BinCAT

BinCAT: purrfecting binary static analysis

Philippe Biondi, Raphaël Rigo, Sarah Zennou, and Xavier Mehrenberger
given_name.family_name@airbus.com

Airbus

Abstract. We present BinCAT, a new open-source static analyzer of binary code. Using abstract interpretation as its core, it currently implements control flow graph reconstruction, value analysis, taint analysis at the bit level and type reconstruction. Analyses can be led either in forward or backward mode. We expect this tool to be more useful than existing ones as it is (i) extensible in terms of analysis capabilities and supported architectures; (ii) scriptable and (iii) fully integrated into IDA. BinCAT is available as free software: `https://github.com/airbus-seclab/bincat/`.

1 Introduction

Motivation We think that most *practical* reverse engineering tools lack advanced analytics features. In particular:

- reliable static data tainting, which can be used to gain quick insight of interesting code paths in a complex function or code base. By tainting data of interest, relevant paths quickly stand out. Backward analysis can also be used to find *where* interesting data comes *from*;
- static value analysis, which is very helpful when code cannot be run dynamically for instance because of architecture constraints, needed privileges, secure boot, etc.;
- type reconstruction and propagation: reconstructing high level types (structures and pointers to structures) and assigning them to variables speeds up the understanding of code.

Nevertheless, a tool offering some of those features but being cumbersome to setup or difficult to integrate in the analyst's workflow would not help. Hence the tool should be able to:

- interact easily with the analysts to allow them to inject their security knowledge and hence increase the analysis capabilities of the tool. Interactions could also be used to guide the tool when it gets too imprecise or for very tricky parts in the program;
- be accessible: computed properties and user assertions have to be expressed in the semantics of the program to be analyzed (i.e. assembly) rather than in the underlying theory of the tool which the analyst may not be familiar with;
- report information at different levels: light weight for quick human processing and more complete, but scriptable, for more advanced research.

Design principles Based on the considerations above we have decided to build BinCAT – which stands for *Binary Code Analysis Toolkit* – a new tool for the static analysis of binary code with the following design principles:

- usable in a standard reverse engineering workflow: we have chosen to be fully integrated in IDA[1];
- guided by real case studies;
- good at tasks where the analyst has no added value;
- emphasizing trust on the analyst, acknowledging that fully automated analysis does not work in practice;
- reporting information at the semantics level of the executable and not in the underlying theory of BinCAT;
- processing the analysis in a theoretical framework, namely abstract interpretation [7], that guarantees the absence of false negatives except those explicitly made by the analyst. This property enables to know which hypotheses have been applied so that the analysts can change their mind and automatically revert (part of) them;
- easily extensible: the mathematical framework of the tool is able to combine as many kinds of analyses as needed with only *one* code exploration engine. All analyses are led in parallel and there is a mechanism that makes different analyses communicate results to each other;

[1] No, radare is not even close to replacing IDA.

- dealing with loop approximation without having to fully unroll them;
- providing only benefits to the user: if BinCAT fails to produce useful results, then the reverser is not blocked and can continue reversing as usual;
- coming with a *scriptable API* to allow advanced users to run the tool in ways that are not reachable from the GUI.

Current features Obviously, those interfaces are only useful if the analysis engine itself provides features that will help the reverser, such as:

- implementation of the semantics of most of the general purpose instructions of the x86_32 instruction set, including segmentation;
- value analysis, both forward and backward;
- data tainting at the bit level, forward and backward;
- type reconstruction and type propagation, forward and backward;
- detection of call stack tampering;
- smart control flow graph reconstruction: BinCAT is able to combine value analysis with the call graph reconstruction, enabling indirect jump resolution and hence going further in code analysis.

Use cases Typical use cases of BinCAT are:

- *static* debugging: the reverser can start the analysis from a concrete state and see what happens as if in a debugger;
- *tainting* a register and/or memory to analyze:
  - where the data is used,
  - where the data comes from;
- using type information provided by the analyst or IDA, combined with value analysis to:
  - recover potential high-level types for data (stack, globals, registers),
  - propagate complex types, for examples from API calls, deep into the code.

Outline The rest of the paper is organized as follows: section 2 illustrates the use of BinCAT and its underlying concepts on several examples; section 3 exposes the main characteristics of BinCAT; section 4 shows its architecture; section 5 concludes this paper while presenting future work.

2 BinCAT in action

The purpose of this section is both to illustrate a typical use of BinCAT and results we may expect on it (see section 2.1), and to explain its functional principle on the analysis of two x86 instructions (see section 2.2).

2.1 A complete example

A typical use of BinCAT is an impact analysis of the parameters of an unknown hash function. For the sake of demonstration, the example is also presented with its source code in Listing 1. This example while artificial motivates the need of a taint analysis at bit-level.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdint.h>
4
5  uint8_t buffer[12] = {0};
6
7  int32_t  __attribute__ ((noinline)) myhash(uint8_t *data, int len)
8  {
9      uint32_t *data_32 = (uint32_t *) data;
10     return data_32[0]^data_32[1]^data_32[2];
11 }
12
13 int main(int argc, char *argv[])
14 {
15     int32_t hash;
16
17     if(argc < 2 || strlen(argv[1]) < 12)
18     {
19         for(int i=0;i<12;i++)
20             buffer[i] = i;
21         memcpy(buffer, "0123456789", 10);
22     } else {
23         memcpy(buffer, argv[1], 12);
24     }
25     hash = myhash(buffer, 12);
26     printf("%x\n", hash);
27     return 0;
28 }
```

Listing 1. Source code of the program to be analyzed

Basic analysis We are interested in the impact of the parameters of the hash function `myhash` when it is called without argument on the command line. That case corresponds to lines 19 to 21 where the buffer to hash is initialized with an increasing counter and partially overwritten by `"0123456789"`. Figure 1 shows the relevant code in IDA.

To statically determine what the code is doing, the analyst runs BinCAT with the default configuration, starting at the initialization instruction (top left `mov eax,0`). Figure 2 shows the resulting listing: every line in gray has been analyzed.

Moreover, each intermediate result can be displayed like a static debugger could do. For example, Figure 3 shows the computed values

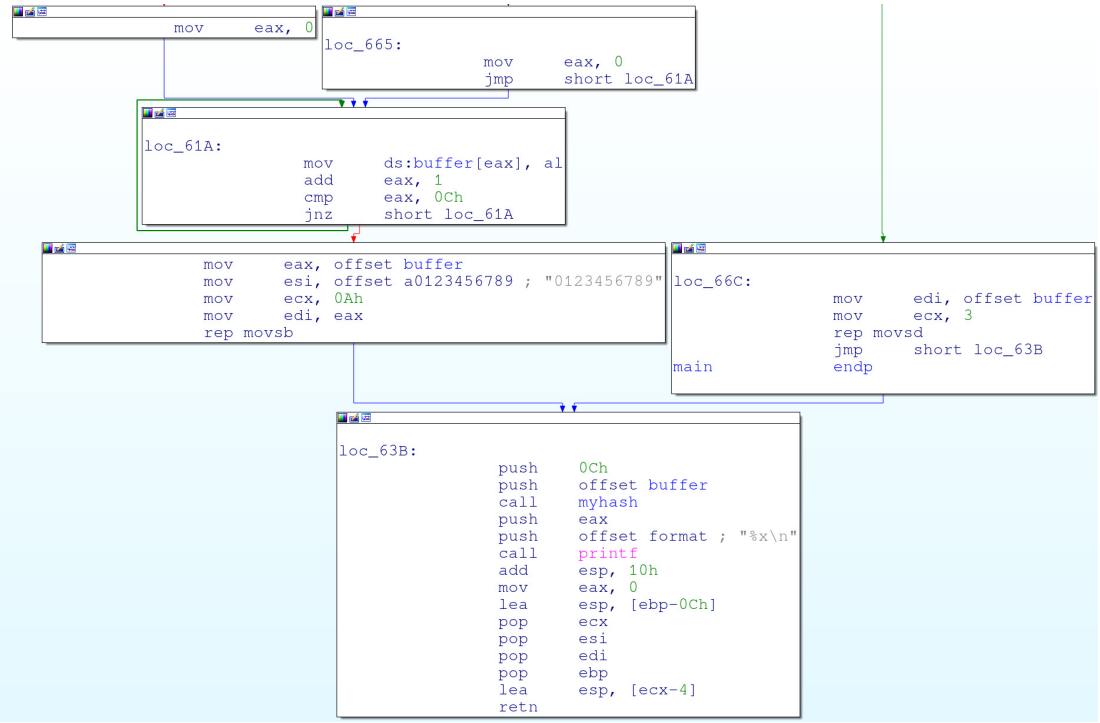

Fig. 1. Simple example disassembly

Fig. 2. Updated disassembly after BinCAT run with default configuration

at the `push eax` instruction, right after the call to `myhash` (4^{th} line of the bottom basic block). Here, we can see the return value of `myhash` in the `eax` register (`0F0E3D3C`). We can also see that `ebp` and `ebx` values are unknown, as represented by the `?` nibbles. As the default configuration sets all registers to an unknown value, this means here that they have never been assigned[2]. We can also see in the corresponding `Hex` view (Figure 3 too) the computed values of the stack:

- the return address of the `myhash` call: `47 06 00 00`
- the first argument (address of `buffer`): `20 20 00 00`
- the size of the buffer: `0C 00 00 00`

Both windows are synchronized and show the state for the currently selected instruction in IDA (`ScreenEA`). When an address is analyzed several times, in case of loops or multiple calls, analysts can select which instance they want to display.

Tainting data In our example, to check the effects of `eax` on the computed hash, we can taint it at the initialization: by right clicking on `eax` in the disassembly `mov eax, 0` and adding a *taint override*, as shown on Figure 4. The keyword `TAINT_ALL` means that all bits of `eax` will be tainted after the analysis of this instruction. Other values are `TAINT_NONE` to untaint all bits and a mask to specify exactly which are tainted. For instance, `0xFF000000` taints only the highest byte of `eax`. To taint the bits 28 to 31 and to mark bits 24 to 27's taint as unknown, one has to use the mask `0xFF000000?0F000000`.

The analysis is then re-run, giving the result shown in Figure 5, where lines in green/light gray indicate that tainted data is manipulated.

Likewise, the registers and memory view use green to indicate *tainted* nibbles (Figure 6).

As we can see, because we initially tainted the `eax` register, taint was propagated to memory (in the `buffer` initialization, the 0A 0B part) and finally back to a *part* of `eax` (`0F 0E`) through the `myhash` calculation.

Restricting analysis If the analyst only wants to study a part of the code, or wants to exclude branches, BinCAT includes a *cut* mechanism, which will stop further analysis from a given point, as shown on Figure 7 where we cut after the initialization loop.

[2] For more complex analyses this could also mean that the analyzer fails to be enough precise.

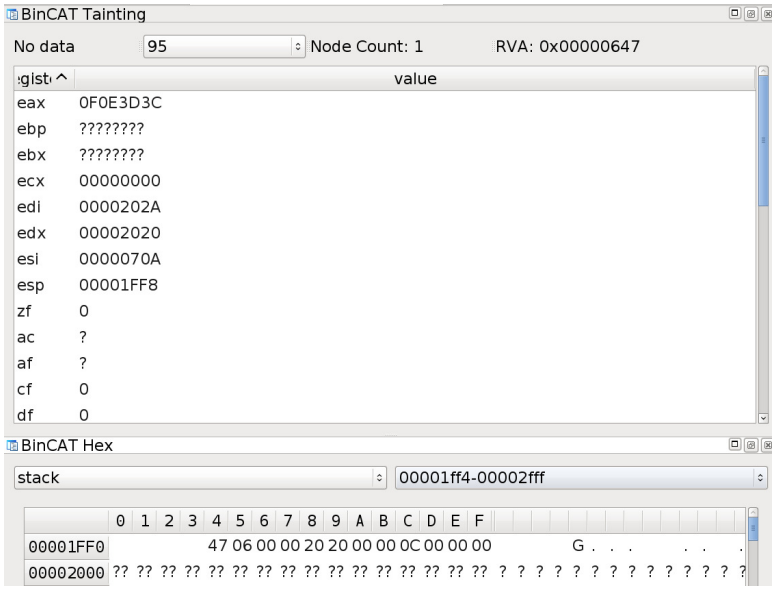

Fig. 3. Computed values of registers and memory.

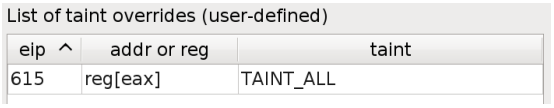

Fig. 4. Analyst-defined taint overrides in IDA

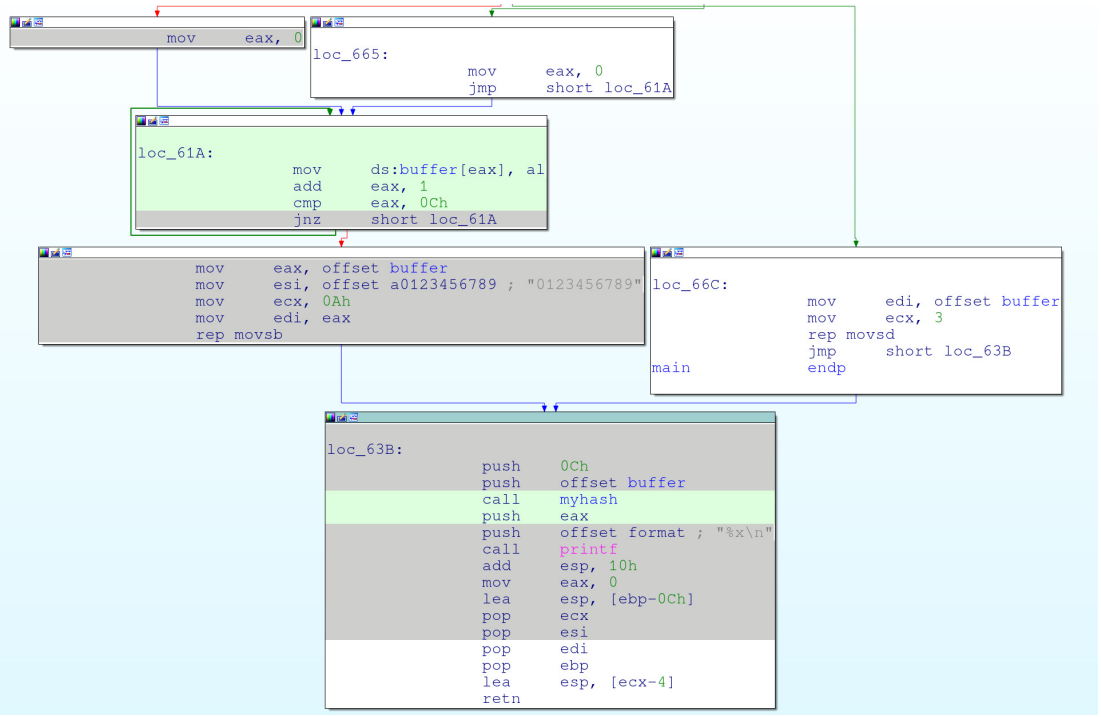

Fig. 5. Disassembly showing instructions processing tainted data

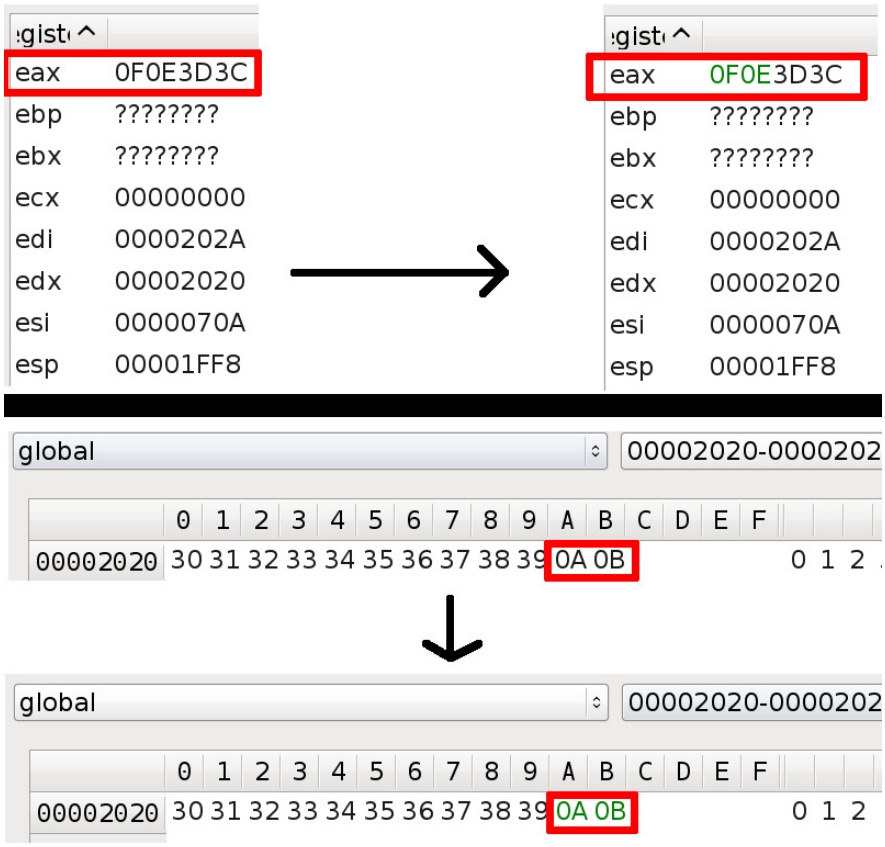

Fig. 6. Registers and memory contents before and after tainting `eax`

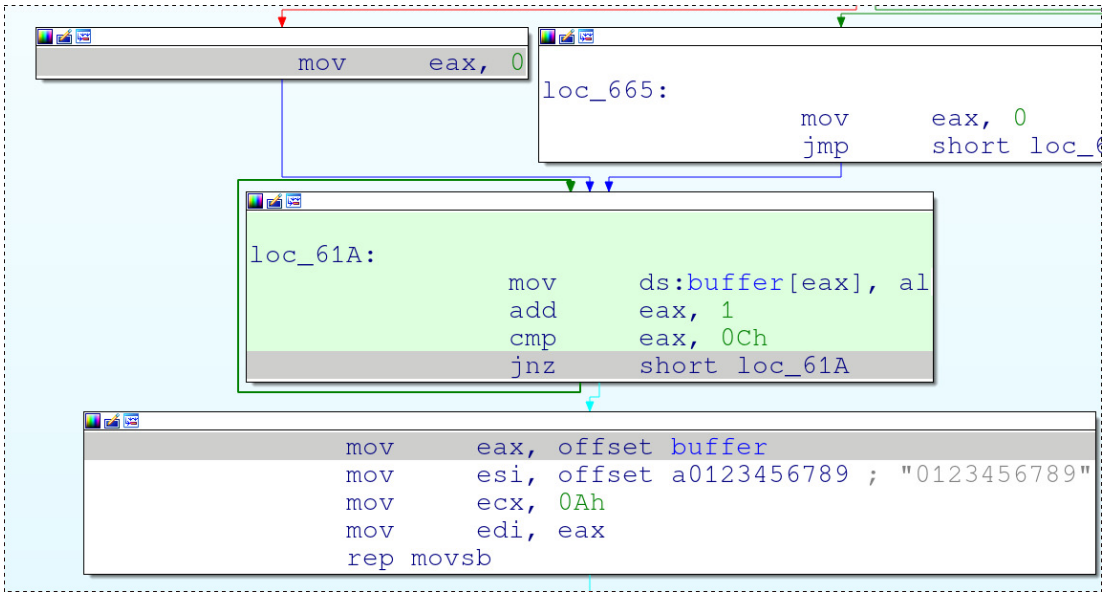

Fig. 7. Analysis stopped after the initialization loop

2.2 Focus on instruction analysis

This section demonstrates in detail the way BinCAT analyses two instructions.

`push eax` In this example, the `push eax` instruction is encountered in a given program. The instruction is decoded into BinCAT's intermediate language:

```
       esp <- (esp - 0x4);
 (32)[esp] <- eax;
```

Listing 2. BinCAT intermediate language representation of `push eax`

The engine will then use the language to compute the side effects of each instruction of the intermediate language. In this case:

1. subtract 4 to `esp`
2. transfer all data information from `eax` to the memory pointed by `esp`, that is
   - the content of `eax` as a 32 bit little endian word
   - the type of `eax`
   - the taint of `eax`

Note that a tainted pointer will taint read and written data when it is dereferenced.

`repe scasd` In this example, the `repe scasd` instruction is encountered. Register `eax` is set to `0` prior to `repe scasd` so `repe scasd` will scan the memory pointed by `edi`, 4 bytes by 4 bytes, until it reaches a 32 bit word different from `0`.

The BinCAT analyzer represents this instruction using one state for each repetition of `scasd`, up to the value specified by the user-supplied `unroll` parameter. Once this value is reached, it applies the widening operator to overapproximate the remaining iterations of the loop (see section 3).

Listing 3 shows the typing metadata that has been deduced by BinCAT, as reported in BinCAT's output file. Since the string operation `scasd` has been performed on `DWORDs`, it has deduced that each accessed address from the memory area should be considered as having a size of 4 bytes.

```
T-mem[0x806d060*1]=DWORD
T-mem[0x806d064*1]=DWORD
T-mem[0x806d068*1]=DWORD
```

Listing 3. Typing data deduced by the analysis of `repe scasd`

3 Main algorithms in BinCAT

This section presents the main algorithms that BinCAT implements.

3.1 Overview

To perform a static analysis on a given executable BinCAT takes the following steps:

1. create an initial abstract state of the program, from user-supplied inputs;
2. compute all reachable states from this initial state until no more new states are discovered. The link between reached states are stored as a graph called an *unfolding of the control flow graph* (uCFG). This uCFG is not exactly the original control flow graph as functions are analyzed as if they were inlined and instructions in loops may sometimes be analyzed several times to improve the precision of the analysis;
3. repeat a fixed number of times:
   (a) explore the computed uCFG backwards to refine results on abstract states,
   (b) explore forward the previously refined uCFG for further refinements of states,
   (c) if the new uCFG is equal to the one of the previous iteration then exit;
4. dump a text version of this uCFG and values into a text file.

The rest of this section contains algorithmic elements on major steps of this algorithm.

3.2 Inputs

To start an analysis BinCAT mainly needs:

- known initial contents of (part of) the memory and registers together with their tainting value. Analysts can mark value, taint or both as uncertain, to perform several analyses in parallel. Note that one does not have to specify every value: if not present a default value is used by the analyzer;
- options for the analyzer (default value for loop analysis, etc.);
- ABI specification: size of addresses, operands, stack width, etc.;
- an entry point address: where to start the analysis;
- import tables and a header file in C to get the signature of the imported functions (used for type reconstruction).

3.3 Intermediate language

The computation of abstract states is not done directly on real machine instructions but rather on instructions of an intermediate language. Using an intermediate language has two major advantages: it allows expressing every side effect of complicated instructions (especially true for x86 instructions), enabling a simpler analysis of the instructions. It also facilitates reusing the analyzer for another architecture by only adding a new decoder for this new architecture to this intermediate language, the exploration engine itself remaining the same.

The intermediate language we use is a variation of REIL [8]. REIL is well suited for the analysis of x86 as it splits complicated x86 instructions into sequences of simple expressions. Nevertheless, in REIL, this normalization is too coarse, leading to a loss of precision during the analysis. Hence, we rather allow some well-chosen complex arithmetic expressions to avoid the generation of temporary fake registers to store intermediate results which would lead to the accumulation of approximations.

Moreover, we have also added a new statement called a *directive* to the intermediate language. It is the core mechanism used to add interaction with an analyst or with results from IDA, for instance. An example of BinCAT intermediate language is shown on Listing 2.

3.4 Abstract states

Remember that statically analyzing a binary program means computing the set of possible values of registers and memory that the program can take at any reachable instruction. We call this set of possible values an *abstract state*.

At a given instruction, an abstract state mainly contains:

- the address of the instruction corresponding to the computed state;
- a context of analysis (like size of operands, addresses, etc.) as it may vary during the lifetime of the program;
- an abstraction of the contents of the memory and the registers called an *abstract value*. An abstract value represents, for each bit of the memory and register, a *set* of possible properties that are true for that bit. Currently, implemented properties are: `Bit`, `Taint`, `Type`:
  - `Bit`-property is either: 0, 1 or unknown ($\top$). Being unknown means having value 0 or 1 which enables leading several analyses in parallel.
  - `Taint`-property is either: tainted, untainted or unknown ($\top$).

- `Type`-property is a property of a sequence of bytes and can be either a C type or unknown ($\top$). `Type`-properties are introduced by either specific instructions such as `repne scasb` or by the analysis of the call of an external function whose signature is known.

In the following these properties are called *abstract domains*.

Being a tool based on the theory abstract interpretation (see [7] for a complete formalization), BinCAT has to respect some characteristics:

- an abstract domain is a complete lattice, that is a partial order $\sqsubseteq$ such that every set of values has an upper bound ($\sqcup$) and hence a lower bound ($\sqcap$). In our case, it means that we have to extend `Bit`, `Taint`, `Type` with a bottom value representing an undefined property. Let us denote $\bot$ this special value;
- it implements a monotonous function that links abstract and concrete states called a concretization function γ. The concretization function of the abstract domain `Bit` is $\gamma(\bot) = \varnothing$, $\gamma(\top) = \{0, 1\}$, $\gamma(\mathrm{Z}) = \{0\}$, $\gamma(\mathrm{O}) = \{1\}$. Concretization functions for `Taint` is very similar. For `Type` it is such that $\gamma(\bot) = \varnothing$, $\gamma(\top) = \{\text{C types}\}$, $\gamma(t) = \{t\}$ with t a C type. Note that an abstract value corresponds to a set of concrete value. This is due to the fact that one wants to avoid false negatives and hence the computation of loops may be overapproximated;
- it has to implement an abstract function for each concrete function (that is functions on real machine values) of the intermediate language. They are used to compute the successor of a given abstract state with respect to a given instruction in the intermediate language. To ensure that the abstract counter part $F^{\#}$ function overapproximates the corresponding concrete function F one has to prove that $F \circ \gamma \subseteq \gamma \circ F^{\#}$. For instance, in the `Bit` domain the result of the addition $+^{\#}$ of Z and O is O is a sound approximation of $+$
- every abstract domain of infinite height has a special operator called a *widening* operator. This operator denoted ∇ is such that any sequence of the form $y_0 = x_0, y_1 = y_0 \nabla x_1, \ldots, y_{n+1} = y_n \nabla x_{n+1}, \ldots$ where x_i are abstract values is ultimately stationary. This operator will be used to compute an overapproximation of the loop behaviors. While our abstract domains are finite, we still define such an operator to make loop computation faster. For `Bit`, we define ∇: $0 \nabla 0 = 0$, $1 \nabla 1 = 1$, $0 \nabla 1 = 1 \nabla 0 = \top$, $\bot \nabla b = b \nabla \bot = b$ and $\top \nabla b = b \nabla \top = \top$ for any b. The widening $a \nabla b$ for the two other abstract domains are very similar: if $a = b$ the result is a, otherwise the result is $\top$ except if one of them is $\bot$. In that latter case, result is the other operand.

Induced properties The above mentioned properties on abstract domains are the core mechanism to ensure that:

1. the main algorithm terminates;
2. the computed uCFG contains all the reachable states from a given initial state, i.e. we can prove it generates no false negative[3] except those due the propagation of choices explicitly made by the analyst.

3.5 Building the maximal unfolded control flow graph

The uCFG is created iteratively starting from a given initial abstract state built from the configuration provided to BinCAT. Then:

1. at step n, a set of states in the uCFG are marked to be explored. They are shown as blue rounded rectangles in Figure 8. By construction, they are always leaves of the uCFG.
2. one state ($state_4$ in the figure) is chosen and removed from this set.
3. (Figure 8 step 1) instruction pointer (EIP), segment registers (CS, DS, SS...) and context (16/32 bit mode for code or data...) are extracted from the chosen state and given to the decoder.
4. (Figure 8 step 2) the decoder translates the pointed instruction into a list of intermediate language statements.
5. (Figure 8 step 3) these statements are then applied to the state ($state_4$) to create another state ($state_5$): abstract values for the memory and registers are updated according to the intermediate language statements. At this point, the analyzer may not be able to interpret a statement because it does not have enough data in the state. For instance, it may encounter a *jump* whose target is the value of a register. If this value is unknown or imprecise, the analysis cannot continue. In this case it will stop, asking the analyst for instructions.
6. finally, the new state is compared to all known states of the uCFG at this instruction. A state $s1$ is lower than a state $s2$ if it has a lower value in every abstract domain (remember that an abstract domain is equipped with a partial order). Given two states and an abstract domain every register and every memory byte are pairwise compared. Two cases are possible:
   - if it is not lower than a previous one, it is added to the uCFG as a child of the state it has been computed from ($state_4$) and it is added to the states to be explored (Figure 8 step 4).

[3] While BinCAT relies on the framework of abstract interpretation that is proven to produce sound analysis results, implementation bugs and unimplemented features (e.g. seldom used processor flags are currently not completely implemented) can cause it to generate false negatives.

– if it is lower, i.e. there exists a state in the uCFG that is a superset of the abstract values held by this state ($state_5$) then it is dropped.

7. at this point, the algorithm loops and the next unexplored state is selected until there are no more unexplored states.

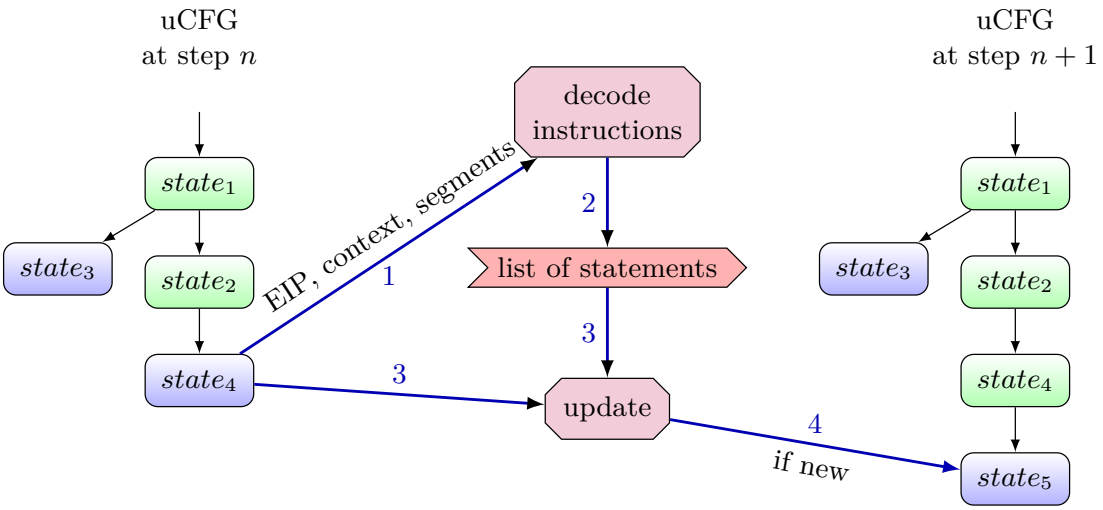

Fig. 8. The uCFG update operation; states to be explored are in blue; fixpoint is reached when there is no state left to explore.

To ensure termination of this analysis (step 6), a counter is held on each instruction of the uCFG. If a threshold is reached, the abstract values of the memory and registers are over-approximated to take into account all past and future values. This is done by joining all the previous abstract values computed at this instruction and then widening the result with the new value. Immediately using the widening operator is also a possibility but the loop computation is more precise when this operation is applied only after a few unrolls of the loop.

3.6 Backward analysis

Backward analysis runs on a given unfolded control flow graph – previously computed during a forward pass – and performs only a refinement of the previously computed abstract states that computes only states that can be lower than previously computed at a given instruction. An example of refinement occurs with type information: when a call of a function having a known signature is encountered, the interpreter can deduce the type of its parameters. This type information has to be propagated backward

along the path as far as possible. Backward propagation stops either because the sink of the control flow graph is reached or the information is lost because of an approximation in the property field of a given abstract state.

4 Architecture

BinCAT has been designed to be fully integrated in IDA, running on Linux, Windows or macOS. Therefore, a few software components have been developed:

- the *BinCAT analyzer* analyzes binaries (see section 3);
- the *python wrapper* allows manipulating analysis results as Python objects;
- the *web service* (optional) enables running the *BinCAT analyzer* on a remote machine;
- the *IDA plugin* is directly used by the analyst, launches analysis from IDA, calling the BinCAT analyzer either directly or through the web service, and displays results.

4.1 BinCAT analyzer

The *BinCAT analyzer* is written in OCaml. It can be easily compiled on Linux using the `Makefile` provided, as a shared object (`.so`) and a standalone command-line executable. Compiling it on other platforms can be challenging and time-consuming.

It takes as inputs (i) an `ini` configuration file (see section 3), typically generated by the IDA plugin; (ii) a user-supplied *analyzed binary*, from which code and mapped sections will be loaded; (iii) optionally a marshalled uCFG containing former analysis results, which is used in case of a backward analysis or a refined forward analysis.

It then produces (i) an `ini` output file; (ii) a file containing a marshalled version of the uCFG, used to run backward analysis from a state contained in this analysis; (iii) a `dot` file containing a representation of the uCFG; (iv) a log file.

Its inputs and outputs are summarized on Figure 9.

4.2 Python wrapper

The python wrapper parses the ini output file of BinCAT, and builds Python objects:

– `CFG` objects represent the unfolded Control Flow Graph, and provide methods to access each State of the uCFG;
– `State` objects represent nodes of the uCFG, and provide access to values and taint stored in memory and registers. There may be several State objects for a given EIP, e.g. for different loop iterations;
– `Value` objects store either register names, memory addresses, or values and taint that are stored in registers or memory.

Objects also contain helper functions, to display stored values, compare states and display a human-readable text similar to the output of the common `diff` tool. These functions are also used by the project's integration tests.

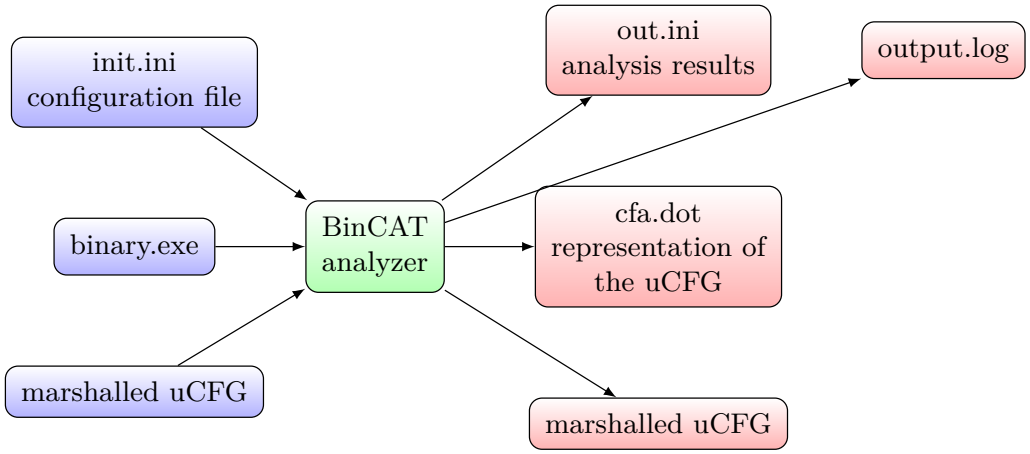

Fig. 9. BinCAT binary inputs and outputs

4.3 Web service

A web service is provided to run BinCAT on a remote machine. This facilitates the following use cases:

– Running IDA on macOS or Windows where compiling BinCAT is not supported;
– Offloading computation on a remote, powerful machine;
– New BinCAT users can get started quickly, by simply installing the IDA plugin.

This service is written in Python, using the flask framework.

Docker image The web service is distributed as a docker image, which may be built using the provided `Dockerfile`.

4.4 IDA plugin

An IDA plugin has been developed in Python. It provides analysts with the following features:

- View value and taint at the current instruction for registers and memory, where colors help analysts quickly identify tainted values;
- Interactively add taint overrides by clicking on a register, to force tainting or removing taint on a register at a given instruction;
- Load default configuration files, depending on the platform the analyzed binary runs on (Windows, Linux, macOS), and generate relevant configuration files;
- Allow analysts to edit the input configuration file for BinCAT to tweak settings;
- Highlight instructions that have been analyzed in gray or green, depending on whether their output depends on tainted data;
- Remember custom configurations as part of the IDB (IDA database) for each instruction;
- Run BinCAT locally, or through the *web service*.

4.5 Testing

Writing the semantics of an instruction set is a tedious, time-consuming and error-prone process. So, to avoid regressions and mistakes, BinCAT includes tests to validate the semantics. Tests are written in Python and use the `py.test` framework.

5 Conclusion

5.1 Related work

Many tools designed for advanced binary code analysis now exist, which was not the case when the BinCAT project was started (2013). So the purpose of this section is *not* to compare BinCAT to those projects but to detail their features and see if they match the goals we discussed in the first section.

Some are based on purely static approaches, others use dynamic ones. [14] provides a good survey of tools and methods in the context of automated vulnerability exploitation. We will only cover tools that are (or could be) relevant to our context of interactive usage. This is summarized in Table 1, where SE stands for *symbolic execution* and AI for *abstract interpretation*. Mode is the way an analysis is run: it can

be either dynamic (code to analyze is executed), static (code to analyze is just looked at) or both. This table also indicates whether these tools support the main features of BinCAT: IDA integration, value analysis, CFG reconstruction, data tainting, and type reconstruction.

Tool	Mode and theory	Computed properties	IDA	Reference
vivisect	Both (SE)	Tainting	No	[3]
angr	Both (SE + AI)	Value, Tainting	No	[14]
Triton	Dynamic (SE)	Tainting	No	[9]
Ponce	Dynamic (SE)	Tainting	Yes	[10]
McSema + KLEE	Static (SE)	Value	Limited	[17] [15]
miasm2	Static (SE)	Value	Yes	[6]
amoco	Static (SE)	Value	No	[16]
BAP	Both (SE + AI)	Tainting, Value	Yes	[5]
BINSEC	Both (SE + AI)	Value	Yes	[1]
BINDEAD	Static (AI)	CFG, Value	No	[13]
Jakstab	Static (AI)	CFG, Value	No	[11]
HexRaysCodeXplorer	Static	Typing	Yes	[2]
HexRaysPyTools	Static	Typing	Yes	[12]

Value here means *value analysis*. For *Abstract Interpretation* it corresponds to finding a set of value that a register or a memory byte can take during an execution while for *Symbolic Execution* it corresponds to finding a model that satisfies a path predicate. CFG stands for CFG reconstruction.

Table 1. Tool comparison

Fully static approaches There are two major categories of static automatic tools: those based on symbolic execution (by using SMT-solvers) and those based on abstract interpretation (like BinCAT).

Tools based on abstract interpretation The principle of leading in parallel a control flow graph reconstruction and a value analysis based on abstract interpretation comes from [11]. Authors of this article also provide an open source implementation of their technique as a tool called *jakstab* which suffers from the following drawbacks: the exploration engine has no interaction with the analyst; results are provided at the mathematical level and there is no GUI.

bindead [13] implements also this technique but with a more expressive set of properties (relational constraints on memory content for instance) and a GUI but without interactivity with the analyst.

Another tool, *BAP* [5] supposes on the contrary a well-formed binary program as their analyses rely on the notion of stack frames that cannot

be trusted in the context of malware analyses. Moreover, this tool exposes several ABIs to lead analyses but corresponding implementations are not very elaborated.

Finally, *binsec* framework [1] is currently frozen for its static part while the DSE (Dynamic Symbolic Execution) part is focused on properties linked to deobfuscation.

Furthermore, for all of them the integration with IDA seems to be an afterthought rather than a design choice, and is limited to displaying analysis results rather than enabling rich interaction with analysts.

Tools based on symbolic execution

McSema, while not a symbolic execution tool in itself, is interesting as it translates x86 and x86_64 code into LLVM intermediate representation. It can thus be combined with tools such as *KLEE* to provide symbolic execution of binary code. It also does not provide any IDA integration or GUI.

miasm is quite powerful: it includes symbolic execution, emulation, data slicing, etc. It is also able to emulate APIs and system calls. Being developed in Python, it interfaces easily with IDA and already includes several scripts to interact with the engine from IDA. However, data tainting support is not yet included in the main branch.

amoco is quite similar to *miasm* but does not implement API emulation nor data tainting and lacks IDA integration.

Misc Two interesting plugins for IDA provide very useful features for the reverser: *HexRaysCodeXplorer* and *HexRaysPyTools*. In particular, they are able to rebuild structures from decompiled code in Hex-Rays. But they do not provide taint analysis, and are limited to code decompiled by Hex-Rays, which is not always accurate (and very expensive!).

Comparing static analyses by symbolic execution and by Abstract interpretation Static analysis by abstract interpretation is a powerful technique to analyze programs as:

- abstract interpretation comes with a way to combine as many data flow analyses as needed (typing, data tainting, etc.) in parallel, while communicating to refine their mutual results. This is not the case for symbolic execution;
- static symbolic execution can be (like any data flow analysis) encoded as an abstract domain and hence provide other analyses with models

computed from a path predicate to make them more precise. The reverse is not necessarily true;
– abstract interpretation deals with loop computation without having to fully unroll them while having a widening operator which ensures no false negatives are generated. This is suitable if one wants to prove the absence of a vulnerability in a code or help the analysts to revert their analysis hypotheses.

Dynamic approaches BinCAT was designed from the start to be a static analysis tool, but dynamic approaches are also interesting as they provide the analyzer with a concrete context. So this section covers some tools that can help reversers but are not directly comparable with BinCAT.

Ponce, which is a wrapper around *Triton* integrated with IDA is very interesting for the reverse engineer as it can be used as part of a standard IDA workflow.

angr is a very complete framework which includes a lot of approaches, including static value set analysis, and includes bridges to combine them. However, it lacks IDA integration and is oriented towards automated vulnerability discovery rather than manual analysis. But, given it is implemented in Python, it would be a great candidate for an integration with IDA.

vivisect completely lacks documentation and does not seem to include any interface with IDA.

5.2 Future work

While BinCAT is already usable in its current state, evolutions are, of course, planned to make it more useful.

Testing While the unit tests written in Python are helpful, they are tedious to write. To speed up testing instruction semantics, we plan to use `testi386` from QEMU, which is conceptually very simple: an ELF binary which computes values using a wide range of CPU instructions and arguments, and which uses `printf` to display the results.

QEMU compares the emulated output with the results of an execution on actual hardware. BinCAT will just need a stub of `printf` to be able to use the tests.

Additional ISA As the analyzers are all based on the intermediate language, adding support for another architecture “only” requires implementing a decoder. The two obvious candidates are AMD64 and ARM.

New analysis domains As the results are heavily tied to the abstract domains, adding new ones can improve certain parts of the analysis or even provide new functionalities. Some options include:

- a more abstract memory domain to model ranges of values for each byte of memory, like the Value Set Analysis [4];
- a domain encoding a symbolic analysis, potentially combined with a SMT solver to get logical invariants;
- a shape analysis to compute how dynamically created objects are linked;
- relational domains like polyhedra to compute invariants into a given stack frame.

C++ helpers `C++` is particularly tedious to reverse, mostly because of indirect calls through *vtable* pointers and complex object structures. BinCAT's value analysis could be very helpful to recover the call graph, while type reconstruction could help recover object structures.

In particular, a full analysis starting from nothing is impossible but starting from parts of a concrete state copied from a live process would give the analysis engine enough to start analysis.

Other improvements Various improvements could also be implemented:

- handling of self-modifying code: while the engine itself does not make any assumption about the code, the decoder does not use the memory abstraction and only gets the code from the binary;
- distinguishing taint sources: currently the taint engine only supports one taint. The idea would be to tag different taint sources to be able to distinguish them, as explained in [18].
- data tainting based on the control flow: i.e. tainting data manipulated in a branch depending on tainted data. As overtainting is likely, testing is needed to decide if a better option would be to allow explicit tainting in a branch based on an analyst's decision.

Acknowledgements This project has been partially funded by DGA-MI[4]. Authors would also like to thank Mouad Abouhali for his active participation to the project, as well as anonymous referees for their valuable comments.

[4] Direction générale de l'armement – Maîtrise de l'information.

References

1. Binsec. http://binsec.gforge.inria.fr/.
2. Rodrigo Branco Alex Matrosov, Eugene Rodionov and Gabriel Barbosa. Hexrays code xplorer. https://github.com/REhints/HexRaysCodeXplorer.
3. atlas. Vivisect. http://visi.kenshoto.com/viki/MainPage.
4. Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, 2010.
5. David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the Conference on Computer Aided Verification*, 2011.
6. CEA. miasm2. https://github.com/cea-sec/miasm.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, NY, 1977.
8. Thomas Dullien and Sebastian Porst. REIL : A platform-independent intermediate representation of disassembled code for static code analysis. *In Proceeding of CanSecWest*, 2009.
9. Jonathan Salwan et Florent Saudel. Triton : Framework d'exécution concolique et d'analyses en runtime. In *SSTIC*, 2015.
10. Alberto Garcia Illera and Francisco Oca. Ponce. https://github.com/illera88/Ponce.
11. Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *Proc. 10th Int. Conf. Formal Methods in Computer-Aided Design (FMCAD 2010)*, pages 43–50, 2010.
12. Igor Kirillov. Hexrayspytools. https://github.com/igogo-x86/HexRaysPyTools.
13. Bogdan Mihaila. Bindead. https://bitbucket.org/mihaila/bindead/wiki/Home.
14. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.
15. KLEE team. Klee. http://klee.github.io/docs/.
16. Axel Tillequin. amoco. https://github.com/bdcht/amoco.
17. McSema Contributors https://github.com/trailofbits/mcsema/blob/master/CONTRIBUTORS. Mc-semantics / mcsema. https://github.com/trailofbits/mcsema.
18. Babak Yadegari. *Automatic Deobfuscation and Reverse Engineering of Obfuscated Code*. PhD thesis, University of Arizona, 2016.