output of `malloc`).
- Propagate the data-flow from any return value of any call of any function during an execution would lead to an important loss of efficiency, and probably lead to unusable results (as every input would be tainted by almost every function).

4.4.3 Intuitions and approximations

1 – *Coupling can be inferred on a finite (small) number of executions.*

We try here to infer two things: couples (f, g) and the parameter ρ corresponding to these couples. For the coupling, this assumption has the same implications that the ones we have done for arity and type inference. However, relatively to ρ, we add another implication: we say that the theoretical value of ρ can be approximated by statistics on the first `NB_CALLS_TO_CONCLUDE` calls of g.

2 – *We only work with addresses.*

This is a direct consequence of section 4.4.1: because we only try to infer strongly-coupled functions, we only instrument functions that return an address (*ie* functions $f \in B_f$

such that $type_f(0) =$ ADDR), and functions that take at least an address as a parameter (*ie* functions $g \in B_f$ such that there exists $i \in [\![1, \#g]\!]$ such that $type_g(i) =$ ADDR).

3 – *Coupling can be inferred by simple value comparison.*

In the same way we detect addresses by replacing dataflow with value comparison in section 4.3.1, we propose to detect strong coupling between f and g by comparing values returned by f with parameter values taken by g. We can do this approximation because we are trying to infer only strong coupling, *ie* coupling between parameters and return values of type address.

4.4.4 Method overview

The dynamic instrumentation of this step consists in storing parameter and return values of functions at each call, in the same way we did in section 4.3.2. Due to approximation **2**, we only instrument functions that have either a parameter or a return value of type address. Due to **3**, we do not keep all values for the parameter values, only the first NB_CALLS_TO_CONCLUDE values. However, we store every return value (for the relevant functions). We end up having a set of values stored in *arg_value*. *arg_value* is a two-dimensions array, where $arg\_value[fid][i]$ contains a list of concrete values for the i^{th} parameter of function fid. The index 0 is used to design the return values ($arg\_value[fid][0]$ contains all concrete return values of f occurred during the execution). Algorithm 9 describes the method we use, at the end of the execution, to find out couples from the set of values we obtained. It is also parameterized by two values: NB_CALLS_TO_CONCLUDE which has the same role as before, and ρ which is the threshold for coupling detection (we do not detect functions that are coupling for a value less than ρ). The influence of those two parameters is discussed in section 5.3.2.

4.4.5 Discussion

Soundness – Approximation **3** leads to a loss of soundness, because of address reuse. Let say that f returns an address a, that is freed later in the execution. If at some point a is reused for another allocation and given to g, we will infer a shared value between f and g (because of the numeric equality between the parameter of g and the return value of f, here a).

Completeness – As for arity and type inference, we are not complete because of approximation **1**.

Remark – For coupling inference, we do not need more information dynamically caught than for type inference. Then, we could perform type and coupling inference on the same execution. However, it is important to note that we need results of type inference to conclude on coupling. So in practice, we would have two parts on the on_exit handler: one to conclude on types, and a second one (that comes after) to conclude on coupling, using the results of type inference. In our implementation, we kept one pintool for each task for clarity and testing purposes.

```
Data: arg_value: two-dimensions array, where
    $arg\_value[i][j]$ is a list of values of the $j^{th}$
    argument of function $i$ arg_type: two-dimensions
    array, where $arg\_type[i][j]$ is the type (ADDR, INT
    or FLOAT) of the $j^{th}$ argument of function $i$
    nb_calls: one-dimension array, where
    $nb\_calls[fid]$ is the number of times $fid$ was
    called during the execution.
/* Iteration on functions returning an address
   */
foreach function fid such that arg_type[fid][0] ==
ADDR do
    if nb_calls[fid] >= NB_CALLS_TO_CONCLUDE
    then
        continue;
    end
    /* Iteration on functions taking an address
       as a parameter                              */
    foreach function gid do
        foreach i such that arg_type[gid][i] == ADDR
        do
            if nb_calls[gid] $\geq$
            NB_CALLS_TO_CONCLUDE then
                continue;
            end
            nblink = 0;
            foreach v in arg_value[gid][i] do
                if $v \in$ arg_value[fid][0] then
                    nblink += 1;
                end
            end
            if $\frac{nblink}{arg\_value[gid][0].size()} > \rho$ then
                /* f and the ith parameter of g are
                   strongly $\rho$-coupled                */
            end
        end
    end
end
```

Algorithm 9: on_exit for coupling inference

speed up the analysis. A performance comparison to quantify the improvement will be the object of a future work. Relatively to coupling inference, the aim is to validate the principle of our approach, but deeper tests will also be the object of a future work. As we focus on the accuracy of our heuristics, we perform our tests on a unique execution of each program at each step (arity, type, coupling). Moreover, we do not aim to recover all functions embedded in a given binary, so we do not try to construct particular inputs that would lead to a high code coverage. Instead, we choose to work on very simple executions with standard inputs (see section 5.2.2).

5. EXPERIMENTS

5.1 Goals

In this section, we present experiments designed to test the validity and relevance of the heuristics we described in section 4. For prototype inference, the question we address is the accuracy of our method, since we use approximations to

5.2 Experimental framework

5.2.1 Set of programs

We propose, as a common benchmark to evaluate accuracy of our heuristics, the following set of programs: Midori (v0.5.10, 45KB), MuPDF (v1.7a, 14MB), grep (v2.21, 704KB), and Emacs (v24.5, 27MB)[5]. The choice of these

[5]Static sizes of binaries on disk (not including DLL).

programs has been driven by the following criteria:

1. They are **well-used programs**, so testing our approach on them makes sense as it is representative to real cases of use.
2. They are **open-source**, so we can compare our results to the source code and therefore test the accuracy of our approach.
3. They are **written in** `C`, which is the most widely-used non-object language. We recall that our approach is currently designed to work on non-object programs (see section 2.4).
4. Their **size is representative** of the programs we daily use (large programs are represented by `emacs`, smaller ones by `grep`).

5.2.2 *Inputs fed to each program*

- For `Emacs`, we open a `C` source file of about 500 lines.
- For `Midori`, we do not give any input (the browser automatically opens `http://google.com` when it starts).
- For `MuPDF`, we open a pdf of 67 slides generated with Keynote.
- For grep, we look for the expression "void" in a folder containing about 15000 files for a total of about 30 millions of lines.

Note that, in this preliminary evaluation, we do not include diversity in our tests through inputs but through multiple programs. Still, we observe homogeneous success rates on those different programs.

5.2.3 *Validation*

To validate our results, we wrote a `Python` script that extracts information from source code and compares it with the values inferred by our pintool. This script uses a syntax analysis to recover arity of functions from prototypes and calls, and `clang` (which performs a semantic analysis of source code) to determine the basic type of each parameter (`int`, `addr` or `float`).

5.2.4 *Platform*

We run our tests on a `Linux Mint 17 64 bits` virtualized with `Virtual Box 4.3.20`. The host characteristics are a `Intel Core i7-4610M` and 16 Go of `RAM`. The virtual machine was attributed two CPU cores and 8 Go of `RAM`. All implementations have been written in `C++` as `pintools`.

5.3 Results

5.3.1 *Arity*

We present here some results of the arity inference with our pintool. Table 1 shows the accuracy of our results on the several programs we tested, for default parameter values. Accuracy is defined as follows:

$$\frac{\#nb\_ok}{\#nb\_fn\_inf}$$

where $\#nb\_ok$ is the number of functions for which the inference was correct, and $\#nb\_fn\_inf$ is the number of functions actually inferred.
It is important to note that:

- If a function is called less than `NB_CALLS_TO_CONCLUDE` times during the execution, then it is not inferred and therefore not taken into account in the given numbers
- Some functions are inferred but cannot be checked, because the prototype of the function was not found in the source code by our script. These functions are not taken into account in the given numbers either.

	`midori`	`grep`	`mupdf`	`emacs`
Accuracy (%)	95,8	95,6	98,7	92,4
Functions inferred	4094	51	526	591

Table 1: Arity inference accuracy (in %)

Influence of parameter – Functions that are called less than `NB_CALLS_TO_CONCLUDE` during the execution are ignored. It means than the more this value is high, the more selective our inference is, but the more precise it is expected to be. Figure 1 shows the variation of the number of functions and the accuracy of the arity results as a function of this parameter. We observe that the number

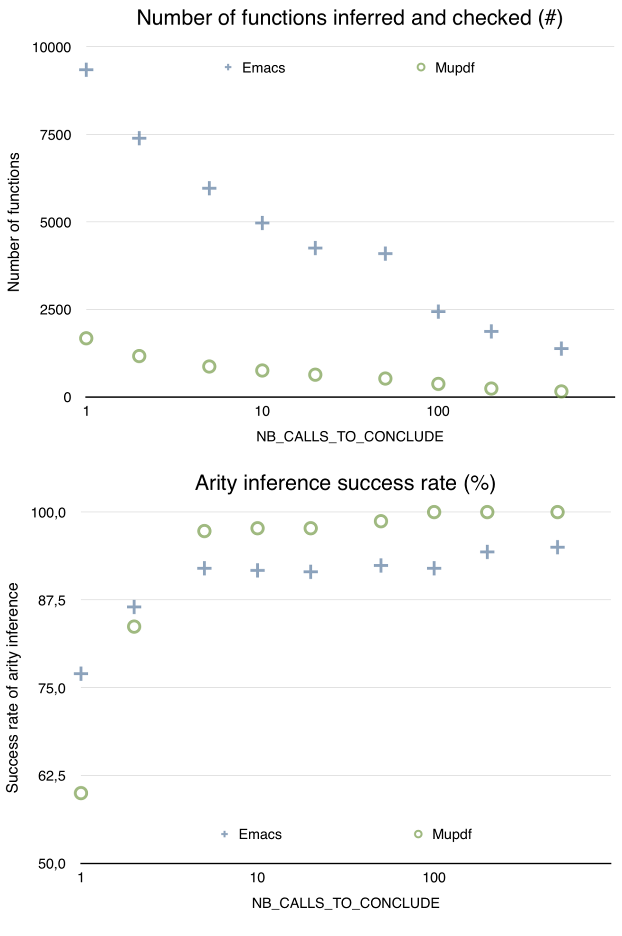

Figure 1: Influence of `NB_CALLS_TO_CONCLUDE` on arity inference

of functions inferred quickly falls down when the parameter increases. Indeed, with `NB_CALLS_TO_CONCLUDE = 10`, the execution of Emacs leads to 4969 functions inferred ; with NB_CALLS_TO_CONCLUDE = 100, only 2437 functions are inferred. On the other hand, for NB_CALLS_TO_CONCLUDE > 10, the gain on the arity inference becomes negligible. For a value of 10, the success rate on Emacs is 91,7%, and for a value of 100 it becomes 92,4%.

5.3.2 Types

Type inference is parameterized by two values:

- NB_CALLS_TO_CONCLUDE - this parameter has the same role as for arity inference.

- THRESHOLD_TYPE - this value sets the proportion of values that must be dereferenced during an execution to conclude that a given parameter is an address. For instance, for a given execution, f being a function of one parameter called n times, if the parameter is dereferenced during more than THRESHOLD_TYPE * n calls to f, then it is inferred as an address.

Table 2 presents the results of type inference on different programs with default values for these two parameters: THRESHOLD_TYPE = 0.01 and NB_CALLS_TO_CONCLUDE = 50. Note that accuracy values represent the proportion of functions for which **the type of each parameter** has been inferred correctly. Statistics on the accuracy at parameter level would then give better results.

	midori	grep	mupdf	emacs
Accuracy (%)	96,2	100	92,5	90,4
Number of functions	4094	51	526	591

Table 2: Type inference accuracy (in %)

Table 3 shows the influence of the parameter THRESHOLD_TYPE on the results of type inference on Emacs. We can observe that the smaller value for THRESHOLD_TYPE (0.01) gives the best results (91,76%). This means that to differentiate an address from an integer, detect one dereference during one call to a function is enough. From a theoretic point of view, this can easily be understood by setting the fact that an integer value would never be used as a pointer in any instruction. However, as we compare numeric values, we do use approximations in the data flow analysis. This results points out that our heuristics in numeric value comparison to follow data gives very good accuracy in practice.

THRESHOLD_TYPE	0.01	0.05	0.10	0.50	1.00
Ratio	91,76	90,80	90,56	88,83	10,51

Table 3: Influence of THRESHOLD_TYPE on type inference accuracy (with NB_CALLS_TO_CONCLUDE = 100)

Table 4 shows the influence of NB_CALLS_TO_CONCLUDE on accuracy results of type inference on Emacs.

Similarly to arity inference, we can observe that increasing the value of NB_CALLS_TO_CONCLUDE leads to a small improvement of accuracy (from 91,25% to 91,76% by increasing the parameter from 10 to 100), and in the same time the number of functions inferred decreases very quickly (from 857 to 466).

NB_CALLS	1	5	10	50	100
accuracy (in %)	89,24	91,25	91,35	90,41	91,76
nb of functions	1022	857	868	591	466

Table 4: Type inference accuracy (in %) on Emacs as a function of NB_CALLS_TO_CONCLUDE (with THRESHOLD_TYPE= 0.01)

5.3.3 Coupling

Coupling inference does not correspond to a piece of information that disappears during compilation and that we try to recover. Instead, it is behavioral information we try to catch for a better understanding of the binary under analysis. Therefore, we cannot present here results in the form of numbers in a table or charts.

Our coupling inference pintool outputs a list of tuples of the form (f, g, i, ρ), where f and g are the two function coupled, i the index of the parameter of g concerned by the coupling with f, and ρ the proportion of coupling between f and $g[i]$ that was dynamically observed during the execution. To have an idea of the scale of our coupling inference, in one execution of MuPDF with a threshold set to THRESHOLD_COUPLE = 0.8[6], we obtain 578 couples. Among these 578 couples, there are 42 different functions as the first element of a couple. For instance, js_malloc is involved (as f) in 50 couples with 50 different functions. Here is a fragment of the output of our pintool relatively to js_malloc:

```
js_malloc -> blit_aa[4] - rho = 1
js_malloc -> ft_char_index[1] - rho = 1
js_malloc -> FT_Get_Char_Index[1] - rho = 1
```

Listing 4: Partial result of coupling inference

An analysis of the 578 couples (f, g) we inferred leads to an interesting result. Indeed, we observe that among the functions f, a significant proportion represent allocating functions (do_scavenging_malloc - 124, fz_malloc_array - 115, js_malloc - 50 times). This is an interesting point, for two reasons:

- Allocating functions are the main example of function whose output is given to other functions as a parameter, so this result shows that the definition of coupling allows to catch interesting behaviors during execution.

- It is also encouraging for a future work to detect allocators, which is one of the aims of our larger project in which this paper takes place.

5.3.4 Time of execution

Table 5 gives some measures we did on time of execution during inference, for NB_CALLS_TO_CONCLUDE = 50 (THRESHOLD_TYPE and THRESHOLD_COUPLE have no influence on the time of execution). We are not able to perform these measurements on the programs we used before (midori, emacs and mudpf), because they require user interaction to complete (e.g. close the window by a click), which distort time measurements. Instead, we use command line programs that do not need any user interaction from start to termination: grep, tar and a2ps. These three programs have been used on data sized such that the normal execution (without instrumentation) lasts about one second. In this tabular, T1 is

[6]We recall that THRESHOLD_COUPLE is the minimum value of ρ to conclude that f and g are indeed coupled

the normal time of execution of the program (with no instrumentation), T2 is the time of execution with arity inference and T3 is the time of execution with type inference. We do not show the cost of coupling inference because it can be done in the same time type inference is performed with no additional instrumentation (we just add some computations on the `on_exit` handler, which is called only once, so the cost is negligible). We also show the number of functions effectively inferred, to give an idea of the amount of work performed in these executions.

	`grep`	`tar`	`a2ps`
nb of functions inferred	46	101	127
T1 (s)	0,80	0,99	0,80
T2 (s)	1,70	2,64	31,6
T3 (s)	1,06	1,79	13,2

Table 5: Time execution as a function of the instrumentation - T1: none ; T2: arity inference ; T3: type inference

First, we can notice that in the worst case (here for arity inference on `a2ps`, the time execution is multiplied by about 30. Relatively to other dynamic instrumentations, this factor is acceptable. For type inference, we have a maximum overhead of about 15. For small tools such as `grep` and `tar`, we can see that the overhead is very small for both arity and type inference (about x2 for each). Regarding to other techniques using dynamic instrumentation, our approach is indeed lightweight. In [18] for instance, where the goal is to retrieve data structures, the time of execution on `grep` is of about 15 minutes. In [6], they analyze 10000 assembly instructions in about 1.7 seconds, which is also much slower than our implementation. However, we recall that these works aim to be accurate whereas we aim to be fast; therefore time comparison is only relevant to validate the lightness of our approach.

5.4 Discussion about errors

5.4.1 Arity

In average, for about 5% of the functions we inferred, the arity is wrongly inferred. These errors have several sources that have been identified:

- For functions that have more than 6 non-float parameters, the arity is wrong because of a limitation of our implementation. Indeed, as we said in section 2.4.2, parameters are passed through the stack from the seventh's one. But our implementation only watches parameters passed through registers.
- Functions that have a variable number of parameters are not inferred correctly. Our approach is currently limited to functions of fixed arity (see section 3.2.1), so this kind of errors was expected.

5.4.2 Type

Relatively to type inference, we have identified a source of error that explains a part of the mistakes. We first observed that a consequent part of the errors occur on the last parameter that is inferred as an integer instead of an address. After a specific dynamic analysis of several of these functions, we observed that the concerned parameters are equal to zero, which explains the fact that our implementation infers its type as an integer and not an address. We propose the following explication, which is still an hypothesis: it seems to concern optional parameters that are given the `NULL` pointer because not needed in the context where the relative functions are called.

6. RELATED WORK

Reverse engineering has been a quite active research area in the last years, and many techniques and tools have been proposed to extract information from binary codes. As stated in the introduction, these techniques may largely differ depending on the nature of the information they aim to retrieve, the assumptions they make on the target program, and the effort they require in terms of computing resource to operate. We discuss here some of these works which are related to our proposal.

First of all, various techniques have been proposed to retrieve information regarding (high-level) data types and (abstract) data structures. These techniques can be partitioned into static analysis (with no code execution), and dynamic analysis (based on execution traces). In the former class we can mention [1], based on a binary-level abstract interpretation technique (VSA) for variable recovery and abstract structure identification. The resulting tool, Divine, allows to obtain much more precise results than the widely-used IDAPro disassembler, based on ad hoc heuristics. However, being based on a quite expensive technique, scalability of Divine is an issue. More recently, a more efficient yet unsound alternative has been proposed in [6]. According to the experiments performed, a better precision versus scalability trade-of is achieved. Another follow-up of [1] has been proposed in [9], which use a variant of VSA for variable identification followed by a type inference to associate a C-like type to each variable identified. All these approaches suffer from usual static analysis limitations, namely how to deal with statically unknown information (such as dynamic jump targets) without losing too much precision. On the dynamic side, a pioneering work has been proposed in [10], which infers data types from known "type sinks" in the code (such as C library calls). This idea has been generalized in [18], where specific analysis are used to retrieve memory allocated chunks (tracking the memory zones addressed by pointers) and to identify arrays using loop access pattern recognition. Results are obtained from several execution traces obtained by running a concolic engine to increase code coverage. All these works infer richer information about data type than our proposal, but they also require much more computational effort.

A second class of work deals with function prototype recovery. For instance, the method described in [3] aims to extract all dependencies of a given function to be able to execute it out of its embedding binary. To retrieve function input and output locations they dynamically track memory addresses or registers that are read before being written between a call and a return to conclude on the location of parameters. Nevertheless, this heuristic does not allow to distinguish parameters from global variables. A more complete algorithm is proposed in [6], to identify register arguments and return values of functions, taking into account saved registers on the stack (using a simplified VSA). The criteria we use to retrieve function signatures are similar to the ones proposed in these work (based on dynamic "liveness" and "reachability definition" computations). However our dynamic analysis is much simpler, with less overhead, and allows to retrieve function prototypes with a very good precision.

Finally, a huge number of tools have been proposed regarding dynamic taint analysis ([14, 17]), which is a problem close to the one we address regarding coupling inference: in both cases, the objective is to retrieve inter-procedural data-flow relations. One of the main challenge here is to ensure scalability by avoiding to instrument each instruction and to associate a shadow memory to each byte. According to our experiments, the aggressive heuristic we propose in this work, based on simple value comparisons, scales very well while providing good results, precise enough in many applications.

7. CONCLUSION

In this paper we proposed a lightweight dynamic analysis technique to retrieve partial inter-procedural data-flow relations and abstract function prototypes from a binary code. This technique relies on very strong heuristics, requiring only low-cost code instrumentation, hence able to operate at large scale. Although this technique is deliberately unsound and incomplete, the data-flow relations obtained are precise enough to be used in several code analysis or reverse engineering processes. In particular, the approach we propose can be viewed as a preliminary step before going deeper on specific parts of the code with more complex techniques such as static analysis or behavioral inference. One of the outcome of this work is therefore that even a few (lightly instrumented) code executions could leak very useful information regarding the structure and semantic of a binary code.

This work can be extended into several directions. First of all, our current prototype, Scat, could be improved to get rid of some current technical limitations. For instance we could take into account several (classical) calling conventions, using some pattern recognition technique to identify which one is used in a given function. Further experiments would also be welcome to better evaluate how far we can go in terms of scalability, and how good are the coupling relations we would obtain. Regarding the approach itself, it would be interesting to investigate if the heuristics we proposed could be (slightly) refined in order to overcome some current limitations, such as handling functions with a variable arity, or solving the "NULL pointer problem" mentioned in section 5.4.2, or dealing with code produced from object-oriented languages (as in [8]). Combining results obtained from several code executions could also improve the results without sacrificing too much the scalability. Finally, our next objectives will be to apply this work to allocator detection (short-term), and to reuse it in a more complete reverse-engineering framework based on active behavioral model inference (long-term).

8. REFERENCES

[1] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *VMCAI*, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. *USENIX Security*, 2014.

[3] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

[4] X. Chen, A. Slowinska, and H. Bos. Who allocated my memory? detecting custom memory allocators in c binaries. In *WCRE*, pages 22–31. IEEE, 2013.

[5] F. de Goër. Implementation of our approach is available on github. https://github.com/Frky/scat.

[6] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, pages 51–60, New York, NY, USA, 2013. ACM.

[7] J. Feist, L. Mounier, and M.-L. Potet. Statically detecting use after free on binary code. *J. Computer Virology and Hacking Techniques*, pages 1–7, 2013.

[8] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *PPREW*, page 1. ACM, 2014.

[9] J. Lee, T. Avgerinos, and D. Brumley. TIE: principled reverse engineering of types in binary programs. In *NDSS*, San Diego, California, USA, 2011.

[10] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, 2010.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm Sigplan Notices*, 40(6):190–200, 2005.

[12] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System v application binary interface. amd architecture processor supplement. *Also available as http://x86-64.org/documentation/abi.pdf*, 2013.

[13] MITRE. Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html.

[14] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[15] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *ETAPS*, CC'08/ETAPS'08, pages 16–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In *AAAI*, pages 798–804, 2008.

[17] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *SP*, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[18] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*. Citeseer, 2011.