A Heuristc Approach for ROP Chain Semantic Search

Stanciu Andrei Călin

Supervisor Conf.univ.dr.ing. Paul Irofti

University of Bucharest

Rezumat

Return Oriented Programming (ROP) este o tehnică de exploatare care necesită o analiză la nivel de instrucțiuni individuale a binarului țintă. În această lucrare va fi prezentată o modalitate de a automatiza acest proces, bazată pe algoritmi euristici care operează cu ajutorul unei reprezentări intermediare, independente de platforma procesorului. Modulul implementat oferă posibilitatea căutării semantice, și acoperă majoritatea cazurilor întâlnite în practică. Eficiența și timpii de execuție sunt analizați, în general având rezultate pozitive.

Abstract

Return Oriented Programming (ROP) is an exploitation method that requires an analysis at the instruction level of the target binary. This documentation aims to present a way to automate this process using heuristic algorithms that operate on a platform independent intermediary representation. The implemented module allows for semantic search and covers most of the use cases that arise in practice. The efficiency and observed runtimes are discussed, generally with positive results.

Contents

1 Introduction

1.1 Short History of ROP

Return Oriented Programming, as first described in [1], is an attack that originates from the first return-to-libc attack [2]. In short, it is a form of reusing (chaining) existent snippets of code, called *ROP gadgets*, usually endingwith a `ret` (or equivalent) instruction, in such a way that the attacker is able to create a malicious execution flow that suits his needs - in some scenarios it provides a turing-complete mechanism of attack, as shown in [3].

Various methods for mitigating (or, at least, increasing the difficulty of executing such an attack) have been proposed: Address space layout randomization ([4]) is one of the most popular and most implemented prevention techniques, randomizing the base of various memory sections (stack, shared objects, and so on) so that the attacker can no longer rely on fixed offsets for the gadgets in the memory - still the ASLR can be easily bypassed when the attacker is able to leak certain addresses; another popular class of mitigations consists of monitoring the control flow (eg. [5]), but, depending on the implementation and the mechanism of action, their efficiency and practicality is still under question ([6], or [5] broken by [7]), and are thus not widely adopted. Moreover, this type of attack has successfully been executed in real scenarios, for example [8], [9]. All of this points to the relatively high importance this attack presents, even decades after being first used.

1.2 Related Work

Besides focusing on the mitigations, another practical solution for avoiding ROP attacks is by simulating attacks and analyzing the already implemented security measures on the existing binaries meant to prevent such malicious scenarios. In this sense, the return oriented programming exhibits a "regular enough" pattern in its payloads, such that various automatization techniques for constructing ROP sequences were proposed. From the perspective of the properties of the generated *ROP chains* (i.e. the sequences of bytes, used to chain the *ROP gadgets* for a pre-determined malicious effect), the following tools can be distinguished:

- *simple gadget searchers* that are, in essence, binary parsers: they search the entire binary for executable sequences of instructions, taking into account some general criteria such as the ending of the sequence with a `ret` instruction on the x86 architecture, and providing the user with a complete list of ROP gadgets, filtering for simple constraints such as exact instruction mathings (eg. on x86_64, search the current binary for exactly `pop rcx; ret;`)
- *gadget searchers with semantics* offer the possibility of searching gadgets by specifying the desired effect, and other properties such as keeping some registers unchanged
- *chain generators for specific calls* that automatically try to construct a chain that calls a specific target function (most of the time, `execve` or related are targeted)
- *chain generators with semantics* that allow the entity executing the attack to choose one (or multiple) specific wanted effects; for example, on x86_64, the user can request for value 0x20 to be loaded in `rax` register, and the payload generator outputs `xor rax, rax; ret; add rax, 0x20; ret;`

When a successful attack implies the usage of very simple gadgets or when the targeted binary is sufficiently small, gadget searchers might be the best choice. In this sense, open-source tools like *Ropper* ([15]) or *ROPgadget* [16] provide the user with such features, including semantic gadget search or even a chain generator method for opening a shell, although the main disadvantage is flexibility: for example *ROPgadget* searches only for "template" gadgets to build a chain, and *Ropper* has similar drawbacks when it comes to more complex scenarios. That being said, such more complicated scenarios

are addressed by other proposed tools, such as [12], [11], and [10]; in these implementations, some aspects that are commonly addressed can be identified:

- usage of some form of intermediary language or symbolic (platform independent) representation to store the instructions / gadgets / chains
- how to test and classify gadgets w.r.t. their semantic effect: gadgets that store or load from memory, gadgets that load constants, and so on
- the problem of controlling jumps and memory addresses
- the problem of clobbering (needed) registers between gadgets execution
- reducing gadget interactions between one another to various graph (tree, directed acyclic graph) representations
- the usage of SMT solvers (most notable the z3 solver - [14]) for a portion of the tasks
- the usage of heuristic (and thus non-exhaustive) algorithms for chaining gadgets

The usage of non-exhaustive algorithms is unfortunately one of the consequences of the vast number of possible chains, with different effects, and is the primary reasons there are many different approaches, each with their own (at least slightly) different functionalities. Besides that, most of the existing tools begin with a binary parsing and gadget analysis phase, which is then followed by the usage of various algorithms that attemp to implement what could be vaguely defined as "chaining rules", all of this while manipulating mostly graph-related data structures.

The implementation described in this paper is inspired by all these remarks, and it aims to serve requests for a semantic search and chain generator. The supported architectures are x86_64 and ARM64 (ARM v8-A), with most of the algorithms being platform-independent. The supported binaries are ELF binaries, although, as with the processor architecture, the implementation can be generalized. Also, most use cases should be covered with a relatively small amount of computational time. The biggest differences between the existing tools and this module is that the latter one manipulates memory considerably more restrictive (i.e. no memory writes), and that it does not aim to create a chain for a function call (and does not interact with any existing function), but rather to satisfy the user requests.

The rest of this documentation is organized as follows: in Section 2, the intermediary representation and the used algorithms are defined; in Section 3, various experiments and results are presented; in the end, Section 4 summarizes a conclusion and adds some observations with regard to possible future directions.

2 Concepts and Implementation

2.1 Intermediary Representation

The intermediary representation offers a convenient (and platform-independent) way of storing the semantic of the gadgets (chains). Every subsequent algorithm is either responsible for converting the data in the intermediary representation, or operates over the intermediary representation (eg. every search algorithm). In the following paragraphs the definition for these basic building blocks will be given, along with any other observation regarding their implementation.

2.1.1 Structured Elements

Structured elements are the atoms with which any intermediary representation of the instructions and their *effects* are expressed. They consist in a type name and a variable number of properties (1), and are the same for both x86_64 and ARM64 architectures (throughout the rest of the documentation, when a concept is presented, it is the same for both architectures, unless specified otherwise; also, thse concepts could hopefully be generalised to other architectures).

One of the more important observations to be made is that the registers that are taken into account are general-purpose registers - the registers specific to floating point or vectorized operations are not supported. The 32 bit registers are converted in operations on the 64 bit corresponding register, masked with an *and* element. With regard to registers, through this documentation, the term "register" might refer to three things, depending on the context: a register's name, an input register (reg_in) element, or output register element (reg_out); usually, when taking about reg_in / reg_out elements, they will explicitly be annotated (R_{in}, R_{out}).

The stack elements with value (*64b_stack_val*) do not actually contain a value, but rather an id with global scope (i.e. any two 64b_stack_val with the same id correspond to the exact same value and (on ARM64) they are associated with the exact same jump).

The "operation" type elements are used to build arbitrary recursive operation trees - more emphasis on how their behaviour is analysed and matched is put on in later sections (2.2.4).

The *deref* elements are exclusively used only in the preprocessing phase of a running session, that will be detailed later (2.3).

2.1.2 Effects

Effects are pseudo-instructions used to express the functionality of one or more real instructions. They consist of a type name, a destination element and a variable number of parameters (2). The LOAD_S effect indicates the loading of a value from the stack - in some circumstances, it can be any expression (most likely containing deref elements); note that the stack is the only accessible portion of memory and it can only be read, write operations being absent from this implementation. The LOAD_CT effect marks the loading of a constant value in a register; MOV_RR indicates the copying the value of a register into another, and ARITH marks any arbitrary arithmetic or logic operation. The JUMP effect contains the expression that will ultimately resolve to the jump address used. One can observe that most of the effects are particular cases of ARITH effect; while this is true, this distinction helps in optimizing the implementation, and also makes it more easy to follow.

Only few mnemonics (instructions) are supported. As mentioned when defining the structured elements, there is no floating point or vectorized arithmetic. Also, the change in value of the stack pointer is restricted to addition with a constant offset.

Throughout the rest of this documentation, the effects might be formally represented in a different number of ways, depending on the context:

Table 1: Structured elements

type	properties
64b_stack_val	id
64b_stack_pad	-
ct_val	value (integer)
reg_in	register name
reg_out	register name
deref	expression (a structured element)
add	term1, term2 (structured elements)
sub	term1, term2 (structured elements)
and	term1, term2 (structured elements)
or	term1, term2 (structured elements)
xor	term1, term2 (structured elements)
neg	term1, term2 (structured elements)
mul	term1, term2 (structured elements)
lsh	term1, term2 (structured elements)
rsh	term1, term2 (structured elements)

Table 2: Effects

type	destination element	parameters
LOAD_S	reg_out	64b_stack_val or any other structured element except pad
LOAD_CT	reg_out	ct_val
MOV_RR	reg_out	reg_in
ARITH	reg_out	any operation element
JUMP	ct_val	any structured element except pad
NO_OP	None	-

$$R_{out} \leftarrow E_i(R_{in0}, R_{in1}, ...R_{ink})$$
$$R_{out} \leftarrow E_i(R_{in0}, R_{in1}, ...R_{ink}; S_0, S_1, ...S_t)$$
$$R_{out} \leftarrow E_i(R_{in0}, R_{in1}, ...R_{ink}; S_0, S_1, ...S_t; CT_0, CT_1, ...CT_q)$$
$$ef_i(jumpId; R_{in0}, R_{in1}, ...R_{ink}; S_0, S_1, ...S_t)$$

All of the above are considered valid representations: the first one expresses an effect as a function parametrized by the reg_out and all k reg_in elements; the second one explicitly mentiones all t 64b_stack_val elements (stack elements might be used even in the first representation, even if not mentioned); the third representation also indicates the constants used; the fourth one is a JUMP effect, whose destination element is an index corresponding to the associated jump (more on that in 2.1.5). For example, the effect $x10 \leftarrow (x8 + S_0 + 3)(x8 - 3)\&S_0$ can generally be represented as $x10 \leftarrow E(x8, CT_0, CT_1, S_0)$, where CT_0 and CT_1 indicate the two constants (with the same equal value of 3) that appear in the expression; the register is indicated only once, as it is the case for the stack element S_0.

The "identity" or "trivial" effect can either be expressed as a NO_OP effect, or implicitly as $R_i \leftarrow R_i$, which helps in defining subsequent concepts without treating the particular case when one or both effects do not exist. This observation applies in all cases where it would be needed without explicitly addressing it (for example, in *effect joining* 2.2.1).

2.1.3 Stack View

A *stack view* is a projection of a single, contiguous portion of a stack, discretized in 64 bit chunks, which contains specific structured elements: 64b_stack_val and 64b_stack_pad. Any stack view is always associated with exactly one *gadget* (or chain) (2.1.4) and its effect list. A stack view contains all the elements that interact with the associated gadget (chain) - more on that in further definitions.

2.1.4 Gadget

A *gadget* (wrt. algorithms presented in this article) is an abstract representation of a contiguous sequence of machine instructions (with some restrictions), along with their effects on the register values, and the layout of the portion of the stack that interacts with them. The associated stack portion is represented as a *stack view*, and the instructions are expressed as a collection of *effect* objects.

Each of the effect objects is independent of any other effect object from the same gadget, and the order of accessing / interpreting these effects does not matter (from the point of view of the operations presented in 2.2, it is a very useful property to be enforced). From this property, it can easily be derived the fact that no two (non-JUMP) effects from the same gadget have the same reg_out register: if there were two such effects, by the property of independence mentioned above, they would either have equivalent properties, in which case one of them would be redundant and thus could be eliminated, or the order of "execution" would matter (the last effect will dictate the value that will ultimately be present in that reg_out) - which violates the hypothesis that the effects are independend from one another and their order does not matter. To achieve such a representation of effects, a model which defines an *register input state* and an *register output state* is defined (fig 1). The gadget is treated as a black-box, the registers $R_0, ...R_k$ enter it with the input values $R_{in0}, ...R_{ink}$, and the only information that is retained consists of the final expressions (effects) that give the register output values $R_{out0}, ...R_{outk}$ - with this in mind, a collection of *independent effects* can be formulated as a collection of *effects* that act on the same *register input state*, and the *register output state* corresponds to all the reg_out elements from all the *independet effects*.

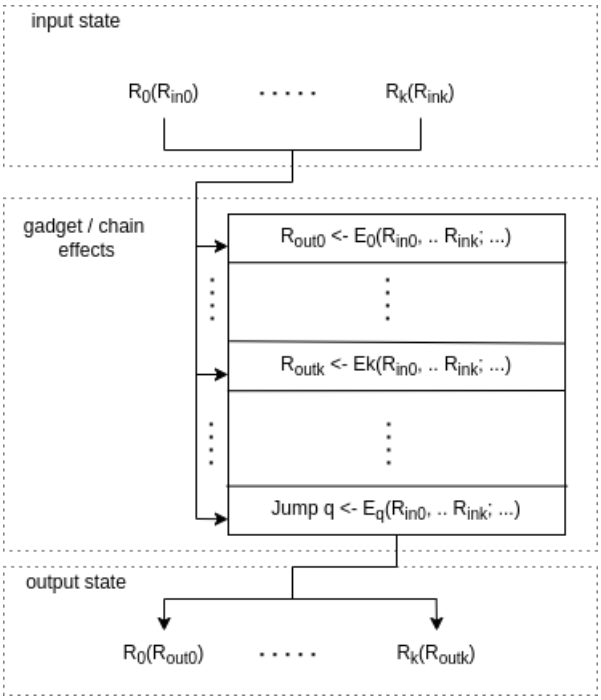

Figure 1: Gadget (chain) register input / output states

Regarding the memory, as stated in the *effect*'s definition, there are no memory writes - this means that the output state can be restricted only to registers, as the above formulation implicitly suggests.

Every gadget must also end with one (or more) specific branching instruction(s). On x86_64, the instructions that mark the end of a gadget are the `ret` and `jmp reg` instructions; on ARM64, the allowed branching instructions are `ret`, `bl` and `blr`. All of them are expressed as a JUMP effect, with the observations that for `ret`, the change in stack pointer is also marked with an ARITH effect, and for `blr` the change in register `x30` value is not taken into account; because of that, searching for wanted effects that contain `x30` register is forbidden (with this restriction, the rest of the implementation is able to handle all particular cases correctly, even without marking the alteration of `x30` value). It can be observed that, prior to executing a branching instruction, the register that contains the destination address can be arbitrarily manipulated, and thus the JUMP effect will have (almost) any structured element as its parameter. This poses a problem for gadget chaining algorithms, whose set of assumptions on which they are based on contain a *jump equivalence* principle: either no jump can take place, or the jump can take place to any address; because of that, the destination address of any gadget needs to be fully controlled by the algorithms; in this sense, a significantly more complicated implementation could address these issues, but this problem is further accentuated when taking randomization of address space into account. Still, some gadgets cannot guarantee these properties, and thus the additional notion of *valid jumps* (2.1.5) is introduced.

Another aspect of gadgets is the memory access: the only way to access the memory is to read from the stack. Writes are completely forbidden, and the reads from any other region of memory that is not guaranteed to be the stack is also not allowed. The implementation is able to represent with the help of "deref" elements any memory access, parametrized by any register, constant, or other stack variable. Still, analogous to the jump-related issues, the gadgets that are used must have their memory accesses completely defined: for this, the *valid stack access* (2.1.6) concept is also introduced. Still, a restriction

imposed by the current implementation mandates that there are no nested "deref" elements at any given moment - a more detailed explanation for why this is imposed is given along with the valid stack access definition.

The stack view (2.1.3) keeps track of exactly all the elements accessed in the gadget that are associated with it. The implementation consists of simply enumerating in an ordered list all the stack elements, including stack padding where the content does not matter. Also, because of the concept of stack joining (2.2.2), the end position of the stack pointer is stored in the gadget's attributes (the start position of the stack pointer is always the first element of the stack view).

2.1.5 Valid Jump

A *valid jump* is a JUMP effect that allows jumping to any non-0 (64 bit) address, and any of those destination addresses must be obtained solely by assigning values to its *jump-related* (see next paragraph) 64b_stack_val elements from the associated stack view.

When talking about valid jumps, it is useful to explain a particular relationship between JUMP effects and the 64b_stack_val elements from the associated stack; as previously mentioned, a 64b_stack_val contains an id which is globally associated with a value, which will ultimately be inserted in the final payload bytes - this value can be null (None) and in this case it will ultimately be treated as padding. But besides the (optionally) associated value, a 64b_stack_val has an *associated jump*, that indicates the fact that the 64b_stack_val in question directly influences the value of its associated jump (i.e. when writing the JUMP effect as an expression that needs to be equal to a particular jump destination address, the corresponding jump-related stack elements are the free variables, and the values assigned to each of those stack elements collectively represent the solution for the equation). This association is both created and used when checking if the jump is valid: if the stack element is pad (or stack value with currently no associated value), it becomes associated to that jump; if the stack element already has a non-null associated value, it will remain "non-associated" with that jump, and treated as a constant when trying to validate that jump; if it is already associated with another jump, it is treated as a random variable with uniform distribution. To justify the approach on the latter case, consider the situation in which there are two different JUMP effects that "want to claim" a 64b_stack_val element. When trying to resolve the first jump to a specific address, the associated stack element will take a value that depends on the destination address, which is (up to a point) random (this implementation is built on the assumption that ASLR is always activated, which corresponds to most of the practical exploitation scenarios nowdays). Because the destination is random, the value that will reside in the associated 64b_stack_val element has a significant chance of appearing random, at least from the point of view of the second jump that also tries to use that "variable". Thus, although being a restriction, choosing to simply treat other jump-related stack values as random variables substantially simplifies the implementation and reduces the number of computations needed to run the algorithm.

Note that the order of validating (checking for validity) for multiple jumps in the same chain (2.1.8) does influence the association of stack elements to jumps. Although this might be rightfully considered a race condition, if two jumps happen to "fight over" the same stack element, either this will not matter and the jumps can successfully be validated anyways, or, most likely, at least one of the jumps will not be able to be validated, case in which any other jump's validity does not matter anymore, and, if the association order would be changed, a symmetric scenario wrt. the jumps that can not be validated will happen - an argument simmilar to the one above can be followed, in which it is assumed particular cases are extremely unlikely and not worth taking into account.

One more thing to mention related to jump ids is that the jump id is unique relative to the associated gadget (chain), but it is not globally unique.

The process of validating an individual jump is defined in 2.2.6.

2.1.6 Valid Stack Access

A *valid stack access* is either a 64b_stack_val element, or a deref element, whose expression, no matter its contents or complexity, is equivalent to an expression of the form "SP + constant offset".

Even when the memory access is still not a valid (stack) access, there is one restriction imposed: no nested deref elements, at any given point. To motivate this fact, suppose there is a nested deref effect, described as $R_i \leftarrow [[SP]]$; to be able to validate this stack access, the effect would need to be brought to a form equivalent with $R_i \leftarrow [SP + c]$, where c is constant; in other words, $[[SP]] = [SP + c] \iff [SP] = SP + c$; this means that the corresponding 64b_stack_val for the $[SP]$ needs to be fixed, but at the same time it depends on the real (runtime) SP value, which, let aside the ASLR, it depends on the moment of time in which the gadgets that contain this effect are executed: if you were to obtain a chain out of two identical gadgets (even the same gadget which jumps back to its beginning), you would need to adjust the value assigned to that particular 64b_stack_val for each gadget, every time a new combination of gadgets is formed (and also, the user that requests these gadgets is forced to also know the stack base address, which in itself is another problem). Because of these reasons, nested dereferences are not taken into account.

The process of validating an individual memory (stack) access is defined in 2.2.7.

2.1.7 Valid Gadget

A *valid gadget* (or resolved gadget) is a gadget whose memory accesses are entirely *valid stack accesses* (2.1.6), and its jump is a *valid jump* (2.1.5) (for chains, all jumps must be valid). The search algorithms use only valid gadgets, which are simply called "gadgets". For the autmatic chain construction process, only valid gadgets can be taken into account (valid jumps is a necessity, because the jumps are used to direct the execution flow to the next gadgets in the chain; the validation of memory accesses could be postponed up to the end of the building process, but this would further complicate the implementation; moreover, some jumps might be invalid because they contain an invalid stack access, so resolving both of these criteria first greatly simplifies the implementation).

2.1.8 Chain

A *chain* is the generalization of a gadget, by which gadgets at different addresses end up branching to other gadgets that end up part of the same chain. Most of what was detailed when defining gadgets remains identical for chains; one major difference is that a gadget has exactly one JUMP effect, while a chain has an arbitrary (greater than 0) number of JUMP effects. Chains are obtained strictly by joining gadgets, by the rules mentioned in 2.2.3.

2.1.9 Wanted Effect

A *wanted effect*, although not as rigurously defined as other concepts, represents the concretization of the "custom semantic" that is imposed over the payload generating algorithms - in short, this entire implementation revolves around finding chains (2.1.8) that satisfy one / more wanted effects. Any wanted effect is represented and stored in memory using the exact same concept (and implmentation) of effects (2.1.2), the difference between a "normal" effect and a "wanted effect" being only at a logical level, when dealing with algorithms manipulating these notions, and not at the lower, per-effect level.

One important thing to note here is that the wanted effects are not allowed to contain any 64b_stack_val or deref elements (they cannot be of LOAD_S type) and also they cannot be JUMP effects.

2.2 Basic Operations

2.2.1 Effect Joining

The *effect joining* is a fundamental operation executed throughout the entire implementation. It takes as input two ordered (relative to each other) effects, and outputs another effect that corresponds to the "execution" of the second effect, taking into account the fact that the first effect has been previously "executed". It can be expressed as follows:

$$
\begin{aligned}
&e_0 := R_{out0} \leftarrow E_0(R_{in0}, ...R_{ink}),\\
&e_1 := R_{out1} \leftarrow E_1(R_{in0}, ...R_{ini-1}, R_{ini}, R_{ini+1}, ...R_{ink}),\\
&\text{reg\_name}(R_{out0}) = \text{reg\_name}(R_{ini}),\\
&e_2 = join(ef_0, ef_1) := R_{out1} \leftarrow E_1(R_{in0}, ...R_{ini-1}, E_0(R_{in0}, ...R_{ink}), R_{ini+1}, ...R_{ink})
\end{aligned}
$$

In short, it represents the substitution of the corresponding reg_in element from e_1 with the reg_out from the e_0. Note that, for simplicity, both e_0 and e_1 are defined as taking the same number (and type) of reg_in as parameters; if the R_{out0} does not have any corresponding R_{ini} for some index i, then $e_2 = e_1$.

Moreover, $\forall e_i, \forall e_j, \forall e_k$, if e_j and e_k are independent, then $join(e_i, e_j)$ and $join(e_i, e_k)$ are also independent (*independence propagation*). This comes from the fact that both $join(e_i, e_j)$ and $join(e_i, e_k)$ act on the same *register input state*, which is the modified original register input state for e_j and e_k.This hints to the property first described when defining *gadgets* in 2.1.4, which will be proven by induction in the section 2.4.3 using the statement from above.

Everything mentioned above can easily be generalized: by having $e_{i_0}, e_{i_1}, ...e_{i_k}$ *independent effects*, $join(e_{i_0}, e_{i_1}, ...e_{i_k}, e_j)$ is the operation in which multiple substitutions inside e_j take place, for every reg_out from e_{i0} up to e_{ik}. Notice that, for every permutation p of the first k arguments:

$$
\begin{aligned}
join(e_{i_0}, e_{i_1}, ...e_{i_k}, e_j) &= join(e_{i_{p0}}, e_{i_{p1}}, ...e_{i_{pk}}, e_j)\\
join(e_{i_0}, e_{i_1}, ...e_{i_k}, e_j) &\neq join(e_{i_0}, join(e_{i_1}, ...join(e_{i_k}, e_j)))
\end{aligned}
$$

From another theoretical point of view, the time complexity for the effect joining operation is of upmost importance due to the fact that (almost) every other algorithm implicitly uses this operation by a significant number of times (in most practical situations, at least thousands of times). The first thing to be said is that this operation is not explicitly implemented inside a function; rather, it is locally implemented in multiple places, but shares the same core idea, that is: for a join expressed as $join(e_{i_0}, e_{i_1}, ...e_{i_k}, e_j)$, for each of the e_{i_t}, every structured element from e_j is checked for its type, and if it is a reg_in, and if it corresponds to the reg_out of e_{i_t} it will be substituted. The time complexity could be written as ($|e|$ denotes the number of structured elements from an effect):

$$O(k|e_j| \max_{0 \le t \le k} |e_{i_t}|)$$

Fortunately, in practice, all of the 3 variables that take part in the previous equation are small: the number of elements in most of the effects is constant (1), the only exception being the ARITH type effects, which can encapsulate an arbitrary amount of structured elements. Because of that, it is expected that only a few instances of ARITH effects inside long chains that accumulated lots of operations will single handedly slow down the entire chain generation process - still, the depth of ARITH effects is indirectly limited by the chain length and other factors such as maximum stack depth. Another observation is that, due to the effect independence of each of the $e_{i_0}, e_{i_1}, ...e_{i_k}$, the number k is bounded by the number of registers for that architecture - for x86_64, $\sim$15 registers are taken into account, and

for ARM64 this number goes up to 31. If a register is not in the output state of that collection of effects, it can be implicitly introduced as the identity effect (2.1.2), proving that the time complexity can be rewritten as:

$$O(k|e_j| \max_{0 \le t \le k} |e_{i_t}|) = O(|R||e_j| \max_{0 \le t \le k} |e_{i_t}|) = O(|e_j| \max_{0 \le t \le k} |e_{i_t}|)$$

This helps underline the more "problematic" parts of the algorithms that are to be presented in the next sections. One more aspect is that, with more sophisticated implementations (say, for example, by using more references and hash tables), the complexity could be reduced so that some outlier ARITH effects would no longer represent a problem, but that would pose the problem of keeping the constant factor for the majority of trivial joins as small as possible.

2.2.2 Stack Joining

Stack joining is another operation that needs to be defined, that mostly takes place "in parallel" with the effect joining (2.2.1). It consists of merging two different stack views. A problem might appear when the stacks that are due to be joined might (partially) overlap: this happens when the end position for the stack pointer of the first stack is not at the end of the first stack view (fig 2). In the mentioned example, $S_{0q}, ...S_{0k}$ and $S_{10}, ...S_{1(k-q)}$ overlap, and need to be joined, analyzing element-by-element pairs of the form $(S_{0i}, S_{1(i-q)})$: if one of the elements from the pair has an assigned value and the other does not, and neither of them have any associated jumps, the final element will take the first element's value; if both of the elements have associated values which are different, then the join fails (3.2.5); another scenario in which joins might fail is when both elements from the pair have associated jumps - if the join would be forced, at least one of the jumps will "lose" that stack element, which would imply valid gadgets / chains can be invalidated, which means the validation procedure would need to be applied again, adding further complexity to the algorithms. Any other combination of assigned values and associated jumps is detailed in the implementation itself, but in short, those rules keep the consistency of the algorithms that process gadgets / chains, all while keeping computations to a minimum. In the end, if the stacks were successfully joined, the end stack pointer will also be updated, taking the value $SP_{joined} = SP_0 + SP_1$.

Regarding the "end SP" value, one might remark that the statement "SP is at the end of the stack view" is ambiguous - after all, the stack view is just a relative projection of what would be the real, runtime stack on which the payload resides. Thus, this projection can be extended "indefinetly". Still, the reason for which the presented arguments hold is because the stack view is *guaranteed* to contain (at least) all the stack elements that influence or interact in any way with the associated gadget (chain). Of course, in this sense, any "end SP" can be considered to end *before* the end of the stack view, as long as the stack view contains any excess of unused stack elements (i.e. 64b_stack_pad that are converted to stack values whenever is needed anyways, or 64b_stack_value with no associated value), but these cases are not considered - even if they were, the excess padding from the stack view would trivially be replaced by the "overlapping" elements from the second stack view.

The time complexity for the stack join operation is linear in the number of elements ($|s|$ denotes the number of structured elements from a stack view):

$$O(|s_i| + |s_j|)$$

2.2.3 Gadget (Chain) Joining

Gadget joining (or combinations of gadget-chain / chain-chain joining) is the operation by which both the effects and the stacks of the two gadgets (chains) are merged into a single chain. As with the effects

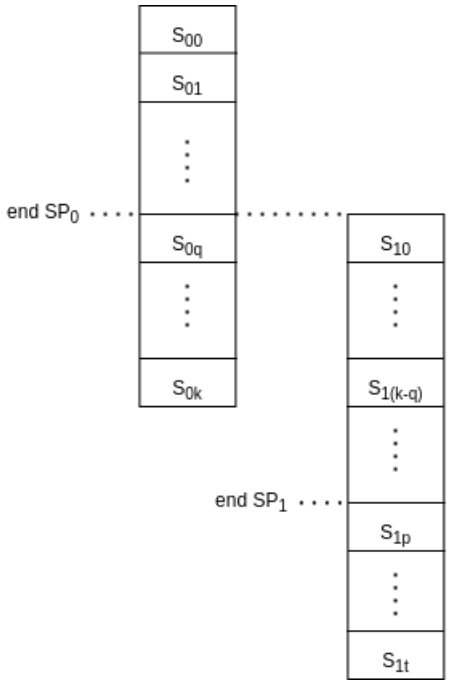

Figure 2: Stacks that overlap when joining

joining (2.2.1) and stack joining (2.2.2), the order of the two gadgets to be joined (passed as parameters) matters. The gadget joining can fail, mostly if the stack joining fails.

The joining between two valid gadgets / chains (2.1.7) is guaranteed to yield a valid chain; if one of the input gadgets / chains is not valid, the validity of the output varies from case to case, but the emphasis is put on the case in which the first input is valid, the second one is invalid, and the output is valid - this is the mechanism by which some invalid gadgets are converted in valid "gadgets" (they are actually valid chains, but can be treated as gadgets from the point of view of the chain searching algorithms because they are the smallest constructions that contain the initial gadgets and are also valid).

The time complexity for gadget joining will be analysed separately due to substantial differences between x86_64 and ARM64 implementations - although the final result is the same, the constant factor for ARM64 is bigger.

The time complexity for joining between g_0 and g_1 with stacks s_0 and s_1 is determined by the following operations (|| denotes the set of strucutred elements, and $||_e$ denotes the set of effects):

- stack and effect list duplication for g_0 and g_1: $O(|s_0| + |g_0| + |s_1| + |g_1|)$
- stack join between (copies of) s_0 and s_1: $O(|s_0| + |s_1|)$
- effect joining between effects of g_0: $e_0^0, e_1^0, ...e_k^0$ and effects of g_1: $e_0^1, e_1^1, ...e_t^1$ is summarized as in pseudocode 1; taking into account the rationale behind the time complexity for one single join operation (2.2.1), and also the inequalities:

Listing 1: Effect joining

```
1                    for j: 0 → t:
2                        join(e_0^0, e_1^0, ...e_k^0, e_j^1)
```

$$\max_{0\le i\le k}|e_i^0| \le |g_0| \le |R| \max_{0\le i\le k}|e_i^0|$$
$$\max_{0\le j\le t}|e_j^1| \le |g_1| \le |R| \max_{0\le j\le t}|e_j^1|$$

the complexity for this step is:

$$\begin{aligned} O(kt \max_{0\le i\le k}|e_i^0| \max_{0\le j\le t}|e_j^1|) &= O(|R|^2 \max_{0\le i\le k}|e_i^0| \max_{0\le j\le t}|e_j^1|) \\ &= O(\max_{0\le i\le k}|e_i^0| \max_{0\le j\le t}|e_j^1|) \\ &= O(|g_0||g_1|) \end{aligned}$$

- stack access validation call for each effect from the joined effects, and each deref corresponding to invalid derefs even if those derefs do *not* appear in joined effects list (as long as the deref elements are "accessed" inside the gadget, it does not matter if they do not appear in the independent effect list of that gadget): $O(S_{val}(|g_{joined}|_e + |\text{hidden derefs}|))$; when only the last gadget is invalid, it can be expressed as: $O(S_{val}(|g_1|_e + |\text{hidden derefs of } g_1|))$; S_{val} is the time complexity of a stack validation call, and is detailed in 2.2.7
- jump validation call for each jump effect from the joined effect: $O(J_{val}|g_{joined}|_e)$; when only the last gadget is invalid, it can be expressed as: $O(J_{val})$; J_{val} is the time complexity of a jump validation call and is detailed in 2.2.6

The analysis for the first 3 steps does not reveal much besides the conclusions drawn from analyzing the complexity of effect join operations by themselves: the number of the structured elements from inside the gadget is the dominant factor when considering gadget (chain) joining. Moreover, in practice, the last 2 steps mentioned above are only executed if at least one of the gadgets (chains) that need to be joined was previously invalid. And, given that the algorithms that work with invalid (or at least not yet validated) gadgets are called only once in the preprocessing phase (2.3), the time complexity for gadget joining can be written as:

$$O(join(g_0, g_1)) = O(|s_0| + |s_1| + |g_0||g_1|)$$

Although this upper bound is tight enough for the joining of 2 gadgets with a lot of ARITH effects, it becomes less meaningful when considering the joining of multiple gadgets:

$$O(join(g_0, join(g_1, ...join(g_{k-1}, g_k)...))) = O(\sum_{0\le i\le k}|s_i| + \prod_{0\le i\le k}|g_i|)$$

Because of that, and with support from the statistics presented in 3.2.2, the *average rate of element count growth* z can be defined, such that:

$$g_2 = join(g_0, g_1) \implies |g_1| \leq |g_2| \leq |g_0||g_1|,\ E(|g_2|) = z|g_1|$$
$$g_{fin} = join(g_0, join(g_1, ...join(g_{k-1}, g_k)...)) \implies |g_k| \leq |g_{fin}| \leq \prod_{0 \leq i \leq k} |g_i|,\ E(|g_{fin}|) = z^k|g_k|$$

Thus, the average time complexity for gadget joining operation is:

$$O(join(g_0, join(g_1, ...join(g_{k-1}, g_k)...))) = O(\sum_{0 \leq i \leq k} |s_i| + z^k|g_k|)$$

2.2.4 Matching Effects

The operation of *matching effects* decides whether two *non-JUMP* effects alter (or can alter) the reg_out in the same way. In other words, it checks whether two effects "do the same thing or not". By previously mentioning "can alter ... in the same way", it means that in some cases the associated stack values might be assigned during the process, if this ends up in the two input effects to match: for example, matching $e_0 : R_i \leftarrow 0xbeef$ (being a LOAD_CT effect), with $e_1 : R_i \leftarrow S_0$ (S_0 being a 64b_stack_value with no associated jump or value), the output of $match(e_0, e_1)$ will be $True$, but in the process, $value(id(S_0))$ will be asigned the value $0xbeef$. Having this in mind, the inputs to the matching operation are treated differently: the first argument is a *wanted effect* (2.1.9), and the second one is any other effect that is matched with - in the above mentioned example, executing $match(e_1, e_0)$ (although possible to implement) is not considered a valid call, because the e_1 would be considered a wanted effect, and at the same time the stack element S_0 does not have any associated value, thus rendering the final payload ambiguous (what bytes to use to fill R_i ?).

Matching between two effects that are of type MOV_RR, LOAD_CT or LOAD_S is done in $O(1)$ time, by checking their properties, and if the second effect is LOAD_S, its stack elements might be assigned values - note that here, LOAD_S refers only to valid memory accesses, and thus the parameter for such effects is always a stack element (and not a deref element or any arbitrary expression). For ARITH effects, a separate internal implementation is called, which determines whether (MOV_RR, ARITH) (LOAD_CT, ARITH) or (ARITH, ARITH) expressions match. The match can be either deterministic, by an exact match of the arithmetic tree, and taking into account only commutativity - this is the fastest match, that does not give false positives but might give false negatives; or, the match can be stochastic.

For the probabilistic matching between a wanted effect and an ARITH effect, the approach is simmilar to other probabilistic checks (2.2.6, 2.2.7): for a constant number of tests T, each of the registers from the input state and jump-related stack values are assigned a random uniform value from the range $[0, 2^{64} - 1]$, the execution of the effects is simulated, and the results are compared. If at least one different pair of results is found, the match returns False, otherwise the effects are considered to match. If the (second) matched effect contains stack values, the execution of the effect is done with the help of z3 module ([14]): the wanted effect is simulated, the result is taken, and the second effect is converted in a z3 expression, which is constrained to be equal to the previous result; if there are unassigned stack values, they can be assigned inside the match by the z3 solver, as exemplified in the first paragraph. The time complexity is (t_{sat} is the time of the z3 execution for checking the satisfiability of the system of T equations, and is constant when z3 is not used):

$$O(M_{check}) = O(T(|e_i| + |e_j|) + t_{sat}) = O(|e_i| + |e_j| + t_{sat})$$

The efficiency of this test, in terms of probabilities, can be analysed analogous to the analysis in 2.2.6, by taking into account the probability of a false match, but also the probability of the runtime register values to invalidate the match.

Listing 2: Fixed register check

```
1    for i: 0 → t:
2        for j: 0 → k:
3            if reg_name(e_j) = r_i and match(e_j, r_i ← r_i) is False:
4                return False
5    return True
```

2.2.5 Checking Fixed Register Values

This is a small subroutine used inside most of the searching algorithms, to check whether the resulting chain (gadget) keeps the register values unchanged. There are two versions: the fast check, which simply verifies the destination elements of the effects, and a more precise (but slower) check, that is described by the following: for a given gadget and its list of effects $e_0, e_1, ...e_k$, and the list of fixed registers $r_0, r_1, ...r_t$, the pseudocode 2 is used.

For the fast check, the time complexity is:

$$O(k) = O(|R|) = O(1)$$

And for the complete check, the time complexity is:

$$O(F_{check}) = O(tM_{check}) = O(|R|M_{check}) = O(M_{check})$$

2.2.6 Jump Validation

Jump validation is the algorithm responsible for both checking for validity and trying to validate any JUMP effect e_{jmp} in relation with its gadget (chain) g. The existence of such algorithm is necessary at least from the point of view of checking the validity, and, although not necessary from the point of view of validating (creating valid) gadgets, it is nonetheless very useful. The algorithm has two variations: a deterministic approach and a stochastic approach.

The deterministic approach has a very limited yield in terms of how many gadgets are recognised as valid (or validated), and thus it is not preferred: it simply checks whether the expression associated with the JUMP effect is a 64b_stack_val and whether it is associated with any other JUMP effect; if both conditions are met, the JUMP is considered valid, otherwise it is considered invalid. The main advantage is that its time complexity is $O(1)$, and that there are never "false valid" gadgets.

The stochastic approach (which is the one used by default) first makes the same checks as the deterministic approach, so that it is as fast as possible when those criteria are met. If not, the following approach is taken: for a constant number of probabilistic tests (the implementation uses the constant 30), the jump address is assigned a random value in the range $(0, 2^{64} - 1]$, and then it is checked whether that value can be obtained solely by assigning values to the stack elements associated with this JUMP effect; the input state registers and the other-jump-related elements are assigned random values - although this assumption might significantly skew the following results, predicting the input state values for these elements is either impractical (by analyzing the entire execution of the binary itself), or even impossible. In an ideal scenario, the whole range of 64 bit values would be checked, but it is obviously impractical.

In this paragraph an attempt at describing the probability of failure (second point in the following list) is made, and, while one could argue it is too inaccurate, the efficiency is supported by empirical evidence (no failed jump has been observed), and also by the fact that most jumps are trivial (they only have one stack element associated with them); statistics in this sense are presented in 3.2.6. To further

simplify the analysis, it will be considered that any address randomization module chooses addresses from the uniform distribution in any arbitrary given range, disregarding any other possible restriction. Moreover, the module that checks the satisfiability of a single test round is considered to be exact and to have the false positive rate of 0 (in the implementation, z3 ([14]) is used, but it can be substituted by any other mechanism). There are a few questions that can be formulated:

1. given the probability p of chosing both an assignation for the input registers and a 64 bit address that invalidates the jump, what is the probability of testing such a tuple in at least one of the T random tests
2. what is the probability of the validation algorithm to give a false positive, and, at the same time, for ASLR to select an address that, in combination with the register values at that moment, it will invalidate the jump, such that the payload building process will fail (at least up to some point - for disambiguation, check the payload building stage 2.5.7)

The first question can readily be answered by the geometric cumulative distrubution function:

$$p_1 = 1 - (1-p)^T$$

The second question arises from the fact that not only the accuracy of the validity test itself is important, but also by taking into account *how important* is to have a "correct" validity answer. For example, if the JUMP effect supports jumping to any 64 bit address, except exactly one single address for a small number of register input values, the probability of a runtime address to be equal to that single address that invalidates the jump is sufficiently small such that it can be ignored. Moreover, when building the payload bytes from the abstract representation (2.5.7) and fixing the jump, the validity will be checked for that specific address anyways, such that in worst case scenario, the gadget is falsely treated as valid, up until the last building steps when the "mistake" is found and the gadget is discarded, so that the probability of failure is even smaller. If we consider the probability of the real address chosen by ASLR at runtime, in combination with the actual register input values to invalidate the jump to be equal to p, then the second probability can be calculated as follows:

$$\begin{aligned} ra &:= \text{(real (runtime) address, actual register values)} \\ fv &:= \text{false validity outcome} \\ Inv &:= \text{set of tuples (address, register values) that invalidate the jump} \end{aligned}$$

$$\begin{aligned} p_2 &= P((ra \in Inv) \wedge fv) \\ &= P(ra \in Inv)P(fv) \text{ (the two events are independent)} \\ &= p(1-p_1) \\ &= p(1-(1-(1-p)^T)) \\ &= p(1-p)^T \end{aligned}$$

If $p = 0$, the gadget will always be vaild, and no other analysis is required, and if $p = 1$, it is impossible for the test to give a false positive. If $p \in (0,1)$, consider:

$$\begin{aligned} &f : (0,1) \to [0,1], f(x) = x(1-x)^T, \text{ where } T \in \mathbb{N}^* \\ &\frac{\partial f}{\partial x} = (1-x)^T - Tx(1-x)^{T-1} = 0 \iff x = \frac{1}{T+1} \end{aligned}$$

The function f is not constant and the derivative is 0 only in $\frac{1}{T+1}$, and thus the global maximum value is given by $f(\frac{1}{T+1}) = (\frac{T}{(T+1)^2})^T$. For $T = 30$, which is the value used in the implementation, we get that

the maximum probability of a failure (p_2) is around 1.2%. For $T = 100$, the same probability would be 0.37%, which is considered "good enough" for this use case.

One alternative option to the stochastic algorithm, considering that the z3 solver was used, was to formulate a logical statement using the universal quantifier, and letting the solver decide the validity all by itself. One downside of such approach would be the an increased instability in the running time of the algorithm, and discarding the observation related to the failure probability (p_2) - still, this comparison remains unclear.

The time complexity for this algorithm is

$$O(J_{val}) = O(T(|e_{jmp}| + t_{sat})) = O(|e_{jmp}| + t_{sat})$$

t_{sat} is the time complexity for checking the satisfiability of the equation given by the jump, including the time taken to assign values to the corresponding jump-related stack values (if it is the case). The explicitation for t_{sat} is beyond the scope of this documentation (a more throughout analysis of the mechanism of action of z3 solver should give an answer to this question), but, overall, it should be taken into account that such an operation is executed T times every time the jump validator function is called.

2.2.7 Stack Access Validation

The *stack access validation* function is in many ways analogous to the jump validation function (2.2.6). It consists of two variations, both being probabilistic. For a deref structured element d with an arbitrary expression inside it, both of those algorithms try to solve an equation that has the following form:

$$[SP + c] = d \iff SP + c = expr(d) \iff c = expr(d) - SP$$

In other words, by subtracting the stack pointer's value (whatever it might be) from the value of the expression that is dereferenced, it needs to be equal to the same constant c everytime. This property ensures that the stack element that is accessed is fully defined w.r.t. the corresponding stack view of the gadget that contains the effect that contains the deref element. The mechanism that is implemented for trying to find such an offset c is the following: for a constant number of K rounds random values are assigned to input state registers (and to SP) and to any jump-related stack elements; in the first approach, the non-jump-related stack elements without value are also assigned random values, and in the second approach (the one used by default) another module (z3) is used to use these stack elements as variables that can be assigned with specific, arbitrary values; after these K sets of equations are introduced, a few more conditions are taken into account: $c\%8 = 0$ (for alignment) and that c should be different from any other offset that corresponds to any jump-related stack element or stack element with assigned (non-null) value, and, lastly, $c <$ MAX_STACK_SIZE. If all of those conditions are met, the value c is extracted, after solving the equations (with z3 for second approach, or simulating execution K times, subtracting SP from all of the results and then comparing the final values, for the first approach). If the second approach was used, the values assigned to (some or all) stack elements are kept, and the entire d deref element is replaced with a 64b_stack_val that has the same id as the stack element on the position (offset) c from the stack view.

Given the stack s associated with the current gadget (and jump), the time complexity for this operation is:

$$O(S_{val}) = O(K|d| + |s| + t_{sat}) = O(|d| + |s| + t_{sat})$$

t_{sat} is the time complexity for checking the statisfiability of the system of K equations with the z3 module. The main difference compared to jump validator's time complexity is that the statisfiability check is done only once, instead of K times.

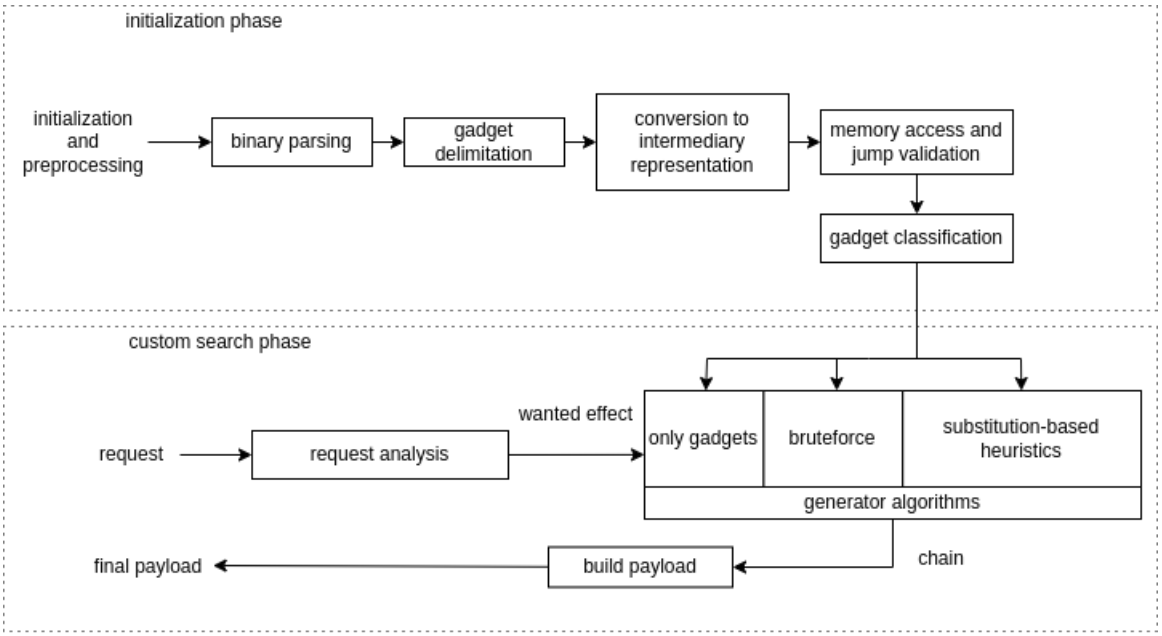

Figure 3: Logic flow

2.3 Logic Flow

The logic flow of the generator (fig 3) can be summarized as follows:

- *initialization phase* in which gadgets are extracted from the binary, converted in an intermediary representation (2.1.4) and are then stored accordingly, so that they can be efficiently accessed later; optionally, gadget validation (2.1.7) takes place here; this phase is executed only once, before any custom search takes place
- *custom search phase* represents the entire search process of a custom chain (or gadget); this phase takes place as many times as the user requests, while the algorithms used take advantage of the preprocessing from the *initialization phase*; it should be noted that different runs of the *custom search phase* are completely independent of one another, and do not influence the results or the performance of any subsequent searches

In the following sections, the preprocessing algorithms (2.4) will be presented, then the search algorithms (2.5). Besides a description, a theoretical time complexity analysis is given, in parallel with the runtimes observed in practice - statistics for the latter claims are presented in section 3. In terms of the assumptions used for all of those routines, they can be formulated as follows:

- The number of gadgets after the preprocessing phase does not surpass $\sim 10^3 - 10^4$ (unique) gadgets.
- The number of effects of a given type and with a given reg_out does not surpass $\sim 10^3 - 10^4$.
- A lot of combinations (effect type, reg_out) have a small (≤ 100) number of gadgets, and a lot of gadgets are clustered inside a few (effect type, reg_out) slots.

- Most of the wanted effects that load values in commonly used registers (registers that are usually part of ABIs and are used to pass integer arguments) can be satisfied just by using gadgets that directly load values from the stack, and thus leaving room for more computations to find more complex effects.
- The usage of search filters and constraints is used in limited amount; for more details, check the search API (2.5.1) and payload building API (2.5.7)

Most of these assumptions can be readily verified by analyzing the distribution of gadgets w.r.t. the destination register and effect type, as in 3.2.1.

2.4 Preprocessing Algorithms

2.4.1 Binary Parsing

The binary parsing module is straightforward and simple: it opens the binary, searches for the ELF magic number in the header and some other properties such as the architecture, finds the offsets for the executable segment(s), and extracts them as an array of bytes. One interesting finding is that for ARM64 ELF binaries, the executable region starts at the beginning of the binary itself, which could theoretically mean the entire header section can be searched for ROP gadgets to be executed. Still, a quick disassemble of such binaries that were *not* specifically crafted to contain gadgets in the headers do not show any spontaneous emergence of any gadget. This stage is usually very fast and takes $O(L)$ time, where L is the binary length in bytes.

2.4.2 Gadget Delimitation

Also being one of the simplest parts, the "delimitation" refers to the delimitation of the instruction sequences that define a gadget: at first, the location where the gadgets *end* is established, then, for each of those endpoint locations, an index is walked backwards until either an unsupported instruction is found (eg. a memory write) or an upper limit for the gadget length is reached. For each of those gadgets, the conversion step is executed (2.4.3), thus rendering the time complexity at:

$$O(|G|C_g I + L)$$

$|G|$ is the number of (candidate) gadgets, $I = \max_{g \in G}(len(g))$ is the maximum number of bytes in a gadget, differing by a constant when compared to the number of instructions per gadget (can be considered constant since the actual implementation imposes an upper bound for it), C_g is the time complexity for the next step (2.4.3), and L is the binary length in bytes.

2.4.3 Conversion to Intermediary Representation of Gadgets

This is the procedure in which byte sequences are turned into gadget representations, as defined in 2.1.4. First individual instruction is taken and converted into an *unordered* list of effects (2.1.2), whose effects are independent of one another. For example, on x86_64, `xchg rax, rcx` instruction is converted in [MOV_RR: `rax` ← `rcx`, MOV_RR: `rcx` ← `rax`], and on ARM64, `ret` is converted in [JUMP(1, `x30`)] (1 is the preliminary jump id for every newly created JUMP effect). This is the stage in which unsupported instructions are detected: if this is the case, the entire candidate gadget is discarded, and the process continues as detailed in 2.4.2.

After converting each instruction into a list L_i of effects, an algorithm that makes use of the *effect joining* operation (2.2.1) is applied as in pseudocode 3.

Listing 3: Effect joining at conversion

```
1
2     input := [L_0, L_1, ... L_I]
3
4     acc = []
5     for i: 0 → I:
6         t = len(acc)
7         new_acc = []
8         for j: 0 → len(L_i):
9             new_acc = new_acc + join(acc[0], acc[1], ... acc[t], L_i[j])
10        acc = new_acc
11
12    return acc
```

This is the code that creates every new gadget. Because of that, a short proof by induction for gadget *effect independence* will be given: The base case is assured by the conversion of each individual instruction to its effect list (every L_i), as mentioned at the beginning of this subsection - this fact is also needed for the generalized *join* operation to be executed (see 2.2.1). The induction step consists of applying the property of *independence propagation* (again, see 2.2.1) of the join operation: all L_i`[j]` from L_i are independent of one another, and each effect from `acc` (obtained from the previous step) contains independent effects, thus all `join(acc[0], acc[1], ... acc[t],` L_i`[j])` are independent of one another, which constitute the `acc` list of effects for the next step.

The above pseudocode illustrates only the manipulation and joining of effects, which can be considered the most important part. That being said, the actual implementation needs to take into consideration the construction of the stack view that is going to be associated with the newly formed gadget, and a first round of checking the validity for the associated jump and the validity for all memory accesses for the current effects takes place. Taking all of that into account, the time complexity is:

- for effect joining algorithm:

$$O(I \max_{0\le i\le I} len(L_i) \max_{0\le i\le I, 0\le u\le t_i} t_i \max_{0\le i\le I, 0\le u\le t_i} (|acc_i[u]|) \max_{0\le i\le I, 0\le j\le l_i} (|L_i[j]|))$$

  where all the variables used have the same meaning as described in the pseudocode. This expression can be further simplified: first, the l_i and t_i can be considered constants, by introducing identity effects where the reg_out is absent; the number of structured elements from any L_i can also be approximated as constant, because each different instruction has at most 3 effects and each of the effects has a constant maximum number m of possible elements, (because these effects reflect exactly one instruction, and thus the arithmetic tree is also limited in depth); nonetheless, the number of elements from each effect from `acc` is bounded by the number of instructions in that gadget: m^I. Applying these simplifications, we obtain:

$$O(Im^I)$$

  Note that the case when the number of elements reaches values comparable to m^I corresponds to the extreme case where all effects are ARITH effects operating on the same gadget, and is virtually never encountered; for statistics regarding the average number of elements inside an ARITH effect, check 3.2.2

- for building the stack view element by element, up to the SP end position: $O(SP_{end})$
- stack access validation calls for all the effects obtained, and a jump validation call:

$$O(S_{val}(|acc_{final}|_e + \text{hidden derefs}|) + J_{val})$$

Summing all of these, we obtain the complexity for the conversion operation C_g. Combining with the complexity detailed in 2.4.2, and that of the binary parsing (which is constant), an approximation of the overall time complexity for creating the initial gadgets from the raw binary is:

$$O(L + |G|I(Im^I + \max_g SP_{end} + S_{val}(|acc_{final}|_e + \text{hidden derefs}|) + J_{val}))$$

Although I appears to have the biggest influence, it fluctuates in a relatively small amount; the biggest performance impact is made by substantially increasing the gadget count), which is a natural consequence of analyzing a bigger binary file.

2.4.4 Gadget Validation

This algorithm aims to validate particular cases of gadgets that were extracted from the binary in an invalid form. The validation of stack accesses and jumps is done in two different stages that work in an analogous way. To define these steps and the goal of the algorithm, let us define:

- the set of elements of all elements from an effect, without the destination element: $elems(ef)$
- the set of all valid gadgets: V, the set of all invalid gadgets: IV
- the *requirement set* Q_g for a gadget $g \in IV$:

$$Q_g^{SP} = \{r_{in} \mid \exists deref \in g.elems \text{ s.t. } r_{in} \in deref\}$$
$$Q_g = \{r_{in} \mid r_{in} \in Q_g,\ r_{in} \neq SP\}$$

- the *advertisement set* A_r for a register (name) r:

$$A_r = \{g \mid g \in V,\ \exists ef \in g.efs \text{ s.t. reg\_name}(ef.dest) = r \text{ and } \forall r_{in},\ r_{in} \notin elems(ef)\}$$
$$A_r^{SP} = \{g \mid g \in V,\ \exists ef \in g.efs \text{ s.t. reg\_name}(ef.dest) = r \text{ and}$$
$$\forall r_{in} \neq SP,\ r_{in} \notin elems(ef) \text{ and } SP \in elems(ef)\}$$

- a *substitution* γ_g for an invalid gadget $g \in IV$ such that $Q_g = \{r_0, r_1, ...r_k\}$:

$$\gamma_g = (g_{v_0}, g_{v_1}, ...g_{v_k}),\ \forall 0 \leq i \leq k,\ g_{v_i} \in A_{r_i}$$
$$\gamma_g^{SP} = (g_{v_0}, g_{v_1}, ...g_{v_k}),\ \exists! 0 \leq j \leq k \text{ s.t. } g_{v_j} \in A_{r_i}^{SP},\ \forall 0 \leq i \leq k,\ i \neq j,\ g_{v_i} \in A_{r_i}$$

- the set Γ_g of all γ_g and the set Γ_g^{SP} of all γ_g^{SP}, for a given $g \in IV$

In short, the algorithm tries to validate a gadget $g \in IV$ by first finding a substitution $\gamma_g^{SP} \in \Gamma_g^{SP}$ (if the gadget has invalid stack accesses), or $\gamma_g \in \Gamma_g$ (if the gadget has invalid jump) of the form $(g_{v_0}, g_{v_1}, ...g_{v_k})$, to join the gadgets from the substitution, and then join with the gadget g:

$$sub_i = join(g_{v_k}, ...join(g_{v_2}, join(g_{v_1}, g_{v_0}))...)$$
$$ch_i = join(sub_i, g)$$

The chain ch_i goes through a validation attempt inside the $join(sub_i, g)$ operation (2.2.3) - if it comes out to be valid, it is added in the V set, and then the whole process is repeated, until there are no validations inside one single round.

The intuition is the following: the expressions that need to be validated for both stack access and jump need to take a value that can be controlled by the payload builder; if the expression were to be composed only from constant values and stack values (that can be assigned whatever value the algorithm considers it is needed), the issue of controlling the expression value would most likely never be a problem - when reg_in elements come into play, their values are not (and cannot be) known to the payload builder, and thus they must be treated as random variables. The other elements that can pose a problem are the jump-associated stack values, but pretty much nothing can be done about them; this leaves the manipulation of reg_in elements as the only realistic method of trying to validate a gadget. Naturally, the easiest way of "getting rid" of them would be to replace them with a constant or non-jump-related stack values, which, as state previously, represent an ideal case. Multiple ways of creating such substitutions can be considered, such as:

- masking the register values in cause with effects such as $R_i \leftarrow R_i$ & 0 or $R_i \leftarrow R_i$ xor R_i; these kind of "masking" or "reducing" effects can happen on a per-instruction base, case in which they can be detected and automatically converted by the gadget conversion phase 2.4.3 (which is actually what happens with r xor r, for example), or they can emerge from joining multiple gadgets, case which can not be readily handled, and would require a heuristic of its own - based on that, it can be concluded this is not a practical solution
- more exotic ways of predicting the values inside the registers might be used, such as analyzing the instructions in the binary that precede the gadget in cause; these methods were simply not tested to draw any meaningful conclusion
- the easiest way, and, fortunately, the most promising one, would be to search for gadgets that execute the substitution "manually"; this is the method that was chosen for the implementation of this algorithm; multiple ways of implementing it exist: on a per-effect level, or on a per-gadget level; also, the substitution can be direct or indirect: for substituting a register R_i, the substituting effect like $R_i \leftarrow CT_0$ can either be found inside a single gadget, or build a chain that obtains it, for example by joining $R_j \leftarrow CT_0$ with $R_i \leftarrow R_j$; as it can be seen from the definitions given at the beginning of this section, the implementation considers only direct substitutions on a per-gadget level, mainly because it is the fastest way; empirical evidence supports its efficiency (3.2.3)

One problem arises, though. Let $g \in IV$ with $Q_g = \{r_0, r_1\}$ (emphasis on $|Q_g| > 1$). Then, for any $\gamma_g = (g_{v_0}, g_{v_1}), \gamma_g \in \Gamma_g$, such that g_{v_0} substitutes r_0 and g_{v_1} substitutes r_1, and the final chain ch_i is obtained as $ch_i = join(join(g_{v_1}, g_{v_0}), g)$. There is no guarantee that $\nexists e \in g_{v_0}$ such that reg_name(r_1) $\neq$ reg_name(e.dest_elem) or $e \in A_{r_1}$. In other words, for any substitution chain consisting of more than one single gadget, there is no guarantee that the substitutions from any gadget (except the last one to be joined) will remain "unaltered" so that they can be joined with the invalid gadget. For every ragister R_i, given the probability p_{R_i} of appearing as a reg_out in the list of effects for a gadget $g \in V$, the probability of having at least one of the k substituting effects replaced by other effect is $(1 - p_{R_0})^0(1 - p_{R_1})^1...(1 - p_{R_k})^k$, which increases exponentially with the number of substitutions. Of course, an actual analysis would look more complicated, because the substituting effects might be replaced by other, equivalent (or even more advantageous) substituting effects by mistake, so the probability from above is the worst case rather than an expected falure rate, but still, it shows that the probability of successfully validating a gadget decreases with the number of registers that need to be substituted. This problem is not directly addressed in the implementation, but rather the algorithm only tries to remove any bias when selecting the substitutions: trying every $\gamma_g \in \Gamma_g$ is obviously impractical, due to the huge number of possible

Listing 4: Gadget validation

```
while (successful validations occured last loop):

    for v in V:
        if v not previously checked:
            check if v can be added in any A_{r_i}

    for iv in IV:
        check if |Γ^SP_iv| > 0
        for γ^SP_iv = (g_{v_0}, ... g_{v_k}) in choices(Γ^SP_iv):
            ch_i = join(g_{v_k}, ... join(g_{v_2}, join(g_{v_1}, g_{v_0})) ... )
            ch_i = join(ch_i, iv)
            if ch_i is valid:
                V.add(ch_i)
                IV.remove(iv)
```

substitutions ($|\Gamma_g| = P_k|A_{r_0}||A_{r_1}|...|A_{r_k}|$, for every $g \in IV$), so a fixed number q of substitutions are chosen at random.

Regarding the validation of stack access versus the validation of jumps, the main difference is that γ_g^{SP} substitutions are used instead of γ_g, and the associated definitions presented at the beginning of this section. This is because to validate a stack access, the presence of SP register as a reg_in is necessary to obtain an expression equivalent with $SP + c$, as detailed in 2.1.6 and 2.2.7. The substitution is executed at a gadget level, so this heuristic clearly does not cover all cases, but forcing more SP registers to be present in the substitution would introduce additional logic on its own which would further complicate any attempt at analyzing the algorithm's behaviour.

Disregarding a lot of the implementation details (such as data structures used to store the substitutions and the other sets, for efficient access), the pseudocode for stack access validation looks like in 4 (the jump validation is analogous and follows the stack access validation phase).

The time complexity will be analysed from two perspectives: the worst case (theoretical upper bound) which could be reached, but with a negligibly small chance, and the average time complexity, which depicts the actual runtimes much more accurately.

First, some observations about the size of the gadget sets:

$|IV| \approx c|V|$
$|V_{final}| = |V| + q|IV|$, where q is the constant number of substitutions tried per iv gadget
$O(|G|) = O(|IV|) = O(|V|) = O(|V_{final}|)$

The time complexity can be generally written as:

$$O(|G|(t_{advertise} + |G| + |G|t_{choice}t_{join})) = O(|G|t_{advertise} + |G|^2 t_{choice}t_{join})$$

The t_{choice} time is the time spent finding a substitution γ_g for a specific gadget g. Due to the probabilistic nature of this routine, besides limiting the number of substitutions to q per gadget, the number of failures is also limited to 3 (i.e. the current substitution to be yielded had already been tried, or some maximum stack size is surpassed):

$$t_{choice} \leq 3q \max_{g \in G} |Q_g| \leq 3q|R| \implies O(t_{choice}) = O(1)$$

The $t_{advertise}$ and t_{join} both depend on the number of elements inside any gadget, be it valid from the beginning or validated on the course of this algorithm. The worst case time complexity for the join operations are given by (as shown in 2.2.3):

$$O(|R| \max_{s \in g.stacks} |s| + \max_{g \in G} |g|^{|R||G|} + S_{val} max_{g \in G}(|g| + |\text{hidden derefs of g}|) + J_{val})$$

Notice the $|G|$ exponent in the product term: this is because the joining rounds can use validated gadgets from the previous rounds of validation, and thus not only the number of joins at a time matters (at most $|Q_g| \leq |R|$ joins). The stack sizes can be considered not affected since there is an upper bound that limits the possible stack size way before any exponential explosion takes place.

Considering the average rate of element count growth z, the upper bound for the average time complexity for the join is:

$$O(|R| \max_{s \in g.stacks} |s| + z^{|R|} \max_{g \in G} |g| + S_{val} max_{g \in G}(|g| + |\text{hidden derefs of g}|) + J_{val})$$

Analogously, the $t_{advertise}$ term can be expressed either as $\max_{g \in G} |g|^{|R||G|}$ or $z^{|R|} \max_{g \in G} |g|$. Finally, the average time complexity for the whole validation algorithm can be expressed as (stack size and validation steps can be considered negligible compared to gadget element count):

$$\approx O(|G|^2 z^{|R|} \max_{g \in G} |g|)$$

2.4.5 Mov Map and Transition Path Finder

The mov map building is also part of the preprocessing phase. The mov map (also called transition graph) serves multiple purposes that will be described later. It aims to create a memory data structure that can answer in $O(1)$ whether the statement $ispath_{ij}$ is true or false, defined as:

$$\exists path_{ij} = (g_0, g_1, ...g_k) \text{ such that } e_{i \leftarrow j} : R_i \leftarrow R_j \in join(g_0, ...join(g_{k-2}, join(g_{k-1}, g_k))...)$$

In other words, it checks for each register pair (R_i, R_j) whether R_j can be substituted by R_i. In practice, it does not succeed to do that for every $ispath_{ij}$ pair; this inconvenience unwillingly arises (as many other complexities in this documentation) from the arbitrary and unpredictable nature of ARITH effects. In this sense, a generalization is applied for the definition of $ispath_{ij}$, such that not only effects that have exactly the structure $e_{i \leftarrow j} : R_i \leftarrow R_j$ are taken into account, but also effects that *match* (2.2.4) with them.

Due to practical considerations (described in substitution-based heursitics sections: 2.5.3, 2.5.4, and 2.5.5), the actual implementation also defines a constant number of subgraphs, with each subgraph having the constaint that one or more registers are fixed; the number of subgraphs is 1 (no fixed register) $+|R|$ (for each supported register) $+|R|(|R|-1)/2$ (for every pair (R_i, R_j), $j > i$ of supported registers):

$$ispath_{ij} = \exists path_{ij} = (g_0, g_1, ...g_k) \text{ such that } e_{i \leftarrow j} : R_i \leftarrow R_j \in join(g_0, ...join(g_{k-2}, join(g_{k-1}, g_k))...)$$
$$ispath_{ij}^{R_u} = ispath_{ij} \text{ and fixed\_reg}(R_u, g_q) = True \forall g_q \in path_{ij}^{R_u}$$
$$ispath_{ij}^{(R_u, R_v)} = ispath_{ij} \text{ and fixed\_reg}(R_u, g_q) = True \text{ and fixed\_reg}(R_v, g_q) = True \forall g_q \in path_{ij}^{(R_u, R_v)}$$

The algorithm is mostly just a particularization of Roy-Warshall algorithm to determine the matrix path of a graph: the nodes are all the registers specific for that architecture, and the edges are (valid)

Listing 5: Roy-Warshall

```
1        for k: 0 → t:
2            for i: 0 → t:
3                for j: 0 → t:
4                    if ispath_ik^fixed and ispath_kj^fixed:
5                        ispath_ij^fixed ← True
```

gadgets that contain an effect that is, or matches with $e_{i \leftarrow j} : R_i \leftarrow R_j$. To discover these kind of effects, the algorithm for searching gadgets with the wanted effect (2.1.9) $e_{i \leftarrow j}$ - the algorithm is described in 2.5.2. After the edges of the graph are discovered, the Roy-Warshall algorithm is applied (for each possible $fixed$ subgraph, including the graph itself), as in pseudocode 5.

It is easy to see that when $ispath_{ik}^{fixed}$ and $ispath_{kj}^{fixed}$ are both true, by joining $e_{k \leftarrow j} : R_k \leftarrow R_j$ (or equivalent) with $e_{i \leftarrow k} : R_i \leftarrow R_k$ (or equivalent) the result is $e_{i \leftarrow j} : R_i \leftarrow R_j$, which is equivalent to $ispath_{ij}^{fixed} = True$.

The algorithm does not keep the shortest path (in terms of number of gadgets) or anything simmilar, except for the list of edges from one node directly to the other. This is because every chain formed by joining $path_{ij}^{fixed}$ gadgets (chains) might alter an arbitrary set of registers, which might affect the results of the algorithm. Because of that, when a path is needed, an algorithm that searches the graph for every possible path to the destination is used; it is true that such a function could be used *without* the path matrix that was calculated above, but executing this stage before still optimizes further algortihms, by providing an instant answer on whether there is a path from any R_j to any R_i, without wasting time searching for a possible path to simply check if there is any of such kind or not, and also this stage precomputes the edges of the graph, without the need to do that every time.

The time complexity for searching the edges is $O(|R|^2 G_{search}) = O(G_{search})$ where G_{search} is the time complexity for searching gadgets (2.5.2) with specified semantics (in this case, effects of type $e_{i \leftarrow j}$). The time complexity for matrix path is $O(|R|^3(1 + |R| + |R|(|R| - 1)/2)) = O(1)$.

The algorithm (implemented as a generator) that yields all paths is bounded by ($|gpath_{ij}|$ indicates the number of gadgets (*not chains*) that satisfy $path_{ij}$):

$$O(P_{find}) = O(\max_{0 \le i \le |R|, 0 \le j \le |R|} (|gpath_{ij}|)^{|R|} |R|!) = O(\max_{0 \le i \le |R|, 0 \le j \le |R|} (|gpath_{ij}|)^{|R|})$$

Even considering that $|R|$ is constant, the runtime of generating all the paths for dense graphs would surpass any reasonable threshold. Fortunately, in practice, most of the time the graph is sparse which allows this procedure to finnish running in a relatively short amount of time (3.2.4).

2.4.6 Gadget Classification

The algorithms that follow access the gadgets in various ways which benefit from the existence of different data structures populated in the preprocessing phase:

- *effect to gadget map*, which stores references to gadgets in buckets delimited by the destination element of an effect and the effect type; a gadget can have references in one or more such buckets
- *mov map (transition graph)*, as detailed in 2.4.5
- *load ct map*, which simply stores for each supported register a dictionary containing every constant loaded with a LOAD_CT effect inside that register, and a list of the gadgets that contain such an effect. As with the mov map, it stores "submaps" that have the fixed register constraints

- *load s map*, which stores for each supported register, a list of gadgets that have a LOAD_S effect with that register as destination
- *arith freestack map*, which stores for each supported register, a list of gadgets that contain an ARITH effect with at least a free (i.e. unassigned, without associated jump) stack element, and which has that register as destination
- *snapshot map*, which stores for each supported register, a dictionary of *snapshot keys* and a list of gadgets that have ARITH effects with these snapshot keys; a snapshot key is defined as a tuple of values for an ARITH effect with no free stack elements, after being repeatedly "executed" with random input state registers (and random jump-related stack elements)

Besides mov map, the most complex map is the snapshot map, which was built with the intention of speeding up the ARITH gadget search: in the preprocessing phase, a few (3) random input register states are imposed; then, when searching for an ARITH wanted effect w, the snapshot key is generated by executing w with the global (random) input register states, then checking the snapshot map for that key. This can only be implemented for the gadgets with ARITH effects with no free stack variable, because a variable on the stack can have an arbitrary controlled value on which the execution depends, and thus a snaphsot key no longer makes sense - because of that, the efficiency of the snapshot map depends on the proportion of "free stack ARITH effects" over the total number of ARITH effects (check 3.2.7 for a statistic in this sense).

2.4.7 ARITH to MOV_RR Conversion

This is an optional step, part of the preprocessing phase. It aims to enrich the register transition graph, presented in 2.4.5, by providing new paths of transferring values from one register to another, that previously did not have any such path (or, better said, it was not discovered). Normally, when the transition register is built, only gadgets that match a specific MOV_RR effect are used. This algorithm comes in aid for this, by adding new "gadgets" to the preprocessed set of gadgets, so that the transition register graph can be recalculated without even "being aware" that this algorithm had been ran before. The concepts used are exactly the same as the ones used for gadget valudation (2.4.4), and the implementation is a direct adaptation of the previously mentioned procedure - because of that, they will not be re-iterated here. In short, it searches through valid gadgets (including the ones obtained from 2.4.4), and tries to find ARITH effects that contain in their exprression registers R_i such that for the destination register R_d of the current effect, $ispath_{di}$ is false, and then tries to find gadgets that, after being joined with the current gadget (containing the said ARITH effect), they will replace every other reg from the ARITH effect with a constant or a stack element, such that in the end the new chain will hopefully contain an effect that matches with $e_{d \leftarrow i} : R_d \leftarrow R_i$; because the transition graph is already built before this algorithm starts running, it could make use of the subgraphs that keep some registers fixed to ultimately increase the yield, but this remains unimplemented. In the end, after a number of such attempts is made, the transition graph can be recalculated, and it will automatically include any new "gadget" (chain) that was built during this phase.

It should be noted that *no* result presented in section 3.2 makes use of this feature, and thus its performance remains largely untested. Still, from the point of view of its performance, it should benefit from the same conclusions drawn that help the gadget validation algorithm (2.4.4) run relatively fast, compared to the worst theoretical running time.

2.5 Search Algorithms

2.5.1 Search API

This is the first step executed after a search call is made, and is responsible for converting the input received through the API of this module into an internal representation. In a typical session, first, a `ROP_util` object must be initialized, giving the name of the binary to be analysed. Then, the preprocessing phase (2.3) begins by calling the `scout_for_gadgets()` method of the ROP utilitary object. Then, to search a chain, the `search_chain()` method is called, which can receive the following arguments:

- `wanted_effects`: a string that encodes the wanted effects; a wanted effect is defined as 2.1.9 (no default value)
- `fixed_regs`: a list of registers that need to keep their values unaltered after the execution of the chain (default empty list)
- `reg_start_values`: a string that encodes the start effects that are joined with any gadget (chain) before checking its properties (default empty string)
- `max_stack_byte_size`: the maximum stack byte size (default 2^{63}, equivalent to unlimited); this observation holds even if SP's end position is past the stack view
- `max_search_cnt`: maximum number of chains to be yielded (default 1)
- `only_gadgets`: whether to search only through gadgets, or also try to build chains (default false)

A subset of these arguments will be found in every searching algorithm; their properties are explained the same, and are explained above. There are some restrictions and also some recommandations for these parameters (for a more detailed explanation, read the sections describing the search algorithms: 2.5.3, 2.5.4, 2.5.5):

- it `only_gadgets` is false, `wanted_effects` must encode exactly one single wanted effect
- if `only_gadgets` is false, `reg_start_values` must be kept empty ("")
- if `only_gadgets` is false, the wanted effect should not contain more than 3 elements inside its expression on the right of "="
- if `only_gadgets` is false, `fixed_regs` should not contain more than 3 elements

The strings `wanted_effects` and `reg_start_values` (with every whitespace removed) are encoded by the following grammar:

S: $R = E \mid , S \mid \lambda$
E: $R \mid I \mid (EBE) \mid E$
B: $+ \mid - \mid \& \mid \hat{} \mid \mid \mid * \mid << \mid >>$
R: rax | rbx | ... | r15 (or the supported registers for ARM64)
I: (integer number in base 10 or 16)

The implementation of this function simply parses the string from above, and then yields chains from the internal method that searches chains. This internal method then decides what algorithms to use and in what order. After yielding the chains that were found, the caller should call the `build_payload()` method, which will turn the chains found into payload bytes (2.5.7).

Listing 6: Try gadget subroutine

```
function try_gadget(g):

    if g.stack_size > max stack size:
        return False

    w = join(start_0, ... start_q, w)

    for ef in g_i.effects:
        if ef.dest_elem = w.dest_elem:
            to_check = ef

    if to_check is not found or match(w, to_check) is False:
        return False

    if check_fixed_regs() is False:
        return False

    return True
```

2.5.2 Gadget Search

This is the algorithm responsible for searching through all the (valid) gadgets found in the preprocessing phase, giving a few arguments (for their explanation, check 2.5.1):

- a single wanted effect (2.1.9); note that in 2.5.1 it was mentioned multiple wanted effects can be used to search for gadgets; while this is true, the actual gadget search receives only one (arbitrary) wanted effect, and the rest of them are simply matched against the gadgets found; here, only the process of finding the gadgets for a single wanted effect is presented
- maximum stack size
- a list of registers whose values must remain unchanged after the execution of the gadget
- register start values
- maximum elements to return

The gadget search can take place in two different ways: with or without the maps described in 2.4.6. Before these maps are built, or when start values are given, or if opted so, the search uses only the effect to gadget map (also detailed in the mentioned section), which is created before the rest of the maps. Independent of this choice, when a gadget is checked whether it satisfies a wanted effect w, the following (6) subroutine is used.

The function try_gadget will be found in a form that resembles this one in every chain searching algorithm that follows. The time complexity for this routine is:

$$O(t_{startjoin} + M_{check} + F_{check}) = O(|w| \max_{0 \le i \le q} |start_i| + M_{check} + F_{check})$$

If it were to explicitate the M_{check} (2.2.4) and F_{check} (2.2.5) and also consider that there are no start effects (which is almost always the case), then the time complexity is reduced to:

$$O(t_{trygadget}) = O(|w| + \max_{g \in G} |e_g| + t_{sat})$$

Listing 7: Gadget search without maps

```
1        result = []
2        for each corresponding effect type t:
3            for each gadget g in gadgets[w.dest_elem][t]:
4
5                if try_gadget(g) is True:
6                    result.add(g)
7
8                if len(result) = max_ret:
9                    return result
10
11       return result
```

The "matching" between a gadget and a wanted effect is proportional with the number of elements from the wanted effect and the corresponding effect from that gadget, and in some cases (matching with an ARITH effect) a call to z3 module is made.

When searching without maps, the pseudocode 7 applies.

The corresponding effect types are:

- (w) LOAD_CT → LOAD_CT, LOAD_S, ARITH
- (w) MOV_RR → MOV_RR, ARITH
- (w) ARITH → ARITH

When maps are used, in an analogous pseudocode, the following maps are checked (in this order):

- (w) LOAD_CT → load ct map, load s map, arith freestack map
- (w) MOV_RR → mov map
- (w) ARITH → snapshot map, arith freestack map

The (worst case) time complexity for searching through gadgets for a wanted effect is the same:

$$O(G_{search}) = O(|G|t_{trygadget}) = O(|G|(|w| + \max_{g \in G} |e_g| + t_{sat}))$$

Still, the usage of maps greatly reduce the constant from this expression, thus greatly speeding up all the searches; empirical evidence in this sense is presented in 3.2.8.

2.5.3 Substitution-Based MOV_RR Search

This heuristic algorithm for searching MOV_RR wanted effects is a direct application of the transition graph concept detailed in 2.4.5. It supports as input the following parameters (for their explanation, check 2.5.1):

- one wanted effect (2.1.9)
- max stack size
- fixed register list

Listing 8: MOV_RR search

```
1   wef := e_{i←j} : R_i ← R_j
2
3   if ispath_ij is False:
4       return
5
6   for path_ij = (g_0, g_1, ... g_k) in transitions(i, j) (2.4.5):
7
8       ch_i = join(g_k, ... join(g_2, join(g_1, g_0)) ... )
9
10      check successful join for ch_i
11      check stack size for ch_i
12
13      for ef in ch_i.effects:
14          if ef.dest_elem = wef.dest_elem:
15              to_check = ef
16
17      if match(wef, to_check) is False:
18          continue
19
20      if check_fixed_regs() is False:
21          continue
22
23      yield ch_i
```

The algorithm can be summarized as in pseudocode 8.

It can be noticed that, although the transit path guarantees the wanted effect is matched, the match is still checked (line 17). This is just a "paranoid" check, to absolutely ensure the returned chain satisfies the wanted effect. Besides that, only the fixed registers are checked (line 20), which is needed if the number of fixed registers requested is bigger than 2 (i.e. when there are no transit subgraphs to guarantee the fixed registers anymore).

Before calculating the time complexity, the context in which the algorithm will be discussed. First of all, when the path does not exist (or at least, it was not found by the algorithm 2.4.5), it will give an answer in $O(1)$ every time (line 3). Now, let us consider first the most relaxed, non-trivial case: only one chain that satisfies $e_{i \leftarrow j}$ is requested, the stack size is unlimited, and there are no registers to keep fixed. In this case, any path $path_{ij}$ is guaranteed to be a solution, simply because the corectness of the path is guaranteed by the algorithm described in 2.4.5, and also because there is no other constraint - there is also the possibility of stack joining failure (2.2.2), but in the experiments done, it does not appear to interfere with the stated conclusion (3.2.5). Considering such use cases, although the theoretical worst case time complexity remains the same, in practice it will work extremely fast (3.2.9). On the other side, if there are start values, a lot (> 2) of registers to keep fixed, and the stack size is neither very small, nor very big (say, such that 1/2 of all possible chains are over this limit), and the caller wants to yield as many chains as possible, then the upper bounds of the theoretical time complexity might be reached, and the algorithm will appear very slow, with the runtime expressed in hours. One more interesting case is when there are no constraints, except one or two fixed registers; in this situation, the expected running time is exactly the same (if not even smaller) as for the first, ideal case (with no constraints), because, instead of using the whole graph comprised of $path_{ij}$ elements, the corresponding precalculated subgraph given by $path_{ij}^{fixed}$ is used (2.4.5); such subgraphs could theoretically be precomputed for a wide range of constraints, such as a limited stack size per gadget, or tuples of more than 2 fixed registers, but there is a limit over which the time spent preprocessing is not worth the runtime benefit.

The time complexity for the general case is:

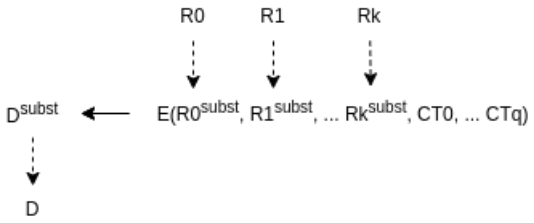

Figure 4: Substitution (register-only) of a wanted effect

$$O(P_{find}(T_{join} + M_{check} + F_{check})) = O(\max_{0 \leq i \leq |R|, 0 \leq j \leq |R|}(|gpath_{ij}|)^{|R|}(T_{join} + M_{check} + F_{check}))$$

Considering less favorable cases (i.e. as described in the paragraph from above), the running times fluctuate a lot and it is relatively common to encounter slow running cases (check 3.2.9 for statistics). On the other side, if any path is guaranteed to yield results (again, check the paragraph from above), then it can be considered $|gpath_{ij}| = 1$ and thus the time complexity becomes (each term can be explicitated according to 2.2.3, 2.2.4 and 2.2.5):

$$O(T_{join} + M_{check} + F_{check})$$

2.5.4 Substitution-Based (Register-Only) ARITH Search

This algorithm, as with other substitution-based heuristics, is based on the transition graph detailed in 2.4.5. It supports as input the following parameters (for their explanation, check 2.5.1):

- one wanted effect (2.1.9)
- max stack size
- fixed register list

In short, it tries to substitute every register from the wanted effect expression, check if there is a gadget that satisfies the modified wanted effect, and, if true, it tries to join that gadget with chains that execute the said substitutions (fig 4). To formalize, let us define:

- for a reg_in element of register R_i, an *(input) register substitution* γ_i:

  $$\gamma_i : R_{i^\gamma} \leftarrow R_i \text{ such that } \exists path_{i^\gamma i} \text{ (as defined in 2.4.5)}$$

- for a reg_out element of register R_d, a *(output) register substitution* γ_d:

  $$\gamma_d : R_d \leftarrow R_{d^\gamma} \text{ such that } \exists path_{dd^\gamma}$$

- an *(ordered) substitution* γ:

  $$\gamma : (\gamma_0, \gamma_1, ...\gamma_k, \gamma_d), \text{ where the last register substitution}$$
  γ_d is always an output register substitution, and the rest are input register substitutions

  By "ordered" it meneans that the substitution takes place in the order that the register substitutions are positioned.

- for a wanted effect (2.1.9) w, a *substituted wanted effect* w^γ:

$$w : R_d \leftarrow E(R_0, ...R_k, CT_0, ...CT_q) \Rightarrow w^\gamma : R_{d^\gamma} \leftarrow E(R_{0^\gamma}, ...R_{k^\gamma}, CT_0, ...CT_q)$$

  (Note that, as specified in 2.1.9, a wanted effect cannot contain stack values)

- for a register substitution γ_i, a *substitution chain* $ch_{i^\gamma i}$:

$$path_{i^\gamma i} = (g_{i_0}, g_{i_1}, ...g_{i_u}) \Rightarrow ch_{i^\gamma i} = join(g_{i_0}, join(g_{i_1}, ...join(g_{i_{u-1}}, g_{i_u})...))$$

  Analogous, ch_{dd^γ} is defined.

- for a wanted effect w and a substitution $\gamma : (\gamma_0, \gamma_1, ...\gamma_k, \gamma_d)$, a *final chain* ch_w^γ:

$$ch_w^\gamma = join(ch_{0^\gamma 0}, join(ch_{1^\gamma 1}, ...join(ch_{k^\gamma k}, join(g_{w^\gamma}, ch_{dd^\gamma}))...)),$$
$$\text{where } g_{w^\gamma} \text{ is any gadget that satisfies } w^\gamma$$

  Notice that, for this definition to be consistent, the substitution γ must be ordered (as mentioned above), so that the chain joining order can be properly defined.

As with the gadget validation algorithm described in 2.4.4, trying to substitute multiple registers gives rise to a problem. Suppose we have a wanted effect $w : R_d \leftarrow E(R_0, R_1)$, and a substitution $\gamma : (\gamma_0, \gamma_1, \gamma_d)$; the order indicates that the first substitution to take place is γ_0, then γ_1, and at the end γ_d. If a gadget g_{w^γ} that validates the substituted wanted effect $w^\gamma : R_{d^\gamma} \leftarrow E(R_{0^\gamma}, R_{1^\gamma})$ is found, ideally, by joining this gadget with the substitution chains $ch_{0^\gamma 0}$, $ch_{1^\gamma 1}$, and ch_{dd^γ}, we would in the end obtain the final chain that satisfies the original wanted effect w. But this is not necessarily the case: for this to happen, the values of R_{0^γ} and R_{1^γ} both need to contain (at the same time) the input values of R_0 and R_1, so that the intermediary effect $R_{d^\gamma} \leftarrow E(R_0, R_1)$ is first satisfied. If the second substitution chain $ch_{1^\gamma 1}$ alters the value of the register R_{0^γ} which contains the input value of the register R_0, or the first substitution chain $ch_{0^\gamma 0}$ alters the yet-to-be substituted register R_1, then that intermediary effect is no longer satisfied, and thus in the end ch_w^γ will not satisfy w, which defeats the whole purpose of this algorithm. Moreover, in the case that registers which need to be kept fixed are given as an argument for the search algorithm, this problem also extends to the substitution chain for the output register (in the scenario above, this is not the case). The situation above can be overcame in two different ways: either try every possible substitution chain, in every order possible, which would obviously represent a problem when taking the performance into account, or try to define additional properties that guarantee the success of the algorithm (or, at least, most of the time). One of the most natural ways to define such an *ordered failproof substitution* is the following:

- the (input / output) register substitution are generalized as follows:

$$\gamma_i^f : R_{i^\gamma} \leftarrow R_i \text{ such that } \exists path_{i^\gamma i}^f \text{ (as defined in 2.4.5)}$$
$$\gamma_d^f : R_d \leftarrow R_{d^\gamma} \text{ such that } \exists path_{dd^\gamma}^f$$

  Here, f denotes a fixed set of registers, including the empty set, case in which it concides with the first definitions. For a more detalied explanation, check 2.4.5.

- an *ordered failproof substitution* γ is defined as follows:

$$\gamma : (\gamma_0^{f_0}, \gamma_1^{f_1}, ...\gamma_k^{f_k}, \gamma_d^{f_d}),$$

Where (f_{req} is the set of registers to be kept fixed, as provided in the arguments of this algorithm):

$$\begin{aligned} f_0 =& f_{req} + [] + [R_1, R_2, ...R_k] \\ f_1 =& f_{req} + [R_{0^\gamma}] + [R_2, ...R_k] \\ & ... \\ f_k =& f_{req} + [R_{0^\gamma}, R_{1^\gamma}, ...R_{(k-1)^\gamma}] + [] \\ f_d =& f_{req} \end{aligned}$$

Although this concept is defined on a general case, any fixed set of registers f that support implementing failproof substitutions (which in turn are based on precomputed subgraphs of transitions) is constrained to have $|f| \leq 2$. Applying this restriction to the set of equations from above, we obtain:

$$|f_{req}| \leq 2 \text{ and } K + |f_{req}| \leq 3,$$
where $K = k + 1$, which is the number of reg_in elements from the wanted effect w

Because of that, the patterns of the possible wanted effects that are supported (in conjunction with the f_{req} set) are:

$$\begin{aligned} & w : R_d \leftarrow E(CT_0, ...),\ f_{req} = [R_a, R_b] \text{ or } f_{req} = [R_a] \text{ or } f_{req} = [] \\ & w : R_d \leftarrow E(R_0, CT_0, ...),\ f_{req} = [R_a, R_b] \text{ or } f_{req} = [R_a] \text{ or } f_{req} = [] \\ & w : R_d \leftarrow E(R_0, R_1, CT_0, ...),\ f_{req} = [R_a] \text{ or } f_{req} = [] \\ & w : R_d \leftarrow E(R_0, R_1, R_2, CT_0, ...),\ f_{req} = [], \\ & \quad \text{for all possible (different) } R_0, R_1, R_2, \text{ and for all possible (different)} R_a, R_b \end{aligned}$$

If there is no register in the wanted effect (for example, when the wanted effect is of type LOAD_CT) and f_{req} is empty, the general case is naturally applied. Also, when the criteria mentioned above are not fulfilled (for example, there are 4 elements in f_{req}), then the general case also applies, degenerating in trying all possible substitutions, until a final chain ch_w^γ that satisfies the wanted effect w and also keeps the required registers fixed is found. One more thing to note is that, as mentioned, even if failproof substitutions are used, there is still a non-zero probability that (at least one of) the join operations might fail (because of stack joining failure 2.2.2) - fortunately, this does not appear to be a problem (3.2.5). Also, the stack size can also influence the running time (and the results) of this algorithm: a sufficiently small stack size would greatly reduce the number of chains to be checked, and while it might run fast, it might not return any results most of the time; on the other side, a very big (unlimited) stack szie (which is the default) allows for all possible chains to be checked, but at the same time the analysis with failproof substitutions from above applies, so the runtime is also small, this time with a bigger chance of results; the case which is the hardest to predict in terms of runtime and number of returned results is when the stack size is neither very big nor very small.

A simplfied pseudocode for this algorithm looks as in 9.

From the pseudocode above, it is worth observing that the search for transitions $path_{i^\gamma i}$ that correspond to the substitution γ is postponed as much as possible (line 18), so that if no gadget g_{w^γ} is found, no joins or checks are executed (line 14); if the failproof substitutions are used, then, most of the time, not finding such gadget g_{w^γ} is the main reason for failure (or at least, for multiple substitutions to be tried), so this order of execution is expected to save a lot of time.

The time complexity can be analysed in a few different cases, depending on the input parameter properties. As explained above, if failproof substitutions are used, the join failure probability is considered negligible, the stack size is unlimited, and the caller only wants to yield a single chain (or a constant number of chains) all `for` loops can be viewed as single operations, and thus the upper bound for this case is:

Listing 9: Substitution based search

```
w := the wanted effect given as argument
f_req := fixed register list given as argument
mss := max stack size given as argument

tgraph = get the transition subgraph
regset = find all reg_in from wef and the reg_out

for each perm in permutation(|regset|):
    for each γ in get_substitutions(perm):

        w^γ = apply(w, γ)
        L_{g_{w^γ}} = search_gadgets(w^γ, f_req, mss)

        if len(L_{g_{w^γ}}) = 0:
            continue

        for each g_{w^γ} in L_{g_{w^γ}}:
            for each (path_{0γ0}, ... path_{kγk}, path_{dd^γ}):

                ch_{0γ0} = join(g_{0_0}, join(g_{0_1}, ... join(g_{0_{u-1}}, g_{0_u}) ... ))
                ...
                ch_{kγk} = join(g_{k_0}, join(g_{k_1}, ... join(g_{k_{u-1}}, g_{k_u}) ... ))

                ch_w^γ = join(ch_{0γ0}, ... join(ch_{kγk}, join(g_{w^γ}, ch_{dd^γ})) ... )

                if len(ch_w^γ) > mss:
                    continue

                for ef in ch_w^γ.effects:
                    if ef.dest_elem = w.dest_elem:
                        to_check = ef

                if match(w, to_check) is False:
                    continue

                if ch_w^γ.check_fixed_regs(f_req) is False:
                    continue

                yield ch_w^γ
```

- $O(1)$ for getting the transition graph (if it is already precalculated) - if not, check 2.4.5 (line 5)
- $O(|w|)$ for extracting the registers from w (line 6)
- $O(1)$ for the permutation loop (line 8)
- $O(|R|^{|regset|}) = O(1)$ for the substitution finder; the regset is the number of registers inside the wanted effect w, which is ≤ 3 when considering failproof substitutions are used, so in the end the time spent to find a substitution can be considered constant (line 9)
- $O(G_{search})$ for searching g_{w^γ} (2.5.2) (line 12)
- $O(1)$ for the third loop which chooses a g_{w^γ} from the ones found (line 17)
- $O(1)$ for the transition chain searcher (2.4.5) (line 18)
- $O(T_{join})$ for the join operations (2.2.3) (lines 20-24)
- $O(M_{check} + F_{check})$ for the "paranoid" matching (2.2.4) and the fixed registers check (2.2.5) (lines 26-36)

In addition to this analysis, the usage of failproof substitutions also has the side effect of imposing an upper bound on the number of registers, which in turn limits the maximum number of possible substitutions, even if the transition subgraphs are equal with the maximal traisition graph. On the other side, even though some of the terms are constant, in certain cases that constant is big enough to impact the runtime on the order of seconds or tenths of seconds, for example.

For the case when failproof substitutions are not used, the time complexity degenerates; compared to above, the following change:

- the permutation loop becomes $O(|regset|!)$
- for the substitution finder, $O(|R|^{|regset|})$ cannot be approximated as constant anymore, and is kept exponential; still, $|regset|$ is expected to remain small
- the loop which iterates over g_{w^γ} becomes linear in their number (instead of selecting only the first gadget which happens in constant time)
- the number of transition chain searcher can no longer be considered constant: $O(\max_{0\leq i\leq|R|,0\leq j\leq|R|}(|gpath_{ij}|)^{|R|})$;
- the number of joins also can increase dramatically, because of the variable value of $|regset|$

The practical implications of using failproof substitutions is more complicated than in theory, though. Even though it certainly helps speeding up worst cases from a practically infinite runtime down to hours or minutes, these cases are not necessarily common - in most of the cases, most of the time is spent trying combinations of possible substitutions.

A small batch of experiments described in 3.2.10 shows the discrepancy in the running times caused by apparently small changes in the wanted effect or the associated search parameters.

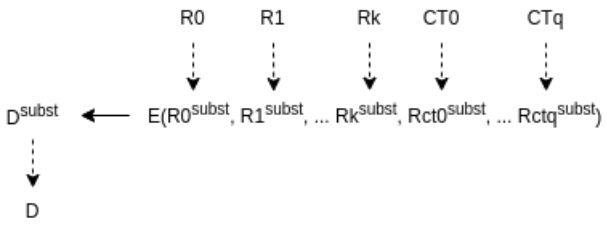

Figure 5: Substitution (register and constants) of a wanted effect

2.5.5 Substitution-Based (Register and Constants) ARITH Search

This algorithm is very similar to the one presented in 2.5.4, the only difference being that besides trying to substitute only registers, constants are also subject to replacement: if a constant (ct_val element, assigned stack values are not taken into account for simplicity, although the algorithm could be extended in this way) is found inside the wanted effect w, it can either be ignored without being replaced (thus reducing this algorithm to the one in 2.5.4), or it can be replaced with a register, which in turn is assigned that constant value. A suggestive illustration is presented in fig 5.

Before anything else, it should be stated that constants are identified (and substituted) by their position inside the wanted effect, and not by value. For example, for the wanted effect $w : rcx \leftarrow (rax + 0x10)0x10$, a valid substituted wanted effect might look like $w^\gamma : rcx \leftarrow (rax + r8)rbp$, or even $w^\gamma : rcx \leftarrow (rax + r15)0x10$.

Because of the generalization compared to 2.5.4, a few notions need to be redefined, and other new introduced:

- a gadget $g_{i \leftarrow c}$:
$$g = g_{i \leftarrow c} \iff \exists e_{i \leftarrow c} \in g, e_{i \leftarrow c} : R_{cti} \leftarrow c$$
- a *constant substitution* has one of the two forms:
$$\gamma_{cti} : CT_i \leftarrow CT_i \text{ (no actual substitution is taking place)}$$
$$\gamma_{cti} : R_{cti^\gamma} \leftarrow CT_i \text{ such that } \exists g_{i \leftarrow CT_i}$$
$$\gamma^f_{cti} : R_{cti^\gamma} \leftarrow CT_i \text{ such that } \exists g_{i \leftarrow CT_i} \text{ and } \nexists e \in g_{i \leftarrow CT_i}, e.dest\_elem \in f$$
- an *ordered substitution* γ can also contain constant substitutions between the input register substitutions, but the output register substitution is always executed at the end
- for a wanted effect (2.1.9) w, a *substituted wanted effect* w^γ:
$$w : R_d \leftarrow E(R_0, ...R_k, CT_0, ...CT_q) \Rightarrow w^\gamma : R_{d^\gamma} \leftarrow E(R_{0^\gamma}, ...R_{k^\gamma}, R_{ct0^\gamma} \mid CT_0, ...R_{ctq^\gamma} \mid CT_q)$$
- for a constant substitution γ_{cti}, a substitution chain $ch_{cti^\gamma cti}$:
$$ch_{cti^\gamma cti} = g_{i \leftarrow CT_i}$$
- the other definitions from 2.5.4 remain the same

As it can be noticed, the concept of failproof substitutions is kept, along with the motivation behind it. Still, the f_i sets need to be redefined to include the constant substitutions. For simplicity, consider the case in which the constants are substituted after all registers are substituted, given by

$\gamma : (\gamma_0^{f_0}, \gamma_1^{f_1}, ...\gamma_k^{f_k}, \gamma_{ct0}^{f_{ct0}}, \gamma_{ct1}^{f_{ct1}}, ...\gamma_{ctu}^{f_{ctu}}, \gamma_{ct(u+1)}^{f_{ct(u+1)}}, ...\gamma_{ctq}^{f_{ctq}}, \gamma_d^{f_d})$ (only the first $CT_0, ...CT_u$ are substituted; $CT_{u+1}, ...CT_q$ remain unchanged, and the corresponding substitutions are the identity):

$$
\begin{aligned}
f_0 =& f_{req} + [] + [R_1, R_2, ...R_k] \\
f_1 =& f_{req} + [R_{0^\gamma}] + [R_2, ...R_k] \\
& ... \\
f_k =& f_{req} + [R_{0^\gamma}, R_{1^\gamma}, ...R_{(k-1)^\gamma}] + [] \\
f_{ct0} =& f_{req} + [R_{0^\gamma}, R_{1^\gamma}, ...R_{(k-1)^\gamma}, R_{k^\gamma}] + [] \\
f_{ct1} =& f_{req} + [R_{0^\gamma}, R_{1^\gamma}, ...R_{(k-1)^\gamma}, R_{k^\gamma}, R_{ct0^\gamma}] + [] \\
& ... \\
f_{ctu} =& f_{req} + [R_{0^\gamma}, ...R_{k^\gamma}, R_{ct0^\gamma}, ...R_{ct(u-1)^\gamma}] + [] \\
f_d =& f_{req}
\end{aligned}
$$

As it can be observed, the difference between substituting a register R_i with R_{i^γ} and substituting a constant CT_i with R_{cti^γ} is that the register R_i is kept fixed before substituting it, while the constant obviously is not. After the substitution, both R_{i^γ} and R_{cti^γ} need to be kept fixed. Unfortunately, this does not help with the practical restrictions inposed on f_{req} and the number of reg_in elements from the wanted effect - it actually tightens them, by introducing the number of constants in the equation:

$$
\begin{gathered}
|f_{req}| \leq 2 \text{ and } K + Q + |f_{req}| \leq 3, \\
\text{where } K = k + 1, \text{ which is the number of reg\_in elements from the wanted effect } w \\
\text{and } Q = q + 1, \text{ which is the number of ct\_val elements from the wanted effect } w
\end{gathered}
$$

These constraints hold even if not every constant is substituted, because the decision of substituting a constant or not is taken during the construction of the substitution itself: the first choice is to not replace it, then every possible register is tested for the existence of a $g_{i \leftarrow c}$. Because of that, another performance issue needs to be taken into account: how to query in the fastest way possible for such gadgets. A simple solution is implemented: the wanted effect w is searched for constants; then, for each distinct constant value (recall that constants are identified per apparition, and not by value) the procedure that searches for individual gadgets is called (2.5.2) - the resulting gadgets are also checked for fixed registers, (individual or pairs of two, as in 2.4.5); the results are finally stored such that querying for the existence and for the list of these gadgets can be done in constant time. The main downside of this approach is that it needs to be executed every search, without the possiblity of precalculating it. Alternative options include the analysis of every LOAD_CT effect from each possible gadget; this would be a fast approach, and it would be executed only once, like the register transition graph; the main downside would be that, besides LOAD_CT effects, constants can also be loaded in registers by using a LOAD_S effect, and assigning a value to the corresponding stack element; because the LOAD_S effects are one of the most abundant type of effects (3.2.1), ignoring them represents a major drawback.

The pseudocode is almost identical as in 2.5.4, with the exception that at the beginning, the step mentioned above is executed. This adds to the time complexity the term $O(|w| + |w|_{ct}|R|G_{search}|R|^2) = O(|w| + |w|_{ct}G_{search})$; by considering failproof substitutions, which limit $|w|_{ct}$, we can consider it constant and thus the complexity can be expressed as $O(|w| + G_{search})$. In practice, this step takes a non-trivial amount of time to execute (3.2.10), but limiting the number of constants that can appear in the wanted effect certainly helps this cause. The rest of the time complexity is subject to the same discussion as presented in 2.5.4.

2.5.6 Bruteforce Search

The bruteforce search is the search routine to be used, if the others have failed. It does not contain any specific logic by itself: it simply checks all possible (different) chains of limited depth (i.e. number of gadgets per chain, considering validated chains created in 2.4.4 as one single gadget). The loop which checks the properties of the chain resemble the mechanism used to check chain properties used in the other algorithms, such as 2.5.4 or 2.5.2. The number of chains checked is around $|G|^2 + |G|^3 + ... + |G|^d$, where d is the bruteforce depth. This search method is not recommended, and its depth (or even whether it will be used or not) can be controlled with the API exposed by the ROP utilitary object.

2.5.7 Payload Building API

This is the last step of the entire payload generator, and is mainly implemented for convenience: this module is responsible for converting a chain with its entire set of properties into the relevant payload expressed as a byte array; it does not create any new chains, nor does it search new gadgets or alter the logic in any way - it simply builds the payload. That being said, there are a few aspects that will be further discussed.

Although the scheme depicted in 3 shows payload construction as a direct next step after the chain searching, it is actually a different stage that requires user interaction: first, the users sends a request for chain searching algortihms, which return a chain object; then, the user calls the payload builder api that converts the given chain in bytes. This ensures payload building api is properly exposed, which will be presented later in this subsection.

One of the main issues when building the payload is the adjustment of addresses to match the runtime virtual address space in which the payload will be deployed. All of the algorithms presented (up until now) are not aware of ASLR, or any other shift in the address of the gadgets, compared to the address indicated in the binary. Because of that, when building the payload, the caller must also provide the offset to be added to the addresses from within the payload; if none is provided, the offset will simply be considered as being 0. This offset is taken as-is by the payload builder and is not checked in any way for corectness (i.e. whether the resulting addresses respect any kind of properties, or if the offset itself is the real one).

Another consideration is the byte values from the final payload. Often, a ROP payload must not contain specific byte values, such as 0x00 or other whitespace characters (0x20, etc). To handle this requirement, the payload builder accepts as argument a list of *forbidden payload bytes*, that are checked agains the resulting payload before returning it to the user (and naturally, after the address offset is added). If at some point the payload builder finds a forbidden byte, it either tries another address for that gadget or chain (if the forbidden byte is inside a chain address), or it fails directly. The bytes inside the payload cannot usually be changed, because this would change the logic itself (or these bytes are padding, but the pad sequence is set by the user anyways), thus trying to use another address at which an identical gadget is found is the only general way of changing the payload contents to avoid forbidden bytes. It should be mentioned that, due to the format of 64 bit addresses, introducing bytes such as 0x00, 0x7f or 0xff (and other very common values) in the list will result in failure virtually everytime.

The api itself has a few more features for x86_64. Still, payload builder receives the following arguments:

- `chains`: a chain, or an ordered list of chains to be joined, for which the payload will be built (the chain objects themselves remain unaltered)
- `exit_address`: the address where the chain will return (jump) after finishing executing; the address is used as-is, without any offset addition
- `forbidden_bytes`: the forbidden bytes list

- `addr_offset`: the address offset (is added to the addresses of all gadgets inside the chains)
- `pad_sequence`: the (repeating) padding sequence to be used wherever padding is needed (or the value is irrelevant)
- *(x86_64 only)* `alignment_on_entry`: it indicates the alignment of the stack pointer when entering the (first) chain - default 8 bytes on x86_64, 16 bytes on ARM64
- *(x86_64 only)* `alignment_on_exit`: in conjunction with `alignment_on_entry`, the payload builder will use a ret-only gadget to align the stack pointer before exiting the chain

On ARM64, due to its particularities, the alignment options are not provided (mainly because there is no direct analog to the x86 return-only gadget). Still, the stack pointer is always 16 byte aligned, (considering its initial value when entering the payload is also 16 byte aligned) so that the stack accesses do not fail.

The payload builder returns a tuple consisting of:

- the payload bytes, ready to be used
- the entry address for the chain (with the address offset added), which must be handled manually; in case it must be part of the payload, it must be manually appended
- the stack pointer position, relative to the rest of the payload; it is useful if one wants to manually concatenate multiple payloads, in which case the user must take into cosideration that after the stack pointer end position, there might still be non-negligible values (i.e. that are used by the chain in order to satisfy the wanted effect) - it is basically the same discussion from the stack joining phase (2.2.2)

3 Experiments and Results

In this section various results will be presented. First, some experiments that analyse the internal behaviour are detailed, to support some of the claims made throughout the entire section 2, mainly those related to average runtime performance or the presumptions that lie beneath some of the algorithms used. Then, some general statistics such as the running time for some specific wanted effects will be shown, to illustrate the efficiency of the whole module.

With regard to the reproductibility of these results, the results contain a sufficiently high degree of randomness such that some particular instances might require repeated execution to be replicated (most of the time, this is the case with an "uncommon" chain obtained by gadget validation defined in section 2.4.4, that is further used by the searching algorithms). That being said, the entire source code for the implementation and for the experiments, along the rest of the specifications are public, and readily available for anyone to use.

3.1 Execution Environment

Every result presented in this section were obtained by executing the code inside the `stats.py` on the following hardware:

- laptop model *Lenovo Legion Y540*
- processor *Intel Core i7-9750H CPU @ 2.60GHz*
- RAM *16GB DDR4*
- (video card is not used)

The software specifications are:

- operating system *Ubuntu 20.04.4 LTS*
- python version *3.8.10* (CPython)
- capstone version *4.0.2*
- z3 version *4.8.13*

The (ELF) binaries that were used for this test are exclusively shared objects. The aim was to create stats for modules that are susceptible to be present inside an address space in which a ROP attack might take place. These libraries have mainly two versions used: an x86_64 binary, and its ARM64 couterpart - notice that even when analyzing an ARM64 binary, the same machine with the x86_64 processor is used, because the python code itself is not platform dependent. The entire list of binaries is shown in table 3 (the x86_64 binaries were made to run on Ubuntu 20.04 LTS, and those on ARM64 are for Linux kali 5.4.83-Re4son-v8l+ for a Raspberry pi 4B).

3.2 Statistics

3.2.1 Gadget Distribution

The distribution of gadgets w.r.t. the effect type and the destination element (as classified in 2.4.6) is important from multiple perspectives: how many gadgets are in a given binary, what are the registers that are altered the most by those gadgets, what type of effects are most commonly found in a gadget, and so on. A lot of algorithms rely on a particular distribution of gadgets underlined in 2.3.

Here are the following results from 4 tests done (note that the results include validated gadgets that are actually indivisible chains):

Table 3: Analysed binaries

x86_64	ARM64
libc-2.31.so (Ubuntu GLIBC 2.31-0ubuntu9.7)	libc-2.33.so (Debian GLIBC 2.33-1)
ld-2.31.so	ld-2.33.so
libpthread-2.31.so	libpthread-2.33.so
libcrypt.so.1.1.0	libcrypt.so.1.1.0
libcurl.so.4.6.0	libcurl.so.4.7.0
libm-2.31.so	libm-2.33.so

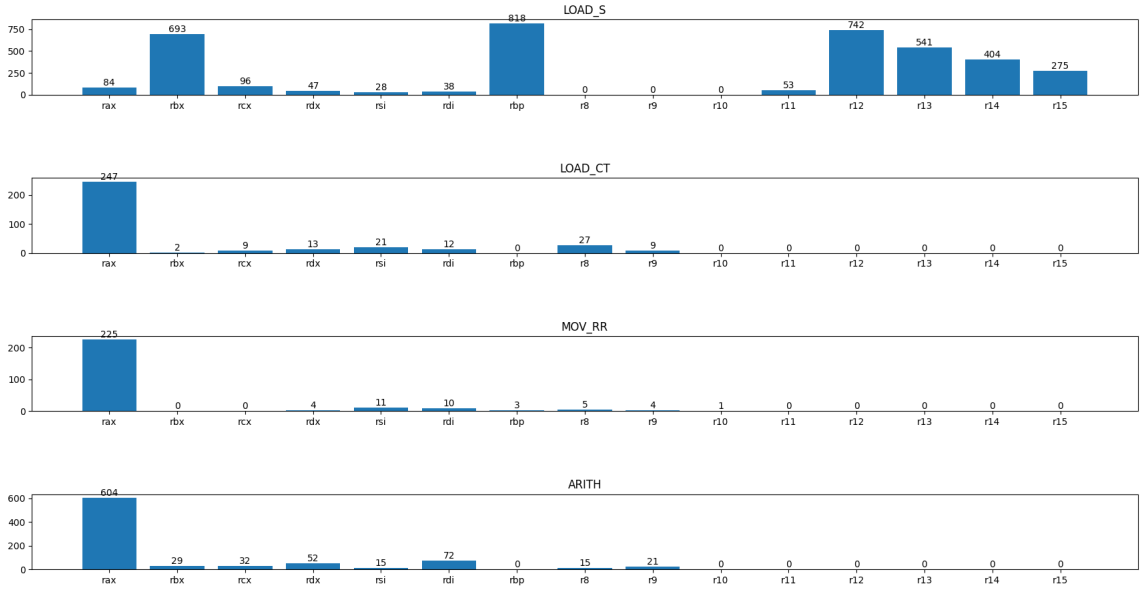

Figure 6: x86_64 libc gadget distribution

- on x86_64, on libc file (fig 6)
- on x86_64, on all the other binaries, with stats summed (fig 7)
- on ARM64, on libc file (fig 8)
- on ARM64, on all the other binaries, with stats summed (fig 9)

On x86_64, even if the smaller number of registers is taken into account, the distribution certainly appears to show a lack of diversity. Fortunately, for simple wanted effects (and fortunately, generally speaking, the most useful effects) like loading a value in a register, there is most of the time a gadget that satisfies it. On the other side, more complicated wanted effects are more likely to fail. The upside of this lack of diversity is speed: compared to ARM64, for most of the requests, the x86_64 analysis appears to move faster (3.2.4, 3.2.8, 3.2.9, 3.2.10).

That being said, the distributions validate a number of hypotheses on which some of the concepts used are based:

- The LOAD_S effect is the most abundant, leaving room for a lot of constant-to-register substitutions to take place in 2.5.5

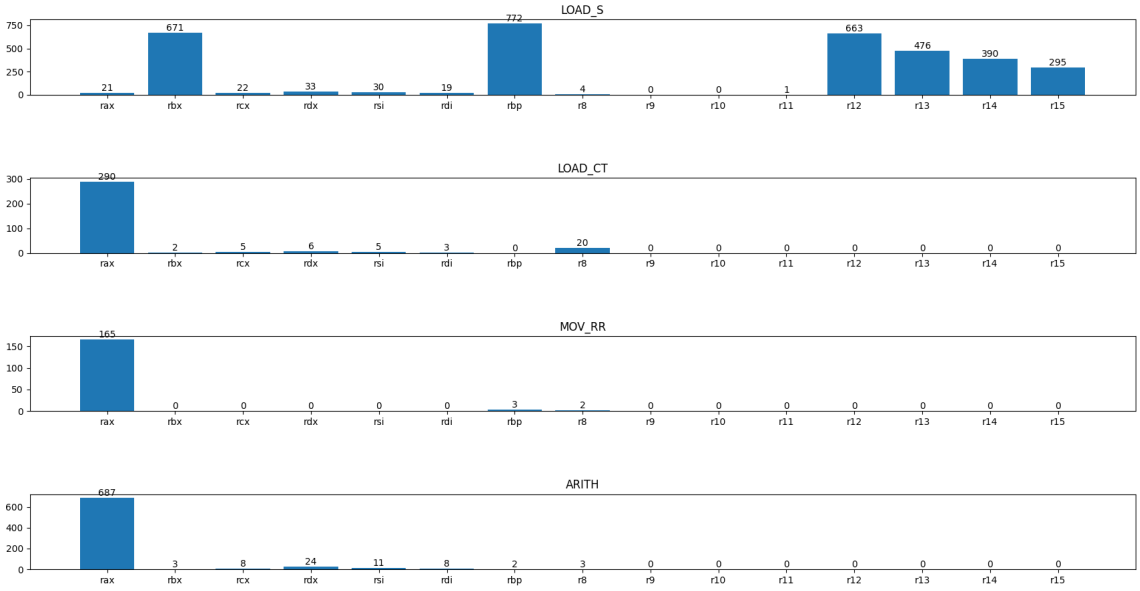

Figure 7: x86_64 gadget distribution on multiple binaries

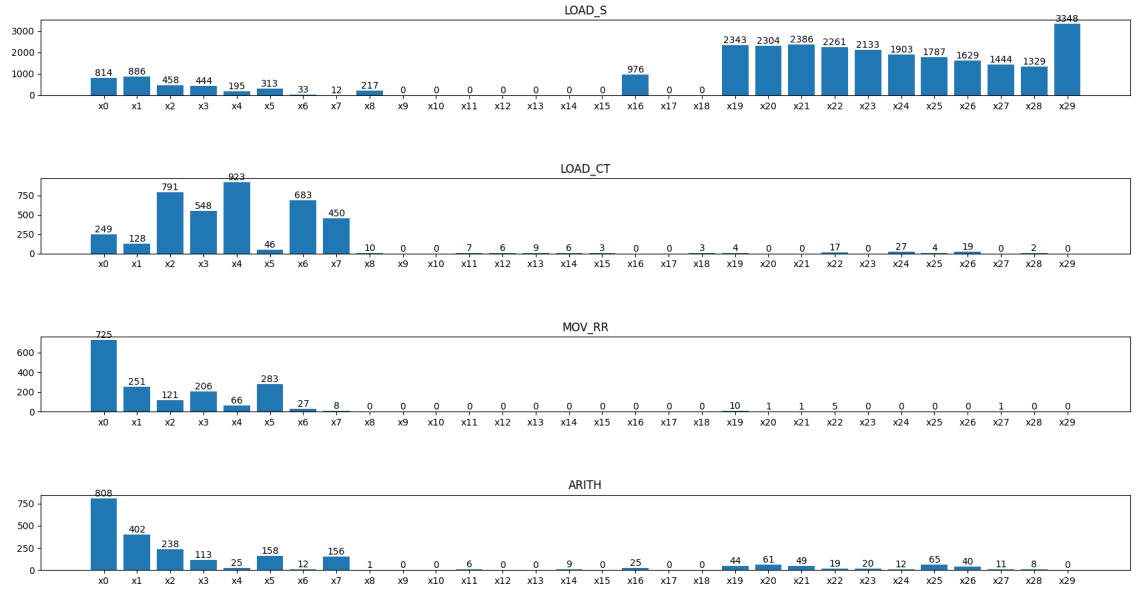

Figure 8: ARM64 libc gadget distribution

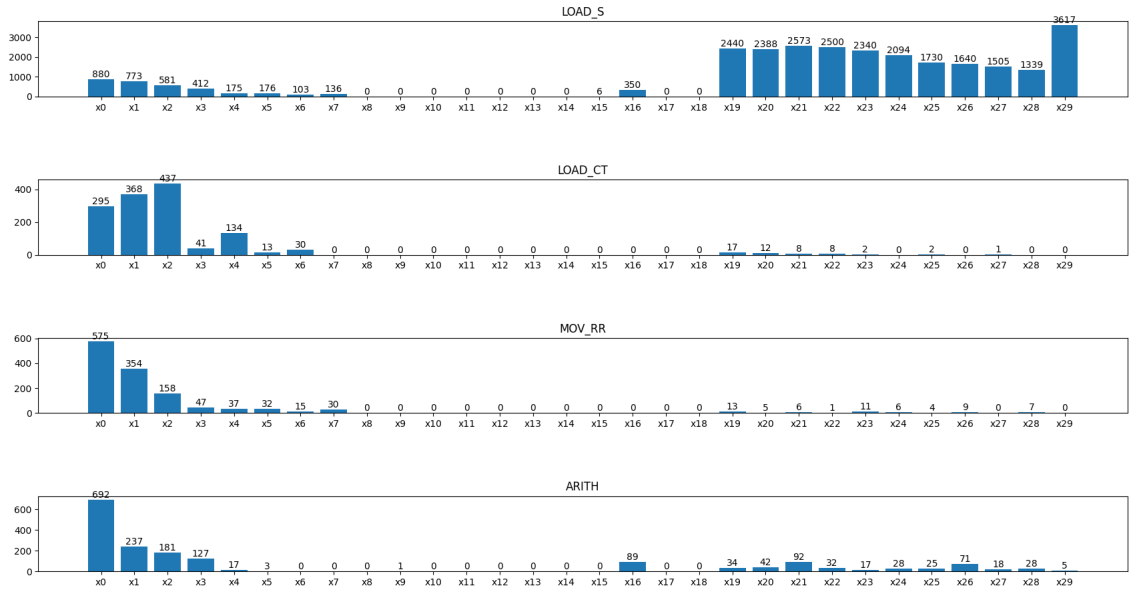

Figure 9: ARM64 gadget distribution on multiple binaries

- Every bucket (effect type, destination element) does not surpass a few thousand gadgets
- Most effects inside gadgets naturally have the destination element the return register (rax or x0); most of the rest of the effects have a register used for argument passing as destination element

3.2.2 Average Number of Elements

This is a small section aimed to analyse the average number of structured elements (2.1.1) inside effects (2.1.2). This statistic is significant mainly because the number of elements in an effect, gadget, or chain is part of the time complexities for most of the operations described in section 2.2. In some of these cases (2.4.4 for example), both a theoretical worst case time complexity and an average time complexity are given, the latter one using the average growth rate for element count. To address this fact, an experiment was done: for each binary from 3, the average number of elements per arithmetic effect (and, separately, per chain) was determined, gadgets or for chains with two gadgets. The results are expressed in table 4 (for average per effect) and table 5 (for average per chain).

From looking at these two tables, it can be seen that the average of elements inside an ARITH effect from single gadgets to chains of two gadgets grows after the gadget join by less than a half. This of course might change for chains longer than 2 gadgets (some gadgets found by search algorithms go up to 15-20 gadgets); still, the more gadgets a chain has, the more likely the ARITH effects are to be replaced by other, more "recent" effects: for example, considering an extreme case of a chain of 100 gadgets, even if an ARITH effect with destination register `rax` manages to accumulate a big expression in the first 10 gadgets, the probability that the `rax` register will not be overwritten with another effect in the following 90 gadgets from the chain is expected to be quite small. Even though a more detailed analysis would be required to calculate the expected number of elements inside an ARITH effect, the statistics presented here, coupled with other chain searching data, point towards the the hypothesis that the average time spent joining gadgets (which depends on the number of elements inside) is far below the theoretical upper bound. Also, the average growth rate strictly for the gadget validation step (denoted as z in 2.4.4) could be indirectly deduced from these stats, by considering that at the end of the validation step

Table 4: Average number of elements in arithmetic effects

binary	average for raw valid gadgets, x86	average for raw valid gadgets, ARM	average for gadgets, x86	average for gadgets, ARM	average for chains of 2 gadgets, x86	average for chains of 2 gadgets, ARM
libc	4.03	2.02	4.28	2.26	6.51	2.91
ld	4.03	2	4.08	2.29	6.46	2.99
libpthread	4.19	2.02	4.19	2.02	6.58	2.73
libcrypt	5.1	2	5.1	2.2	8.8	2.87
libcurl	3.9	2	4.01	2.27	6.33	3.10
libm	3.39	2	3.41	2	5.08	2.85

Table 5: Average number of elements in gadgets (chains)

binary	average for raw valid gadgets, x86	average for raw valid gadgets, ARM	average for gadgets, x86	average for gadgets, ARM	average for chains of 2 gadgets, x86	average for chains of 2 gadgets, ARM
libc	9.75	10.63	11.34	15.7	19.38	23.80
ld	8.97	9.87	9.61	15.85	16.55	23.01
libpthread	9.54	9.59	9.83	9.59	16.87	14.13
libcrypt	9.97	8.76	9.97	16.11	17.14	23.42
libcurl	9.16	10.31	9.79	14.33	16.63	20.92
libm	7.86	7.81	8.06	7.81	14.14	11.73

Table 6: Validation times (ARM64)

binary	runtime
libc	$\sim$ 80 s
ld	$\sim$ 18 s
libpthread	$\sim$ 0.01 s
libcrypt	$\sim$ 14 s
libcurl	$\sim$ 8 s
libm	$\sim$ 0.1 s

the valid gadgets have around 50% more elements for ARM64 and around 10% for x86_64.

If it were to compare the averages obtained on x86_64 and ARM64, on a per-effect basis the average is two to three times higher on x86_64, which is readily explained by the nature of the instructions on x86, which tend to be represented with more complicated arith effects, while on a per-gadget (chain) basis, the averages tend to hover around the same values; this happens because, even if inside a single ARITH effect the x86_64 contains more elements, the ARM64 tends to have a higher number of chains, correlated with the number of supported registers which is two times higher than on x86_64.

3.2.3 Gadget Validation Efficiency and Performance

The efficiency (yield) of the gadget validation algorithm described in 2.4.4 can be defined as the number of validated "gadgets" (chains) of a given type and destination element, compared to the raw distribution of valid gadgets. In this sense, the theory presented in the previous section does not offer any real clue to what the yield would be. Rather, the statistical properties of the gadgets that are typically found determine the outcome of this algorithm. That being said, for simplicity, only ARM64 results will be presented - the mechanism of action for x86_64 is exactly the same, and the overall results are also expected to be the same.

To analyse both the efficiency and the running time, the 6 different (ARM64) binaries from 3 were analysed. The experiment consists of simply running the preprocessing phase, and then checking the count of each type of gadget. By default, the preprocessing phase calls the validation method with coefficient $q = 3$ (which means that for every invalid gadget, 3 different substituting sequences are tried - for more details, check 2.4.4). The running times are shown in 6, and for the efficiency, the results for libc are shown separately (tables 7, 8, 9 and 10), and the results for the other binaries are summed (tables 11, 12, 13 and 14). By looking at the total number of gadgets, the same runs presented in the mentioned tables give the following results: for libc, 3000 raw gadgets were found, out of which 1565 were valid, and after validation 2048 new valid "gadgets" were obtained; for the other binaries, 3995 raw gadgets were found, out of which 1563 were valid, and after validation 2196 new valid "gadgets" were obtained. Note that by summing the results from the tables one obtains a much bigger number than the ones presented above - this is because most of the validated gadgets end up in a lot of different (effect type, reg out) buckes at once.

That the running times are quite unstable, but they ultimately hover around the same order of magnitude. This is due to the randomness that comes into play: in some runs, it might happen that a lot of chains fail early, and thus the final runtime is on the lower spectrum, or the opposite might happen. Still, out of all the runs that were done throughout the entire implementation and testing, there were no outliers found, neither in terms of yield, nor runtime.

The main disadvantage of this function is not necessarily the speed, but rather the unpredictability of results: even if most of the time a considerable amount of gadgets are validated, a mechanism that prioritizes the gadgets with "uncommon" effects (for example, an arithmetic effect present only in a specific invalid gadget) remains to be implemented. Because of that, some searches for "uncommon"

Table 7: Validation results for libc, LOAD_S effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
LOAD_S	x0	143	54	723
LOAD_S	x1	116	69	844
LOAD_S	x2	57	27	422
LOAD_S	x3	19	6	424
LOAD_S	x4	23	7	213
LOAD_S	x5	18	14	326
LOAD_S	x6	10	6	7
LOAD_S	x7	9	6	4
LOAD_S	x8	26	26	160
LOAD_S	x9	0	0	0
LOAD_S	x10	0	0	0
LOAD_S	x11	0	0	0
LOAD_S	x12	0	0	0
LOAD_S	x13	0	0	0
LOAD_S	x14	0	0	0
LOAD_S	x15	0	0	0
LOAD_S	x16	99	99	875
LOAD_S	x17	0	0	0
LOAD_S	x18	0	0	0
LOAD_S	x19	1081	937	1449
LOAD_S	x20	1039	902	1435
LOAD_S	x21	1084	914	1462
LOAD_S	x22	991	845	1414
LOAD_S	x23	956	789	1368
LOAD_S	x24	832	688	1220
LOAD_S	x25	823	653	1151
LOAD_S	x26	718	569	1057
LOAD_S	x27	620	484	905
LOAD_S	x28	543	412	825
LOAD_S	x29	1580	1431	1917

Table 8: Validation results for libc, LOAD_CT effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
LOAD_CT	x0	189	166	86
LOAD_CT	x1	72	3	132
LOAD_CT	x2	176	78	703
LOAD_CT	x3	88	66	491
LOAD_CT	x4	206	60	856
LOAD_CT	x5	15	0	53
LOAD_CT	x6	181	22	658
LOAD_CT	x7	120	0	480
LOAD_CT	x8	3	0	9
LOAD_CT	x9	0	0	0
LOAD_CT	x10	0	0	0
LOAD_CT	x11	2	0	8
LOAD_CT	x12	2	0	6
LOAD_CT	x13	3	0	12
LOAD_CT	x14	2	0	6
LOAD_CT	x15	1	0	3
LOAD_CT	x16	0	0	0
LOAD_CT	x17	0	0	0
LOAD_CT	x18	1	0	5
LOAD_CT	x19	1	0	5
LOAD_CT	x20	0	0	0
LOAD_CT	x21	0	0	0
LOAD_CT	x22	6	0	18
LOAD_CT	x23	2	0	0
LOAD_CT	x24	7	0	28
LOAD_CT	x25	1	0	3
LOAD_CT	x26	5	0	21
LOAD_CT	x27	0	0	0
LOAD_CT	x28	1	0	1
LOAD_CT	x29	1	0	0

Table 9: Validation results for libc, MOV_RR effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
MOV_RR	x0	666	280	470
MOV_RR	x1	275	48	164
MOV_RR	x2	171	22	105
MOV_RR	x3	151	41	163
MOV_RR	x4	49	21	48
MOV_RR	x5	97	49	226
MOV_RR	x6	13	9	18
MOV_RR	x7	2	2	2
MOV_RR	x8	0	0	0
MOV_RR	x9	0	0	0
MOV_RR	x10	0	0	0
MOV_RR	x11	0	0	0
MOV_RR	x12	0	0	0
MOV_RR	x13	0	0	0
MOV_RR	x14	0	0	0
MOV_RR	x15	0	0	0
MOV_RR	x16	76	0	0
MOV_RR	x17	0	0	0
MOV_RR	x18	0	0	0
MOV_RR	x19	4	0	13
MOV_RR	x20	4	0	0
MOV_RR	x21	1	0	1
MOV_RR	x22	3	0	4
MOV_RR	x23	1	0	0
MOV_RR	x24	1	0	0
MOV_RR	x25	0	0	0
MOV_RR	x26	0	0	0
MOV_RR	x27	2	1	0
MOV_RR	x28	0	0	0
MOV_RR	x29	0	0	0

Table 10: Validation results for libc, ARITH effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
ARITH	x0	559	326	499
ARITH	x1	160	21	375
ARITH	x2	125	12	236
ARITH	x3	56	1	113
ARITH	x4	9	0	22
ARITH	x5	47	0	176
ARITH	x6	5	0	12
ARITH	x7	22	17	144
ARITH	x8	1	0	1
ARITH	x9	0	0	0
ARITH	x10	0	0	0
ARITH	x11	2	0	6
ARITH	x12	0	0	0
ARITH	x13	0	0	0
ARITH	x14	2	0	6
ARITH	x15	0	0	0
ARITH	x16	28	0	21
ARITH	x17	0	0	0
ARITH	x18	0	0	0
ARITH	x19	11	0	45
ARITH	x20	17	0	67
ARITH	x21	15	0	42
ARITH	x22	12	0	21
ARITH	x23	2	0	5
ARITH	x24	6	1	17
ARITH	x25	4	4	75
ARITH	x26	5	2	59
ARITH	x27	3	0	10
ARITH	x28	1	0	5
ARITH	x29	0	0	0

Table 11: Validation results for other binaries, LOAD_S effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
LOAD_S	x0	271	19	851
LOAD_S	x1	39	4	859
LOAD_S	x2	27	3	600
LOAD_S	x3	38	4	386
LOAD_S	x4	13	0	195
LOAD_S	x5	7	1	187
LOAD_S	x6	12	0	115
LOAD_S	x7	4	0	148
LOAD_S	x8	5	0	0
LOAD_S	x9	5	0	0
LOAD_S	x10	0	0	0
LOAD_S	x11	0	0	0
LOAD_S	x12	0	0	0
LOAD_S	x13	0	0	0
LOAD_S	x14	0	0	0
LOAD_S	x15	2	0	7
LOAD_S	x16	0	0	420
LOAD_S	x17	6	0	0
LOAD_S	x18	0	0	0
LOAD_S	x19	1031	902	1649
LOAD_S	x20	965	855	1639
LOAD_S	x21	1034	887	1820
LOAD_S	x22	939	793	1820
LOAD_S	x23	859	712	1677
LOAD_S	x24	752	611	1557
LOAD_S	x25	651	510	1263
LOAD_S	x26	595	465	1160
LOAD_S	x27	527	409	1086
LOAD_S	x28	480	359	1004
LOAD_S	x29	1650	1552	2190

Table 12: Validation results for other binaries, LOAD_CT effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
LOAD_CT	x0	279	167	186
LOAD_CT	x1	137	6	375
LOAD_CT	x2	203	2	415
LOAD_CT	x3	33	0	44
LOAD_CT	x4	31	0	148
LOAD_CT	x5	27	1	14
LOAD_CT	x6	5	0	22
LOAD_CT	x7	9	0	0
LOAD_CT	x8	0	0	0
LOAD_CT	x9	0	0	0
LOAD_CT	x10	0	0	0
LOAD_CT	x11	0	0	0
LOAD_CT	x12	0	0	0
LOAD_CT	x13	0	0	0
LOAD_CT	x14	0	0	0
LOAD_CT	x15	0	0	0
LOAD_CT	x16	0	0	0
LOAD_CT	x17	0	0	0
LOAD_CT	x18	0	0	0
LOAD_CT	x19	6	0	20
LOAD_CT	x20	5	0	14
LOAD_CT	x21	7	0	13
LOAD_CT	x22	8	0	7
LOAD_CT	x23	1	0	4
LOAD_CT	x24	1	0	0
LOAD_CT	x25	1	0	3
LOAD_CT	x26	0	0	0
LOAD_CT	x27	0	0	1
LOAD_CT	x28	0	0	0
LOAD_CT	x29	0	0	0

Table 13: Validation results for other binaries, MOV_RR effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
MOV_RR	x0	547	127	453
MOV_RR	x1	250	5	344
MOV_RR	x2	117	1	150
MOV_RR	x3	48	1	36
MOV_RR	x4	35	1	38
MOV_RR	x5	16	0	32
MOV_RR	x6	11	0	25
MOV_RR	x7	16	0	28
MOV_RR	x8	0	0	0
MOV_RR	x9	0	0	0
MOV_RR	x10	0	0	0
MOV_RR	x11	0	0	0
MOV_RR	x12	0	0	0
MOV_RR	x13	0	0	0
MOV_RR	x14	0	0	0
MOV_RR	x15	0	0	0
MOV_RR	x16	57	0	0
MOV_RR	x17	0	0	0
MOV_RR	x18	0	0	0
MOV_RR	x19	22	0	18
MOV_RR	x20	9	0	7
MOV_RR	x21	8	0	4
MOV_RR	x22	5	0	5
MOV_RR	x23	8	0	11
MOV_RR	x24	3	0	5
MOV_RR	x25	2	0	4
MOV_RR	x26	8	0	11
MOV_RR	x27	2	0	1
MOV_RR	x28	8	0	6
MOV_RR	x29	2	0	0

Table 14: Validation results for other binaries, ARITH effect (ARM64)

effect	reg_out	total raw	total valid raw	total validated
ARITH	x0	640	304	485
ARITH	x1	106	0	242
ARITH	x2	51	0	143
ARITH	x3	38	0	143
ARITH	x4	19	0	18
ARITH	x5	5	0	3
ARITH	x6	20	0	0
ARITH	x7	1	0	0
ARITH	x8	0	0	0
ARITH	x9	1	1	0
ARITH	x10	0	0	0
ARITH	x11	0	0	0
ARITH	x12	0	0	0
ARITH	x13	0	0	0
ARITH	x14	0	0	0
ARITH	x15	0	0	0
ARITH	x16	632	0	90
ARITH	x17	0	0	0
ARITH	x18	0	0	0
ARITH	x19	13	0	30
ARITH	x20	22	0	48
ARITH	x21	11	0	82
ARITH	x22	16	0	33
ARITH	x23	17	1	18
ARITH	x24	14	0	41
ARITH	x25	13	0	29
ARITH	x26	18	0	87
ARITH	x27	12	1	22
ARITH	x28	14	1	27
ARITH	x29	0	0	6

Table 15: Transition graph construction time

binary	x86_64 time	ARM64 time
libc	7.66s	305.26s
ld	0.39s	40.23s
libpthread	0.23s	1.54s
libcrypt	0.12s	121.07s
libcurl	1.23s	65.86s
libm	5.94s	1.4s

wanted effects sometimes yield results, sometimes they don't, simply based on the random choices made in this module (but, fortunately, running it is always better than not running it, in terms of gadget diversity).

3.2.4 Transition Graph (Mov Map) Structure Statistics

Throughout the entire section 2.5, the concept of register transition graph and register transition path (as defined in 2.4.5) have been used, based on the presumption that the graph has a sparse structure, and even algorithms with a high (high degree polynomial, or even exponential) theoretical time complexity that use these concepts will be able to run in a relatively short amount of time. To support these claims, this subsection will present statistics a different transition graph, alog with an analysis of some of the path requests that might be executed inside the search algorithms.

Only the libc binary will be tested here - the other binaries are treated analogous, and their transition graphs are much simpler. For both of the architectures, the transition graph will be presented (which might slightly differ from one run to another, due to the randomness described in 2.4.4 and 3.2.3). Some requests will be explicitly analysed for the ARM64 version. It is worth mentioning that transition subgraphs (which have the constraint of keeping a subset of registers fixed) will *not* be shown, mostly because the graph itself already represents a worst-case scenario w.r.t. path finding procedure, and because the subgraphs are used in an analogous way.

The running time for building transition graphs for all 6 binaries from 3 (including all transition subgraphs) is illustrated in 15. As it could have been predicted by looking at the gadget distribution w.r.t. MOV_RR effects in 7 compared to 9, a discrepancy can be observed between x86 and arm. This is most likely due to the fact that on x86 there are two times less supported registers, and the number of gadgets for each MOV_RR effect with a speficic destination register is significantly smaller.

The adjacency matrix and the path matrix for libc are shown in fig 10 (for x86_64) and in fig 11 (for ARM64); the colors from the adjacency matrix indicate the number of gadgets that satisfy that transition - for the path matrix, "yellow" indicates there is a path, and "purple" that there is not.

The sparsity of graph paths can be observed on the adjacency matrices for both x86_64 and ARM64 - only a few registers have gadgets that can transfer their value to other few registers. The fewer paths, the more difficult it is to find paths (or even arithmetic effects) that are not directly found in gadgets, but on the other hand it guarantees the fastness of the algorithms. If the path existence is the only priority, other heuristics could be applied to reduce ARITH effects into MOV_RR effects (or, better said, to match with a MOV_RR effect) such that these matrices will be populated with a few more entries - such example is the algorithm described in 2.4.7, but which was not used in any of the statstics presented in this documentation.

To test the transition path finder, a number of 5 arbitrary transitions were chosen. The time taken to generate all paths was at most a few (~ 4) seconds long, and thus the time taken to generate all possible paths is small and not affected by the theoretical upper bound expressed in 2.4.5. That being said, the number of paths itself and their length also matters, because these paths (expressed as lists

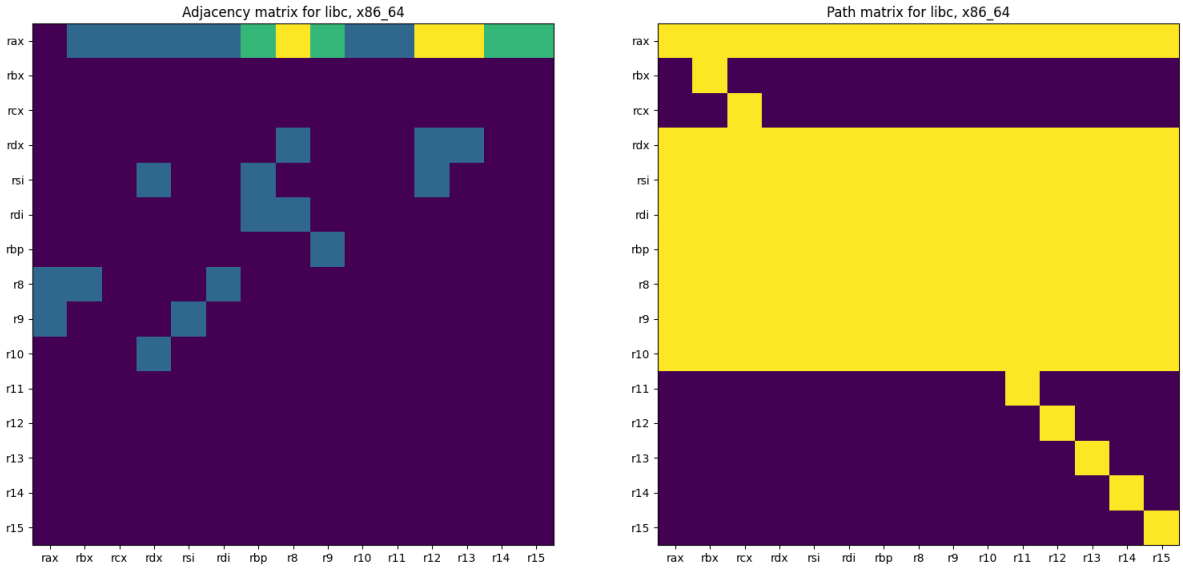

Figure 10: Transition graph matrices for libc, x86_64

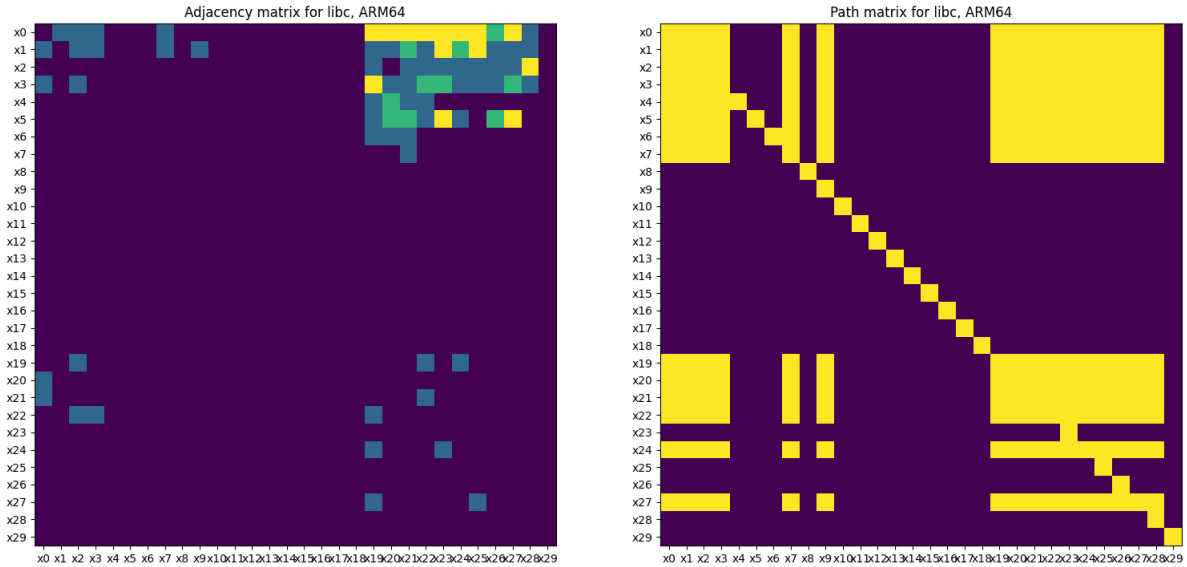

Figure 11: Transition graph matrices for libc, ARM64

Table 16: Transition generator statistics (ARM64)

transition	total number of possibilities found	maximum path length
$x27 \leftarrow x0$	4194	7
$x21 \leftarrow x0$	8198	7
$x7 \leftarrow x3$	19230	8
$x22 \leftarrow x9$	16396	8
$x19 \leftarrow x27$	837639	9

Table 17: Stack joining statistics

architecture	% SP < end, 1g	SP < end, 2g	% join failure
x86_64	0%	0%	0%
ARM64	12%	17%	1%

of gadgets) are further joined to form the corresponding chain, which is further processed - this takes a very large amount of time, compared to the time it takes to generate the paths themselves. These two mentioned properties can be found in 16. Note that in some other runs, the number of possible paths for some transitions went up to a few million. This underlines the necessity for a limited use of these paths, as described in 2.5.4.

3.2.5 Stack Join Failure Stats

Throughout the documentation, especially in the time complexity analysis of substitution algorithms (2.5.4, 2.5.5), the stack join failure probability plays an important role. In this sense, the average probability of stack join failure was determined for libc binary, for both x86_64 and ARM64. The results are presented in 17, whose columns are explained as follows:

- *% SP < end, 1g*: percentage of (valid) gadgets that have the end stack pointer position before the end of the stack view
- *% SP < end, 2g*: percentage of (valid) chains composed of exactly 2 gadgets, that have the end stack pointer position before the end of the stack view (the total is the number of chains that were obtained by successful joining)
- *% join failure*: the percentage of join operations that failed, compared to the total number of join operations (which is gadget count, squared)

A very interesting thing to observe right away is that for x86_64, there was no gadget found to have the stack pointer end before the stack view, which guarantees that every other chain formed in any circumstance will maintain this property - this was also observed in multiple runs. This validates the approach described in 2.5.3, and 2.5.4. On ARM64, this is not the case anymore, and a few gadgets fail when trying to join them; Still, the number is small ($\sim$ 1%), which points towards the assumption that the probability of stack joining failure can be neglected in the algorithms referenced above. Unfortunately, the generalization of the ARM64 statistics for chains larger than 2 gadgets, or even when the stack elements are assigned valuesis unclear, but, still, these statistics are complemented with the run times observed when executing a chain search (3.2.9, 3.2.10), which do not show any tendency of degradation in terms of the number of failed stack joins in a "practical" scenario.

Table 18: Snapshot map statistics for x86_64

destination register	effects with free stack elements	effects w/o free stack elements	snapshot key count
rax	15	586	313
rbx	0	29	12
rcx	18	13	9
rdx	38	9	6
rsi	0	18	9
rdi	5	70	22
rbp	0	0	0
r8	2	12	9
r9	21	0	0
r10	0	0	0
r11	0	0	0
r12	0	0	0
r13	0	0	0
r14	0	0	0
r15	0	0	0

3.2.6 Jump Statistics

Some statistics about the shape of (valid) JUMP effects are provided, so that the efficiency of the jump validation procedure (2.2.6) can be better understood. For both architectures, the number of jump elements that have only a stack element as a parameter is determined, and for the jumps that do not, they are simply printed and manually analysed, for possible jump validation failures. In a single run, for x86_64, 1645 jumps consisted only in a stack value, while 99 jumps did not, and on ARM64, 6408 jumps consisted only in a stack value, while 225 did not. For both architectures, out of the jumps which did not consist of a stack value, most of them were either an addition / xor-ing of two stack elements, or an addition / xor-ing between a stack element and a constant. The most complicated jump expression consisted in an addition / xor-ing of a stack element and a constant arithmetic expression; for example, (256 ^ (stack elem + 216)), (stack elem + (1 & (2^{32} - 1))), and so on - all of them were successfully recognised by the jump validation mechanism as being valid (there were no false positives).

3.2.7 Snapshot Map Statistics

As stated in 2.4.6, the efficiency of using the snapshot map for the gadget search defined in 2.5.2 directly depends on the number of ARITH effects that do not contain any free stack element. In this sense, a few experiments were executed, to determine this distribution of arithmetic effects. Also, the number of snapshot keys was also determined. The results are executed for libc, with no fixed registers, and they are presented in tables 18 and 19, and they represent a detaliation of a subset from the gadget distribution from 3.2.1.

3.2.8 Gadget Search Statistics

Searching for wanted effects through the (valid) gadgets extracted in the preprocessing phase is one of the simples algorithms, but at the same time one of the most important ones, both in terms of results and runtime. While the results returned exhaust all the possibilities (there are no heuristics and approximations - if this function noes not return any result it means there is certainly no gadget that satisfies the corresponding property), the runtime must be at the same time small because this routine

Table 19: Snapshot map statistics for ARM64

destination register	effects with free stack elements	effects w/o free stack elements	snapshot key count
x0	39	768	317
x1	104	266	121
x2	30	209	114
x3	27	83	45
x4	0	21	9
x5	21	136	50
x6	0	14	3
x7	0	169	52
x8	0	2	1
x9	0	0	0
x10	0	0	0
x11	0	6	2
x12	0	0	0
x13	0	0	0
x14	0	7	2
x15	0	0	0
x16	25	0	0
x17	0	0	0
x18	0	0	0
x19	0	35	18
x20	0	62	26
x21	1	43	25
x22	1	19	18
x23	0	6	5
x24	0	15	13
x25	0	51	20
x26	0	34	23
x27	0	9	8
x28	0	4	3
x29	0	0	0

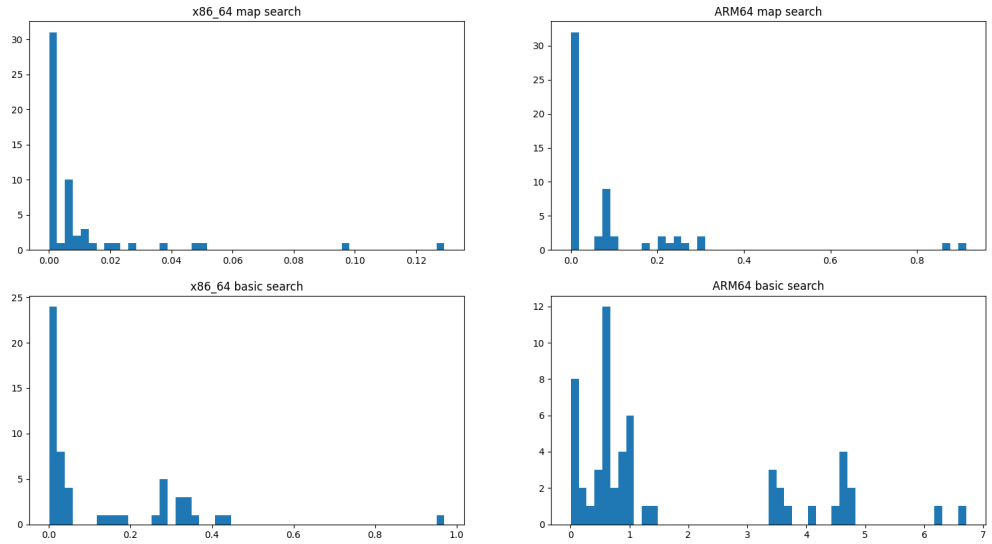

Figure 12: Gadget search timings

is called internally many times by every other search algorithm presented in section 2.5, and also by the transition graph generator (where, for example, it is called almost 900 times, on ARM64).

Due to the arbitrary nature of the wanted effect that can be searched for, a few examples were chosen, that would hopefully depict some typical searches this function is supposed to execute. The results are presented in the histogram 12 (the X axis indicates the time it takes for a search, in seconds), indicating the rest of the parameters chosen; note that, when searching exclusively through gadgets, even though multiple wanted effects are supported at once, this feature is not tested here. The examples were searched without any other constraint, then with stack size limited, then with fixed registers. The examples were chosen strictly from libc, with the observation that the time it takes for the search grows linearly with the number of gadgets in the binary.

It can clearly be observed that the gadget search using the maps described in 2.4.6 is substantially faster than the "basic" method of gadget search for both architectures.

3.2.9 Substitution-Based MOV_RR Search Statistics

This algorithm (2.5.3) can be considered one of the "most successful" heuristics described in this documentation, in the sense that it returns a result in a very short amount of time (at least compared to the other heuristics), and covers a lot of cases. To support this claim, runs were executed for all of the register pairs in "ideal" circumstances (check 2.5.3 for what this means); then, a few runs were intentionally executed with "less than ideal" parameters, to illustrate the runtime degeneration of these situations. The first table (20) contains, for each architecture, and for a number of fixed registers (0, 1, or 2) given as a `fixed_regs` (check 2.5.1 for an API description), the average search time for each effect for which at least a chain (gadget) was found (if there is no such gadget or chain to be found, the negative answer is returned in constant time, as explained in the previous section). The fixed registers were chosen at random out of all the supported registers. On ARM64, searches took a few times more on average, but the timings are well below one second for both architectures; the discrepancy between the two architectures might be explained by a bigger graph on ARM64.

In the second table (21) some searches that do not respect the condition of `len(fixed_regs) < 3` are shown: only one gadget is requested, and there are 5 (randomly chosen) fixed registers for each

Table 20: Substitution based MOV_RR average search timings

architecture	no fixed register	one fixed register	two fixed registers
x86_64	0.0011s	0.0009s	0.0012s
ARM64	0.0042s	0.0035s	0.0039s

Table 21: Examples of MOV_RR searches in "less than ideal" circumstances

wanted effect	runtime
x0 ← x27	14.13s
x0 ← x26	0.0003s
x0 ← x25	942.79s
x0 ← x24	0.44s
x0 ← x23	899.83s
rax ← rdx	5.4s

search. These examples do not represent an average, neither a minimum or maximum runtime value - they are meant only to illustrate the instability of the time it takes to run this algorithm when the constraint is not respected. As you can see, the search can be as fast as the "ideal" searches, or they can be orders of magnitude slower.

3.2.10 Substitution-Based ARITH Search Statistics

The substitution-based search represents the main searching algorithm for this implementation, with two slightly different versions: 2.5.4, where only registers are substituted, and 2.5.5 where constants can also be substituted. For these searches, "ideal" requests were defined, that use "failproof" substitutions, which guarantee a smaller upper bound for the time complexity (or, at least, speed up the algorithm in most of the situations). In this section, a few examples were tested for both architectures on the libc binaries - the search was parametrised in different ways, to highlight the relationship between running times and parameters, and the impact of using failproof substitutions. The results are presented in table 22.

The searches with the fastest timings are the ones without any (stack size) restriction (lines 1, 2, 3, 5, 6, 7, 8, 10), and which use failproof substitutions, as predicted in 2.5.4; when the stack size is restricted, at least for these cases it appears to not influence the timing too much, although it still remain unclear whether changing the stack size has a negative or a positive impact. When failproof substitutions are *not* used (lines 4 and 9), as stated in 2.5.4, the running times become very unstable and can become extremely big: for the test on line 4, the total runtime was only 1.6 seconds, but for the test on the line 9 (which is on arm), the search was stopped even before its ending, 2 hours after it began. Regarding the randomness that originates from the validation phase, it unfortunately influences the running times greatly, especially for ARM64: the effect on line 10 in one run lasted for about 20 minutes, and in another run it lasted more than 2 hours.

For x86_64, the running times are small, compared to ARM64 - this can be observed in other statistics also; here, it can be readily explained by the differences in timings when searching gadgets (3.2.8) and the total number of possible substitutions.

Table 22: Substitution based ARITH search stats

wanted effect	fixed registers	maximum stack byte size	result	load ct map building runtime	total runtime
rax ← rdx + 0xdeadbeef	-	2^{64}	found	1.29s	1.83s
rax ← rdx + 0xdeadbeef	r10	2^{64}	found	1.46s	2.01s
rax ← rdx + 0xdeadbeef	-	0x400	found	1.26s	2.03s
rax ← rdx + 0xdeadbeef	rbx, rdx, r9	2^{64}	not found	1.40s	1.64s
rdx ← (rcx & 0xbadcaffee) + r9	-	2^{64}	not found	1.4s	24.31s
x0 ← x2 + 0xdeadbeef	-	2^{64}	found	14.47s	19.76s
x0 ← x2 + 0xdeadbeef	x25	2^{64}	found	14.81s	15.41s
x0 ← x2 + 0xdeadbeef	-	0x400	found	14.93s	20.19s
x0 ← x2 + 0xdeadbeef	x25, x5, x27	2^{64}	not found	16.45s	$\geq$ 7200s
x2 ← (x3 & 0xbadcaffee) + x19	-	2^{64}	not found	14.53s	1237s

4 Conclusion and Future Directions

In this documentation, an entire module for ROP chain searching has been presented: the intermediary representation allows for platform-independent logic to be implemented, on top on which heuristic algorithms are built, which successfully allow for fast searches for basic effects of arbitrary shape. The generated chains also take into account the maximum payload size, bytes that should not appear in the final payload, and ASLR offset.

That being said, there are a lot of ways to improve the current implementation, or the used concepts: multithreading (multiprocessing) routines could be implemented to significantly speed up some portions of the algorithms; the internal representation can be reworked to store the effects, structured elements, or the stack view more efficiently; the existing heuristics (especially the jump / memory access validation phase) could be improved, in the sense of a faster, or a more stable (deterministic) implementation; new search algorithms (based on DAGs, or completely different) could also be introduced.

References

[1] SHACHAM, Hovav, et al. *Return-oriented programming: Exploits without code injection.* Black Hat USA Briefings (August 2008), 2008.

[2] DESIGNER, Solar. *Getting around non-executable stack (and fix).* http://ouah.bsdjeunz.org/solarretlibc.html, 1997.

[3] TRAN, Minh, et al. *On the expressiveness of return-into-libc attacks.* In: International Workshop on Recent Advances in Intrusion Detection. Springer, Berlin, Heidelberg, 2011. p. 121-141.

[4] SPENGLER, Brad. *Pax: The guaranteed end of arbitrary code execution.* G-Con2: Mexico City, Mexico, 2003.

[5] PAPPAS, Vasilis; POLYCHRONAKIS, Michalis; KEROMYTIS, Angelos D. *Transparent ROP exploit mitigation using indirect branch tracing.* In: 22nd USENIX Security Symposium (USENIX Security 13). 2013. p. 447-462.

[6] CARLINI, Nicholas, et al. *Control-Flow Bending: On the Effectiveness of Control-Flow Integrity.* In: 24th USENIX Security Symposium (USENIX Security 15). 2015. p. 161-176.

[7] CARLINI, Nicholas; WAGNER, David. *ROP is still dangerous: Breaking modern defenses.* In: 23rd USENIX Security Symposium (USENIX Security 14). 2014. p. 385-399.

[8] LINCOLN. *Sygate Personal Firewall 5.6 build 2808 - ActiveX with DEP Bypass.* https://www.exploit-db.com/exploits/13834, 2010. (visited 06.06.2022)

[9] SINTSOV, Alexey. *ProSSHD 1.2 - (Authenticated) Remote (ASLR + DEP Bypass).* https://www.exploit-db.com/exploits/12495, 2010. (visited 06.06.2022)

[10] NURMUKHAMETOV, Alexey, et al. MAJORCA: *Multi-Architecture JOP and ROP Chain Assembler.* arXiv preprint arXiv:2111.05781, 2021.

[11] WEI, Yuan, et al. *ARG: Automatic ROP Chains Generation.* IEEE Access, 2019, 7: 120152-120163.

[12] SCHWARTZ, Edward J.; AVGERINOS, Thanassis; BRUMLEY, David. *Q: Exploit hardening made easy.* In: 20th USENIX Security Symposium (USENIX Security 11). 2011.

[13] FRANCO, Joao Vasco Costa. *ROP Chain Generation: a Semantic Approach.* 2019. PhD Thesis. Master's thesis, Computer Science Department, Técnico Lisboa. 2019. url: https://fenix.tecnico.ulisboa.pt/downloadFile/281870113705327/75219-joao-francotese.pdf.

[14] MOURA, Leonardo de; BJØRNER, Nikolaj. *Z3: An efficient SMT solver.* In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008. p. 337-340.

[15] SCHIRRA, Sascha. *Ropper-rop gadget finder and binary information tool.* https://scoding.de/ropper/, 2014. (visited 06.06.2022)

[16] SALWAN, Jonathan. *ROPgadget-Gadgets finder and auto-roper.* http://shell-storm.org/project/ROPgadget/, at least as early as Mar, 2011, 12: 8. (visited 06.06.2022)